Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework

1. GENERAL HOOK PLACEMENT PROBLEMS
   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

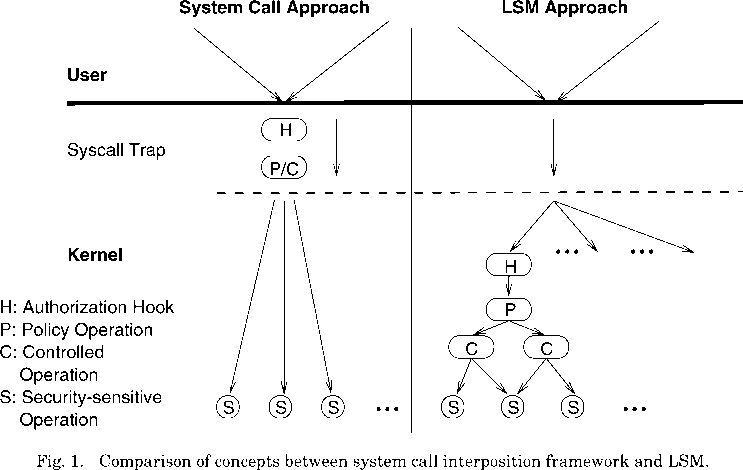
a mediation interface.

—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 opera^^ons. Such authorizations test 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 poZicy 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.

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 are 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 [Bishop and Dilger 1996] 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.

* 1. 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.

⑶ Verify Complete Authorization: For each controlled operation, ver­ify that the correct authorization requirements are authorized by LSM hooks.

⑷ Verify Hook Placement Clarity: Controlled operations implementing a policy operation should be easily identifiable from their authorization

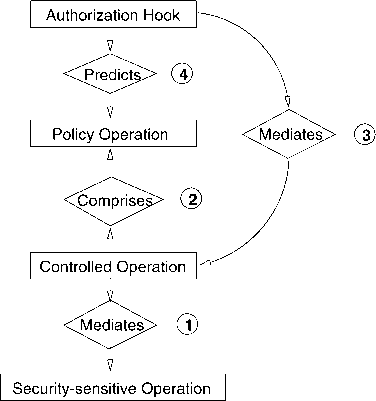


Fig. 2. Relationships between the authorization concepts. The verification problems are to: (1) identify controlled operations; (2) determine authorization requirements; (3) verify complete au­thorization; and (4) verify hook placement clarity.

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 to identify the representa­tive 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.

The mapping of controlled operations to authorization requirements is not necessarily static. For example, a number of the same operations may be ex­ecuted on a file open for reading as on a file open for writing. Thus, context also is a determining factor in mapping controlled operations to authorization requirements. Our approach must enable context-dependencies to be managed effectively, such that the expected relationships between controlled operations and authorization requirements can be tested.

* 1. Related Work

Recently, static analysis to verify security properties has shown promise. First, existing program analysis tools have been used to find common security errors, such as buffer overflows and printf vulnerabilities [Ball et al. 2003; Das et al. 2002; Larochelle and Evans 2001; Shankar et al. 2001; Wagner et al. 2000].

Second, specialized tools have been developed for finding security vulnerabil­ities, such as xgcc [Engler et al. 2000], ITS4/RATS [Viega et al. 2000], MOPS [Chen and Wagner 2002], MAGIC [Chaki et al. 2003], and so on [Ganapathy et al. 2003].

Static analysis tools are based on formal properties of programming lan­guages, so they can be used for complete analysis (i.e., no false negatives). However, static program verification is computationally expensive, so simplifi­cations are often made in the analysis models. These simplifications can result in more conservative analyses (i.e., more false positives) or abstraction of cer­tain properties (i.e., false negatives). Also, static analysis tools can require a significant amount of effort for code annotation, which is necessary to build the desired analysis model.

Specialized analysis tools focus on specific types ofbugs. Engler et al. enables extension of GCC, called xgcc, to do source analyses, which they refer to as meta-compilation [Ashcraft and Engler 2002; Engler et al. 2000; Hallem et al. 2002]. A rule language, called metal, is used to express the necessary analysis annotations in a higher-level language. Rather than annotate the code directly, the metal specifications define finite state automata that guide the analysis engine. 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 [Ashcraft and Engler 2002].

Most of the specialized analysis tools lack completeness (i.e., may result in false negatives), but MOPS specifically aims for ease of specification and com­pleteness of analysis [Chen and Wagner 2002]. Using MOPS, security proper­ties are expressed as finite state automata and programs are represented as pushdown automata. Data flow is not represented, so aliasing and value re­lationships are ignored. However, for many analyses useful bugs can still be found [Chen et al. 2004], and it is often possible to show that many data flow relationships do not exist via other means [Zhang et al. 2002].

In another effort, we use one program analysis tool, CQual [Foster et al. 1999], in an approach to finding LSM hook placement bugs statically [Zhang et al. 2002]. Using GCC analysis to automate CQual annotation, we can then perform a CQual analysis that verifies that all controlled operations are medi­ated by at least one LSM hook. In general, we also want to verify that a con­trolled operation is only run when its required authorization hooks have been checked. CQual provides a type lattice that could be used for defining expected authorizations, although it is conceptually complex to get it right. Further, the context-dependency on the relationship between controlled operations and au­thorization requirements is beyond what CQual can handle.

A Java static analysis tool, called JaB.A [Koved et al. 2002], has been used to collect the actual authorizations on controlled operations for Java. For our purposes, this approach has two shortcomings: (1) it does not analyze the C code of the Linux kernel and (2) it does not provide guidance about whether the authorizations made were correct. On the first point, we have actually defined a translation from C to JaBA analysis concepts [Zhang et al. ], and built a prototype implementation. On the second point, JaBA does provide a context- sensitive control flow graph and a context-sensitive data flow graph that can be leveraged for any analysis. Thus, we will examine use of these graphs in generating an analysis log in Section 5.

Due to the complexity of using these approaches, we found that run-time data collection assisted us in getting accurate data quickly, so that we could ex­plore possible analysis options. From examining the data collected, we have developed a consistency analysis approach that we describe in this paper that enables us to determine whether the appropriate authorization hooks are checked for controlled operations. Ultimately, the approach is indepen­dent of whether we do consistency analysis on data collected at run-time or via a static analysis of the code. In this paper, we examine both means of data collection.

Another related problem is the certification of systems. Historically, the Orange Book [NCSC 1985] was used for guidance in the construction of secure operating systems, but this is now being supplanted by the Common Crite­ria [ITSEC 1998]. 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 [Gutmann 2000] 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 run-time 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.

1. SOLUTION DESCRIPTION

The key insight we leverage in run-time analysis for the LSM framework is that the LSM authorization hook placement is largely correct, such that cases that are inconsistent with respect to 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. While this insight does not guarantee that we find all LSM hook placement bugs (see Section 6), it has enabled us to identify some bugs and has served as a valuable guide for the tool development.

In all of the discussion below, we use the following assumptions. First, we leverage an assumption that the objects in controlled operations are handled in a type safe manner in the 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 set of controlled operations (i.e., the mediation interface). These data types are the ones that correspond to sys­tem call concepts (e.g., files, inodes, sockets, skbuffs, ipc message queues, and so on). Access to kernel data is designed to go through these data structures. While we have not explicitly validated this, we have done a more detailed anal­ysis presented elsewhere [Edwards et al. 2001].

* 1. Authorization Consistency

We first define consistency between a controlled operation and a set of autho­rization requirements.

Definition 1 (Authorization Consistency). The relationship between a con­trolled operation and a set of authorization requirements (i.e., policy opera­tions that are authorized) is cons^s^en^ if whenever the controlled operation is executed authorization hooks associated those authorization requirements are called.

We find that this form of consistency is not absolutely required. Execution of a controlled operation may occur in the context of a different system call, which has different authorizations. Clearly, in this case the authorization re­quirements met will be different.

Thus, it is necessary to be able to define contexts in which consistency is expected. In general, contexts can be arbitrary, but our experience is that three types of contexts matter:⑴ system call;⑵ system call with specific inputs (e.g., flags); and (3) a specific set of controlled operations. In the first case, the authorization is for the system call at large (e.g., fcntl). Such authoriza­tions apply to all the controlled operations in the system call. In the second case, the authorizations depend on some parameter to the system call, usu­ally a flag (e.g., open for read). Thus, some system calls come under one con­text and some under another. In the third case, the appearance of the set of controlled operations, independent of the system call in which they appear re­quires specific authorizations (e.g., the operations associated with accesses to the set owner fields). In these cases, the consistency ignores the system call information.

Definition 2 (Execution Context). An execution context defines a set of ex­ecution paths. An execution context can be defined by (1) a system call (all executions); (2) a system call with particular argument values (or ranges of values); and (3) a set of controlled operations (all paths that include them). Other context definitions are possible.

Our solution must support the description of contexts where we expect con­sistent authorization. Typically, context-sensitive data flow in static analy­sis refers to distinguishing between different inputs to the same function. In this case, context sensitivity is much narrower (depends only on the sys­tem call) or may ignore large parts of the remaining context (for the con­trolled operation set). Such analyses require significant amounts of annotation for a static analyzer and may depend on variables outside the understand­ing of the static analyzer (e.g., user-supplied flags). For example, JaBA com­pletely ignores the values of primitive types, but clearly those can influence analysis.

* 1. Authorization Consistency Levels

An execution context usually consists of many controlled operations, so it is helpful to aggregate controlled operations that are consistent in the same way.

Table I. Authorization Consistency Levels: Names and Effects on Authorizations

|  |  |
| --- | --- |
| Level | Authorizations |
| System call | All controlled operations in system call |
| Syscall inputs | All controlled operations in same system call with same inputs |
| Data type | All controlled operations on objects of the same data type |
| Object | All controlled operations on the same object |
| Member | All controlled operations on same data type, accessing same member, with same operation |
| Function | All same member controlled operations in same function |
| Intrafunction | Same controlled operation instance |
| Path | Same execution path to same controlled operation instance |

For example, if all the controlled operations in a context have the same autho­rizations, then we can view consistency relative to the context at large rather than the individual operations.

We find that we can describe the consistency between each controlled op­eration and the authorization hooks that are called when it is executed in a particular context by a set of discrete values we call consistency levels. Further, the consistency levels form a total-order as follows.

Definition 3 (Consistency Level Total Order). If two different controlled op­erations are authorization-consistent for the same value oflevel i, then they are authorization consistent for any value of level j where i > j in the consistency level total order (see Table I).

If two different controlled operations are executed on the same object, but they have consistent authorizations, then the values of the member and access for those operations do not affect the consistency. For example, if all controlled operations on a particular object have the same authorization requirements, then it does not matter what the member access is. Table I lists the discrete consistency levels. We refer this group of levels collectively as the authoriza­tion consistency levels. These levels include various aspects of a controlled op- erationJs execution, including the context under which it was executed (system call, system call inputs, function, location in function, path to controlled opera­tion), the object it was executed upon (data type and object), and the operation performed (member and access).

The consistency levels aggregate controlled operations into a consistency class where all the controlled operations have the same authorization hooks called given the current placement.

Definition 4 (Consistency Classes). Two different controlled operations be­long to the same consistency class for an execution context, if they have the same authorization hooks called every time they are executed in that context.

* 1. Authorization Consistency Impact

The classification of controlled operations by their authorization consistency divides the controlled operations into two categories:⑴ known anomalies

and (2) consistency classes whose authorization requirements need verifica­tion. In the first case, we consider some of the authorization consistency lev­els to be illegal. We define invariants below for these cases. In the second case, we must determine whether the maximal consistency level for each con­trolled operation in an execution context indicates acceptable authorization requirements or not. For example, if a controlled operation belongs to a group of controlled operations at an object consistency level, this indicates that all the controlled operations on the object have the same authorizations checked. It is then a manual task to determine if this is correct. However, the num­ber of consistency aggregates indicates a partition the controlled operations into maximal-sized classes with the same authorizations. These classes en­able verification of authorization requirements and identification of anomalous classifications.

* + 1. Anomalies. The consistency of authorizations to the levels below the double line in Table I, intrafunction and path, are always considered to be anomalous. Sensitivities of these types mean that the execution path (path) or location within a function (intrafunction) determines the authorization re­quirements of a particular controlled operation on the same member.

The following invariant formally expresses our path inconsistency invariant.

Definition 5 (Path Inconsistency Invariant).

Vc1，c2 e C, e1,e2 e E, (c1 = c2)八（e1 = e2) — ^(c1, e1) = E(c2, e2) (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 operationJs authorization requirements.

Similarly, we define an invariant for intrafunction inconsistency.

Definition 6 (Intrafunction Inconsistency Invariant).

Vc1,c2 e C, e1,e2 e E, (F(d) = F(c2))八

(M(c1) = M(c2))八（e1 = e2) — K(c1,e1) = E(c2,e2) (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.

* + 1. Authorization Consistency Classes. For the other cases, we cannot easily identify them as errors. Instead, we partition the controlled operations into their authorization consistency classes and determine whether their au­thorization requirements are correct.

The authorization consistency class computation is as follows. For each con­sistency level starting at the highest (system call), we partition the controlled operations into consistency classes where all controlled operations have the same value for the consistency level, then we test whether the class also has the same authorizations. If not, then we try the next lower level and parti­tion based on both levels and test again. This approach repeats until we have assigned every controlled operation to a consistency class.

Classifications are defined by consistency levels. For the system call level, 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 3.4). For the data type level, the controlled operations are classified by the system call, inputs, and data type of the operation’s object. Thus, successively finer partitions are created in each step of the analysis.

A classification succeeds (i.e., is x-consistent where x is the level) if it is the first level in which all the controlled operations in that class have the same authorizations. Note that other classes at the same consistency 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 are 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 data-type consistent 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’s controlled operation is not authorized. It is on a different object, so inode operations may be object-consistent. 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.[[1]](#footnote-1)

Other than, when 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\_〇wner field (see Section 4.2.4).

* 1. Necessary Data Collection

By logging system call entry/exits/arguments, function entry/exits, controlled operations (i.e., object, data type, member, and operation), and authorizations, we collect all the necessary values for the consistency levels. 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 data type we define a specific approach for computing their

Table II. Log Record Types

|  |  |
| --- | --- |
| Record Type | Data |
| Controlled operation Authorization Function entry Function exit | Context ID Controlled operation ID OID Context ID Authorization ID OID Context ID Instruction address Context ID |

object identifiers. These identifiers are used for determining all operations and authorizations on an object.

Across system call instances, 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 (since the variable would be the same in the two controlled operations). 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.

1. Actually, this object should also be authorized by the read LSM hook, so we add it to the set of objects authorized by this hook. [↑](#footnote-ref-1)