SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Reflections on The Virtues of Modularity: A Case Study in Linux Security Modules

Andrew Blaich, Douglas Thain, and Aaron Striegel*†

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556

SUMMARY

Developing a modular system that properly supports a range of security models is challenging. The work presented here details our experiences with the modular *Linux* security framework called Linux Security Modules, or LSMs. Throughout our experiences we discovered that the developers of the LSM framework made certain tradeoffs for speed and simplicity during implementation, consequently leaving the framework incomplete. Our experiences show at which points the theory of the LSM differs from reality, and details how these differences play out when developing and using a custom LSM.

KEY WORDS: Security Modules; LSM; Modularity; Linux

1. INTRODUCTION

Modularity is a "virtue" in software engineering. Using a system that was designed with modularity in mind benefits [6] both the users and the administrators. A modular system is capable of being modified dynamically via swappable modules without changing the underlying code or structure of the system. However, modularity is extremely difficult to get absolutely right. There are numerous traps and pitfalls that a modular system can be susceptible to, depending on its implementation. Yet, the end benefit of modularity, "plug and play", is still a desired property that keeps such systems around.

Linux Security Modules, LSMs, are an example of software modularity. LSMs seek to supply a modular design for security in the *Linux* kernel. Linux Security Modules can be loaded at

*Correspondence to: Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556
†E-mail: ablaich@cse.nd.edu & dthain@cse.nd.edu & striegel@cse.nd.edu
Contract/grant sponsor: NSF; contract/grant number: CNS-05-49087 and CNS-03-47392

Received 29 September 2008
Revised 29 September 2008

runtime, dynamically, or can be statically compiled into the kernel. With the recent releases of the *Linux* kernel, 2.6.24 and later, the LSM interface has been made static, thus eliminating the dynamic modular nature of LSMs for the foreseeable future. However, the discussion points presented in this paper still apply to the LSM framework whether allowing modules to be compiled into the kernel, statically, or loaded during run-time, dynamically.

LSMs provide a modular interface with which different security models can exist in the kernel or be swapped out for other models. Loaded modules utilize the LSM framework via "hooks" placed in the kernel's system calls. These hooks allow a security module to validate an action and further disallow it if it violates system policy.

Recently, we utilized the LSM framework to develop a security module for our project called Lockdown [1,10], which is presented as a case study in this paper. Lockdown provides full local context[†], see Figure 1 enforcement and monitoring of the network activities for an end-user on an enterprise network. The LSM framework was chosen for Lockdown because of the unique view a kernel loadable module has of the network activity produced by an application that other solutions are not as capable of achieving.

However, the LSM framework is not without problems, which were encountered during the development of Lockdown. The first issue deals with Linux Security Module stacking (layering); how it is described versus how it is actually accomplished. We ran into several problems when using multiple LSMs on the same system. While being a supported feature of the framework we discuss why theory differs from reality here. The second issue involves the location of the LSM hooks within the *Linux* kernel source code and where certain hooks should be relocated. The location of a security hook is very important as it needs to be placed before any important decisions on the system have been made. However, the hook also needs to be located where the entire context needed to make a decision on that hook is available. The third and final issue involves appropriate error messages when using LSMs and additionally how to recover/respond to denied system calls based on the error codes that are returned.

Based on our experience with the aforementioned problems, we propose several solutions and discuss what lessons can be learned from them in regards to designing and using a modular security framework. The general lessons that can be learned are summarized in three points. First, the design of the security hooks in such a modular system should be consistent throughout the entire design. A few of the hooks in the framework do not catch or act upon the error codes that are returned which is in contrast to the semantics of the majority of the hooks. Secondly, in theory, providing the ability to do a task, such as module stacking and enforcing how it should be done are very different things. While module stacking is available in the LSM framework, it is up to the designer of an LSM module to implement support to allow for additional modules to stack against it. Critically, this needs to be moved into the kernel so that an LSM is not reliant on any other LSM. Finally, being able to convey to the end-user, administrator, and developers easy to understand and intuitive error messaging and additionally handling these "errors" gracefully should be a desirable goal for all software engineers. Throughout the work it was difficult to determine at times if the reason for a

[†]Local context is defined as the who and what (user/application) of a connection, whereas traditional firewalls are only concerned with the where (IP Address/Port numbers)

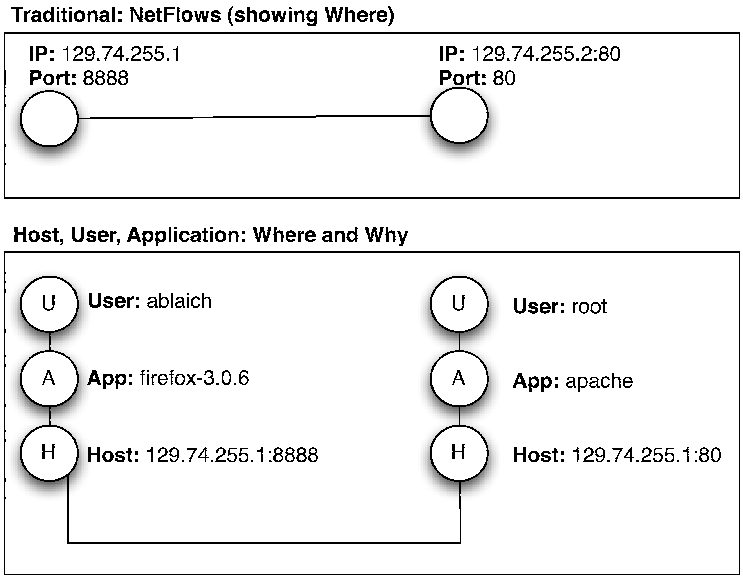

Figure 1. What is local-context?

problem was due to a policy violation in an LSM or something much worse. Through the use of coherent and useful error messaging, the cause of a denied action can be narrowed down quickly when typical error messaging is used.

2. Case Study: Lockdown

The administrator of an enterprise network has a responsibility to enforce corporate policies on the network. Yet, traditional security mechanisms have problems enforcing the intended policies. The problem with enforcing policy has been due to the prevalence of tools that have trouble mapping corporate policy to the available enforcement mechanisms. However, these tools have been easier to manage than more complicated mechanisms, which explains their continued usage. While advanced solutions do exist, e.g. *SELinux*, that have strong enforcement, they are significantly harder to manage. To this end, we developed Lockdown, a policy-oriented security approach that builds on the concept of local context to deliver a lighter weight approach to enterprise network security while striking a balance between the level of enforcement and level of management available to the network administrator.

Typical tools, such as firewalls that rely on rule sets constructed from IP addresses and port numbers, typically infer application usage via the port number. The tradition of using such tools has lead administrators to develop a *fuzzy* picture of what is occurring on their network,

since application usage is inferred. This inference leads to poorly enforced policy controls, since many different applications can use the same port number. Thus, extra information is needed in order to tighten up the enforcement and gain a clearer picture of the network.

Lockdown is made up of several components that include one for: monitoring, enforcing, auditing, analysis and visualization. The monitoring component runs on each workstation and observes network activity such as the application and its path, pid, gid, uid, source and destination IP address/port, etc. All of the observed data by the monitoring component is logged and sent to a central database for analysis. The enforcing component, built as an LSM, enforces policy that has been infused with local-context on the host. Using the hooks provided by the LSM interface, portions of the policy are validated against the context of an application trying to achieve network connectivity through kernel level socket system calls. The auditing, analysis and visualization components are involved in validating the effectiveness of current corporate policy on the workstations under the control of Lockdown. Additionally, these components provide in-depth details to the administrators on what is actually occurring on their network.

With Lockdown policy can be tied to specific usage scenarios and tightly enforced. Figure 1 demonstrates the power of utilizing local context when creating and enforcing policy/rules over a system that does not, such as traditional firewalls like *iptables*.

An example policy is the following:

`Allow web-browsing to be done to any address with only application` *`Firefox version 3.0`* `that belong to the employees in group` *`Full-Time`*.

The previous policy is then transformed into a rule set so that it is readable by the computer and local context enforceable on a host. The translated policy would look like (* indicates for all):

```
ALLOW out to ADDRESS * with APP "/usr/bin/firefox-3.0" by GROUP "fulltime";
DENY out to ADDRESS * with APP * by GROUP *.
```

Essentially, the first rule allows the previously stated policy to occur while the second rule prevents any other network activity by any other permutations of the local context from occurring. Like traditional firewalls such as *iptables* if a rule is not matched then the last entry in the chain of rules is what dictates the action taken on a connection.

As will be discussed next, the vantage point that an LSM has as opposed to a tool like *iptables* enables it to provide a more robust suite of enforcement tools and options. The type of policy as shown previously is best enforced by an LSM. The three other options that we considered before developing our LSM included: *iptables*, system call interposition/trapping, and *SELinux*.

The first option, *iptables*, is a user-space application that uses the netfilter [13] *Linux* kernel hooks. Netfilter is a set of callback functions located within the network stack kernel code. Whenever a packet enters or leaves the network stack, the netfilter hooks are called. While *iptables* uses kernel hooks in order to manage network traffic, the detail of the rules used are only at the level of IP address and port number. Using only the IP address and port number

for a rule when analyzing a packet is extremely fast since the hooks need to parse every packet that traverses the network stack and bit-wise operations can be performed against the packet and the rule, which is must faster than doing a string compare against an application name or path for every packet. However, the coarseness of the rule set and packet by packet level of parsing that *iptables* does lead us to explore other alternatives. For example, observe the following rule constructed for *iptables* for the corporate policy of allowing web-browsing:

```
iptables -A INPUT -p tcp -m tcp --sport 80 -j ACCEPT
iptables -A OUTPUT -p tcp -m tcp --dport 80 -j ACCEPT
```

Application usage is inferred in the above *iptables* rule set that only web-browsing will be done over port number 80. Whereas the equivalent rule in Lockdown is finer grained, since the application is now taken into account:

```
ALLOW out to ADDRESS * with APP "/usr/bin/firefox-3.0" via PORT 80
```

The second option, system call interposition, involves dynamically modifying the system call table through the use of a kernel module. Whenever a system call is made the call is redirected to the module that is claiming responsibility for it and in turn executes the corresponding code. This method was essentially overkill for our needs. Rewriting entire portions of system call code in order to trap the network socket operations occurring in the system calls is tricky since the code base for *Linux* is in constant flux and changes made for either security or performance in a newer kernel version may not be reflected in our module as quickly. The traps and pitfalls of system call interposition are detailed in [4].

The third option, *SELinux*, is the NSA's (National Security Agency) answer to a secure version of *Linux* [7, 18]. *SELinux* is robust and feature rich, but the policy files/modules are tricky to write [12] and manage. The commonly found LSMs, which include: *AppArmor* [14], *Smack* [17], and *Tomoyo* [3]; help to show that *SELinux* may be very robust, but portions of the feature set are improved on by other groups to allow for wider adoption of robust security solutions and methods in the modern operating system. As shown in [9] when comparing and contrasting *AppArmor* versus *SELinux*, SELinux is typically not favored due to its poor usability making it difficult to manage. The complexity of *SELinux* is further seen in a comparison to *Multics* [2]. *Multics* was designed originally with security in mind , while current operating systems have security grafted on. In [8], the authors discuss how the original *Multics* kernel was only 628KB in size, whereas in comparison the sample *SELinux* policy (not including the linux kernel) is approximately 1767KB, 2.5 times bigger. The difficult nature required in adding security to modern operating systems outside of the original design results in the increased size and complexity of such systems as they exist today.

Additional downsides with using SELinux include the use of labels instead of pathnames for applications. While the SELinux model of using labels that are "stored as extended attributes for filesystem objects" is considered more secure since references are made to the actual filesystem objects. However, labeling/relabeling of a filesystem takes time and is not intuitive when auditing the log files on a system. To this end we chose to develop our own LSM since the main goal of Lockdown is to provide a manageable, usable, and intuitive security solution.

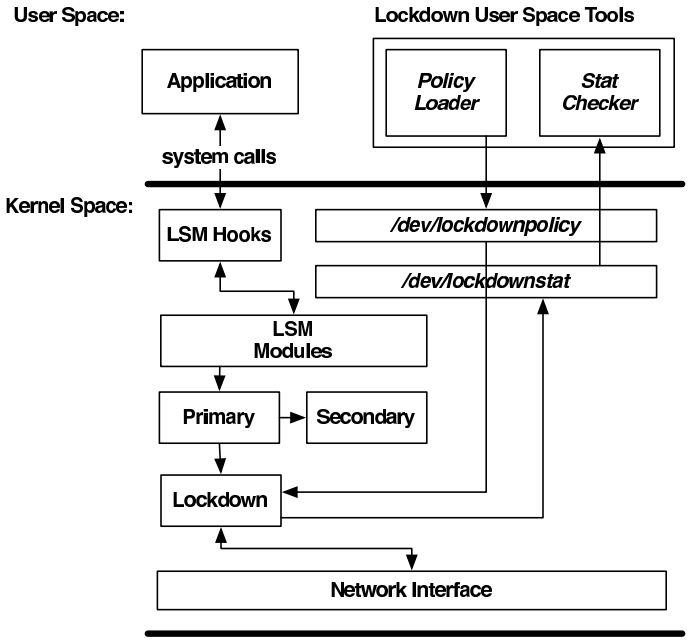

Figure 2. The Lockdown LSM Enforcement module

Additionally, since the majority of the this work relies on the intuitive auditing and logging of data a label based approach to application security was not chosen.[‡]

3. Linux Security Modules

Before we get into the underpinnings of the LSM, it would help for a brief history lesson on where Linux Security Modules came from. The framework for LSMs was initially proposed and developed by Cowan and Wright [19, 20, 21]. The goal of the LSM framework was to provide sufficient hooks into the *Linux* kernel to allow a module to provide for a method of access control; while also keeping changes to the kernel minimal. The need for this method of access control is fueled by comments made by Linus Torvalds [11], creator of Linux, that an ultimate security standard does not exist. Essentially, since measuring security is still very subjective

[‡]The debate of labels versus pathnames for application security continues to be waged on the Linux kernel mailing list as well as security forums.

and not quantitative, it is hard to judge how much "more" secure one security model is from another. So, rather than having one security model rule the kernel, it would be better if the kernel did not have to rely on any single hardcoded standard.

Now that we know where LSMs came from, how exactly do they work? The first thing to discuss is how are LSMs loaded and what happens during the loading phase. LSMs are inserted into and removed from the kernel via the standard *insmod*, *rmmod*, and *modprobe* utilities. When an LSM is first inserted, it needs to check whether it can be loaded or not. The LSM interface provides functions which allow a module to register into one of two available slots. The main interface slot that a module can register with is called the *primary module* and the alternative slot is called the*secondary module*. A module that is attempting to load into the kernel as an LSM needs to first check the primary slot for any other previously loaded LSMs; if none are found then the module can load. However, if the primary slot is occupied then the secondary slot needs to be checked before being loaded. If the secondary slot is free, then the module will load into there. However, the secondary slot represents a module registering with the primary module, because the kernel can only in reality handle one security module, this is explained later in the module stacking section. It is the job of the primary module to handle any secondary module.

An overview of the LSM architecture between user space and kernel space as it pertains to Lockdown can be found in Figure 2. If a LSM needs to communicate with user-space (since modules are loaded into kernel space), such as having to read in policy files or output statistics there are a few ways to accomplish this. The method which we used was to create several special files (/dev/filename) that a user space application is able to write data to and read data from; depending on the permissions. These "devices" serve as transports between user space and kernel space. The initialization of these "devices" is done during the setup phase of the module.[§]

Once a LSM is inserted properly, the descriptor table is loaded that tells the kernel which LSM hooks the current module is providing functions for. The LSM hooks exist as upcalls to a loaded module in the kernel that has designated itself responsible for handling all or a sub-set of the available hooks. The hooks are scattered throughout the common system calls that exist in the kernel which are responsible for a range of tasks including: program execution, filesystem, inodes, tasks, files, netlink, unix domain networking, sockets, key management, and System V operations. Any pertinent structures or variables that are necessary for making a decision for a particular LSM hook are passed into the hook, like a function call, and the module that implements the hook has access to them when it gets called. For example, the *socket_create* hook as found in the *sock_create* system call is shown below (LSM hooks are preceded by `security_` in the kernel source, but not in the actual LSM code):

```
...
err = security_socket_create(family, type, protocol, kern);
if(err)
```

[§]The setup and creation of devices can be found in Linux Kernel Module documentation pertaining to device drivers [16].

```
return err;
...
```

Once a hook is called in the kernel source, the module designated to handle that hook is called and the corresponding code is executed in the appropriate LSM. If the LSM wishes to prevent the system from proceeding, it returns a non-zero error code. LSM hooks can typically only further deny actions that have not yet been denied by the kernel itself. However, there is some support for authoritative hooks, where a previously denied action can be allowed, but these are "limited to the extent that the kernel consults the POSIX.1e capable() function" as stated in [20].

For a better idea of how the LSM hooks work, consider the following example. Assume we have loaded an LSM onto our machine that implements the LSM hooks for creating a network socket. Whenever an application attempts to create a socket, our LSM code gets called and we have a chance of saying whether the socket should be allowed or denied. Assuming we have a policy in place that says to allow only the application *mozilla* to create sockets, any application that attempts to create a socket is validated against our policy in the LSM. As a sidebar, while getting the name of the process attempting to create a connection is trivial, `current->comm` (where current is an instance of the `task_struct` data structure in the kernel that indicates the current process), obtaining the full path from which the application is running from is not. The application path is obtained by walking the `vm_area_struct` until the appropriate entry is found in virtual memory, the entry is then fed into the `d_path` function which assigns the application path to a *c-string* that we return to the calling LSM hook. Walking the `vm_area_struct` is typically a process you would want to avoid since it is time consuming, but the path is not stored like the name of the process is in the `task_struct`. The reasons for the path name not being stored in the `task_struct` at the time of execution is that the currently running process can have its pathname changed since in Linux a process can be moved around during its lifetime.

If the LSM determines that the application is *mozilla* then it returns an error code value of 0, meaning to allow it. However, if the application is not *mozilla* then our LSM will return a negative error code value such as *-EPERM* to indicate that permission was denied for the socket create operation. The return value is caught by the socket create system call within the kernel and since the value is less than 0 the system call terminates and the request made by the application to create a socket will have been denied for the current instance.

Now that we have presented the background information on what LSMs are, where they came from, and what they are capable of; it is time to discuss the three separate issues that were discovered during the course of the development of the module for Lockdown.

3.1. Points of Enforcement in the Kernel

3.1.1. Current Implementation

The first issue we came across dealt with the placement of the LSM security hooks within the *Linux* kernel source. Choosing where and when to provide a security hook is at times a non-trivial task. If too many hooks are used, the system is at risk of being slowed down significantly,

due to the inherent delay involved with policy checking before allowing or denying a system call to execute. If too few hooks are used, then the ability to provide a sufficient security framework is hindered.

Linux has 175 hooks for the LSM interface scattered throughout its 327 system calls. A large majority of the LSM hooks occur when the functional code of a system call begins to execute. If a hook is implemented in a currently loaded LSM, program flow is diverted to the block of code that provides functionality for the hook, which is capable of making a decision to allow or deny the system call. If the system call is allowed, the LSM returns a *0* and the system call continues as normal. If however the call is denied, then a negative value error code is returned from the LSM and the system call terminates execution.

However, not only are the number of hooks important, but also the placement of these hooks within the body of code for a system call and the nomenclature used to describe them. As will be discussed momentarily, certain LSM hooks end up providing almost no advantage over other hooks and essentially have no ability to deny an action due to their placement in the code. Additionally, the naming of some of the hooks we dealt with were slightly misleading with what the perceived action was.

3.1.2. Problems Observed

Our continued work with the LSM brought up some interesting issues with the LSM hooks and how they are placed throughout the kernel source. While the LSM security hooks are not available in every system call, they are available in a decent portion to provide for the development of a fairly robust security solution. Our work mainly concentrated on the hooks relevant to socket activity for *AF_INET* (IPv4) and *AF_INET6* (IPv6) which can be found in Table I. The usage of the socket hooks during the communications between a server and client running *Linux* can be found in Figure 3.

The names used for the hooks are relatively straightforward when determining their purpose. For example, the *socket_connect* hook, found on the client side, is responsible for security operations with the connection phase of a client connecting to a server socket. The connect hook works as expected, if the policy on the client side restricts a socket connection based on a specific context: then the *socket_connect* LSM hook simply needs to return a negative error code value and the connection will not be allowed. Some hooks, however, have a few problems. With hooks like *socket_create* and *socket_accept*, the context of a connection is not known at the time these hooks are called, which is at the beginning of the corresponding system call.

For our work with validating if a certain socket should be allowed based on the address, port, application, user, etc (local context) responsible for the socket a decision cannot be made at the time of these two hooks. However, two other hooks associated with create and accept called *socket_post_create* and *socket_post_accept* occur at the end of the corresponding system calls, hence the "post" name being used. However, a major problem with these hooks is that the typical LSM semantics of returning an error code to either validate or invalidate an action is useless at these points for two reasons. The first reason is that the socket has already been established. If the local context violates policy, the socket needs to be torn down via the socket shutdown function call from within the LSM, after it has already been established. The application that established the socket ends up having the system call return successfully only

Table I.

LSM Hook	Parameters
.socket_create	int family, int type, int protocol, int kern
.socket_post_create	struct socket * sock, int family, int type, int protocol, int kern
.socket_bind	struct socket * sock, struct sockaddr * address, int addrlen
.socket_connect	struct socket * sock, struct sockaddr * address, int addrlen
.socket_listen	struct socket * sock, int backlog
.socket_accept	struct socket * sock, struct socket * newsock
.socket_post_accept	struct socket * sock, struct socket * newsock
.socket_sendmsg	struct socket * sock, struct msghdr * msg, int size
.socket_recvmsg	struct socket * sock, struct msghdr * msg, int size, int flags

to have the socket not be accessible since it was shutdown via another method. The second reason is that the error code is not caught by the system call utilizing these LSM security hook. Despite the context of the socket being known in these "post" calls, the standard method of returning an error value will not work in these cases; which contrasts with the rest of the LSM security hooks that have their error code values caught in order to prevent further action by the executing system call.

The current security operations were meant to be minimally invasive in regard to their addition to the kernel source. Keeping the hooks minimally invasive within the source code explains the reasons why the *post* security operations do not have their return values caught. If a *post* security call were to be undone based on the return value, the amount of code required to switch the state of the system back would increase and would complicate matters for the LSM framework designers. Keeping the overhead of implementing a security framework minimal, but still allowing it to be powerful, is however a tough balancing act.

While overall, the points of enforcement for the LSM hooks within the kernel source are sufficient, the advantage of how *post* hooks were implemented becomes the greatest disadvantage in this regard. An LSM designer is required to check an additional hook for certain system calls, *post* hooks. These hooks exist for event notification, so for the socket accept system call the *socket_post_accept* LSM hook indicates that a socket has been accepted on the system. Yet, this is the closest spot, in terms of execution path, for which a connecting socket that violates system policy can be shutdown.

3.1.3. Solutions

Possible solutions to the problem of having the *post* hooks (which are only good for auditing as far as LSM semantics are concerned) are the following: rename the hooks, change the return type, save state for rollback, or move the hook into the decision part of the system call.

The first two suggestions are essentially cosmetic work. Renaming the hooks to reflect their true nature, such as *socket_post_accept* to *socket_audit_accept* or *socket_monitor_accept* would

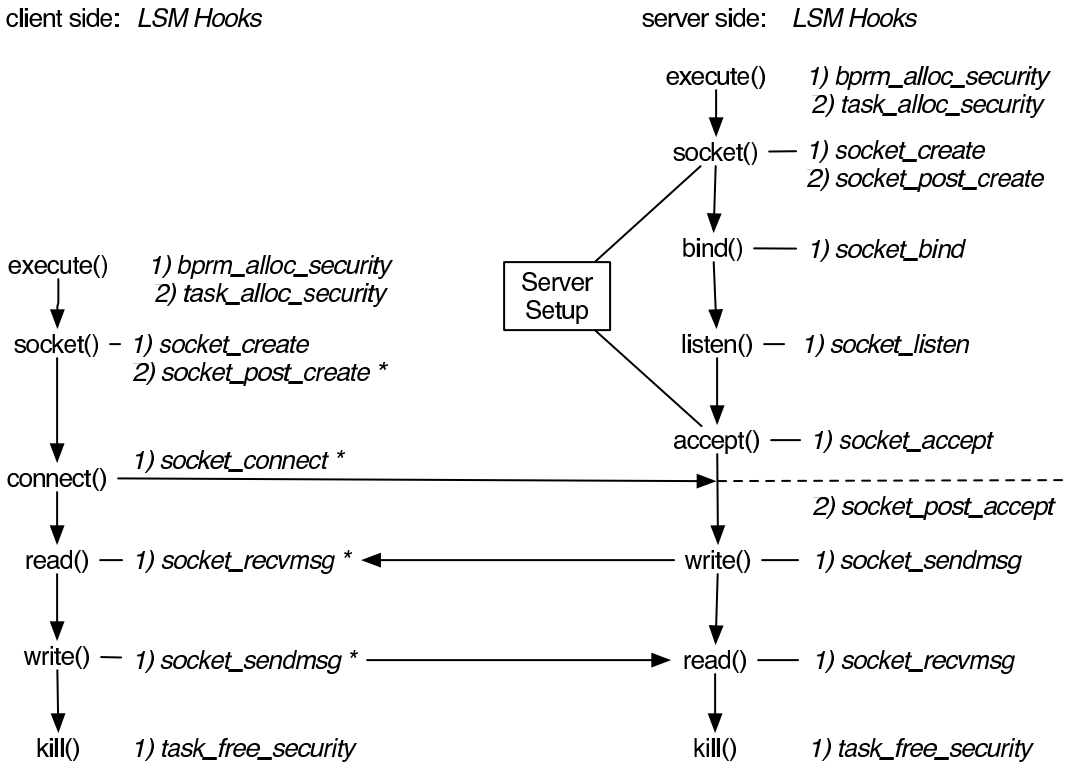

Figure 3. LSM hooks used for socket connectivity.

be of greater use to an LSM developer. The second suggestion of changing the return type to "void" is essentially another cosmetic change that would indicate to the developer that a return code is irrelevant in these situations.

The final two suggestions actually require more thinking on the parts of both the LSM framework designer as well as the kernel source system call designers. Saving state for rollback would require maintaining the previous state before the system call is executed and then, based on the return value from the *post* hook, either roll the state back or keep things as they are. The final suggestion of moving the *post* hook to the decision portion of the system call in order to ascertain the proper context and make a decision before the action is actually accomplished in the kernel is the better of the two. Keeping with our work with the socket LSM hooks, the *socket_post_accept* hook code should be moved into the portion of code that determines where a connection is coming from before the socket is established. This sort of modification requires doing a check in another function than the current system call. So that the context of

an attempt to create a connection is known before the kernel blindly accepts a connection for a socket simply because something is listening and waiting to accept connections. While the last two suggestions may require extra work in their implementations, they would keep the LSM hook semantics consistent. Whereas, the first two suggestions are simpler to implement, the semantics of the LSM hooks would still differ for the post hooks.

3.2. Module Stacking

3.2.1. Current Implementation

The second issue we came across with Linux Security Modules deals with how the stacking or layering of multiple modules on a single host is accomplished. Why would a user want to use multiple LSMs? A user may want to use multiple modules, each for a dedicated task, in order to increase the security of their system.[¶]. A single security product that attempts to do to much typically will result in a solution that is poorly configured by administrators due to the complex nature of the product. Other problems resulting from "do-it-all" solutions is that they, at times, lack the ability to accomplish a single task any better and, in fact, may be worse than a solution that is designed for a specific task. Thus, layering is an accepted practice in the security community, since different groups/companies are more apt at solving certain types of security issues than others.

The stacking policy of LSMs is first-come first served. As discussed previously, the LSM interface in the kernel is capable of supporting both a primary module and a secondary module; where the first security module to load gains control over the primary spot and the next module to load, in time, is assigned to the secondary spot. While the LSM interface allows for up to two security modules to be loaded, it does not attempt to manage or police the interface other than for the primary module. The reason for this is that the kernel allows for the security operations table to have their hooks pointed to the primary loaded module, it doesn't allow the table to redirect to multiple modules. So the primary module is registered with the kernel via the `register_security` function, which corresponds to the primary spot. In order to allow a secondary module there is another registration function call `mod_reg_security` that lets the secondary module register with the primary module, if the primary module supports that feature.

3.2.2. Problems Observed

To further clarify the stacking interface, the first LSM that is loaded into the kernel wins and is made the primary module. It is then the responsibility of the primary module to provide functionality to allow for a secondary module to be loaded into the kernel. The interaction between the primary module and the secondary module is better observed in the

[¶]The traditional approaches of applying security to a system involve a multi-layered defense.

real world LSM modules of *SELinux* and *Capabilities*[‖]. *SELinux* is written to be loaded as the primary module and *Capabilities* is required to be the secondary module; *SELinux* provides functionality to *Capabilities*, but if another module were to take the place of *Capabilities* it would not be allowed since *SELinux* is written to be compatible with only the *Capabilities* module.

For any other LSM that may need to be loaded, the primary module needs to be replaced or re-written in order to interact with a different LSM. We ran into this trouble when attempting to load our custom LSM onto a *Linux Fedora* based development system that was pre-configured with *SELinux* by default during the installation.

Essentially, if we wanted to load a module that did security monitoring/logging only, it would need to stack in with the other LSMs on a system. While some of the commonly found LSMs explained earlier include their own logging features that could be modified, since the code is typically open source (the problem is worse when the source is closed), changing how the logs are produced could break other dependent tools as would be the case with *SELinux* and its automatic policy generation and auditing programs.

It would be ideal however to allow for a secondary module to be deployed without having to rely on the primary module for support. If a primary module chooses not to "play" nicely and allow a secondary module, administrators are forced to make compromises and tradeoffs with the sorts of modules they are able to have deployed, as opposed to a true layered/stacking solution. This seems almost counter to the points made by Linus Torvalds that no single security solution should rule the kernel [11], but if developers are constantly forced to re-write already written features, it may in the end result with a single security solution "ruling all".

3.2.3. Solution

Now that we know how module stacking is currently done, how should it really be done? With the current implementation of module stacking for the LSM interface, in order for a custom LSM to be loaded into the kernel we first had to disable SELinux. While this was not a problem for our group since we do not have SELinux configured for any sort of enforcement. This may be a problem for other institutions that do use it and cannot afford to have it disabled.

Solutions have been proposed in the past to tackle the module stacking issue. Some of the solutions involved research into developing a third-party module that is responsible for handling the stacking between modules so as to not rely on a primary LSM module to offer it. This work seemed to have cooled and no longer appears to be maintained [5, 15]. However, allowing a primary LSM module to orchestrate and conduct the flow of security operations through multiple LSM modules that can be loaded onto a system would be a good approach. In this way, different paths could be taken through the multiple modules loaded on a system depending on what kind of security operation is being called. However, having this ability by default in the Linux kernel its a better fit since the kernel provides the LSM interface.

[‖] *Capabilities* refers to the LSM module implementation of POSIX Capabilities; which indicate wether an object or process is "capable" of doing an action based on a set of privilege bits.

The way in which we propose fixing the issue involves three cases for which the new module stacking model needs to take into account. What should happen if a secondary module wants to:

1. reject something accepted
2. accept something rejected
3. log something rejected

The first thing that needs to be done is for the module stacking to be moved into the kernel from which it is able to handle multiple redirects for the security operations table to the appropriate security modules. A priority list can be used for each hook where the head of the list points to the primary module loaded and the tail keeps track of the secondary (or last loaded module if support is enabled for more than two LSMs). Each list points to the appropriate module in the order it should be called. With this list if a primary module makes a decision on whether to allow or deny an action the result is passed down the list through each module that implements that hook, from head to tail. The module at the tail of the list for a specific hook then has its result returned to the calling system call.

Each module needs to do a check on the value being passed in from the previous module in the list to ensure that the result is not changed unless it needs to be. For example with case number 1, listed above, if the primary module allows the creation of a socket, but there is also a secondary module the result from the primary is passed into the secondary which then decides to deny the creation of the socket, so the return value is changed, and since there is only two modules in this instance the system call ends up being rejected after first having been accepted. The same is true for case 2 with the reverse action occurring, something that was accepted is rejected later on. With case 3 it might also be necessary to plug in a custom logging LSM that simply audits the system. With this redesign in the stacking interface the logging module can load and observe any activity occurring in the LSM framework by plugging in at the tail of the lists for each hook and simply passing the result it gets from modules before it onto the system call without modification. With this method, the logging module acts as a pass through as it is capable of auditing the activity occurring without having to modify any of the data passing through it, except for recording it to disk.

3.3. Proper Error Messages and Event Handling

3.3.1. Current Implementation

The third and final issue we dealt with involves appropriate error messaging and event handling. Being able to properly notify the user and or application of errors or denied actions meaningfully is a desired property in *any* system. Administrators that are attempting to debug issues with a system have a limited amount of time. By being supplied with appropriate, non-cryptic error messages, they can debug and solve problems faster. Additionally, there is the problem of event handling. If an application is expecting a system call to finish, but it fails due to an LSM being loaded, how does the application recover? Application designers need to be made aware if an LSM is responsible for denying an action so that a program can terminate certain actions gracefully, or know not to attempt them.

As detailed during the LSM hooks section, the majority of the hooks have their error return values caught. These values are typical of those found when programming in user-space, except that the values are negative as opposed to positive, but with zero still signifying a successful action (no error). Developers are encouraged to capture these error codes when using system calls in order to determine wether an action was successfully completed or not so that their application does not crash unexpectedly.

3.3.2. Problems Observed

Error code reporting was very important in the work we did with the LSM. When dealing with network sockets we wanted the applications a user runs to be notified when a socket was unable to be created due to a policy violation and that data is not being sent through the socket. Traditionally, software based firewalls such as *IPTables* receive packets that are sent out by a host application. If the packets violate the firewall rules then they are silently dropped. However, the applications a user uses have no idea that packets are being dropped and will continue to send them. This behavior will lead to applications either waiting for a timeout that they themselves set or the application will continue to wait until it crashes or is killed.

Throughout the use of our LSM, we were stopping the creation of any sockets before packets were sent. However, if a socket was allowed and then at a point later in time policy changed during the run of the application, the send and receive socket message hooks (see Table I) would terminate the socket activity. As a result, the application is instantly made aware of something occurring, and can terminate any policy violating actions, gracefully, if it is properly written to check for error codes returned from the system calls.

In Figure 4, we demonstrate the ability and ease for an administrator to further debug a solution that provides error reporting as to one that simply times out. However, throughout our experiences with using the LSM hooks and additionally with loading an LSM, it became clear that more useful error message reporting is needed. For example, in a typical enterprise network setting, there are numerous types of security solutions, some hardware and some software, in addition to possible misconfigurations of routers, improper software coding, and more. It would be ideal if an administrator knew that an application was unable to send data over the network due to a policy violation of the Linux Security Module loaded on that system and which LSM was responsible for the denial. This could easily be resolved with additional error code values that include an "LSM denied action" message, as opposed to a generic statement informing the application that is was unable to create a socket; while better than a timeout, it still leaves some mystery as to what the reason for the inability to create a socket was.**

**By default, Java does not have a timeout value set for sockets.

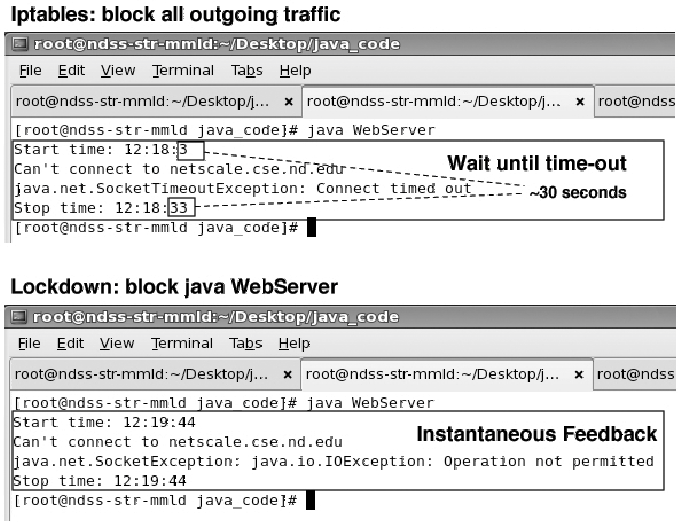

Figure 4. The application view of enforcement comparing *iptables* (above) versus an LSM (Lockdown) (below)

3.3.3. Solution

Enabling the LSM framework to report to the application that a socket it was trying to create or utilize was denied due to an LSM and to further specify which LSM was responsible is the ideal solution to this problem. The first part can be done by expanding on the *linux/error.h* file to include additional error codes to indicate a denied LSM action. Being able to tell the application which LSM, if many are loaded, could be a trickier problem, but who is responsible for doing this depends on if any of the module stacking suggestions are taken into account. For example a third party module could keep track of denied actions and denote who was responsible for what, the kernel could keep track of it, or, if no stacking is involved, the primary module loaded would typically be found to be the culprit since it has the final say in the current "further deny" LSM semantics.

While the current LSM framework was created by keeping the kernel source code changes minimal, a simple addition of adding additional error messages as they pertain to security, the LSM framework can be further enhanced thus empowering both users and administrators. The users and administrators would have better help in debugging and tracking down the cause of denied actions. This could be done easily by duplicating the error code values and separating them by an offset value from their counterparts, such as returning the value: *-EPERM+LSM* would translate to a denied permission by an LSM. The original *-EPERM* error code would exist, the offset of *+LSM* would just refer to LSM specific error codes.

Since many different security modules, programs, daemons, etc may be running at any single instant on a system, it is at times difficult to debug the root cause of a problem if you are unfamiliar with what is currently running and, even if you are familiar, it may still be difficult. In the case of setting up a network socket, the socket connection could fail for many reasons: a server denies the connection; a hardware or software based security solution on or off the host system denies the connection; the application developer has a bug in his code; the network configuration on the host system is not configured properly; etc. Trying to track the cause of a problem down to any single one of these reasons is difficult, thus providing a way to narrow it to a specific portion of the system is beneficial for the end-user.

4. Conclusion

Throughout the course of this paper, we discussed our experiences with the Linux Security Module framework. We posit that modularity is beneficial for complex systems, but is difficult to get absolutely right. The three problems presented in this paper about the LSM framework highlight the difficult nature of modularity as it pertains to security. However, as all systems and software evolve over time, the lessons learned from the experiences presented here can further evolve the LSM framework to enhance its robustness.

The lessons learned during the development of our LSM were the following: First, the security hooks interface should be consistent for all hooks in regard to having the ability to return an error code in order to dictate how a particular action is handled by the kernel. The *post* hooks, as discovered, are in a better position of making an informed decision on an action than their non-post counterparts. These hooks, however, do not follow the semantics of normal LSM hooks isince the error code returned is not checked and acted upon. The solutions presented here attempt to solve the problem via a renaming or a relocation of the hooks in the kernel source so that they can deny an action rather than only serve as auditors. Second, the LSM interface, while in theory provides for module stacking to occur, leaves it up to the module developer to provide the functionality. A module developer becomes reliant on other LSM module designers to allow for stacking, whereas an LSM developer should instead have to depend on the kernel to provide such a feature since it provides the interface. The solutions proposed enable a chain of modules to be loaded where each LSM can handle a different task. A custom logging LSMs could thus be plugged into a chain of security enforcing LSMs without altering the security of the system. Third, and finally, relaying appropriate error messages back to the developers, users, and administrators of a system so that they are made aware as to why certain actions were denied because of an LSM policy violation needs to be added into the framework. While the current iterations are an improvement over an *IPTables* implementation of denying data across a network socket, more intuitive error messaging can still be put into place to ease debugging and frustration by all of those involved.

From the experiences presented it can be further ingrained in the software engineering community that modularity is a great idea, but it is still hard to achieve a perfect system. There are lots of tradeoffs made when designing a modular system, such as keeping the modular interface code simple; providing enough "power" to the modular interface so as to not hinder its ability; and maintaining proper semantics through the entire design process. While not all

of these may be 100% achievable it helps to balance all aspects of the design to essentially give users of the modularity, wether developer or end-user, the best possible experience.

REFERENCES

1. A. Blaich, Q. Liao, B. Sullivan, A. Striegel, D. Thain, and T. Wright. Lockdown: Simplifying Enterprise Network Management with Local Context. Technical report, University of Notre Dame, 2007.
2. F. Corbato and V. Vyssotsky. Introduction and Overview of the Multics System. In *AFIPS Conference Proceedings*, volume 27, pages 185–197. Spartan Books, 1965.
3. N. D. CORPORATION. Tomoyo linux. Technical report, Japan, 2008.
4. T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security*, 2003.
5. S. Hallyn. Linux LSM stacker . http://sourceforge.net/projects/lsm-stacker.
6. C.-C. Huang and A. Kusiak. Modularity in design of products and systems. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 28(1):66–77, January 1998.
7. T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.
8. P. A. Karger and R. R. Schell. Thirty Years Later: Lessons from the Multics Security Evaluation. In *ACSAC*, December 2002.
9. A. Leitner. AppArmor vs SELinux. *Linux-Magazine*, 69:40–42, August 2006.
10. Q. Liao, A. Blaich, A. Striegel, and D. Thain. ENAVis: Enterprise Network Activities Visualization. In *Usenix LISA*, 2008.
11. Linux Kernel Mailing List. Pluggable Schedulers vs. Pluggable Security http://kerneltrap.org/Linux/PluggableSchedulersvsPluggableSecurity, September 2007.
12. J. Morris. Networking in NSA Security-Enhanced Linux. *Linux Journal*, 2005.
13. Netfilter. http://www.netfilter.org/.
14. Novell. AppArmor Application Security for Linux. Technical report, Novell, 2008.
15. M. Quaritsch and T. Winkler. Linux Security Modules Enhancements: Module Stacking Framework and TCP State Transition Hooks for State-Driven NIDS. In *Secure Information and Communication*, 2004.
16. P. J. Salzman, M. Burian, and O. Pomerantz. The linux kernel module programming guide. *http://tldp.org/LDP/lkmpg/2.6/html/*, May 2007.
17. C. Schaufler. The simplified mandatory access control kernel. Technical report, March 2008.
18. S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical Report 01-043, NAI Labs, December 2001.
19. C. Wright. Lsm bitkeeper repository.
20. C. Wright and C. Cowan. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
21. C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Module Framework. In *Ottaw Linux Symposium*, 2002.