Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Maintaining the Correctness of the Linux Security Modules Framework

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

Abstract

In this paper, we present an approach, supported by software tools, for maintaining the correctness of the Linux Security Modules (LSM) framework (the LSM community is aiming for inclusion in Linux 2.5). The LSM framework consists of a set of function call hooks placed at locations in the Linux kernel that enable greater control of user-level processes' use of kernel functionality, such as is necessary to enforce mandatory access control. However, the placement of LSM hooks within the kernel means that kernel modifications may inadvertently introduce security holes. Fundamentally, our approach consists of complementary static and runtime analysis; runtime analysis determines the authorization requirements and static analysis verifies these requirements across the entire kernel source. Initially, the focus has been on finding and fixing LSM errors, but now we examine how such an approach may be used by kernel development community to maintain the correctness of the LSM framework. We find that much of the verification process can be automated, regression testing across kernel versions can be made resilient to several types of changes, such as source line numbers, but reduction of false positives remains a key issue.

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 (i.e., generic function pointers) 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 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). Also, it is necessary to link these operations with their access control policy (e.g., write data) to ensure that the correct authorizations are made for each controlled 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.

We have examined both static and runtime analysis techniques for verifying LSM authorization hook placement [6, 20]. Our static analysis approach identifies kernel variables of key data types (e.g., inodes, tasks, sockets, etc.) that are accessed prior to authorization. The advantage of static analysis is that its complete coverage of execution paths (both data and control) enables it to find potential errors more easily. Many successes with static analysis have been reported recently [7, 11, 16]. The effectiveness of static analysis is limited by the manual effort required for annotation and the number of false positives that are generated [1]. Also, some tasks are very difficult for static analysis. However, runtime analysis requires benchmarks that provide sufficient coverage and also creates false positives that must be managed. Thus far, our experience has been that runtime analysis provides a useful complement for static analysis, so both types of analyses need to be performed to obtain effective verification.

While our initial results have been positive [2], ultimately, we believe that it is necessary that such analysis become part of the kernel development process to really maintain the effectiveness of the LSM framework. As the Linux kernel is modified, the LSM authorization hooks may become misplaced. That is, some security-sensitive operations that were previously executed only after authorization may now become accessible without proper authorization. Since the subtleties of authorization may be non-trivial, the kernel developers need a tool that enables them to verify that the authorization hooks protect the system as they did before or identify the cases that need examination. Further, kernel developers need a way of communicating changes that need to be examined by the LSM community.

In this paper, we outline the analysis capabilities of our static and runtime tools and describe how they are used together to perform LSM verification. We do not provide a detailed discussion of the analysis tools, so interested readers are directed elsewhere for that information [6, 20]. We would also like to make such tools available and practical for the kernel development community, so we examine how effectively the analysis steps can be automated and what issues the users of the analy-

[1] Static analysis is overly conservative because some impossible paths are considered which can lead to some false positives.

[2] Five LSM authorization hooks have been added or revised due to the results of our analysis tools.

sis tools must resolve in order to complete the analysis. We find that much of the verification process can be automated, regression testing across kernel versions can be made resilient to minor changes, such as source line numbers, but reduction of false positives remains a key issue. While the analysis tools are not yet available as open source, we are working to obtain such approval.

The remainder of the paper is structured as follows. In Section 2, we review the goals and status of the LSM project. In Section 3, we define the general hook placement problem. In Section 4, we review the static and runtime analysis verification approaches. In Section 5, we outline how LSM verification experts use the static and runtime analysis tools in a complementary fashion to perform a complete LSM verification. In Section 6, we examine how the analysis tools can be made practical for use by the kernel development community. In Section 7, we conclude and describe future work.

2 Linux Security Modules

The Linux Security Modules (LSM) framework is being developed to address insufficiencies in traditional UNIX security. Historically, UNIX operating systems provide a single authorization mechanism and policy model for controlling file system access. This approach has been found to be lacking for a variety of reasons, and these inadequacies have been exacerbated by emerging technologies. First, the UNIX policy model lacks the expressive power necessary for some security requirements. UNIX file mode bits enable control of file accesses based on three types of relationships that the subject may have with the file: file owner, file group owner, and others. Some reasonable access control combinations cannot be expressed using this approach, so extension have been created (e.g., access control lists (ACL)). Second, the UNIX access control model provides discretionary access control (DAC) whereby the owner of the objects controls the distribution of access. Thus, users can accidentally give away rights that they did not intend, and the all-powerful user *root*, as which a wide variety of diverse programs run, can change access control policy in the system arbitrarily. Third, with the advent of new programming paradigms, such as mobile code, the UNIX assumption that every one of the users' processes should always have all of the users' rights became flawed [3], and it was found that the UNIX access control model was too limited to enable the necessary level of flexibility [10, 1, 9]. Fourth, controlling access to a variety of other objects besides files was also found to be necessary, and, in some cases, restricting the relationships that objects may enter is necessary [17]. For example, the ability to mount one file system on another is a controlled operation on the establishment of that relationship between the two file systems.

Initially, the authorization mechanisms proposed to address these limitations were inserted at the user-system boundary (e.g., by wrapping system calls [1] or callbacks [9]). By not integrating the authorization mechanisms within the kernel, the authorization mechanism lacks the kernel state at the time that the operation is performed. Attacks have been found that can take advantage of the interval between the time of the authorization and the time at which the operation is invoked [2]. Further, the performance of the system is degraded because the kernel state must be computed twice if the authorization mechanism is placed at the system call interface. Recent research work on improving the UNIX authorization mechanism in Linux has focused on inserting hooks to the authorization mechanism in the kernel directly [4, 13, 14, 15, 18]. However, the variety of authorization hook placements and styles resulted in ad hoc modifications to the Linux ker-

nel.

Another major advancement has been the separation between the authorization mechanism and the policy model used. The work on DTOS and Flask security architectures demonstrated how the authorization policy server can be separated from the authorization mechanism [12, 17]. Thus, a variety of access control policies can be supported. In particular, a variety of mandatory access control (MAC) policies can be explored. An advantage of MAC policies is that provable containment of overt process actions is possible, so protection of the TCB and key applications can be implemented. Various flavors of MAC policy models have been examined, but no one approach has been shown to be superior. The design of effective policy models and policies themselves remains an open research issue.

The LSM project includes several of the parties working on independent Linux kernel authorization mechanisms, in particular Security-Enhanced Linux (SELinux) and Immunix SubDomain, to create a generic framework for calling authorization modules from within the kernel. Motivation to unite these mechanisms came when Linus Torvalds outlined his goals for such a framework [19]. Linus stated that he wants authorization to be implemented by a module accessible via generic hooks. The hope that an acceptable authorization framework would be integrated with the mainline Linux kernel has resulted in a comprehensive LSM implementation.

As of Linux 2.4.16, LSM consists of 216 authorization hooks inserted in the kernel that can call 153 distinct authorization functions defined by the authorization modules (i.e., loadable kernel modules). The authorization hooks enable authorization of a wide variety of operations, including operations on files, inodes, sockets, IPC messages, IPC message queues, semaphores, tasks, modules, skbuffs, devices, and various global kernel variables. Authorization modules for SELinux, SubDomain, and OpenWALL have been built for LSM, so LSM is capable of enforcing MAC policies already.

3 General Hook Placement Problems

3.1 Concepts

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

- **Authorization Hooks:** These are the authorization checks in the system (e.g., the LSM-patched Linux kernel).
- **Policy Operations:** These are the operations for which authorization policy is defined in the authorization hooks. Because we would like to identify code that is representative of the policy operation, they are practically defined as the first controlled operation (see below) requiring this policy.
- **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*.

The definition of these concepts is made clear by a comparison between system call mediation and the in-kernel mediation used by LSM shown in Figure 1. When authorization hooks are placed at the system call interface, the policy operations (e.g., the conceptual operation

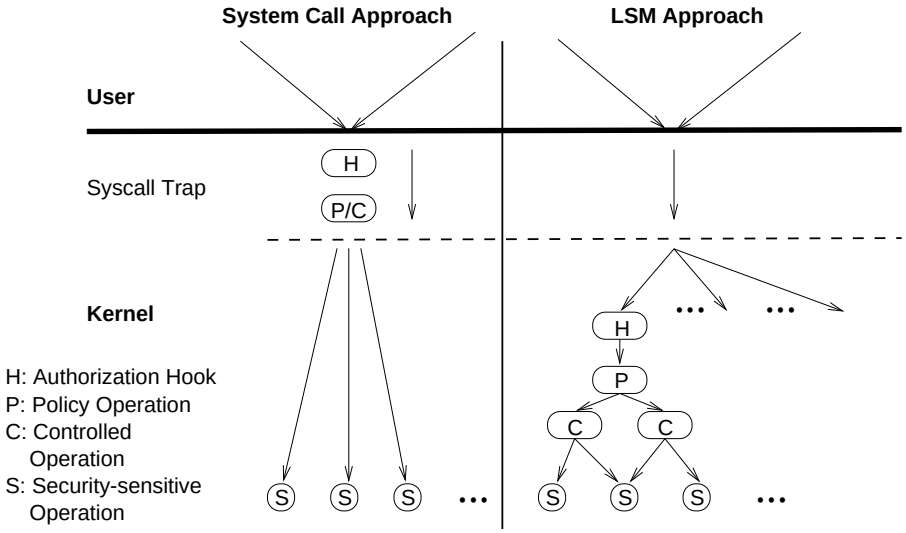

Figure 1: Comparison of concepts between system call interposition framework and LSM.

`write`) and controlled operations (e.g., where mediation of all file opens for write access occur at the system call `sys_open` with the access flag `WRONLY`) are effectively the same. This is because policy is specified at the system call interface, and the system call interface also provides complete mediation. The security-sensitive operations in both cases are the data accesses made to security-relevant kernel data, such as files, inodes, mappings, and pages.

When authorization hooks are inserted in the kernel, the level of complete mediation is the kernel source code, so the policy operations and controlled operations are no longer necessarily the same. 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 open are performed at the time these operations are to be done. This eliminates susceptibility to TOCTTOU attacks [2] and redundant processing. The kernel source is complex, however, so it is no longer clear that all security-sensitive operations are actually authorized properly before they are run.

Given the breadth and variety of security-sensitive operations, we would like to identify a higher-level interface for verifying their proper LSM authorization. This interface must mediate all access from the authorization hooks to the security-sensitive operation. This interface is referred to as the *mediation interface* and is defined by a set of controlled operations.

3.2 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 policy operations that must be authorized by the LSM hooks.
3. **Verify Complete Authorization:** For each controlled operation, verify that the policy operations (i.e., authorization

requirements) are authorized by LSM hooks.

4. **Verify Hook Placement Clarity:** Policy operations should be easily identifiable from their authorization hooks. Otherwise, even trivial changes to the source may render the 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. This verifies, that the LSM hook placement is correct with respect to this set of controlled operations and authorization requirements. When the mediation interface is shown to be correct, it verifies LSM hook placement with respect to all security-sensitive operations. These tasks are complex, so it is obvious that automated tools are necessary.

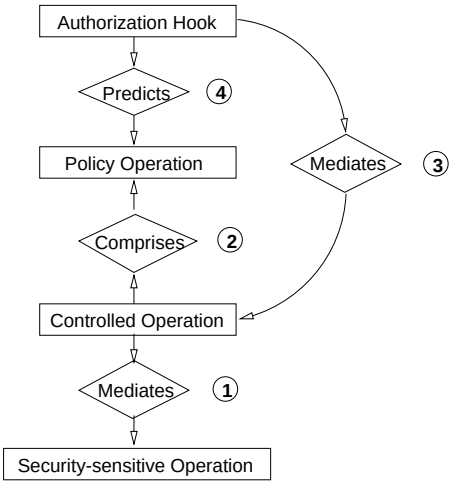

Figure 2: Relationships between the authorization concepts. The verification problems are to: (1) identify controlled operations; (2) determine authorization requirements; (3) verify complete authorization; and (4) verify hook placement clarity.

In addition, we found that additional automated support is necessary to identify the controlled operations and their authorization requirements. First, manual identification of the controlled operations is a tedious task. We must develop an approach by which controlled operations can be selected from the set of security-sensitive operations. Once this approach has been determined automated techniques are needed to extracted these operations from the kernel source. Second, because the controlled operations are at a lower level than the policy operations, we need to determine the policy operations (i.e., authorization requirements) for a controlled operation. Since we expect a large number of controlled operations, it is necessary to develop an approach to simplify the means for identifying their authorization requirements.

Lastly, to ensure maintainability of the authorization hooks we can verify that the policy operations can be easily determined from the authorization hook locations. This work is has been done, but in interest of focus it is outside the scope of this paper. This is work is presented elsewhere [6].

3.3 Related Work

We are not aware of any tools that perform any of the tasks outlined above. While static analysis has had some promising results lately [7, 11, 16], the problems upon which they have been applied have been different and narrower in scope (e.g., buffer overflow detection). We believe that static analysis tools will eventually provide some important improvements in the verifications described above, but some analyses will be easier to do with runtime tools (e.g., due to reduced specification for comprehensive tests).

4 Solution Background

In this section, we review the approaches we devised for using static and runtime analysis to verify the placement of LSM authorization

hooks.

4.1 CQUAL Static Analysis

We use the CQUAL type-based static analysis tool as the basis for our static analysis [8]. CQUAL supports user-defined *type qualifiers* that are used in the same way as the standard C type qualifiers such as `const`. We define two type qualifiers, `checked` and `unchecked`. The idea is that a variable with a `unchecked` qualifier cannot be used when a variable with a `checked` qualifier is expected. This simulates the need to authorize variables before they are used in controlled operations.

The following code segment demonstrates the type of violation we want to detect. Function `func_a` expects a `checked` file pointer as its parameter, but the parameter passed is of type `unchecked` file pointer.

```
void func_a(struct file
            *checked filp);

void func_b( void )
{
        struct file * unchecked filp;
        ...
        func_a(filp);
        ...
}
```

As input to CQUAL, we define type relations between the `checked` and `unchecked` type qualifiers that represent the requirement that a `checked` type cannot be used when an `unchecked` is expected. Using its inference rules, CQUAL performs *qualifier inference* to detect violations against these type relations. These violations are called *type errors*. CQUAL reports both the variables involved in the type errors and the shortest paths to type error creation for these variables. For a more detailed description of CQUAL, please refer to the original paper on CQUAL [8].

In order to do this analysis, CQUAL requires that the target source be annotated with the type qualifiers. This is an arduous and error-prone task for a program like the Linux kernel, so we use GCC analysis to automate the annotation process. There are three GCC analyses we perform to prepare the source code for CQUAL processing.

1. All controlled object must be initialized to `unchecked`.
2. All function parameters that are used in a controlled operation must be marked as `checked`.
3. Authorizations must upgrade the authorized object's qualified type to `checked`.

In order to ensure that static analysis is sound (i.e., no type errors are missed by the analysis), we perform some additional GCC analyses. For example, we verify no reassignments of variables and check for intra-procedural type errors. These analyses are sometimes primitive, but they have limited the amount of manual work required sufficiently. We are working with the CQUAL community and others to improve the effectiveness of static analysis for this purpose. For a more detailed description of our static analysis, see our paper [20].

With our static analysis, we have identified some LSM vulnerabilities in Linux 2.4.9, but since the runtime analysis tool was done first we have found only one new, exploitable vulnerability. It has since been fixed in later versions of LSM [5]. Figure 3 shows the vulnerability.

The code fragment demonstrates a time-of-check-to-time-of-use [2] (TOCTTOU) vulnerability. In this case, the `filp` variable is authorized in `sys_fcntl`. However, a new version of `filp` is extracted from the file descriptor and used in the function `fcntl_getlk`.

Since a user process can modify its mapping between its file descriptors and the files they reference this error is exploitable.

4.2 Vali Runtime Analysis

We have developed a tool, called Vali [3], for collecting key kernel runtime events (Vali runtime) and analyzing this runtime data (Vali analysis) to determine whether LSM authorization hooks are correctly placed [6]. The key insight of the Vali analysis is that most of the LSM authorization hooks are correctly placed, so it is anomalies in the authorization results that enables us to identify errors. Using this approach, we have found 5 significant anomalies in LSM authorization hook placement, 4 of which have been identified as bugs and fixed.

Vali consists of kernel instrumentation tools, kernel data collection modules, and data analysis tools. The kernel instrumentation tools build a Linux kernel for which kernel events (e.g., system calls and interrupts), function entry/exits, LSM authorizations, and controlled operations can be logged by the data collection modules. We use the same kind of GCC analysis as we did for the static analysis to find controlled operations in the kernel. Other events are easily instrumented through GCC instrumentation (functions), breakpoints on entry addresses (kernel events), and the LSM authorization hooks themselves (authorizations).

Loadable kernel modules for each type of instrumentation collect these events. The main problem with data collection is not the performance overhead, but the data collection bandwidth. The performance overhead of instrumentation is only about 10%, and since the analysis kernel is not a production kernel this is quite acceptable. However, the rate at which data is generated can exceed the disk through-

[3]Vali is the Norse God of Justice and the first four letters in "Validate."

```
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
               unsigned int cmd,
               unsigned long arg)
{
  struct file * filp;
  ...
  filp = fget(fd);
  ...
  err = security_ops->file_ops
        ->fcntl(filp, cmd, arg);
  ...
  err = do_fcntl(fd, cmd, arg,
                 filp);
  ...
}
static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETLK:
      err = fcntl_setlk(fd, ...);
    ...
  }
  ...
}
/* from fs/locks.c */
fcntl_getlk(fd, ...) {
  struct file * filp;
  ...
  filp = fget(fd);
  /* operate on filp */
  ...
}
```

Figure 3: Code path from Linux 2.4.9 containing an exploitable type error.

```
# open for read access
1 = (+,id_type,CONTEXT)
(+,di_cfm_eax,sys_open)
(+,co_ecx,RDONLY)
2 (D,1) = (+,ALL,0,0)
# open for read-write
1 = (+,id_type,CONTEXT)
(+,di_cfm_eax,sys_open)
(+,co_ecx,RDWR)
2 (D,1) = (+,ALL,0,0)
```

Figure 4: Filtering rules for open system call (sys_open) with read and read-write access flags. The (D,1) means that the rule should use only the records that have matched this rule number in the second argument. There is also a negation counterpart (N,x) where the specified records are excluded.

put rate, so we enable event filtering. Currently, this is simply collecting 1 out of n events where n can be tuned. Ultimately, we would like to be able to control the types of events collected to ensure that rarer events are not missed.

The logged data identifies the objects used in controlled operations and the authorizations made upon those objects. While different object instances are used in different system call instances, objects referenced by the same variable (i.e., used in the same controlled operations) should normally have the same authorizations. This is not entirely true as some system calls (e.g., `open`, `ioctl`, etc.) may imply different authorizations based on the flags that are sent. Therefore, we have defined a simple filtering language to identify the kernel events that should have the same authorizations for all objects (see Figure 4 for examples). Per filter, all objects should have the same authorizations. Therefore, we can identify anomalous cases that do not have the expected authorizations, and these cases are often errors. Because the filters enable focusing on a small set of operations, we have had more success finding problems using the runtime analysis than the static analysis. We have found errors ranging from missing authorizations for an obscure system call `getgroups16` to a missing authorization for resetting the fowner of a file using one of the flag variants of `fcntl` (see Figure 5).

The runtime analysis also does something that the static analysis does not: it identifies the expected authorizations for an object in a system call. Cases that are consistent identify a belief in the set of authorizations that are required on an object. These authorization requirements can be used as input to the static analysis tool which can then be used to verify the correct authorizations, not just the existence of an authorization. While most of the controlled operations require just one authorization, the error in the `fcntl` case above was in the lack of a second authorization to check the permission to set the owner.

5 LSM Community Analysis Approach

As might be gathered by the previous section, we find that the two analysis approaches are quite complementary. In this section, we outline the approach intended for use by LSM verification experts to verify LSM authorization hook placement using our analysis tools. We discuss how the kernel development community might use the analysis tools to maintain LSM correctness in the following section.

Verification of LSM authorization hook placement involves the following steps:

1. **Checked/Unchecked Static Analysis**: We first apply our static analysis approach to find variables for which no authoriza-

```
/* from fs/fcntl.c */
static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETOWN:
     /* set fowner is authorized
        for filp */
     err = LSM->file_ops->set_fowner(
           filp);
     ...
     filp->f_owner.pid = arg;
     ...
    case F_SETLEASE:
      err = fcntl_setlease(fd, filp,
            arg);
    ...
  }
  ...
}
/* from fs/locks.c */
fcntl_setlease(unsigned int fd,
       struct file *filp,
       long arg) {
  struct file_lock *my_before;
  ...
  if (my_before != NULL) {
    error = lease_modify(my_before,
                    arg, fd, filp);
  ...
}
lease_modify(struct file_lock
             **before,
             int arg, int fd,
             struct file *filp) {
  ...
  if (arg == F_UNLCK) {
     /* ERROR: could set active
        fowner to 0 */
     filp->f_owner.pid = 0;
     ...
  }
...
}
```

Figure 5: Code path from Linux 2.4.16 containing an exploitable error for the system call `fcntl(fd, F_SETLEASE, F_UNLCK)` whereby the `pid` of an active lock can be set to 0.

tions are performed. Since there are a significant number ($\tilde{3}0$ for the VFS layer alone for $\tilde{2}50$ controlled variables) of type errors, these must be further classified to eliminate all those that are known not to be a real error.

2. **Runtime Analysis for Authorization Requirements**: Using benchmarks that cover as much of the kernel source as possible plus potential exploits derived for testing the remaining static analysis type errors, perform the runtime analysis to derive the kernel authorization requirements.

3. **Static Analysis Using the Authorization Requirements**: More complete coverage of the kernel is possible using static analysis, so repeat the static analysis using the authorization requirements.

4. **Runtime Verification Using All Exploits**: Repeat the runtime analysis using any newly derived exploits from the second static analysis.

5.1 Checked/Unchecked Analysis

In the first step, the static analysis is applied to the kernel source, and some number of type errors are identified by CQUAL. We have fully automated this process, but the problem of examining type errors and determining whether they are exploitable must ultimately be done by an expert. The type error rate (type errors per variable of a controlled data type) varies from subsystem to subsystem, but it is 12% for the VFS layer (higher than usual) in Linux 2.4.9. Therefore, we have 30 variables in the VFS layer that are not explicitly authorized before they are used in a controlled operation.

Many of these type errors are not exploitable errors, however. In the VFS layer, these type

errors come in three kinds: (1) use in initialization or other "safe" functions; (2) extraction of inodes from checked dentries; and (3) unknown type errors. The first two kinds of errors are not exploitable errors, so we need to change our analysis to prevent them from being generated. In the first case, we can relabel these functions, so they no longer require a `checked` variable. Since some functions are not obviously "safe," so there is some possibility for error here. When these functions are modified, some re-evaluation is necessary to maintain is status. In the second case, we need to change CQUAL to infer a `checked` variable if it comes from a `checked` field, and no user process can modify this relationship. For example, if we check a dentry then later extract the inode from this dentry, the LSM hooks believe that the inode is authorized also. This is because the dentry inode relationship is fixed (i.e., not modifiable by user processes). For situations of this type, we can infer the variable is `checked`. CQUAL does not support this kind of reasoning, but we are working with the CQUAL community to do this.

For other type errors, we need to find other means to distinguish whether they are real errors or not. For many of these cases, we develop exploit programs to try to find vulnerabilities with respect to the LSM authorizations. At present, this is a manual process, but we would like to automate some aspects of this process based on the nature of the type error. Ultimately, some degree of manual effort will always be required in processing type errors.

5.2 Authorization Requirements Generation

Second, we then perform the runtime analysis given the benchmark and exploit programs to discover vulnerabilities, identify anomalies in authorizations, and determine the authorization requirements of the controlled operations. The exploit programs identify vulnerabilities automatically, so we do not detail them further here (see the detailed static analysis paper [20] for the program that exploits the vulnerability in Figure 3). We discuss anomaly identification and the determination of authorization requirements here.

The Vali runtime analysis identifies the authorizations that are made on each object in a kernel event (i.e., system call with the same expected authorizations). Any variations in the authorizations signal either a miss definition a kernel event (i.e., the authorizations really change when we did not expect) or an anomaly in authorization. Recall that kernel events are defined by filter rules. Since writing these rules depends on deep knowledge of the kernel and LSM authorization, we expect that the LSM verification experts will write such filter rules to correctly generate anomalies. For example, we defined rules for open read-only and read-write cases in Figure 4.

An authorization anomaly is a case where an authorization only occurs in some of the cases of the kernel event. In Figure 5, the `set_owner` authorization was missing even though the inode field `f_owner` was accessed in a `fcntl` system call. While different flags to `fcntl` may result in different authorizations, we found that because the same fields were accessed with different authorizations, there could be a potential problem. Thus, this error was found in trying to write an appropriate filter for the different variants of `fcntl` invocations. A complete discussion of the different types of anomalies and their use in the classification of kernel events is provided in our runtime analysis paper [6].

Once the filters are written, they can be used by the Vali analysis tool to generate the object authorizations and any anomalous authorizations. In general, an object's authorizations may vary depending on the operations performed (i.e.,

```
DFN d 0 0 384 -1
DFN d 0 0 384 1
DFN d 0 0 400 -1
...
SFN(ALWAYS) d 0 0xc
```

Figure 6: Vali analysis output aggregating all inode and file operations with the same authorization. The DFN fields indicate: (1) "d" is *datatype-insensitive*, meaning all operations on the datatype have the same requirements; (2) first 0 is aggregate id; (3) second 0 is the class id for "inode"; (4) next number is the member id accessed; (5) last is the access type code.

field and access type) and the functions in which the operations are performed. The Vali analysis tool aggregates the common authorizations first by operation type, if their authorizations are always the same, then by function. That is, we hope that all operations in an event have the same authorizations. If not, then other operation attributes are used to aggregate when the authorizations are the same for operations with the same attribute value. We use operation datatype, object, member access, and access+function as the aggregation attributes. Figure 6 shows a datatype aggregation for inodes in the `read` system call (files are also aggregated). Figure 7 shows that authorizations may vary depending on the member access or the function in which the access is performed . The aggregation attributes are totally-ordered, so we try to aggregate at the attribute that yields the largest aggregate.

Maximizing aggregation also has the positive outcome that it reduces the number of regression differences. For example, if a controlled operation has the same authorization requirements regardless of the function in which it is run, then moving or adding the operation to a new function does not signal a regression difference.

```
DFN f 0 0 320 -1
SFN(ALWAYS) f 0 0x37

DFN f 1 0 1152 1
SFN(ALWAYS) f 1 0x13

DFN i 0 0 1216 1 ext2_lookup
...
SFN(ALWAYS) i 0 0x1a

DFN i 1 0 1216 -1 find_inode
SFN(ALWAYS) i 1 0x37
```

Figure 7: Vali analysis output showing four groups of *function-insensitive* ("f") and *function-internals-sensitive* ("i") operations for stat64. Function-insensitive accesses have the same authorizations regardless of the function in which the dangerous operation appears. Function-internals-insensitive operations have different authorizations as accesses to member 1216 in the two functions `ext2_lookup` and `find_inode` identify.

When all anomalies have been resolved, then the output defines the authorization requirements for the controlled operations in the kernel events in which they are run. Of course, some authorizations could be missing entirely for all runs, but we expect that the aggregated requirements will make it possible to verify this with reasonable effort.

5.3 Static Analysis Using Requirements

The authorization requirements found using the Vali runtime analysis can be used as input to the CQUAL static analysis. Three changes must be made to use the authorization requirements:

1. Authorizations must result in variables of a qualified type of the authorization made.

2. Functions annotations must be changed to

expect parameters of qualified types depending on the authorizations expected.

3. A type qualifier lattice must be built that represents the legal relationships between type qualifiers.

In the first case, we must now change `unchecked` variables to a qualified type commensurate with the authorization (e.g., `read_authorized`). Given that a variable can have multiple authorizations that depends on the kernel's control flow, such annotation itself is a subject of static analysis. CQUAL does not help with annotation (i.e., it is an input to CQUAL), so we must devise another technique for proper annotation. Fortunately, objects almost always have only one check, and no more than three, so this problem is handled manually at present.

The Vali authorization requirements for controlled operations generated from the Vali runtime analysis are used to identify the type qualifier requirements of functions. Figure 8 displays the output data from Vali used as input to this process. Variables are identified by their line number, data type, member access, and access type. While this is not completely unambiguous, it is sufficient for the kernel currently and we can identify ambiguities that cannot be resolved automatically. The SFNs identified as ALWAYS indicate the authorization requirements to be enforced on this variable.

Since multiple kernel events may use the same functions, the type qualifiers are, in general, the OR of these cases. This is represented using CQUAL's type qualifier lattice [8]. Since CQUAL's granularity is a function, code within a function that is called only when different authorization requirements are expected will not necessarily be handled properly by CQUAL. An example of this is `lease_modify` in Figure 5 where the authorization for `set_owner`

```
DFN(namei.c, 207)(OT_INODE, 1152, -1)
SFN(ALWAYS): SCN_INODE_PERMISSION_EXEC
SFN(ALWAYS): SCN_INODE_PERMISSION_WRITE
SFN(ALWAYS): SCN_INODE_UNLINK_DIR
SFN(NEVER): SCN_INODE_UNLINK_FILE
SFN(NEVER): SCN_INODE_DELETE
```

Figure 8: Vali runtime output for the authorization requirements for a controlled operation in the `unlink` system call. The DFN indicates the line number, variable type, operation, and access which can be used to identify the variable in most cases. The ALWAYS SFNs indicate the authorization requirements.

is only necessary if `(arg == F_UNLCK)`. Where a combination of authorizations to a function are of the form $A \vee (A \wedge B)$ where A and B are authorization types, then we may need to manually annotate the code where $A \wedge B$ is required. We can do this by creating a dummy function call that requires $A \wedge B$. Ultimately, better intra-procedural analysis is required to find blocks of code within functions that require different authorizations, however, because such manual annotation limits regression testing (see Section 6.1).

5.4 Further Exploit Verification

Any exploit programs derived from the second static analysis are added to the Vali runtime analysis benchmarks, and runtime analysis is rerun. Since these programs are mainly looking for vulnerabilities, rather than anomalies, the output will be largely unchanged from step 2.

6 Kernel Community Approach

As the kernel evolves, the placement of the LSM authorization hooks may be invalidated. Since the kernel development community at large will modify the kernel, we need an ap-

proach in which the kernel modifications can proceed while maintaining the verification status of the LSM authorization hooks. Clearly, the kernel development community will not be inclined to perform the verification process of the LSM community as described above. However, it is possible for the kernel development community to leverage this work to maintain LSM verification.

Basically, we envision that kernel developer's task in maintaining the LSM authorization hook verification will involve regression testing on the static and runtime analyses. As part of an extended kernel build, the static analysis process can be run as in step 3 of the LSM community process above. The type errors generated above can be compared to the existing classifications to verify that no new type errors or type error paths are created.

Since some unresolved type errors are likely to remain for a while, it is ultimately necessary to perform runtime regression testing. While this task requires more work because the new kernel must be run, much, if not all, of this task can also be automated. In this case, the goal of the kernel development community is to identify any new anomalies or new authorization requirements (e.g., if a new object is added) to the LSM community. As described below, the Vali runtime analysis tool can identify such differences automatically.

6.1 Static Regression Testing

Since the GCC annotation process, CQUAL analysis, and output classification can be performed automatically, static LSM regression testing can be integrated as an extension to the automated build process. We first describe the tasks that are necessary to automate static analysis as part of the build process. While the analysis will generate the output automatically, some situations arise in manual analysis is currently required. We list these cases and examine their implications.

The build process for static analysis consists of the following steps:

- **GCC analysis**: Our extended GCC compiler must be used to build the kernel. The compiler creates a log of controlled operations, controlled variable declarations, and LSM authorization hooks. The following Makefile modifications are necessary:

  ```
  CC = $(CROSS_COMPILE)/\
  $(VALI_GCC)/gcc \
      --param ae-analyses=8243
  ```

  The parameter `ae-analyses` indicates the types of information that our extended GCC compiler gathers.

- **Perl annotation**: Perl scripts have been written to pre-process the GCC analysis output into a form that is then used by a second set of Perl scripts to automatically annotate the Linux source code with CQUAL type qualifiers.

  The GCC analysis generates output for each controlled operation, such as seen in Figure 9.

  The first set of Perl scripts processes such output into the form:

  ```
  /usr/src/linux-2.4.9-\
      .../fs/attr.c:61 \
      inode_setattr *inode
  ```

- **CQUAL Linux build**: CQUAL requires some pre-processing of the Linux source code before it can perform the analysis (e.g., removal of blanks and comments). The standard GCC compiler can be used for this step.

```
DEBUG_ACCESS: controlled operation:
   file = /usr/src/linux-2.4.9.../fs/attr.c
   current_function = inode_setattr.
   function_line = 63.
   current_line_number = 66
   access_type = write
   name = (*inode)
   member = i_uid (384)
   is_parameter = 1.
DEBUG_ACCESS: controlled operation:
   file = /usr/src/linux-2.4.9.../fs/attr.c
   current_function = inode_setattr.
   function_line = 63.
   current_line_number = 68
   access_type = write
   name = (*inode)
   member = i_gid (416)
   is_parameter = 1.
```

Figure 9: The GCC analysis output.

```
$(CC) $(CFLAGS) -E $< | \
  $(CQUALBINDIR)/remblanks >$*.ii
```

- **CQUAL analysis**: CQUAL can then be used to perform the static analysis. The first step runs the CQUAL analysis. The second step generates the type error path information. See Figure 10.

- **Authorization requirement annotation**: Vali runtime analysis generates authorization requirements per controlled operation as shown in Figure 8. We are in the process of writing Perl scripts to apply these requirements to the annotation of authorizations specific to the requirements described above. We hope to be able to report on this at the symposium.

While we have not done detailed time measurements, we have found that the entire analysis adds about 10 minutes to the build time. GCC analysis adds little overhead to the kernel build. Perl processing takes about 5 minutes for the kernel. CQUAL analysis takes approximately another 5 minutes for the kernel.

The current static analysis process verifies that the LSM authorization hook placement is correct, but some situations need further, manual examination. These cases are listed below:

1. **Changes to "safe" functions**: The GCC analysis identifies the addition of a new controlled operation to a function formerly classified as "safe," see Section 5.1.

2. **Changes to manually annotated functions**: A source comparison detects a change to a function with any manual annotation (e.g., the addition of dummy functions for ORed authorization requirements, see Section 5.3).

3. **New type error variables**: The CQUAL analysis identifies any new variables that have type errors.

4. **New shortest type error paths**: The

```
$(CQUALBINDIR)/cqual -prelude \
    $(CQUALDIR)/config/prelude.i.security -config
$(CQUALDIR)/config/lattice.security attr.ii 2>attr.path
```

Figure 10: Performing static analysis with CQUAL.

CQUAL analysis identifies any new shortest type error path for a variable with an existing type error.

The first two situations are cases where the dependencies of the analysis have changed, such that the analysis may no longer be sound. While we hope to eliminate such dependencies through further analysis, we expect the analysis will always be subject to some number of dependencies. In fact, as the analysis becomes more elaborate, the complexity of dependencies increases, so the current set may prove to be the best option.

The second two situations are the identification of a new type error that may indicate a real vulnerability. In order to reduce the number of false positives, secondary analyses are necessary to identify them. These analyses may have dependencies (e.g., that is the cause of case 1), so the cost of managing the dependencies must be less than the value of removing the false positives.

While it is not completely clear where the balance between manual effort on the part of the kernel developer and LSM community is in this process, we anticipate the following. Our goal is that most notifications of case 1 and 2 can be handled trivially by the kernel development community and the LSM community can verify. Errors of case 3 and 4 may also be handled by the development community in many cases, but again the LSM community may do deeper verifications and develop classifiers to eliminate identifiable false positives.

6.2 Runtime Regression Testing

Since we expect that there will always be some number of type errors for which exploits can become possible and some tasks that are more easily or better done by runtime analysis, we strongly recommend performing the runtime regression analysis. However, this analysis is more time-consuming than the static analysis in two key ways: (1) the instrumented kernel must be built and (2) the runtime analysis benchmarks must be executed on the instrumented kernel.

At present, the build process for a Vali-instrumented kernel, runtime logging modules, and analysis tools is completed automated. However, the execution of the analysis is not automated at present. The main tasks that are not automated are: (1) the collection of instruction pointer locations for kernel entry/exit points used to identify the kernel events and (2) the runtime execution. The first task only involves "grepping" the generated object dump for a few well-known instruction, so it appears straightfoward to automate this. We are looking into how to automate the runtime data collection using VMware [4].

In order to enable regression testing, the Vali runtime analysis tool generates output that does not include line number or instruction pointer information, so that regression can be done across minor kernel modifications. Further, aggregation of controlled operations that are not function-sensitive enables regression across kernel modifications regardless of func-

[4] VMware is a trademark of VMware, Inc.

tions executed in a kernel event.

Figures 6 and 7 give an idea of how the output from the Vali runtime tool enables regression. The output shows the controlled operations with the same authorization requirements. In cases where the authorization requirements of controlled operations are sensitive to the function in which the operation is run, more information is displayed. In this case, if the controlled operation is moved from one function to another, the regression test identifies the change.

Given aggregation, the following types of changes between regression tests are possible:

- **New controlled operation in an aggregate**: An operation has been added, and it has been classified with an existing aggregate.
- **Remove controlled operation from an aggregate**: An operation has been deleted, so it no longer appears.
- **Move controlled operation to another aggregate**: Either an authorization or an operation has been moved such that a different set of authorizations are active when the operation is performed.
- **Create a new aggregate**: A new set of authorizations has been created or a new sensitivity has been triggered such that a new aggregate of operations and permissions has been created.

The addition and removal of controlled operations is not a major change if they adhere to the existing aggregates. However, it is always wise to verify that the operations are consistent with the aggregations assigned to them. The move of operations to other aggregates or the creation of new aggregates are significant changes that warrant review.

7 Conclusions and Future Work

In this paper, we outline static and runtime analysis tools that we have developed to verify the correctness of LSM authorization hook placement. These tools have been used to find five, since fixed, errors in LSM hook placements. We believe that such verification should not be a one-time process, but rather it should practical for kernel developers to perform regression testing as the kernel is modified. The problem is to automate the analysis process as much as possible and only provide test results that really require examination by the development community, as much as possible. We demonstrate that static analysis process and most of the runtime analysis process are automated already. We also identify the types of analysis results that the tools will report to the developers. While it is nice to eliminate as many false positives as possible, we are limited by the Halting Problem as to how many can be removed in general and the means for identifying false positives introduces dependencies that also must be verified. At present, we do not eliminate most false positives automatically, but expect that the LSM community will identify them as such and regression over these will be sufficient (i.e., as long as few, new false positives are introduced little effort will be required to handle them). The generated output is low-level which enables quick comparison, but still makes is difficult for developers. Interfaces for handling this information are a significant area of future work.

References

[1] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

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

[3] N. S. Borenstein. Computational mail as a network infrastructure for computer-supported cooperative work. In *Proceedings of the Fourth ACM CSCW Conference*, pages 67–74, 1992.

[4] Wirex Corp. Immunix security technology. Available at `http://www.immunix.com/Immunix/index.html`.

[5] A Edwards. [PATCH] add lock hook to prevent race, January 2002. Linux Security Modules mailing list at `http://mail.wirex.com/pipermail/linux-security-module/2002-January/002570.html`.

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

[7] 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.

[8] 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.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *The Sixth USENIX Security Symposium Proceedings*, pages 1–12, 1996.

[10] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, 2001.

[12] S. Minear. Providing policy control over object operations in a Mach-based system. In *Proceedings of the Fifth USENIX Security Symposium*, 1995.

[13] NSA. Security-Enhanced Linux (SELinux). Available at `http://www.nsa.gov/selinux`.

[14] LIDS organization. Linux intrusion detection system. Available at `http://www.lids.org`.

[15] A. Ott. Rule set-based access control (RSBAC) for Linux. Available at `http://www.rsbac.org`.

[16] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–216, 2001.

[17] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lepreau. The Flask security architecture: System support for diverse policies. In *Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[18] Argus Systems. Argus PitBull LX. Available at `http://www.argus-systems.com`.

[19] L. Torvalds and C. Cowan. Greetings, April 2001. Linux Security Modules mailing list at `mail.wirex.com/pipermail/linux-security-module/2001-April/000005.html`.

[20] 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.