Linux Security Modules Enhancements: Module Stacking Framework and TCP State Transition Hooks for State-Driven NIDS

Markus Quaritsch
Graz, University of Technology, Austria
quam@qwws.net

Thomas Winkler
Graz, University of Technology, Austria
tom@qwws.net

Abstract

Until the availability of Kernel 2.6 the Linux operating system lacked general support to integrate security mechanisms into the kernel. The Linux Security Module Framework (LSM) was designed to overcome this limitation. Although LSM provides a solid baseline for kernel security, it lacks important features. In this paper two of these limitations are addressed: First a framework-managed module stacking mechanism is proposed that allows multiple security policies to be present in the kernel at the same time. The second aspect this paper deals with is the addition of LSM hooks to the Linux TCP layer. This extension was chosen because it allows the implementation of a State-Based Network Intrusion Detection Mechanism which is outlined at the end of the article.

Keywords: Linux Security Modules, Managed Module Stacking, TCP LSM Hooks, State-Driven Intrusion Detection

1 Introduction

Along with many other new features Linux Kernel 2.6 introduced the *Linux Security Modules* (LSM) [1, 2, 10] mechanism that provides general security support for the kernel. Although LSM provides a universal and flexible security framework it nevertheless lacks some features. Among those is a framework-managed and enforced module stacking mechanism that allows multiple security modules to be loaded into the kernel without relying on module cooperation. The second aspect that is addressed in this paper is the addition of security hooks to the Linux TCP layer. These hooks allow tracking of state transitions to implement a state-driven network intrusion detection system as proposed in [3].

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the concepts of the existing *Linux Security Modules* framework along with an outline of its implementation. Section 3 describes the principles and architecture of the proposed module stacking extension and gives a summary of the implementation. Another enhancement of LSM is described in Section 4: The addition of security hooks to the TCP layer to track TCP state transitions. The subsequent section 5 at first describes the basic concepts of State-Based Network Intrusion Detection Systems followed by a description of a possible implementation of such a system based on the LSM TCP hooks from section 4. Section 6 presents the current state along with possible further extensions while section 7 gives a conclusion of our work.

2 LSM: Goals, Concepts and Implementation

LSM was designed as a general framework which allows the integration of different security concepts into the Linux Kernel. LSM by itself only provides a number of security hooks located in different kernel subsystems without implementing a security concept itself. This is entirely left to the security modules. Among the best known of those modules is *Security Enhanced Linux* (SELinux) [4,5,6,7,8,10] developed by NSA[1] .

LSM Architecture: The main goal of LSM is managing the access to kernel objects. To achieve this, calls to security hook functions were placed at various points within the kernel to mediate access to kernel objects. Function pointers for the hook functions are stored in a global structure called `security_ops`. Those function pointers are called by the kernel. To provide great flexibility it is possible to load so-called *security modules* into the ker-

[1] NSA - National Security Agency, Web: http://www.nsa.gov

nel. Those modules can provide and register functions for the function pointers in the `security_ops` table. It is not required that a module implements all available hook functions, but it can limit itself to any subset that is sufficient for its goals.

There are about 130 such security hooks available in the kernel. The most important categories of security hooks are:

Task Hooks: provide control over process related functionality such as *kill*, *setuid* or *nice*

Program Loading Hooks: allow access control prior to loading programs and add hook calls at critical points of the *execve* function

IPC Hooks: allow access control for interprocess communication

Filesystem Hooks: allow fine grained access control for filesystem operations such as *read* or *write*

Network Hooks: provide several hooks at different points of the network stack

In addition to the security hooks the LSM framework provides so-called *security fields*. Those are simple `void` pointers which are attached to important kernel data structures. This mechanism permits security modules to add information to kernel objects and thereby label them or attach security relevant information to a specific object. There are special *alloc* and *free* hooks that are called whenever a kernel object is created or destroyed to allow the security module to initialize and clean up the security fields.

In theory it is possible to load more than one security module into the kernel. This mechanism is called *module stacking*. This functionality however relies completely on the cooperation between modules. The LSM framework by no means enforces a proper stacking. The first security module that is loaded registers itself with the kernel. The following modules then register themselves with the first (primary) security module. This means that the kernel itself is only aware of the primary security module. It is up to the primary module to pass on hook calls to subsequent modules. A module does not have to implement this cooperative behavior or can ignore security decisions of subsequent modules which means that the primary module has ultimate control.

3 Framework Managed Module Stacking

As described in section 2 LSM currently lacks a module stacking mechanism that provides and enforces correct module stacking. We propose an extension to the current LSM implementation to overcome these limitations.

A mechanism is required that allows to register multiple security functions from different security modules for a single security hook. It should be the task of the LSM framework to call all the registered functions for a certain LSM hook. The next step is the computation of the final result of all the access decisions (grant access or not) which then is returned to the calling kernel function. The security modules themselves should not need to be aware of the fact that there are multiple modules loaded into the kernel, otherwise existing modules would require modifications to fit the new stacking mechanism which is not desired.

The current LSM framework maintains a global data structure called `security_ops` where function pointers to the security functions of the loaded primary security module are held. A security module registers its functions by using the `register_security` function of the kernel. The kernel framework is only aware of the first loaded module, which is also called the primary module. Subsequent modules that are loaded into the kernel have to use the `mod_reg_security` function to register themselves. This function calls `security_ops->register_security` which essentially is a LSM hook function. The primary LSM module might or might not have registered a function for this hook. And even if it has, this still will not guarantee that subsequent security modules are handled correctly by the primary module. There is no enforcement that hook functions of subsequent modules get called or their access decisions are taken into account.

The proposed approach to implement correct module stacking is to introduce a special LSM multiplexing module which is loaded as primary security module. Subsequent modules register with the multiplexor module which guarantees that all security function calls from the kernel are passed to all loaded security modules. This concept is shown in Figure 1.

3.1 Implementation

Considerations for the implementation of the multiplexor module can be separated into two parts: Firstly every module has to be consulted for every security hook call that occurs. Secondly every func-

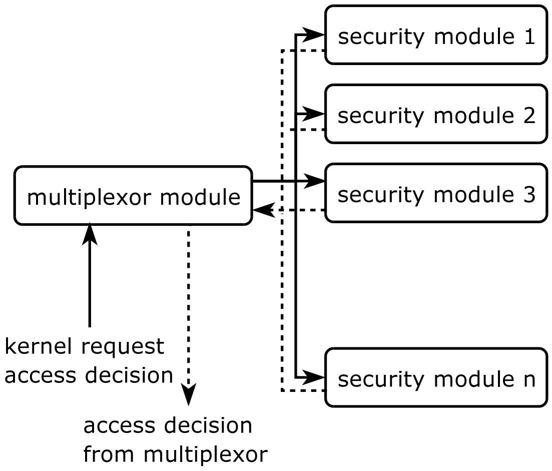

Figure 1: The multiplexor passes access decision requests to all loaded modules, computes the result and returns it to the calling kernel function.

tion has to be able to store data in the `void` pointer security field of the related kernel object. The second constraint requires the multiplexor module to restore the correct state of the security field of a security module ahead of a security function call. After the call the multiplexor has to save the content of the security field. Since the current LSM implementation only provides simple `void` pointers for the security field, a more flexible data structure is required.

3.1.1 Multiple Modules and Hook Calls

The multiplexor module has to be loaded as primary security module. It then registers special functions for the `register_security` and the `unregister_security` hooks in the global `security_ops` table of the kernel. For any subsequent security module that tries to register itself with the kernel, the *register_security* function will fail since there has already been a primary module loaded. As the next step the module tries to register with the kernel using the `mod_reg_security`. This function actually calls the `security_ops->register_security` function pointer. This enables the multiplexor module to build up a list of all loaded security modules along with the security functions they provide.

The LSM multiplexor provides security functions for all the security hooks currently available in the kernel which get registered in the global `security_ops` table. Whenever the kernel asks the LSM framework for an access decision, the matching multiplexor security function gets called. This function then calls its counterparts from all registered modules. The access decisions of those calls are aggregated into a single value which is returned to the calling kernel function. Since the return values are sensitive in terms of granting access to certain kernel regions or objects the mutliplexor only will grant access if all the security modules grant access. Otherwise access will be denied by the multiplexor.

3.1.2 Saving and Restoring Fields

The LSM framework introduced security fields which are attached to certain kernel data structures. Those fields are used to store security-related information in kernel objects. With multiple modules in the kernel a mechanism is required that manages access to the security fields of kernel objects.

To allow more than one security module the single `void` pointer that is provided by the current LSM in certain kernel structures is replaced by an array of `void` pointers. This replacement is handled transparently by the multiplexor. Before a certain security function of a module is called the multiplexor saves the array of `void` pointers and sets the kernel object security field to the correct entry for the module. After the call returns, the multiplexor writes the values of the security field back into the array.

For both, the handling of the loaded security modules and the handling of the multiple security fields of kernel-objects, arrays were chosen as data structures. To quickly access array elements every security module is assigned a unique identification number. Those numbers are maintained in a bitmap.

3.2 Discussion

The LSM multiplexor module described in the preceding sections offers a simple and elegant way to extend the LSM framework as currently available in the Linux 2.6 kernel series by a mechanism that guarantees and enforces correct module stacking without relying on module cooperation. In addition, this approach avoids grave changes to the existing LSM implementation and therefore doesn't require modifications of existing security modules.

By implementing the multiplexor as a module great flexibility is achieved. Users are able to load the multiplexor when required but can fall back to the default LSM behavior whenever they are employing only a single security module. This also eliminates any performance impact of the multiplexor due to additional indirections for "single module setups".

4 LSM Hooks for the TCP Stack

Although the LSM consists of about 130 hooks placed in all parts of the kernel, there are no hooks in the TCP layer. To implement an intrusion detection system based on changes of TCP states it is required to insert hooks in this part of the kernel as well.

The integration of the TCP hook function into the LSM framework as well as the additional hook calls to this function from within the TCP layer should be done with minimum impact to the existing code.

4.1 Extending the LSM Framework

As described in section 2 the global structure `security_ops` (defined in *include/linux/security.h*) stores pointers to all security hooks available. When introducing a new security hook, the `security_ops` structure has to be extended to hold the additional function pointer.

These function pointers in the `security_ops` structure are not accessed directly but by means of inline functions which are defined in *include/linux/security.h* as well. Such an inline function has to be provided for the newly introduced TCP hook function.

In a first version only one hook function has been introduced to keep track of the changes in the TCP layer. The parameters for this security function are a pointer to the socket which changed its state, the old state and the new state. This information is sufficient to keep track of the state transitions in the TCP-Layer. This single hook function is called from various places within the TCP layer.

4.2 Placing the Hooks

Besides extending the LSM-Framework it is required to insert the calls to the new hook functions into the TCP layer. The goal was to keep these changes as little intervening as possible. Although the Linux Kernel supports not only IPv4 but also IPv6 the following approach focuses on IPv4 to describe the basic principles.

When taking a closer look on the TCP layer, there are two main entrance functions. Incoming packets from the IP layer enter the TCP layer through the function `tcp_v4_rcv`. Packets for already established connections get processed by the function `tcp_rcv_established` while all other packets are passed on to `tcp_rcv_state_process`. This function takes into account the current TCP state of the connection and carries out the required actions. If this leads to changes in the current TCP-State the function `tcp_set_state` (defined in *include/net/tcp.h*) is called. Therefore this function turns out to be an ideal location to call the TCP security hook.

Placing the call of the TCP security hook in the `tcp_set_state` function covers almost any state transitions in the TCP layer (see [9]). One exception is the transition from the state *LISTEN* to the state *SYN RECEIVED*. This transition is special because upon receiving a *SYN*-Packet on a socket which is in state *LISTEN* this socket remains in state *LISTEN* and a new socket which is in the state *SYN RECEIVED* is created. To track this state transition too it is necessary to insert one more call of the hook in the function `tcp_v4_syn_recv_sock` (defined in *net/ipv4/tcp_ipv4.c*).

Another exception is the transition from state *CLOSE* to state *LISTEN*. This transition is invoked from the userspace by the systemcall *sys_listen*. In the case of a TCP socket the systemcall passes the request to the function `tcp_listen_start` which in turn changes the state to *LISTEN*. By inserting the TCP security hook in the `tcp_listen_start` function (defined in *net/ipv4/tcp.c*) it is possible to track this state transition.

By calling the TCP security hook within the functions described above it is possible to track all state transitions within the TCP-Layer. However, it might be necessary to insert additional hook calls to cover all invalid state transitions (e.g. the socket remains in the current state when receiving invalid TCP packets).

5 State-Driven NIDS based on LSM TCP Hooks

The additional LSM TCP hooks described in section 4 provide the necessary facilities to implement a Network Intrusion Detection System (NIDS) based on TCP state transitions. A similar approach was outlined in [3]. In contrast, the current proposal focuses on a concrete implementation for the Linux Kernel on top of the LSM framework.

Before the actual implementation is described some more general considerations about NIDS systems based on state transitions will be presented.

5.1 State-Transition based NIDS: The Concepts

The TCP protocol is implemented as a state machine with well defined state transitions [9]. Many attacks try to exploit this behavior by performing invalid or incomplete sequences and therefore leaving the TCP stack in an undefined state. For common intrusion detection systems located in the userspace it is not possible to detect such attacks since there is no information available about those events outside the kernel.

For example, it is easy to detect a portscan with a state-transition based intrusion detection system. Portscans are characterized by a lot of incoming connections on different ports which are closed again instantly. They are used by intruders to get information about open ports and the corresponding services of their point of attack. When trying to detect a portscan with an intrusion detection system based on packet inspection it is necessary to trace the connection establishment and termination on IP-layer level. This means that the intrusion detection system has to keep track of at least six IP-packets corresponding to the connection establishment and termination.

Using a state-transition based intrusion detection system it is only required to keep track of the transitions of the opened ports. The intrusion detection system counts the number of sockets changing their state from *LISTEN* through *ESTABLISHED* to *CLOSED*. If the gradient of this value exceeds a given threshold the reason can be a portscan. Variants of the attacks are so-called stealth-scanner where the actual establishment of the connection is not concluded. Of course it is also possible to take only a few ports into account or offer some additional *honeyports* with no real services behind.

By keeping the approach as lightweight as possible it should be feasible to deploy the NIDS infrastructure on a whole network without the risk of a significant performance impact for single machines. This allows the implementation of a distributed network of sensors. This is especially interesting for switched networks where there is no single point that sees all the traffic. Even if this would be possible there is still the problem that this single monitoring point requires lots of computing power to analyze and process all the network traffic.

What we propose is the deployment of the lightweight state based NIDS module onto all machines of the network. While those NIDS modules are running in kernel mode to be able to monitor all the traffic rules and thresholds can be easily set from userspace by employing the Netlink [11] functionality of the Linux Kernel. Since it is not desirable that every single sensor has to be manually configured we propose a central point of control. This master station provides rule sets along with other configuration parameters such as thresholds for all the sensor stations. The sensor stations obtain this information via network communication from the central master station and configure themselves according to those rules. An overview of the setup is depicted in Figure 2

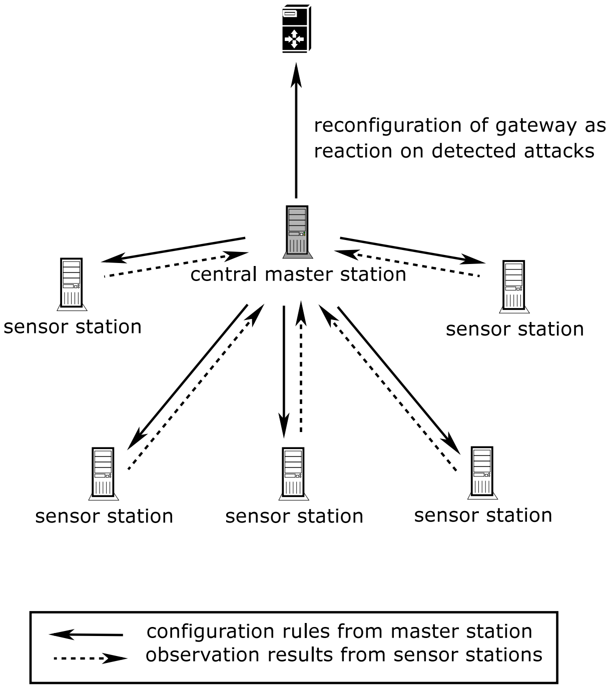
reconfiguration of gateway as
reaction on detected attacks
central master station
sensor station
sensor station
sensor station
sensor station
sensor station
configuration rules from master station
observation results from sensor stations

Figure 2: The NIDS modules is deployed on all machines of the network. A central master station provides rules and acts upon observation results from the sensor stations.

Aside from providing the rules the master station also acts as a central decision maker. All sensor stations report detected attacks to the master station. Then, it can decide how to react on the attack. This could reach from a reconfiguration of the other sensor stations by providing updated rule sets to changing the configuration of firewall systems running on certain gateway machines to block incoming traffic from suspicious hosts.

5.2 Implementation: LSM Module

After describing the advantages of intrusion detection systems based on the state transitions in the TCP layer we introduce two kernel modules. These modules register themselves with the LSM framework and provide functions for the TCP security hooks described in section 4.

The first module introduced implements logging facilities for state transitions while the latter implements a portscan detection. Using the LSM multiplexor module described in section 3 both modules can be loaded simultaneously.

Both modules consist of two main parts. While the first part implements the logging facility respectively portscan detection mechanism, the second part handles the communication with the userspace. The principal task of each module is described in the following two sections, while the communication mechanism is described in section 5.3.

5.2.1 Logging TCP state transitions

The fist module is rather simple. Its only intention is to log each TCP state transition of a socket. As a side effect this module can be used to check if all state transitions are reported correctly by the new security hooks. Therefore the module has to register itself as security module and provide an appropriate implementation of the security function for the TCP hook. All other security hooks can be ignored.

As counter for the transitions from state A to state B a two dimensional array where the first index is the old state and the second index is the new state is employed. Because the number of states in the TCP layer is fixed the array can be statically allocated at compile time.

Furthermore, a userspace tool was implemented to interact with the kernel module. With this tool it is possible to read the current values stored in the array or reset all values to zero from userspace.

5.2.2 Detecting portscans

The original idea of introducing new security hooks was to provide a lightweight mechanism for detecting network intrusions. This module follows this intention and presents a mechanism to detect portscans only by observing state transitions in the TCP Layer.

As described earlier, portscans are characterized by lots of incoming connections on different ports which are closed instantaneously. This connection establishment and termination leads to a quick transition from *LISTEN* through *ESTABLISHED* to *CLOSED*.

By measuring the time between the transitions from *LISTEN* to *SYN RECEIVED* and *TIME WAIT* or *LAST ACK* to *CLOSED* it is possible to detect all connections which are opened and closed "quickly". Therefore, the security field of the socket structure is used to store the timestamp when a transition from *CLOSED* to *SYN RECEIVED* occurs. When the socket is closed the stored timestamp is compared to the current time and if the difference is below a given threshold this connection could be considered as part of a portscan. If the number of short connections reaches a second threshold value, an intruder may be looking for open ports.

Another userspace utility was introduced for communication with this LSM module as well. On the one hand this tool can be used to configure the thresholds and on the other hand to get informed about possible portscans. In reaction to that special events (e.g. inform some system/network monitoring programs) could be triggered.

5.3 Implementation: Communication with Userspace

While the core parts of the NIDS are implemented in the kernel it is required that the configuration of the modules can be modified and the current data collected by the module can be read from userspace. There are many ways to achieve a communication between kernel modules and userspace tools.

The first approach might be using systemcalls to pass data between the userspace and kernelspace. But implementing this communication mechanism for a kernel module would require a lot of changes to the whole kernel. Systemcalls are very dependent upon the architecture and therefore not suitable for our communication needs.

A more flexible way to exchange data between kernelspace and userspace are the so called netlink sockets [11]. This framework is already part of the kernel and can be used by kernel modules without any needs of changes to the kernel itself. It also defines the basic packet-structure and contains all mechanisms to exchange data between the kernelspace and userspace. Therefore we decided to use this communication mechanism.

The kernel module has to register itself with the netlink framework to be able to receive data from the userspace. The connection endpoint is the kernel-side counterpart of a user-socket. Sending data to the userspace is like writing to the socket. Receiving data is a bit more complicated. The kernel module receives a `sk_buff` structure [10] and has to divide it into the individual netlink packets.

From the point of view of the userspace the communication is a socket communication with datagrams (connectionless). After opening a new socket with a specific protocol type the application can write to and read from this socket as usual.

6 Current State and Further Work

Currently the LSM Enhancments described in previous sections are freely available as "proof of concept" implementations from http://uni.qwws.net/linpro/project. While at the moment only portscans are detected the flexibility of the framework allows an easy extension to detect a whole class of attacks.

Building upon the proposed concepts there are several topics that could be investigated further:

- An Implementation of the multiplexor as a kernel patch. This could probably allow a more efficient implementation and reduce performance overhead but might break existing security modules along the way.
- For the detection of illegal state transitions additional LSM hooks in the TCP layer might be required.
- A deployment of attack-detection kernel modules on a whole network to act as a distributed intrusion detection system could be of interest.
- For performance reasons it might be useful to integrate the LSM multiplexor as a patch into the kernel.

7 Conclusion

The LSM multiplexor presented in the first portion of the paper provides a mechanism that allows multiple LSM modules to be present in the kernel at the same time without relying on module-cooperation. This property is especially valuable for NIDS modules presented in the second part of the paper.

The given example of portscan detection based on the examination of TCP state transitions actually stands for a whole class of network based attacks that can be identified by state-driven NIDS systems. Among those attacks are for example SYN-flooding, FIN-stealth scans or the ”X-Mas tree” attack. By using the LSM multiplexor and the new TCP LSM hooks it is possible to write detection modules for each of those attacks.

8 Acknowledgements

This paper is based upon results from our Seminar/Project at the Institute for Information Systems and Computer Media (IICM), Graz University of Technology. We would like to thank Harald Krottmaier for his support and encouragement.

References

[1] Wright, Cowan, Smalley, Morris, Kroah-Hartman: ”Linux Secuirty Modules: General Security Support for the Linux Kernel”, USENIX Association, 11th USENIX Security Symposium, San Francisco (2002)

[2] Wright, Cowan, Smalley, Morris, Kroah-Hartman: ”Linux Security Module Framework”, 11th Ottawa Linux Symposium, Ottawa, Canada (2002)

[3] Udo Payer: ”State-Driven Stack-Based Network Intrusion Detection System”, 7th International Conference on Telecommunications - ConTEL 2003, Zagreb, Croatia (2003)

[4] Loscocco, Smalley: ”Integrating Flexible Support for Security Policies into the Linux Operating System”, National Security Agency, NAI Labs (2001)

[5] Stephen Smalley, Chris Vance, Wayne Salamon: ”Implementing SELinux as a Linux Security Module”, NCSC, NAI Labs (2002)

[6] Oliver Tennert: ”Hinter Schloss und Riegel” [German], iX 12/2003, Heise Zeitschriften Verlag GmbH, Hannover, Deutschland (2003)

[7] Carsten Grohmann, Konstantin Agouros: ”Regel-recht” [German], Linux Magazin 01/2003, Linux Media AG, München, Deutschland (2003)

[8] Carsten Grohmann: ”Regel-Praxis” [German], Linux Magazin 02/2003, Linux Media AG, München, Deutschland (2003)

[9] Jon Postel (Editor): ”Transmission Control Protocol”, RFC 793, Defense Advanced Research Projects Agency (DARPA), Virginia, USA (1981)

[10] Markus Quaritsch, Thomas Winkler: ”Linux - Ein Einblick in den Kernel” [German], Seminar/Projekt IICM (2004), http://uni.qwws.net/linpro/seminar

[11] Gowri Dhandapani, Anupama Sundaresan: ”Netlink Sockets - Overview”, Department of Electrical Engineering & Computer Science, The University of Kansas (1999)