Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework

Antony Edwards * Trent Jaeger Xiaolan Zhang
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {jaegert,cxzhang}@us.ibm.com

May 17, 2002

Abstract

We present runtime tools to assist the Linux community in verifying the correctness of the Linux Security Modules (LSM) framework. The LSM framework consists of a set of authorization hooks inserted into the Linux kernel to enable additional authorizations to be performed (e.g., for mandatory access control). When compared to system call interposition, authorization within the kernel has both security and performance advantages, but it is more difficult to verify that placement of the LSM hooks ensures that all the kernel's security-sensitive operations are authorized. We have examined both static and runtime analysis techniques for this verification, and have found them to be complementary. Static analysis is more complex to implement and tends to generate more false positives, but coverage of all type-safe execution paths is possible. Runtime analysis lacks the code and input coverage of static analysis, but tends to be simpler to gather useful information. The major simplifying factor in our runtime verification approach is that we can leverage the fact that most of the LSM hooks are properly placed to identify misplaced hooks. Our runtime verification tools collect the current LSM authorizations and find inconsistencies in these authorizations. We describe our approach for performing runtime verification, the design of the tools that implement this approach, and the anomalous situations found in an LSM-patched Linux 2.4.16 kernel.

1 Introduction

The Linux Security Modules (LSM) project aims to provide a generic framework from which a wide variety of authorization mechanisms and policies can be enforced. Such a framework would enable developers to implement authorization modules of their choosing for the Linux kernel. System administrators can then select the module that best enforces their system's security policy. For example, modules that implement mandatory access control (MAC) policies to enable containment of compromised system services are under development.

The LSM framework is a set of authorization hooks inserted into the Linux kernel. These hooks define the types of authorizations that a module can enforce and their locations. Placing the hooks in the kernel itself rather than at the system call boundary has security and performance advantages. First, placing hooks where the operations are implemented ensures that the authorized objects are the only ones used. For example, system call interposition is susceptible to time-of-check-to-time-of-use (TOCTTOU) attacks [2], where another object is swapped for the authorized object after authorization, because the kernel does not necessarily use the object authorized by interposition. Second, since the authorizations are at the point of the operation, there is no need to redundantly transform system call arguments to authorize kernel objects.

While placing the authorization hooks in the kernel can improve security, it is more difficult to determine whether the hooks mediate and authorize all controlled operations. The system call interface is a nice mediation point because all the kernel's controlled operations (i.e., operations that access security-sensitive data) *must* eventually go through this interface. Inside the kernel, there is no obvious analogue for the system call interface. Any kernel function can contain accesses to one or more security-sensitive data structures. Thus, any mediation interface is at a lower-level of abstraction (e.g., inode member access). In addition to mediation, it is also necessary to ensure that the proper access control policy (e.g., write data) is enforced for each security-

*Work done with the author was at the IBM T.J. Watson Research Center.

sensitive operation. If there is a mismatch between the policy enforced and the controlled operations that are executed under that policy, unauthorized operations can be executed. We believe that manual verification of the correct authorization of a low-level mediation interface is impractical.

Much recent effort has focused on how static analysis tools may aid in the verification of various security properties [4, 9, 12]. As is the trend now, we expect that static analysis will be used where possible, and runtime analysis will be used to complete the analysis [1]. Thus, we are proceeding with the development of both static and runtime verification tools, and have found that static and runtime analysis have complementary features. Our static analysis approach enables comprehensive verification that the variables used in security-sensitive operations have been authorized [15]. However, it is difficult to statically determine the authorization requirements that should be checked. This is because different data and control flows within functions may require different authorizations. A useful insight for performing runtime analysis is the assumption that LSM authorization hooks are correctly placed in most cases. Thus, verification is a matter of finding and resolving inconsistencies in authorization requirements and verifying that the resultant authorization requirements are correct. Runtime analysis enables to implement this approach by collecting the authorizations that are actually performed and displaying the actual authorizations, so anomalous cases (i.e., missing or inconsistent authorizations for an operation) can be identified.

In this paper, we present a runtime verification approach and tools to assist the LSM community and Linux kernel developers in verifying that the LSM authorization hooks completely authorize accesses. The runtime analysis approach involves: (1) instrumenting the Linux kernel to collect security relevant runtime events (e.g., major kernel events, such as system calls, LSM authorizations, and controlled operations) and (2) analysis of the collected data to identify potential errors. We extend GCC to perform analyses of its abstract syntax tree to add instrumentation to the Linux kernel as necessary. Kernel modules collect the runtime events generated by the instrumentation. We also have analysis programs that use a basic filtering language to extract the events of interest for analysis (e.g, for a particular system call), generate *authorization graphs* that show anomalous authorizations, and *sensitivity class lists* that aggregate authorization requirements as comprehensively as possible to minimize the effort to verify authorization requirements. We have found three bugs in LSM hook placement in the file system that have since been fixed, and another anomaly that resulted in significant discussion. We demonstrate the use of these tools on LSM-patched Linux version 2.4.16.

The remainder of the paper is structured as follows. In Section 2, we define the general hook placement problem. In Section 3, we develop an approach to solving the general hook placement problem. In Section 4, we outline the implementation of the tools and discuss the analyses performed and their results. In Section 5, we conclude and describe future work.

2 General Hook Placement Problems

2.1 Concepts

We identify the following key concepts in the construction of an authorization framework:

- **Security-sensitive Operations:** These are the operations that impact the security of the system.
- **Controlled Operations:** A subset of security-sensitive operations that mediate access to all other security-sensitive operations. These operations define a *mediation interface*.
- **Authorization Hooks:** These are the authorization checks in the system (e.g., the LSM-patched Linux kernel).
- **Policy Operations:** These are the conceptual operations authorized by the authorization hooks.

Correct authorization hook placement must ensure that the *authorization hooks* authorize all *security-sensitive operations*. Such authorization tests whether the system's authorization policy permits the requesting principal to execute the particular security-sensitive operations. It is more convenient to express authorization policy at a higher level (e.g., file read or write), so rather than authorizing the individual security-sensitive operations we authorize conceptual operations, which we call *policy operations*. Further, since the number of security-sensitive operations can be large, it is preferable to authorize them once at an interface that mediates all the security-sensitive operations. The set of *controlled operations* defines such a mediation interface. Thus, we define our problem to verify that all controlled operations are authorized for the expected policy operations using the LSM authorization hooks.

[1] Consider the static checking of type-safe C code where possible and the runtime checking of other code used by Ccured [11].

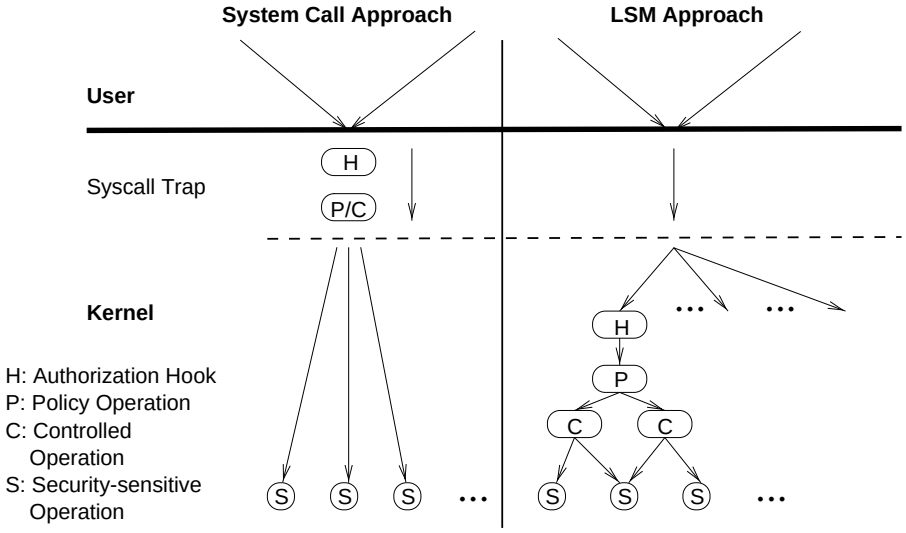

Figure 1: Comparison of concepts between system call interposition framework and LSM.

Identifying the controlled operations is more difficult for the in-kernel mediation of LSM than for the system call mediation mechanisms of the past. As shown in Figure 1, the system call interface is well-known for providing mediation of all the security-sensitive operations in the system call. Therefore, the system call interface can be used both as the controlled operations and the policy operations.

When authorization hooks are inserted in the kernel, a mediation interface is no longer obvious, so the controlled operations and their mapping to policy operations is no longer so easy to identify. For example, rather than verifying file open for write access at the system call interface, the LSM authorizations for directory (exec), link (follow link), and ultimately, the file (write) are performed at the time these operations are to be done. This approach has the benefits of eliminating susceptibility to TOCTTOU attacks [2] and redundant authorization processing, but in order to verify the hook placement more work is necessary to identify the controlled operations, the policy operations they correspond to, and verify that the authorization hooks authorize them properly.

2.2 Relationships to Verify

Figure 2 shows the relationships between the concepts.

1. **Identify Controlled Operations:** Find the set of operations that define a mediation interface through which all security-sensitive operations are accessed.
2. **Determine Authorization Requirements:** For each controlled operation, identify the authorization requirements (i.e., policy) that must be authorized by the LSM hooks.
3. **Verify Complete Authorization:** For each controlled operation, verify that the correct authorization requirements are authorized by LSM hooks.
4. **Verify Hook Placement Clarity:** Controlled operations implementing a policy operation should be easily identifiable from their authorization hooks. Otherwise, even trivial changes to the source may render a hook inoperable.

The basic idea is that we identify the controlled operations and their authorization requirements, then we verify that the authorization hooks mediate those controlled operations properly. First, we need an approach to find the controlled operations in the kernel. Second, because the controlled operations are at a lower level than the policy operations (i.e., authorization requirements), we need an approach by which the authorization requirements of each controlled operation can be determined. Third, we need to compare the LSM hook authorizations made to the expected authorization requirements. These tasks are complex for in-kernel authorization, so it is obvious that automated support is required.

Lastly, to ensure maintainability of the authorization hooks we must verify that the controlled operations representative of each policy operation can be easily determined from the authorization hook locations. This work has been done, but in interest of focus it is out-

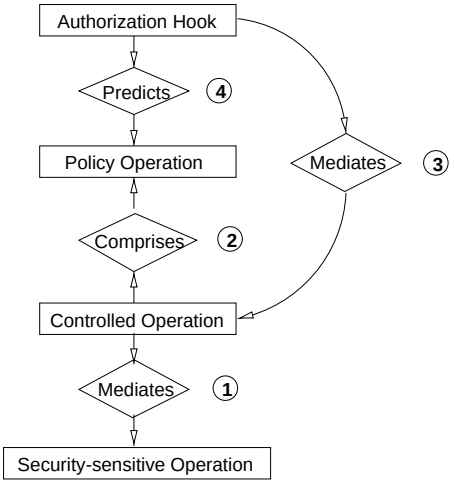

Figure 2: Relationships between the authorization concepts. The verification problems are to: (1) identify controlled operations; (2) determine authorization requirements; (3) verify complete authorization; and (4) verify hook placement clarity.

side the scope of this paper. This is work is presented elsewhere [3].

2.3 Related Work

Recently, static analysis has shown promise in a variety of ways. First, existing program analysis tools have been used to find common security errors, such as buffer overflows and `printf` vulnerabilities [9, 12, 13]. We also use one of these tools, CQUAL [5], in our static analysis approach [15]. These tools require a significant amount of code annotation prior to their use (i.e., scales with size of the code). We perform GCC analysis to automate the annotation task, such that it is practical for the Linux kernel. Such analyses tend to error on the conservative side (i.e., no false negatives) which means that more false positives than reasonable may be generated. We are working on secondary analyses to eliminate obvious false positives. Also, some analysis tasks are difficult to do with static analysis tools. In our case, determining the authorization requirements of individual operation would require complex data and control flow analysis beyond that intended by CQUAL.

Second, Engler *et al* enables extension of GCC, called *xgcc*, to do source analyses, which they refer to as *meta-compilation* [4]. A rule language, called *metal*, is used to express the necessary analysis annotations in a higher-level language. Since the rules match multiple statements, the amount of annotation effort is reduced. A variety of software bugs, including security vulnerabilities, have been found by this tool [1]. While it appears that *xgcc* could be used for the static analysis we perform, the *metal* rules would be more complex than the CQUAL annotation and the GCC analyses. Also, *xgcc* is not intended to derive authorization requirements.

Another related problem is the certification of systems. Historically, the Orange Book [10] was used for guidance in the construction of secure operating systems, but this is now being supplanted by the Common Criteria [7]. However, the certification task is ad hoc and laborious, and has generally not been successful in improving the security of commonly-used operating systems. Gutmann argues in his thesis [6] that certification approaches, including formal verification tools, are doomed to failure unless they represent concepts at the level of the source code. Gutmann also advocates a combination of static and runtime analyses. The approach that we use differs from certification in the sense that it checks for particular errors rather than providing a top-down assurance that the overall system meets its requirements. An interesting research question is whether a sufficient breadth and depth of such checks could provide a confidence comparable to certification. Unlike certification, such confidence could be maintained as the source code evolves.

3 Solution Description

The key insight we leverage in runtime analysis for the Linux Security Modules (LSM) framework is that the LSM authorization hook placement is largely correct, such that cases that are inconsistent with the norm are likely to be indicative of an error. For example, it would be considered unusual if a particular controlled operation has different authorization requirements on different runs of the same system call.

We have found that the attributes of controlled operations can be totally-ordered with respect to their impact on authorization requirements. For example, if all the controlled operations in a system call have the same authorizations, then the value of the other attributes of a controlled operation do not affect the authorizations (i.e., system call is at the top of the order). We use this knowledge to identify cases that are anomalous (i.e., authorizations are sensitive to attributes that they should not be) and to partition controlled operations into their maximal-sized classes by common authorizations. Further unexpected sensitivities in these classes are used to identify errors.

In all of the discussion below, we use the following assumptions. First, we leverage the type safety of much

of the Linux kernel. This does not invalidate any of the errors we find, but there could be other errors as well. Second, we assume that accesses to objects of the authorized data types define the mediation interface. These data types are the ones that correspond to system call concepts (e.g., files, inodes, sockets, skbuffs, ipc message queues, etc.). Access to kernel data is designed to go through these data structures. While we have not explicitly validated this, we have done some more detailed analysis presented elsewhere [3].

3.1 Authorization Sensitivity Attributes

Table 1 lists the attributes of controlled operations to which authorization requirements may be sensitive. We refer this group of attributes collectively as the *authorization sensitivity attributes*. Each controlled operation has information about the conditions under which it was executed (system call, system call inputs, function, location in function, path to controlled operation), the object it was executed upon (datatype and object), and the operation performed (member/access).

These attributes are totally-ordered, such that if the authorizations of controlled operations differ when the value of one factor is changed, then the authorizations also differ when a higher factor is changed. For example, if two controlled operations on a particular object have different authorizations, then that datatype will also have different authorizations for the two controlled operations.

Conversely, if the authorization requirements of controlled operations are insensitive to changes in one factor, then they are also insensitive to changes in all lower factors. For example, if all controlled operations on the same datatype have the same authorizations, then so do all controlled operations on the same (structure) member.

3.2 Authorization Sensitivity Impact

The classification of controlled operations by their authorization sensitivity divides the controlled operations into two categories: (1) known anomalies and (2) sensitivity classes whose authorization requirements need verification. In the first case, sensitivity to some of the authorization sensitivity attributes is considered illegal. We define invariants below for these cases. In the second case, we partition the controlled operations into maximal-sized classes with the same authorizations. These classes enable verification of authorization requirements and identification of anomalous classifications.

3.2.1 Anomalies

The sensitivity of authorizations to the attributes below the double line in Figure 1, *intra-function* and *path*, are always considered to be anomalous. Sensitivities of these types mean that the execution path (path) or location within a function (intra-function) determine the authorization requirements of a particular controlled operation on the same member.

The following invariant formally expresses our path insensitivity invariant.

Path Insensitivity Invariant

$$\forall c_1, c_2 \in C, e_1, e_2 \in E, (c_1 = c_2) \wedge (e_1 = e_2) \rightarrow R(c1, e1) = R(c2, e2) \quad (1)$$

This invariant states that the same controlled operation ($c1 = c2$) run in the same event ($e1 = e2$ defined by the system call and its inputs) must have the same authorization requirements (defined by the function R). That is, the execution path within an event cannot affect a controlled operation's authorization requirements.

Similarly, we define an invariant for intra-function insensitivity.

Intra-Function Insensitivity Invariant

$$\forall c_1, c_2 \in C, e_1, e_2 \in E, (F(c_1) = F(c_2)) \wedge (M(c_1) = M(c_2)) \wedge (e_1 = e_2) \rightarrow R(c1, e1) = R(c2, e2) \quad (2)$$

In this case, two controlled operations in the same function (computed by the function F) and which make the same member access (computed by the function M) must have the same authorization requirements R.

3.2.2 Authorization Sensitivity Classes

For the other cases, we cannot easily identify them as errors. Instead, we partition the controlled operations into authorization sensitivity classes based on their authorizations and attribute sensitivity and determine whether their authorization requirements are correct.

The authorization sensitivity class computation is as follows. For each sensitivity level starting at the highest (system call), we partition the controlled operations into sensitivity classes where all controlled operations have the same value for the sensitivity attribute, then we test

Factor	*Authorizations are same for:*
System Call	all controlled operations in system call
Syscall Inputs	all controlled operations in same system call with same inputs
Datatype	all controlled operations on objects of the same datatype
Object	all controlled operations on the same object
Member	all controlled operations on same datatype, accessing same member, with same operation
Function	all same member controlled operations in same function
Intra-function	same controlled operation instance
Path	same execution path to same controlled operation instance

Table 1: Authorization Sensitivity Factors: names and effects on authorizations

whether the class also has the same authorizations. If not, then we try the next lower attribute and partition based on both attributes and test again. This approach repeats until we have assigned every controlled operation to a sensitivity class.

Partitioning depends on the attribute. For the system call attribute, all the controlled operations of a system call are in one class. For system call inputs, all controlled operations of the same system call and with the same type of inputs are aggregated (see Section 3.3 below). For the datatype attribute, the controlled operations are classified by the system call, inputs, and datatype of the operation's object. Thus, successively finer partitions are created in each step of the analysis.

A classification succeeds (i.e., is *x-sensitive* where *x* is the attribute) if it is the first attribute in which all the controlled operations in that class have the same authorizations. Note that other classes at the same sensitivity that have the same authorizations are aggregated to form the maximal-sized classes. Once the classes are created it is a manual process to verify that the authorizations for each class is correct. For the file system, the number of classes is small enough that manual verification is practical.

As an example, consider the `read` system call. File operations are datatype-sensitive because all controlled operations on file objects are authorized for `read`. Manual verification involves checking that read permission for files is sufficient. Since the read authorization also is intended for the file's inode, we mark the file's inode as authorized for `read` as well. However, after classification, one inode controlled operation is not authorized. It is on a different object, so inode operations are object-sensitive. This is an operation on the directory inode of the file to determine whether a signal should be sent as a result of a read in this directory. Several other file system calls also perform test for notification, and notification is only performed if the original file operation is authorized. Therefore, we can say that this directory inode should also be authorized for file read. The same goes for the current task and superblock as well. It is straightfoward to extend the collection to do this, however. Ultimately, we would expect that all controlled operations in the `read` system call are authorized for read access.

Other than finding an authorization completely missing, the most common way for identifying an error is to find two classifications (i.e., two aggregates with different authorizations) that perform an important common operation. This situation occurred in `fcntl` where two different classifications (based on different system call inputs) operate on the same `f_owner` field (see Section 4.2.4).

In comparison to static analysis, we both verify that the objects are authorized and verify what the authorizations should be in a single step. Both the static and runtime approaches enable quick verification that the file and most inodes are authorized properly. Both identify that the directory inode is not authorized. In both cases, manual examination is necessary to determine whether there is an exploitable situation. However, the runtime approach has an advantage that it is easier to state additional authorizations, such as for the directory inode in the `read` system call. Also, the verification of the specific policy operation authorized is easier in the runtime analysis.

3.3 Necessary Data Collection

By logging system call entry/exits/arguments, function entry/exits, controlled operations (i.e., object, datatype, member, and operation), and authorizations, we collect all the necessary values for the sensitivity attributes. All the information can be easily logged, but the identification of meaningful object identifiers and system call

input changes need some further analysis.

During execution, objects are referenced via function pointers, but this is not necessarily a sufficient identification of an object. For example, an inode has a persistent identifier (i.e., device, inode number) that is used in authorization. Therefore, for each datatype we define a specific approach for computing their object identifiers. These identifiers are used for determining all operations and authorizations on an object.

Across system calls, we assume objects that are used in the same variable have the same authorization requirements. To simulate this we use the first controlled operation in which an object appears as an identifier. If two objects are first accessed in the same controlled operation they must be assigned to the same variable. However, different execution paths may result in the same variable being used in a different controlled operation first. However, aggregation of classes with the same authorization requirements will merge these cases, so this assumption has proven effective.

The system call arguments change on almost every call, but only a few of the arguments really impact authorizations (e.g., the access flag on `open`). Therefore, we collect the arguments, but only use the arguments that we have found impact authorization requirements to do partitioning. Only a few system calls that we have examined have different authorizations based on their input arguments, such as `open`, `ioctl`, and `fcntl`. Because different authorizations are used based on different inputs, these system calls are more complex, and hence, more prone to errors.

4 Implementation

Complete authorization is verified by analyzing (offline) a kernel execution log. This section describes the implementation of the tool that creates this log, the implementation of the log filtering tool used to prepare and display analysis data, and the results of our analysis thusfar.

4.1 Collecting Runtime Information

4.1.1 Log Contents

Table 2 shows the information collected during runtime analysis. Controlled operations are identified by the tuple *(instruction pointer, object type, member, access)*. A *controlled operation ID* is assigned to each unique combination. Authorizations are uniquely identified by *(LSM_hook, policy operation)*. Like controlled operations, a unique authorization ID is assigned to each. Function entry and exit are recorded as well. The function entry address uniquely identifies the function.

Record Type	**Data**		
Controlled Op.	Context ID	Controlled Op. ID	OID
Authorization	Context ID	Auth. ID	OID
Function Entry	Context ID	Instruction Addr.	
Function Exit	Context ID		

Table 2: Log Record Types

For each controlled operation or authorization performed, the log must include the identity of the object (e.g., inode) involved. *Object identities (OIDs)* are defined per object type, for example, inodes are identified by (device ID, inode number) while tasks are identified by process-ID. OIDs are only required to be unique within a context.

We use the concept of a *context* to mean the processing of a kernel event (e.g., a system call). Authorizations are obviously only valid in the context in which they are executed, therefore, the log entries must also include the context of controlled operations and authorizations.

4.1.2 Collection Overview

Figure 3 presents an overview of the tool. Creation of the log involves three stages: the required information must be generated, it must be collected, and it must be written to the log.

Information is generated in three different ways. First, authorization information is generated by the LSM hooks. Second, controlled operation details are generated by compiling the kernel with a modified version of GCC that identifies controlled operations, and instruments the kernel with calls to a handler function before all such operations. Control-flow information is also generated by instrumenting the kernel at compile-time. Third, context information is generated by placing breakpoints in the kernel. These three methods are discussed in more detail in the following sections.

Four kernel modules are loaded to receive the information shown in Figure 3. These modules perform coarse-grained filtering, and arrange the information into the correct format, before passing the record to the logging module. The logging module assigns a context ID to the incoming records and writes the information into a buffer.

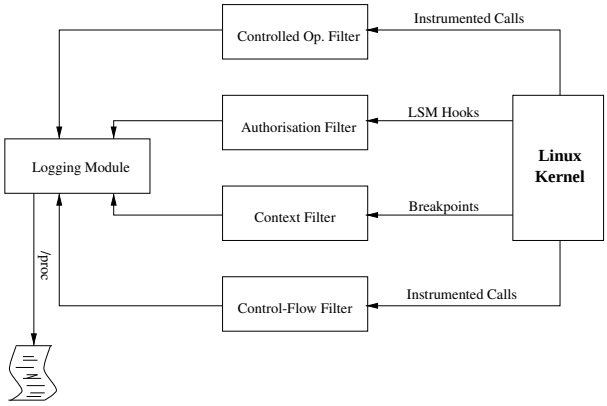

Figure 3: Implementation Architecture

4.1.3 Authorization Information

Hooks to log authorization information are already provided by the LSM patch, so little additional implementation is required. The authorization filter is simply an LSM module that adds a log entry for each authorization. These log entries identify the authorization that was performed (e.g., RMDIR_PARENT, RMDIR_TARGET) and the object authorized.

4.1.4 Controlled Operations

To log controlled operations, we first have to locate controlled operations in the kernel, and then provide a mechanism for detecting the execution of these operations.

Identifying controlled operations in the kernel requires source analysis. Rather than a direct source-code analysis (which is difficult), we chose to identify controlled operations by analyzing GCC's intermediate tree representation. As Linux depends on GCC extensions, a source-code analysis would require using the GCC parser, therefore making use of the tree it already builds seems logical. To identify controlled operations, we traverse the tree looking for expressions in which members of mediated data types are accessed [2]. When a controlled operation is detected we insert a call to a function `__controlled_op` that includes the object, type, member, and access, before the statement in which the expression exists. If the expression is the condition statement of a loop, then a call is inserted before the loop and at the end of each iteration. This call contains all the information required to identify the controlled operation and allow the handler to extract the identity of the object.

[2] These are COMPONENT_REF nodes where the resultant type of the first operand is a mediated type.

A couple of accesses cause problems for this approach. First, it is possible to modify a structure member by taking the address of a member, storing it to a pointer, and changing the member via the pointer. Since the initial access is a read into the pointer variable, it is possible that we may miss the subsequent write. Rather than performing more extensive source analysis to identify these cases, we simply detect when aliasing occurs. Second, it is also possible that we miss accesses to controlled data structures when they are cast to a non-controlled type. This is also detected. Our initial analysis shows that these cases occur in a small number of ways (although for the first, a large number of times), so they can be handled as special cases.

4.1.5 Control Flow

Control flow information is generated by compiling the kernel with the `-finstrument-functions` switch provided by GCC-3.0. This option causes the compiler to insert calls to handler functions at the entry and exit of every function. These handler functions then pass the information to the appropriate module.

4.1.6 Context Information

As there may be multiple execution contexts in the kernel at anytime, all log entries must contain a context ID, so the analysis can tell which entries relate to one another. Unfortunately, no key is available that will uniquely identify a single execution context, therefore, we must choose a non-unique key and define an approach to distinguish contexts with the same key.

We chose the base of the current kernel stack as the non-unique key as we need a key that is at least unique among concurrently active executions, and it would seem impossible for this property to be violated for the stack. While it is unique among concurrently active executions, the kernel stack is not unique per-context for three reasons: all system-calls from the same process use the same kernel stack, once a process dies its kernel stack may be allocated to a new process, and interrupts execute with the kernel stack of the process they interrupt. The critical property here is that although the context key is not unique, contexts with the same key are never interleaved. Therefore, by recording the beginning and end of a context (and the associated key), we can unambiguously assign log entries to contexts.

Fortunately, there are only a few points where a context can begin (all located in `entry.S`), and a roughly

equal number of places that contexts can end. The exit system call is an exceptional case since it never returns, therefore, the `schedule()` call in `do_exit()` is also identified as a context exit point. To generate this information at run time, the context filter inserts breakpoint instructions into the (memory-image of the) kernel at all entry and exit points. When a breakpoint is executed, the context filter creates a log entry containing the context key, and whether this is the beginning or end of a context.

4.1.7 Performance

We did a simple performance check to determine the performance degradation in the instrumented kernel. On an unmodified Linux kernel, LMBench configured for a "fast benchmark" took 3 minutes and 4 seconds to run. The instrumented kernel took 3 minutes and 24 seconds to run the same benchmark for a degradation of slightly over 10%. We believe that this overhead is quite acceptable for such analyses. In this test, as in the results collection described above, we sample 1 out of 20 system calls. The reason for this is to keep the log growth rate lower than the disk throughput rate. Since these benchmarks perform the same system calls many times, we did not notice that we "lost" any security-relevant information. If necessary, a policy for determining when to drop a log entry can be devised.

4.2 Log Analysis

We have also built a tool that enables log analysis for identifying sensitivities in authorization requirements as described in Section 3.1. The tool enables specification of rules for extracting the desired log entries, called *log filtering rules*, and computes the authorization sensitivities given the extracted entries. We can generate two types of displays for sensitivities: (1) *authorization graphs* that show the sensitives between each authorization and controlled operation and (2) *sensitivity class lists* that show the aggregation of controlled operations by authorizations and sensitivity attribute.

While the analysis tool enables flexible analysis, we have found that an optimistic approach is the easiest to manage. That is, we write rules to identify sensitivities at the highest level attribute, system call. If all the controlled operations in the system call execution have the same authorizations (i.e., are system call sensitive), then we only have to verify that the authorizations are correct. If not, we examine whether system call inputs are responsible for the sensitivity. Analysis for system call input sensitivity is somewhat ad hoc, since there are a large number of possible inputs, but very few have an effect on authorizations. Authorization graphs are useful for this task because they give an overall view of the authorization status. After tuning the log filtering rules to handle system call input sensitivities, we then generate partitions (i.e., sensitivity class lists) for controlled operations to do the remaining sensitivity analysis.

4.2.1 Log Filtering Rules

The log filtering tool takes an execution log and set of filtering rules as input, and outputs the log entries that match the rules. The rule language is currently rather low-level, as we have been concerned more with demonstrating feasibility rather than creating a nice high-level rule language. However, we demonstrate the rule language to give a sense of the types of analyses that are possible.

A rule base is defined by a set of rules that define matching requirements. A rule consists of: (1) an index; (2) a dependency specification; (3) a set of statements. The index identifies the rule within the rule base. The dependency states relationships to other rules by index. We can state that a rule can only match entries that are also matched by another rule, (D, i), where i is the index of the other rule. Also, we can state that a dependency that a rule does not include entries matched by another rule i, as (N, i). Lastly, the statements describe the matching conditions for entries. These are specified by identifying the entry type (id_type), and then matching type-specific attributes. Entry types include: events (`CONTEXT`), authorizations (`SEC_CHK`), functions (`FUNC`), and controlled operations (`CNTL_OP`).

Figure 4 shows some example rules. The path sensitive rule finds all authorizations in the context of a `read` system call when a controlled operation at the specified address is run. The first line collects all context entries for a read system call (i.e., the start of the system call). The second line collects all entries of controlled operations at the specified location. The `(D,1)` means that this statement is dependent on statement 1, so only entries within the `read` system call context will be collected. The third line collects all authorizations within the `read` system call context. In this case, each execution of this controlled operation should have the same authorizations or there is a violation of the *path insensitivity invariant* that prohibits a controlled operation from having multiple sets of legal authorizations.

The function sensitive rule collects all authorizations and controlled operations of "read inode member i_flock" within a `read` system call context. The specification

```
# Path sensitive rule for operation at
0xc014f046
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,READ)
2 (D,1) = (+,id_type,CNTL_OP)
(+,di_dfm_ip,0xc014f046)
3 (D,1) = (+,id_type,SEC_CHK)

# Member sensitive rule for inode member
i_flock read access
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,READ)
2 (D,1) = (+,id_type,CNTL_OP)
(+,di_dfm_class,OT_INODE)
(+,di_dfm_member,i_flock)
(+,di_dfm_access,OP_READ)
3 (D,1) = (+,id_type,SEC_CHK)

# Input sensitive rule for open for read ac-
cess, but not path_walk
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,OPEN)
(+,co_ecx,RDONLY)
2 (D,1) = (+,id_type,FUNC)
(+,di_ffm_ip,path_walk)
3 (D,1)(N,2) = (+,ALL,0,0)
```

Figure 4: Example authorization sensitivity filtering rules

of `(D,1)` on the second line means that all controlled operations of this type within a `read` system call will be extracted. If the authorizations associated with this controlled operation are not the same, then the member access is sensitive to its location.

The system call input sensitive rule collects all the log entries in each `open` system call for read-only access. The authorizations of the `open` system call depend on the access for which the file is opened, so `open` is system call input sensitive. Further, we also show a negative filter in this rule that eliminates all entries within the scope of the `path_walk` function. The authorizations for file lookup, including any link traversal, can be separated from those for authorizing the open of this file. Such filtering capabilities enable us to choose our analysis scope flexibly.

4.2.2 Graphical Log Analysis

The analysis tool can also generate graphs that enable visual analysis of the filtered data. Using these graphs, it is possible to verify the authorization sensitivities by inspection, as we will describe below. An *authorization graph* consists of two sets of nodes in a filtered log:

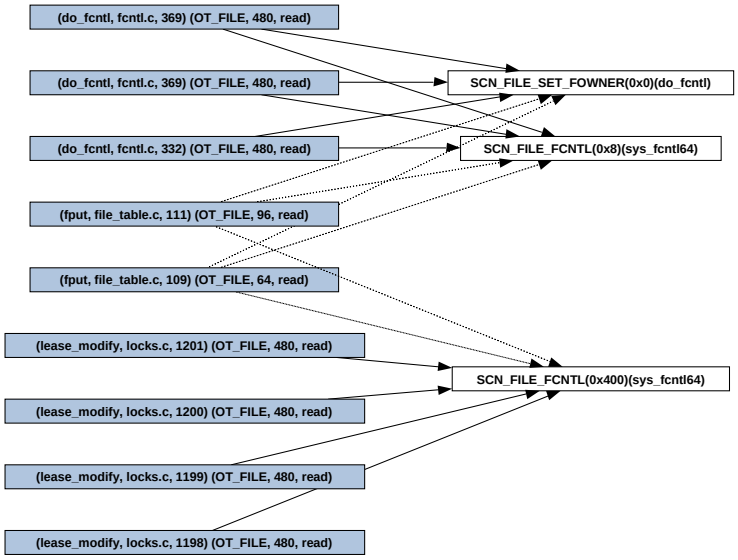

Figure 5: Authorization graph for `fcntl` calls for `F_SETLEASE` (controlled operations in `lease_modify` and `fput`) and `F_SETOWN` (controlled operations in `do_fcntl` and `put`). When command is `F_SETOWN` both `FCNTL` and `SET_OWNER` are authorized, but only `FCNTL` is authorized for `F_SETLEASE`.

(1) the controlled operations and (2) the authorizations made. Edges are drawn from each controlled operation to the authorizations that have been satisfied when it is run. There are two types of edges: (1) *always* edges mean that the associated authorization is satisfied every time the controlled operation is run and (2) *sometimes* edges mean that the associated authorization is satisfied at least once when the controlled operation is run.

An always edge (as well as the lack of an edge) means that the authorization is not sensitive to lower-level attributes. A sometimes edge indicates a sensitivity. The lack of an edge where an edge would be expected would indicate a missing authorization.

Figure 5 shows an example authorization graph. The example graph is displayed using the daVinci graph visualization tool [3] [14]. In this case, the authorization graph shows the controlled operations and the authorizations for two types of `fcntl`

[3] daVinci is only a temporary solution. It has been taken out of the freeware domain, so we are planning on switching to another graph visualization tool before any final publication. Therefore, these graphs are included to show what an authorization graph looks like, and we will include graphs based on another tool at publication.

calls: (1) `fcntl(fd, F_SETOWN, pid_owner)` and (2) `fcntl(fd,F_SETLEASE,F_UNLCK)`. The controlled operation nodes include location (function name, file name, line number) and operation (data type, member offset, operation type) information. The authorization nodes include the authorization, command, and function containing the authorization. *Always* edges are indicated by a solid line and *sometimes* edges are indicated by a dashed line. If no edge exists between a controlled operation and an authorization, then that authorization is never performed for that operation.

By visually analyzing this graph we can identify whether the invariants described in Section 3.2.1 hold for the current graph or not. In this case, the *sometimes* relation between `fput` and its authorizations may indicate a problem. Also, the fact that different sets of authorizations are made for the same field (member offset 480 which happens to be `f_owner`) may be indicative of a problem. Manual investigation is then required to identify whether any inconsistency is due to an error or a legitimate sensitivity.

4.2.3 Sensitivity Class Lists

The sensitivity class lists show the partition of the controlled operations by sensitivity level in which authorizations are consistent and the authorization requirements at those levels. This partition is computed using the algorithm described in Section 3.2.2. The sensitivity class lists provide a different view than the authorization graphs of the same authorization results. Whereas an authorization graph shows the relationship between each individual controlled operation and authorization, the sensitivity class lists show the collection of controlled operations with the same authorization requirements. The sensitivity class lists makes more obvious the number of different authorization cases that exist in the data. Also, the sensitivity class lists are easier to use in regression testing since they are textual [8].

Figure 6 shows the partition of controlled operations for the `read` system call. This partition is used as the example in Section 3.2. As described there, the sensitivity class list shows two classes that are sensitive at the datatype level: one for tasks and superblocks with no authorizations and one for files with read authorization. Then, the sensitivity class list has two classes that are object-sensitive: one for the inode that is read authorized and one for its directory that has no authorizations. Ultimately, we expect to annotate current task, file's directory, and file's superblock as read authorized which will result in all controlled operations having the same

```
DFN d 0 FILE f_dentry -1
DFN d 0 FILE f_dentry 1
DFN d 0 FILE f_vfsmnt -1
DFN d 0 FILE f_op -1
...
SFN(ALWAYS) d 0 FILE_READ
-----------------------
DFN d 1 SUPERBLOCK s_blocksize -1
DFN d 1 SUPERBLOCK s_type -1
...
DFN d 1 TASK state -1
DFN d 1 TASK state 0
DFN d 1 TASK flags -1
...
SFN() NONE
-----------------------
DFN o 0 INODE i_blocks -1
DFN o 0 INODE i_blocks 1
DFN o 0 INODE i_version -1
...
SFN(ALWAYS) o 0 FILE_READ
-----------------------
DFN o 1 INODE i_dnotify_mask -1
SFN() NONE
-----------------------
```

Figure 6: Sensitivity class list for `read` system call with the following fields: (1) entry type (DFN or SFN); (2) sensitivity (d for datatype and o for object); (3) class number; (4) datatype; (5) member; (6) access identifier.

```
# fcntl for F_SETOWN with just the field
f_owner
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,fcntl)
(+,co_ecx,F_SETOWN)
2 (D,1) = (+,id_type,SEC_CHK)
3 (D,1) = (+,id_type,CNTL_OP)
(+,di_dfm_member,f_owner)
# fcntl for F_SETLEASE with just the field
f_owner
4 = (+,id_type,CONTEXT) (+,di_cfm_eax,fcntl)
(+,co_ecx,F_SETLEASE) (+,co_edx,F_UNLCK)
5 (D,4) = (+,id_type,SEC_CHK)
6 (D,4) = (+,id_type,CNTL_OP)
(+,di_dfm_member,f_owner)
```

Figure 7: Rules for finding the `f_owner` anomaly.

authorization (i.e., being system call sensitive).

Most of our experience is with the file system although we have also examined task authorizations. Most objects have either one or no authorizations, so the sensitivity class lists are not too complex. The system call `unlink` is one of the few where an object has multiple authorizations. Using sensitivity class lists it is easy to see that the directory inode has three authorizations (exec, write, unlink_dir) and the inode being removed has one (unlink_file) because they are object-sensitive and placed in different classes. Thus, for the file system and the task operations we have examined, authorization graphs and sensitivity class lists have been sufficient to verify authorizations.

4.2.4 Sample Analysis

We briefly demonstrate a sample analysis for an anomaly that we found. While the approach to finding anomalies was developed concurrently to actually finding anomalies, we used roughly the same approach as described although some of it was not automated. This anomaly occurs in the `fcntl` system call. The sensitivity class list for `fcntl` shows that its authorizations are system call input sensitive. The values of the `cmd` and `arg` parameters to `fcntl` can change the authorizations that are required. We use authorization graphs to look at the authorizations under the different inputs since it is easier to see coarse-grained problems – lots of sometimes edges occur.

Figure 7 contains two sets of rules: (1) one which collects all authorizations and controlled operations of the file structure field `f_owner` in a `fcntl(fd, F_SETOWN, pid_owner)` system call and (2) one which collects all authorizations and controlled operations on the field `f_owner` in a `fcntl(fd, F_SETLEASE, F_UNLCK)` system call. Note that this is same rule (less the `fput` controlled operations) used to generate the graph in Figure 5.

In Figure 5, we see that some of the controlled operations are authorized for the `fcntl` and `set_fowner` authorizations and some are only authorized for `fcntl`. This is despite the fact that the controlled operations access the same field, `f_owner` (offset 480). Given this anomaly, we examined the kernel source to determine whether an exploit of this anomaly is possible. We discuss the results of this analysis in the next section.

4.3 Results

We applied the December 10, 2001 LSM patch to the Linux 2.4.16 source and compiled the kernel using our modified version of GCC-3.0 [4]. To create an execution log to analyze, we executed in parallel three instances of LMBench, the SAINT vulnerability tool (www.wwdsi.com/saint/), a kernel compile, some regular usage, and some test programs that we wrote as we became suspicious of anomalies. Since the effectiveness of runtime analysis depends on running enough code, the development of benchmarks that cover the enough of the interesting paths must be developed. For example, LMBench only runs about 20% of the kernel code. Also, our static analysis tool finds some other potential errors for which benchmarks should be written to determine if they can be exploited.

We have instrumented the kernel to collect controlled operations on the major kernel data structures: files, inodes, superblocks, tasks, sockets, and skbuffs. Thusfar, we have only done a detailed analysis on the file system authorizations, and an initial analysis on task authorizations. Since the file system is fairly well-understood, we did not expect a large number of anomalies, but we found some nonetheless.

- **Member Sensitive (multiple system calls):** We found that there is no authorization hook in the function `setgroups16`, but that we can reset the task's group set. An authorization protects this operation in `setgroups`. This hook was missed because these backwards ABI-compatible 16-bit task operations, such as `setuid16` and `setchown16`

[4] Keeping up with kernel version is not a great deal of work. We have the system running on Linux 2.4.18 now, and the only thing we had to do was update our authorization filter to the current LSM interface.

usually convert their 16-bit values to 32-bit values and call the current versions that do contain authorizations. However, since `setgroups16` sets an array, it is easier not to convert the array, so the current version (that contains a hook) is not called. Note that there is no `setgroups16` call in the current version of libc, so we had to write an assembler program to perform this exploit.

- **Member Sensitive (single system call):** The `f_owner.pid` member of `struct file` tells the kernel which process to send signals to regarding IO on this file. Setting this field is authorized by `security_ops->file_ops->set_fowner` if the user tries to set it directly via `fcntl(fd, F_SETOWN, pid_owner)`. However, if a user removes a lease from a file via `fcntl( fd, F_SETLEASE, F_UNLCK)`, the owner is set to zero without the authorization being performed. Furthermore, a process can set the owner of a Universal TUN device (drivers/net/tun.c) to itself without the authorization being performed. To achieve this, the process calls `ioctl(fd, F_SETFL, FASYNC)` on an open, attached, TUN device.

- **Member Sensitive (single system call):** During our investigation of the sensitivity of `filp.f_owner` described above, we we found that access to `filp.f_owner.signum` (the signal that should be sent upon IO completion) can be set without the authorization via `fcntl(fd, F_SETSIG, sig)`.

- **System Call Sensitive (missing authorization):** A `security_ops->file_ops->read()` authorization is performed at the beginning of every read system call. This authorization is required since the authorization performed when the file was originally opened may no longer be valid, due to the process changing its security attributes, the file changing its security attributes, the file being used by a new process, or a change in the security policy. This authorization, however, is not performed during a page-fault on a memory-mapped file. Therefore, once a process has memory-mapped a file it can continue to read the file regardless of changes to security attributes or security policy.

We engaged in a discussion with that resulted in a patch to all the anomalies, except the one for reading memory-mapped files. The community decided that a file that requires read authorization must not be memory-mapped. We are encouraged that we have been able to help find and fix hook placement problems. We have found that the analysis approach can document the current state of an LSM kernel, so future LSM kernels can be regression tested. Also, we are developing an approach that takes into account both the static and runtime analyses. Lastly, we are also encouraged that our initial assumption that LSM is mostly correct appears valid, at least for the file system.

5 Conclusions

In this paper, we presented tools for assisting the Linux community in verifying the correctness of the Linux Security Modules (LSM) framework. The LSM framework consists of a set of authorization hooks placed inside the kernel, so it is more difficult to identify the complete mediation points. We leveraged the fact that most of the LSM hooks are properly placed to identify misplaced hooks. We used structure member operations on major kernel data structures as the mediation interface and collected the authorizations on these operations. By analyzing the output of a runtime logging tool, we identified the operations whose authorizations were inconsistent. We have analyzed the file system and some task operations and found some anomalies that could have been exploited. Working with the LSM community, these problems have since been fixed. For example, we found that some variants of `fcntl` enabled operations to be performed that were authorized in other cases. Ultimately, we found that runtime analysis is useful for verifying systems where a inconsistencies from the norm are likely to be errors. Further development of benchmarks for runtime analysis remains an challenge.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy 2002*, May 2002.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[3] A. Edwards, T. Jaeger, and X. Zhang. Verifying authorization hook placement for the Linux Security Modules framework. Technical Report 22254, IBM, December 2001.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.

[5] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 192–203, May 1999.

[6] P. Gutmann. The design and verification of a cryptographic security architecture, August 2000. Submitted thesis. Available at www.cs.auckland.ac.nz/ pgut001/pubs/thesis.html.

[7] ITSEC. *Common Criteria for Information Security Technology Evaluation*. ITSEC, 1998. Available at www.commoncriteria.org.

[8] T. Jaeger, X. Zhang, and A. Edwards. Maintaining the correctness of the Linux Security Modules framework. In *Proceedings of the 2002 Ottawa Linux Symposium*, 2002. To appear.

[9] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, pages 177–190, 2001.

[10] NCSC. *Trusted Computer Security Evaluation Criteria*. National Computer Security Center, 1985. DoD 5200.28-STD, also known as the Orange Book.

[11] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, January 2002.

[12] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, pages 201–216, 2001.

[13] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium (NDSS 2000)*, 2000.

[14] M. Werner. The graph visualization system daVinci 2.1. Available at http://www.informatik.uni-bremen.de/ davinci.

[15] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002. To appear.