Securing the Networking System using Linux Security Modules

Bogdan Davidoaia, Silviu Grigore
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Bucharest, Romania
Email: bogdan.davidoaia@cti.pub.ro, silviu.grigore@cti.pub.ro

***Abstract*—The security of computer systems is becoming more and more important due to the increased usage and integration of computers in a wide range of domains. A key aspect is represented by the security of distributed systems where network communication can give rise to vulnerabilities.**

This paper proposes a solution for securing network system access in Linux that is easy to use and allows flexible configurations. This is achieved through a kernel module that enforces Mandatory Access Control rights by using Linux Security Modules. The implementation uses security hooks to restrict user access to specific socket operations and filter network packets.

I. Introduction

As the number of computer systems used across various fields increases, so does the security risks associated with using these systems. Large distributed systems can be vulnerable both to failures and premeditated malicious attacks. Many of these systems manage large amounts of money or confidential data and, as such, they require additional security measures.

Security in Linux uses a Discretionary Access Control (DAC) [1] model, which restricts access to objects based on user or group identity. Objects in Linux are represented by regular files, sockets, character and block devices, network devices, etc. This DAC approach allows users and groups to change access restrictions and also delegate them to other users.

Mandatory Access Control (MAC) [2] also restricts users and groups access to resources, but it doesn't permit users to make policy decisions and/or assign security attributes. The restrictions on resources are implemented by enforcing a set of static rules. This approach is preferred for securing network systems because it is less vulnerable to attacks generated by malicious or flawed applications ran by privileged users that can damage or destroy the system.

Systems can enforce both DAC and MAC at the same time, where DAC restricts access to objects in a discretionary manner and MAC imposes additional restrictions upon the DAC layer. This layered approach is commonly used in security systems as it offers much more control and flexibility in deploying different security policies.

Linux Security Modules (LSM) [3] is a general purpose security policy framework, which is a standard part of Linux since kernel version 2.6. LSM was designed to be truly generic, conceptually simple, easy to use, efficient while introducing minimum overhead. LSM facilitates the creation and stacking of loadable kernel modules that implement various access control mechanisms and security policies.

The interface provided by LSM consists of a set of callback functions that are called before certain operations are performed on kernel objects. These callbacks return an 'yes' or 'no' value, which indicates if the checked operation is allowed. The objects managed by LSM can be tasks, programs, filesystems, files, sockets, network devices and packets, interprocess communication objects.

The security solution proposed in this paper uses LSM to implement MAC over sockets and filter network packets. It is implemented as a loadable kernel module that extracts its security policy from a configuration file. In addition to the LSM API, the kernel module uses NetFilter hooks to analyze and filter network packets.

The main advantage of this solution is that offers fine-grained control over socket operations and, thus, allows complex security policies to be implemented. This presents an advantage over the standard Linux network security module that provides only network packet filtering capabilities.

The rest of this paper is structured as follows. Section II describes related work. Section III presents the architecture of the proposed solution. Section IV describes the implementation details of the developed application. Section V presents the experimental results. Finally, Section VI presents conclusion and issues for future work.

II. Related Work

The increased growth of security needs has given birth, over time, to many security patches and modules for the Linux platform. This section presents the most relevant security solutions developed so far.

A. SELinux

Security-Enhanced Linux (SELinux) [4] is a security mechanism implemented in the Linux kernel, which was originally developed by the United States National Security Agency. It was initially implemented as a patch, but it was redesigned as a kernel module based on LSM API. SELinux is an implementation of flexible access control architecture that supports Type Enforcement, Role-Based Access Control and Multi-Level Security.

The architecture of SELinux consists of two components: the policy decision-making component, which is encapsulated in a security server and the policy enforcement component, which is integrated in objects to which the policy is applied such as processes, filesystems, sockets, IPC objects.

SELinux assigns a security label composed of strings for role, user name and domain to every subject (user and process) and kernel object. The security label components are used in defining rules in policy file, which is loaded in the SELinux kernel module.

The solution presented in this paper is similar to SELinux because it is implemented as a loadable kernel module, it is based on LSM API and it loads its policy rules from a configuration file. However, access control component is simpler and rules are defined using only Linux user and group IDs as subjects.

B. AppArmor

AppArmor [5] is a security module for the Linux kernel, which can be seen as an easier to configure and maintain alternative for SELinux. AppArmor is also implemented using LSM interfaces and provides MAC capabilities upon the traditional Linux DAC.

In the AppArmor security model restrictions are set only on programs and not on users. For each application a profile is defined as a file, which contains rules that specify the access rights to system objects. A profile only specifies restrictions to the access privileges given to the application by the operating system and cannot grant other new rights.

AppArmor can provide two modes: enforcement mode that applies the restrictions and complaining mode that only logs policy violations, but doesn't prevent the operation from being executed. Thus, the complain mode can be used to observe access habits of certain applications.

The solution proposed in this paper also logs attempts at policy violations, but always prevents them from taking place. However, an application level policy was not adopted by this proposal, because the goal is to restrict socket or packet access for all applications running in an user session.

C. Smack

Simplified Mandatory Access Control Kernel (Smack) [6] is a kernel based implementation of mandatory access control that includes simplicity in its primary design goals. Smack is also implemented using the LSM interface and works best with filesystems that support extended attributes.

The Smack security model uses four basic elements: subject (Linux tasks), object (files of all types, Linux tasks and IPC objects), access (read, write, execute, append), label (data that identifies a subject or an object). In the case of a file the label can be stored as an extended attribute.

Using these elements, access rules can be defined by specifying subject and object labels and the access mode. Generic rules can be defined by using special labels such as *, ^, _, ? (e.g. any access requested by a task labeled "*" is denied).

Although the solution proposed in this paper uses rules similar to Smack's, it doesn't use labels to identify subjects or objects as it uses the identity given to them by the operating system (user or group ID for subjects, IP address and port number for sockets and packets).

D. Iptables

Iptables [7] is an user space application that is part of the standard Linux distributions and allows packet filtering, network address translation and general packet header modification rules to be defined in order to implement complex firewall policies.

Iptables operates on the kernel firewall tables for IPv4 and defines filtering rules for user defined chains or predefined chains such as PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING. For each predefined chain Iptables uses a Netfilter hook in the Linux kernel, which is called for each packet that traverses that chain.

The solution presented in this paper also uses Netfilter [8] hooks to filter incoming and outgoing packets. It doesn't provide packet header modification capabilities as its goal is only to drop or accept network packets according to the configured network security rules.

III. Design

The main goal of this paper is the design and development of a Linux network security module that implements Mandatory Access Control that is easy to configure, flexible and offers a fine-grained control over socket operations. This solution tries to distinguish itself from previous Linux security modules by being conceptually simple, easy to extend and very specific to network security.

To achieve this, an analysis of security mechanisms provided by the Linux kernel has been done and implementation as a loadable kernel module seems to be the most flexible and non-intrusive because it doesn't require modifying the whole kernel. The alternatives presented in Section II reinforced the idea of implementing the security mechanism as a loadable kernel module.

The solution is built upon Linux Security Modules to restrict socket access and Netfilter to filter packets. The analysis showed that LSM proved to be a powerful framework that can be used to address a variety of security problems, while Netfilter is standard and simplest way to filter network packets in the kernel.

The main advantages of LSM are the fact that it is a standard part of the Linux kernel and, as such, it doesn't require explicit installation and is available for most Linux distributions. LSM is easy to use and introduces little overhead.

A. Security module features

The main goal of the security module is to manage subject access to network sockets and packets in a MAC fashion and, hence, its features are focused on enforcing these types of security policies. The objects managed by the module could

also include types of system resources other than network sockets and packets, but this is subject for future work.

The subject access to network sockets and packets is restricted using a set of security rules that are defined in a configuration file. The configuration file is read when the module is loaded. The security rules can be changed while the module is loaded by using a helper program, which specifies a new configuration file.

The security rules can be divided into socket rules and packet rules. A socket rule refers to socket operations, IP addresses and/or port numbers. A packet rule refers to transport protocols, source and destinations IP addresses and/or port numbers. A security rule can specify whether an operation is permitted or forbidden.

For system administrators to be able to track potential access violation attempts, the security module records all the operations prevented from being executed and dropped packets after security rules checks. These are logged using the syslog daemon.

B. Security module components

The security module consists of five major elements: configuration file parser, rule set checker, dynamic configuration loader, LSM hooks component and Netfilter hooks component.

The configuration file parser is responsible analyzing the configuration file and generating a set of rules that dictates the security policy. It is invoked when the module is loaded with a predefined path to the configuration file and each time a new configuration is set dynamically.

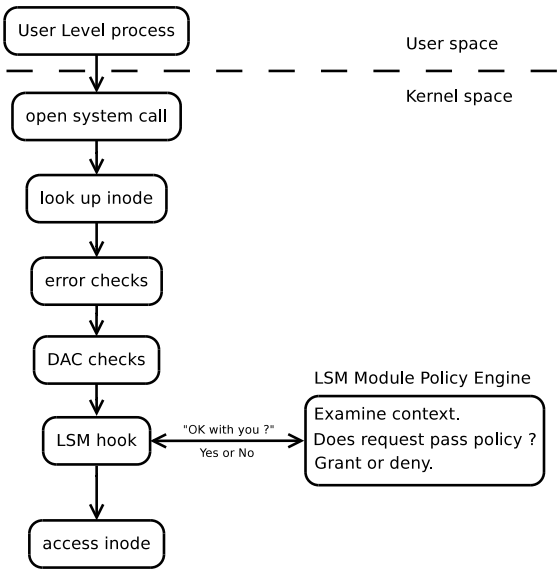

Fig. 1. LSM Hook Architecture

The parser generates a rule set stored in an internal representation, which allows efficient rule lookup. The rule set checker is called for every access to a network object and it looks for rules that may deny that access. If no such rule is found, then the default decision is to permit the access.

The dynamic configuration loader is associated with a character device that permits ioctl operations. These operations are used to specify a new configuration file to generate new rules that would replace the old ones. To facilitate an easier usage of this feature, a helper program was developed to communicate with the character device.

The LSM hooks component consists of a set of callback functions that overrides the corresponding default LSM hooks. These hooks are called just before the kernel would have accessed the object as seen in Figure 1. The callback functions use the rule set checker component in order to verify if access should be granted.

The defined callback functions cover all socket system call operations: create, bind, connect, listen, accept, sendmsg, recvmsg, getsockname, getpeername, getsockopt, setsockopt, shutdown. All these individual hooks can be used to offer fine-grained access control over sockets.

The Netfilter hooks component is conceptually similar with the LSM hooks component, the difference being that the callback functions are used to accept or drop incoming or outgoing packets. This component alone can offer firewall-like functionality.

IV. Implementation

As was stated in Section III, the application is composed of five major modules, for which implementation details will be presented in the following subsections.

A. The configuration file parser

The configuration file parser is implemented as a user-space process, which is invoked when the kernel module is loaded by using the *call_usermodehelper* function provided by the Linux kernel. The module waits for the invoked process to finish loading all the rules from the configuration file. The user-space process can also be started independently, in which case it reloads the kernel module configuration.

The reason why a user-space implementation was chosen for the configuration parser is because reading and processing files directly in kernel-space is unsafe and the kernel doesn't support it by default. Therefore, file processing is done in user-space and the extracted data is transferred to the kernel module through ioctl calls.

An example of the configuration file format can be seen in Figure 2. The configuration file is composed of lines containing statements that specify the default policy enforced for cases not covered by the rules, the scope of the rules (user or group) and operation specific rules. The configuration file can also include single line comments started by the '#' symbol.

The operation specific rules have the following format:

- SOCKET CREATE ⟨protocol⟩ ACCEPT | DENY
- SOCKET BIND | LISTEN ⟨source ip⟩ ⟨source port⟩ ACCEPT | DENY
- SOCKET CONNECT | ACCEPT | SENGMSG | RECVMSG ⟨source ip⟩ ⟨source port⟩ ⟨dest ip⟩ ⟨dest port⟩ ACCEPT | DENY

- SOCKET GETSOCKOPT | SETSOCKOPT ⟨option⟩ ACCEPT | DENY
- SOCKET SHUTDOWN ⟨how⟩ ACCEPT | DENY
- PACKET CONNECTION ⟨protocol⟩ ACCEPT | DENY
- PACKET PROTOCOL ⟨protocol⟩ ⟨source ip⟩ ⟨source port⟩ ⟨dest ip⟩ ⟨dest port⟩ ACCEPT | DENY

The operation specific rules are associated with the user or group declared before them by using either the USER or GROUP statement. If there is no such previous statement, the defined rules are applied to all groups. In addition, if a rule is malformed, the parser ignores it and displays a notification message to the standard output.

```
DEFAULT_POLICY ACCEPT

# rules for user root
USER root
SOCKET CREATE tcp DENY
SOCKET BIND 12.212.123.45 * ACCEPT
SOCKET CONNECT * 123 12.212.113.45 * DENY
SOCKET LISTEN * * DENY
SOCKET ACCEPT * * * * DENY
SOCKET SENDMSG * * * * ACCEPT
SOCKET RECVMSG * * * * ACCEPT
SOCKET GETSOCKOPT KEEPALIVE ACCEPT
SOCKET SETSOCKOPT BROADCAST ACCEPT
SOCKET SHUTDOWN * * RD DENY

# rules for group student
GROUP student
PACKET PROTOCOL tcp * * * * DENY
PACKET CONNECTION tcp ACCEPT

SOCKET * DENY
PACKET * DENY
```

Fig. 2. Configuration file example

Due to the fact that the kernel only operates with user and group IDs and not their name, a name to ID translation step is required when parsing the configuration file. The translation is done by parsing the */etc/passwd* and */etc/group* files and storing the associations between user and group names and IDs.

The data extracted from the configuration file is stored in a structure that keeps the default policy, the policies that apply to all users and groups and the lists containing the rules specific to each user and group.

B. The dynamic configuration loader

The dynamic configuration loader is the part of the kernel module that handles the ioctl call through which the data extracted by the parser is received. The structures used to store the rules are slightly different for the parser and the kernel rule set.

This means that the dynamic configuration loader can't copy all the data using a single *copy_from_user* call. Instead, the loader has to navigate all the pointers from the user-space structure and copy the data step by step while building the kernel-space structure.

The kernel-space structure consists of the default policy, the list of rules that apply to all users and groups and two hash tables, which contain the rules specific to each user and group. Hash tables were used instead of list in order to provide faster rule lookup. Each hash item contains the default policies applied to the associated group or user and an array of lists for each operation type.

C. The rule set checker

The rule set checker provides a lookup function for the rule set. This function is called in every LSM and Netfilter hook to check access rights for the requested operation.

The input values for the rule set checker are the operation type and a structure containing operations specific data. The structure is the same as the one used internally by the dynamic rule loader. Depending on the combination of rules, the returned result can be 'accept' or 'deny'.

To determine if an operation is permitted or not, the rule set checker first obtains the user and group identifiers associated with the process that initiated the operation. The user ID is used to check if there are any operation rules specific to that user, in which case the rule set checker searches for a matching rule. If there multiple matching rules, the one defined last in the configuration file is used to determine the result.

If there is no rule that matches the parameters of the operation, the default policy associated with that user is consulted. If the result cannot be inferred by using this information, the rules that apply to all users are checked in the same fashion. Unless the result can be determined, the same procedure is applied for the group specific rules. The default policy is consulted when no rule from the configuration file matches.

For each operation specific rule parameter, a '*' symbol can be declared in the configuration file, which means that the rule matches for any of the values permitted for that parameter.

D. The LSM hooks

The implemented LSM hooks are set of socket specific functions that are called before any socket operation is performed, to check if it is permitted. These hooks are stored in a *security_operations* structure, which is registered with the LSM framework.

As not all available LSM hooks are needed, the unused hooks from the *security_operations* structure are set to dummy functions, provided by the Linux kernel.

In each security hook the following operations are performed:

- the data required for rule matching is extracted from the hook's arguments
- the rule set checker is consulted in order to determine if the operation is permitted or not
- the result is logged using the syslog daemon
- the result is returned to the LSM framework

E. The Netfilter hooks

The implemented Netfilter hooks operate only the PRE-ROUTING and POSTROUTING chains, in order to filter only the sent and received packets. These hooks are stored in a *nf_hook_ops* structure, which is registered to the Netfilter framework.

These hooks are associated to two types of rules: connection rules and protocol rules. The protocol rules are static rules that dictate whether a packet will be sent or received (similar to the socket send and receive rules).

The connection rules refers to packets sent and received over a particular connection. If a restricting rule is declared for a certain protocol, then packets associated to connections that were initiated by remote peers are dropped.

For each user, the module keeps a list of connections initiated by him, which are described by the used protocol, the local and remote ip addresses and port numbers. These elements are inserted in the list when a connection is initiated by the local entity and removed when the connection is closed. Sent packets are not restricted, but received packets are checked against the connection list.

Connection initialization and termination is protocol dependent. For TCP, the connection is started when the local entity sends a message with SYN flag set and the ACK number 0, and is terminated by a message with the FIN flag set.

As UDP is not a connection oriented protocol, packets are considered part of the same message stream if they are sent and received only a few milliseconds apart. Thus, when a packet is sent, a timer is started, which is rearmed when the response is received. If the timer expires without a response, the connection is considered closed and its entry is removed from the connection list.

V. Experimental Results

The solution proposed in this paper was implemented and tested on a system that runs Ubuntu 7.10 (with the kernel version 2.6.22-14). The reason this version was chosen and not a more recent one is that, since version 2.6.24 the LSM framework can't be used anymore to create loadable kernel modules, instead LSM modules need to be compiled together with the kernel.

This approach was used to ease the development process as compiling the whole kernel and reinstalling Linux after every change takes a significant amount of time. Still, the solution presented in this paper will work with little changes with newer kernel versions.

The conducted tests were oriented towards validation and not performance measurement. The main reason for this was that our efforts were concentrated on first having a functional version of the solution and future work will include performance analysis.

The validation tests were done by using a predefined configuration file for the security module and writing an user-space program, which tries to invoke each of the operation defined in the rule set. Checking whether a test was successful was done by inspecting the rule set checker results, which were printed by the syslog daemon.

```
DEFAULT_POLICY ACCEPT

# rules for user student
USER student
SOCKET * ACCEPT
PACKET * ACCEPT
SOCKET RECVMSG * * * * DENY
```

Fig. 3. Test configuration file

For example, the kernel module was loaded using the configuration file listed in Figure 3 that forbids the user 'student' from using the socket receive message operation. For this test case two test applications were written.

One of the test applications was deployed on a system on which the security module was not enabled. This application listens for connections on a specific port and for each accepted connection it reads a message and sends back a reply.

The other test application was executed by the user 'student' on a system were the security module was loaded using the configuration specified in Figure 3. This application connects to the first one, sends a message and waits for a reply.

The behaviour was that the message sent by the second application was successfully received by the first one, which in turn sent a reply that was not received by the remote peer. Because the security module was configured to deny socket receive message operation for all connections belonging to the user 'student', the *recvmsg* function returned with an error code.

Similar tests were done to check the functionality of all the other rules. The tests showed that the security module is behaving as expected. However, performance tests are still important and remain an issue for future work.

VI. Conclusion

The approach presented in this paper offers a solution for implementing a network security module that enforces the MAC schema. In order to achieve this, we used the LSM and Netfilter frameworks, which provide a set of hooks for restricting user access to specific socket operations and filtering network packets.

The main advantage of the proposed solution is that it allows complex security policies to be implemented by offering fine-grained control over socket operations. This presents an advantage over the standard Linux network security module that provides only network packet filtering capabilities.

Future work can be aligned to two directions: conducting thorough performance testing as well as optimizing the application and adding new features such as extending the rule set to support filesystem access. In addition, porting the security module to a newer kernel version should also be taken into account.

REFERENCES

[1] D. D. Downs, J. R. Rub, K. C. Kung, and C. S. Jordan, "Issues in discretionary access control," *Security and Privacy, IEEE Symposium on*, vol. 0, p. 208, 1985.

[2] Y. Jiang, C. Lin, H. Yin, and Z. Tan, "Security analysis of mandatory access control model," in *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, vol. 6, pp. 5013–5018, IEEE, 2004.

[3] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium*, vol. 2, p. 44, San Francisco, CA, 2002.

[4] S. Smalley, C. Vance, and W. Salamon, "Implementing selinux as a linux security module," *NAI Labs Report*, vol. 1, p. 43, 2001.

[5] M. Bauer, "Paranoid penguin: an introduction to novell apparmor," *Linux Journal*, vol. 2006, no. 148, p. 13, 2006.

[6] C. Schaufler, "Smack in embedded computing," in *Proceedings of the 10th Linux Symposium*, 2008.

[7] G. Purdy, *Linux iptables Pocket Reference*. Pocket References Series, O'Reilly Media, 2004.

[8] J. Engelhardt and N. Bouliane, "Writing netfilter modules," *Revised, February*, vol. 7, 2011.