Using CQUAL for Static Analysis of Authorization Hook Placement

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

June 10, 2002

Abstract

The Linux Security Modules (LSM) framework is a set of authorization hooks for implementing flexible access control in the Linux kernel. While much effort has been devoted to defining the module interfaces, little attention has been paid to verifying the correctness of hook placement. This paper presents a novel approach to the verification of LSM authorization hook placement using CQUAL, a type-based static analysis tool. With a simple CQUAL lattice configuration and some GCC-based analyses, we are able to verify complete mediation of operations on key kernel data structures. Our results reveal some potential security vulnerabilities of the current LSM framework, one of which we demonstrate to be exploitable. Our experiences demonstrate that combinations of conceptually simple tools can be used to perform fairly complex analyses.

1 Introduction

Linux Security Modules (LSM) is a framework for implementing flexible access control in the Linux kernel [3]. LSM consists of a set of generic authorization hooks that are inserted into the kernel source that enable kernel modules to enforce system access control policy for the kernel. Thus, the Linux kernel is not hard-coded with a single access control policy. Module writers can define different access control policies, and the community can choose the policies that are most effective for their goals.

The code segment in Figure 1 shows an example of how LSM hooks are inserted in the kernel. The function `sys_lseek()` implements the system call `lseek`.

```
/* Code from fs/read_write.c */
sys_lseek(unsigned int fd, ...)
{
  struct file * file;
  ...
  file = fget(fd);
  retval = security_ops->file_ops
           ->llseek(file);
  if (retval) {
    /* failed check, exit */
    goto bad;
  }
  /* passed check, perform operation */
  retval = llseek(file, ...);
  ...
}
```

Figure 1: An example of LSM hook.

The security hook, `security_ops->file_ops->llseek(file)`, is inserted before the actual work (call to function `llseek()`) takes place.

System administrators can provide an implementation of the corresponding hook functions (e.g. `security_ops->file_ops->llseek()`) by selecting a kernel module that implements their desired policy. Examples of LSM modules under development include SubDomain [4], Security-enhanced Linux [13], and OpenWALL.

While much effort has been devoted to placing hooks in the kernel, this has been a manual process, so it is subject to errors. Even though the LSM developers are highly-skilled kernel programmers, errors are unavoidable when dealing with complicated software. Thus far, little work has been done to verify that the hooks indeed provide complete mediation over access to security-

sensitive kernel objects and enforce the desired authorization requirements. Such verification would help gain acceptance for the LSM approach and enable maintenance of the authorization hooks as the kernel evolves. The verification task for LSM is not a simple one because LSM authorization hooks are embedded within the kernel source, rather than at a well-defined interface like the system call boundary. While this improves both performance and security, it makes it impractical to verify the hook placements manually [6].

As a first step, we began the development of runtime analysis tools for verifying LSM authorization hook placement [6]. These tools are easy to run, have helped us identify the requirements of a verification system, and have enabled us to find some hook placement errors. However, runtime analysis is limited by the coverage of its benchmarks and requires some manual investigation of results to verify errors. Given the recent spate of efforts in static analysis tools [7, 11, 14], we were curious whether any of these tools could be applied effectively to authorization hook verification. Given a brief evaluation of tools, we chose to use CQUAL [9], a type-based static analysis tool. It was chosen mainly because it was conceptually simple (type-based and flow-insensitive), available to use without significant modification, and was supported by formal foundations.

This paper presents a novel approach to the verification of LSM authorization hook placement using CQUAL. We have found that with a simple CQUAL lattice and some additional analyses using GCC we can verify complete mediation of operations on key kernel data structures. *Complete mediation* means that an LSM authorization occurs before any controlled operation is executed. Further, we have found that using the authorization requirements found by our runtime analysis tools, we can build a manageable lattice that enables verification of complete authorization. *Complete authorization* means that each controlled operation is completely mediated by hooks that enforce its required authorizations. Our results reveal some potential security vulnerabilities of the current LSM framework, one of which we demonstrate to be exploitable. The findings and a code patch were posted to the LSM mailing list [5], and the fix was incorporated in later kernel releases. The resultant contribution is that through the use of a small number of conceptually simple tools, we can perform a fairly complex analysis.

The rest of the paper is organized as follows. Section 2 defines the verification problem. Section 3 describes our approach in detail. Section 4 presents the potential vulnerabilities discovered through our static analysis. Section 5 discusses effectiveness of our approach and possible extensions to CQUAL. Section 6 describes related work, and Section 7 concludes the paper.

2 Problem

We aim to enable two kinds of verification: (1) verification of complete mediation and (2) verification of complete authorization.

2.1 Complete Mediation

For complete mediation, we must verify that each controlled operation in the Linux kernel is mediated by some LSM authorization hook. A *controlled operation* consists of an object to which we want to control access, the *controlled object*, and an operation that we execute upon that object. An LSM authorization hook consists of a hook function identifier (i.e., the policy-level operation for which authorization is checked, such as `security_ops->file_ops->permission`) and a set of arguments to the LSM module's hook function. At least one of these arguments refers to a controlled object for which access is permitted by successful authorization (sometimes these objects are referred to indirectly).

The first problem is to find the controlled objects in the Linux kernel. In general, there are a large number of kernel objects to which access must be controlled in order to ensure the system behaves properly. Based on the background work done for the runtime analysis tool [6], we have found that effective mediation of access to kernel objects is provided through user-level abstractions identified by particular controlled data types and global variables. Operations on these objects define a mediation interface to the kernel objects at large. Of course, there may be a bug that enables circumvention of this interface, but this is a separate verification problem beyond the scope of this paper.

We identify the following data types as *controlled data types*: files, inodes, superblocks, tasks, modules, network devices, sockets, skbuffs, IPC messages, IPC message queue, semaphores, and shared memory. Therefore, operations on objects of these data types and user-level globals compose our set of controlled operations. In this paper, we focus on the verification of controlled operations on controlled data types only. Now we

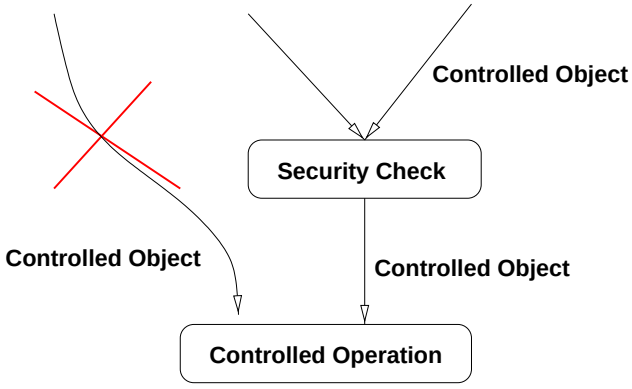

Figure 2: The complete mediation problem.

can define our complete mediation verification problem: *verify that an LSM authorization hook is executed on an object of a controlled data type before it is used in any controlled operation.* For example, because the variable `file` in Figure 1's function `sys_lseek` is of a controlled data type, any operations on this variable must be preceded by a security check on `file`. Figure 2 shows the problem graphically.

In order to solve the complete mediation verification problem, there are a few important subproblems to solve. First, we must be able to associate the authorized object with those used in controlled operations. In a runtime analysis, this is easily done by using the identifiers of the actual objects used in the security checks and controlled operations. In a static analysis, we only know about the variables and the operations performed upon them. Simply following the variable's paths is insufficient because the variable may be reassigned to a new object after the check.

Next, we need to identify all the possible paths to the controlled operation. While the kernel source can take basically arbitrary paths, in practice typical C function call semantics are used. Thus, we assume that each controlled operation belongs to a function and can only be accessed by executing that function.

Thus, all inter-procedural paths are defined by a call graph, but we must also identify which intra-procedural paths require analysis. Note that the only intra-procedural paths that require analysis are those where authorization is performed or those where the variable is (re-)assigned. These are the only operations that can change the authorization status of a variable. Since variables to controlled objects are typically assigned in the functions where their use is authorized and are rarely reassigned, this often limits our intra-procedural analysis to the functions containing the security checks. Further, security checks should be unconditional with respect to the scope for which the check applies, so such analyses should be straightforward.

Thus, we envision that the complete mediation problem will be solved by following this sequence of steps for each controlled object variable:

1. Determine the function in which this variable is initialized (*initializing function*).
2. Identify its controlled operations and their functions (*controlling functions*).
3. Determine the function in which this variable is authorized (*authorizing function*).
4. Verify that all controlled operations in an authorizing function are performed after the security check.
5. Verify that there is no re-assignment of the variable after the security check.
6. Determine the inter-procedural paths between the initializing function and the controlling functions.
7. Verify that all inter-procedural paths from an initializing function to a controlling function contain a security check.

If a re-assignment is found in step #5, then the verification is restarted from the location of the new assignment.

2.2 Complete Authorization

Given a solution to complete mediation, the problem of verifying complete authorization is straightforward, but finding the requirements is difficult. Each controlled operation requires prior mediation for a set of authorization requirements. The verification problem is to ensure that those requirements have been satisfied for all paths to that controlled operation. In this case, multiple security checks may be required (and thus, multiple authorizing functions), but the overall mechanism is basically the same. We need to ensure that the set of authorizing functions that provide the necessary security checks must occur between the initializing function and the controlling function.

Collection of the authorization requirements for the controlled operations is the more complex task. Our runtime

analysis tool [6] enables determination of the authorization requirements of controlled operations, so rather than developing a new analysis tool, we use our runtime results to find the authorization requirements.

2.3 Summary

When we first examined this problem, it appeared that an extensive static analysis tool with inter-procedural dataflow analysis capability was needed. Such tools either are not available to the public, do not work on Linux kernel (due to scalability issues or C coding style issues), or are too complicated to customize for our problem. A closer look at the nature of the verification problem, however, reveals that a less-powerful static analysis tool might be sufficient. For verification purposes, we do not care about the exact value of the controlled object. We only care about its authorization state (i.e., authorized or non-authorized) and that its variable is not re-assigned. Some limited source analysis may be necessary to verify that the expected conditions apply, but this should be quite simple in most cases.

3 Approach

3.1 CQUAL Background

CQUAL is a type-based static analysis tool that assists programmers in searching for bugs in C programs. CQUAL supports user-defined *type qualifiers* which are used in the same way as the standard C type qualifiers such as `const`.

The following code segment shows an example of a user-defined type qualifier: `unchecked`. We use this qualifier to denote a controlled object that has not been authorized. This declaration states that the file object (`filp`) has not been checked.

```
struct file * $unchecked filp;
```

Typically, programmers specify a type qualifier *lattice* which defines the sub-type relationships between qualifiers and annotate the program with the appropriate type qualifiers. A lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound. For example, Figure 3 shows a

```
partial order {
  $checked < $unchecked
}
```

Figure 3: A lattice of type qualifiers.

lattice with two elements, `checked` and `unchecked`, and the subtype relation $<$ as the partial order. Here it means `checked` is a subtype of `unchecked`.

CQUAL has a few built-in inference rules that extend the subtype relation to qualified types. For example, one of the rules states that if `Q1 < Q2` (meaning qualifier `Q1` is a subtype of qualifier `Q2`) then type `Q1 T` is a subtype of `Q2 T` for any given type `T`. Replacing `Q1` and `Q2` with `checked` and `unchecked` respectively, we have that `checked T` is a subtype of `unchecked T`. From an object-oriented programming point of view, this means that a `checked` type can be used wherever an `unchecked` type is expected, but using an `unchecked` type where a `checked` type is expected results in a type violation. The following code segment shows a violation of the type hierarchy. 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);
  ...
}
```

Using the extended inference rules, CQUAL performs *qualifier inference* to detect violations against the type relations defined by the lattice. For a more detailed description of CQUAL, please refer to the original paper on CQUAL [9].

3.2 Approach

CQUAL is employed to perform the central task of statically verifying that all inter-procedural paths from any initializing function to any controlling function, contain an authorization of the controlled object (steps 6 and 7 from Section 2). This is achieved using the lattice configuration shown in Figure 3. Figure 4 shows a graph-

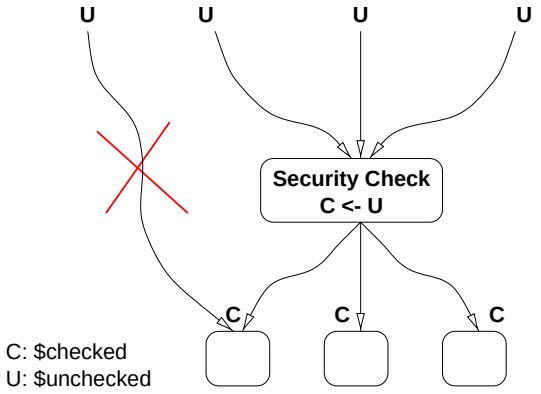

Figure 4: Detecting Security Violations via Type Inferencing.

ical depiction of our approach. All controlled objects are initialized with an `unchecked` qualifier. The parameters to controlling functions that are used in controlled operations are specified as requiring `checked` qualified objects (as `func_a` was above). Authorizations change the qualified type of the object they authorize to `checked`. Using these qualifiers, CQUAL's type inference and analysis will report a type violation if there is any path from an initializing function (where the object is `unchecked`) to a controlling function (where the object must be `checked`) that does not contain an authorization (a cast from `unchecked` to `checked`).

There are three requirements for this solution (equivalent to steps 1, 2, and 3, in the previous section):

1. All controlled objects 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`.

If the number of controlled objects and controlling functions was small, we could manually annotate the source (as was done by Wagner et. al. to detect format string vulnerabilities using CQUAL [14]). Unfortunately, both are far too numerous for manual specification to be feasible. Therefore, we use a modified version of GCC and a set of PERL scripts to automate this process.

In the following subsections we detail our approach to each of the seven steps outlined in the previous section.

3.2.1 Step 1: Initializing Controlled Objects to Unchecked

We locate the origin (i.e., declaration) of all controlled objects and qualify them as `unchecked`. There are three different kinds of variables that a function can access: global variables, local variables, and parameters. Currently we do not consider global variables, which account for less than 2% of controlled objects.

All locally declared variables of a controlled type are qualified as `unchecked`. A special case of this is when reference to a structure member of a controlled data type is passed as a parameter to a function (e.g. `f(dentry->d_inode)`, where field `d_inode` is of controlled type). It should also be qualified as `unchecked`, because it is equivalent to declaring a local variable, initializing it to be a reference to the structure member, and then passing the variable to the function. To qualify such cases, we explicitly cast the parameter to `unchecked` at the function call (e.g. `f((struct inode * $unchecked)dentry->d_inode)`).

The task of marking local variables of controlled types is automated using two tools: one for controlled local variables and one for the passing of structure member references to functions. First, we modified GCC to output the location (file and line number) of any local variable declaration with a controlled type. To achieve this, we inserted code that traverses the abstract syntax tree (AST) for each function as it is compiled. The code scans the AST for local declarations (`VAR_DECL` nodes) and prints the location details if the type (`TREE_TYPE`) of the declaration is a controlled type (independent of the level of indirection). In the case of structure member references, our GCC code scans the AST for function calls (`CALL_EXPR` nodes). If any parameter is a reference to structure member (`COMPONENT_REF` node, see Section 3.2.2 for more discussion), and the type of the referenced field is one of the controlled types, then GCC prints out detailed location and type information about the parameter. Next, this information is input to a PERL script that inserts appropriate annotations into the source code.

For parameters in function declarations, we leave their types unqualified. CQUAL then automatically infers their type during the analysis process. There are a few exceptions to this rule, where we manually annotate function prototypes (in two header files) that we know expect `checked` type parameters.

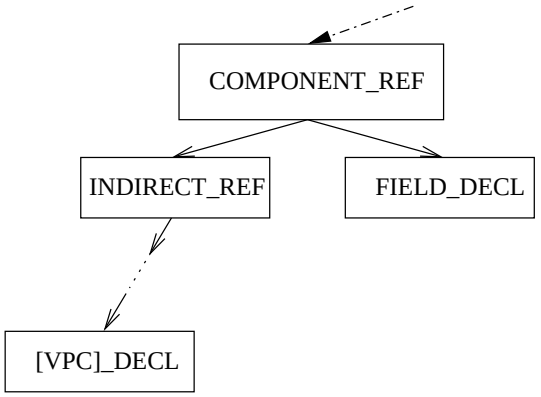

Figure 5: Detecting Controlled Operations in the AST

3.2.2 Step 2: Annotating Checked Parameters

Controlled operations occur whenever a member of a controlled type is read or written (all controlled data types are structures). Controlled operations must only be performed on `checked` objects. With current version of CQUAL, we cannot specify type requirements for variables at individual statement level, instead, we specify type requirements on any function parameters that are used in controlled operations within that function. This analysis verifies complete mediation in the inter-procedural case (i.e., where the controlling function is different from the authorizing function) but, it cannot verify complete mediation for controlled operations within an authorizing function. Our approach to intra-procedural analysis is described in step 4 below.

To automate the annotation process, we again added code to GCC to output the details of controlled operations, and then input this information into a series of PERL scripts. These scripts aggregate the controlled operations to the function parameters, and add `checked` qualifiers to those parameter declarations. The type inference engine then propagates this up the call graph, raising an error if an `unchecked` local variable is passed to a `checked` parameter.

Figure 5 shows the subgraph structure that our analysis searches for in the AST. Access to structure members is represented in the AST by `COMPONENT_REF` nodes. These nodes have two children, the first is an expression which specifies the variable being accessed, and the second is a `FIELD_DECL` node which specifies which field is being accessed. The expression that specifies the variable being accessed is a chain of `INDIRECT_REF` and `ADDR_EXPR` nodes corresponding to the C dereference (*) and address (&) operators, respectively. At the end of this chain is either a `VAR_DECL` corresponding to a local variable, a `PARM_DECL` corresponding to a parameter, or a `COMPONENT_REF` if we are accessing a member of a structure embedded in another structure.

Our analysis searches for `COMPONENT_REF` nodes in the AST. When one is found, it determines the type of the structure being accessed (the left subgraph in Figure 5). If this is a controlled type, then the expressions is accessing a member of a controlled type, and the location information (file, function, and line number) is reported. We also output whether this operation is on a local variable (`VAR_DECL`) or a parameter (`PARM_DECL`).

This information is then input to a series of PERL scripts. These scripts scan the GCC output for controlled operations on parameters (i.e., those that contain `PARM_DECL` nodes). Using the location information provided by GCC, they find the function declaration, and annotate the parameter with the `checked` qualifier.

3.2.3 Step 3: Authorizations

In theory, once an authorization is performed on a controlled object, its qualified type is changed from `unchecked` to `checked`. However, the current version of CQUAL we use is flow-insensitive, i.e., the qualifier type of a variable remains the same throughout its scope (e.g., the scope of a local variable is its defining block, typically the function). To get around this limitation, following an authorization, we declare a new, `checked` qualified variable with the same base type as the object authorized. All uses of the original controlled variable following the authorization are replaced by the new variable. This process is automated using a PERL script that replaces uses of the original variable via simple pattern matching.

The simple approach of replacing all uses of the variable on source lines following the authorization makes two assumptions about the function's control-flow graph that must be verified. Firstly, that there are no back-edges from below the authorization to above it. This ensures that the authorization is not inside a loop and that there are no `goto` statements below the authorization that jump to above the authorization. Secondly, that there is no control-flow path from above to below the authorization that does not execute the authorization. This ensures that the authorization is not inside a conditional or switch statement.

These assumptions are verified by adding code to GCC to build the function's control-flow graph from its register transfer language (RTL) description. Once the graph is created, the two properties described above are verified. While the vast majority of authorizations possess these properties, exceptions do exist. Fortunately, the number of exceptions is small enough that they can be handled manually.

3.2.4 Step 4: Verifying Controlled Operations Within Authorizing Functions

The analysis so far verifies mediation in the inter-procedural case, but, it does not verify intra-procedural mediation. Intra-procedural analysis is required to verify that controlled operations within an authorizing function occur after the authorization.

Our approach in step 3 makes this analysis simple. In step 3 we replaced all uses of the controlled object (co) following the authorization with a new variable (co'). An intra-procedural control-flow analysis verified the validity of this replacement. The intra-procedural analysis reduces to finding all controlled operations within the function that operate on local variables (parameters are handled by the inter-procedural analysis). If the local variable is an introduced variable (co') then it is mediated, otherwise a warning is generated.

3.2.5 Step 5: Verifying Assignments to Checked Objects

As described in Section 2, complete mediation requires verification that a variable is not re-assigned between an authorization and a controlled operation. From the CQUAL perspective, the right hand side (RHS) of an assignment takes one of four forms:

1. An `unchecked` object.
2. A `checked` object.
3. A structure member (e.g. `dentry->d_inode`).
4. An explicit type cast (e.g. `(struct inode*)0xc2000000`). Since explicit casts in the Linux source obviously don't include our qualifiers, CQUAL treats them as unqualified.

CQUAL correctly handles the first two cases, as the objects are qualified. If the left hand side (LHS) of the assignment is `checked` then CQUAL will raise a type violation for the first case and allow the second case.

In the third case, however, the structure member has no type qualifiers to cause type violations. With no other information, CQUAL will therefore infer that the RHS has the same qualified type as the LHS, and report no errors. As an example of how this can produce false-negatives, consider the code fragment below.

```
void func_a(struct inode * $checked
  inode);

void func_b(struct inode * $checked
  inode)
{
  ...
  inode = dentry->d inode;
  ...
  func_a(inode);
}
```

The variable `inode` in `func_b` has already passed security check since it has a `checked` qualifier. However, it is assigned a value `dentry->d_inode`, before being passed to `func_a` which expects a `checked` inode. Clearly we would like CQUAL to raise a type violation, since `dentry->d_inode` is not an authorized variable. However, according to CQUAL inference rule, CQUAL will infer that `dentry->inode` is `checked` and allow the function call.

The solution is to treat `dentry->d_inode` as an unauthorized local variable by typecasting it to `unchecked`. At present we have not implemented the interim solution and so this source of false-negatives remains in our results.

The fourth case fails to report type violations for the same reason. Explicit casts in the Linux kernel do not include our type qualifiers, therefore, CQUAL infers their type. To address this problem, we wrote a PERL script that scans the source for explicit casts, and inserts the `unchecked` qualifier. Any assignment of such an expression to a `checked` variable or parameter will result in a type violation.

3.2.6 Steps 6 and 7: Determining and Verifying All Inter-procedural Code Paths

CQUAL performs interprocedural inferencing to verify that between an initializing function and the controlling

function, there exists a security check. The controlled object variable has an `unchecked` qualifier when it's defined in the initializing function. When the initializing function calls other functions passing the controlled variable as a parameter, the `unchecked` qualifier is propagated down the calling chain, until the authorizing function is reached, at which point, a new `checked` variable is defined and used after the security check (Step 4 in Section 2). When the authorizing function calls other functions passed the new `checked` variable, the `checked` qualifier is again propagated along the calling chain, until it reaches the controlling function. If a controlling function is reached without passing through an authorizing function, then an error will be raised, because the variable will have an `unchecked` type and the controlling function expects a `checked` type.

3.3 Complete Authorization

Verification of complete authorization is basically carried out in the same way as complete mediation, with slight modification to the lattice structure based on the authorization requirement information. Rather than having a generic `checked` type qualifier for all security checks, we assign a type qualifier for each unique security check. A controlled operation that requires multiple security checks will then have a type qualifier that is a subclass of the corresponding type qualifiers of the checks required. For instance, if a system contains two security checks, denoted by $C1$ and $C2$ respectively, assuming that the controlling function `f(file)` requires both security checks to be performed on the `file` object, then the type qualifier lattice should be:

```
partial order {
  $checkedForC1C2 < $checkedForC1
  $checkedForC1C2 < $checkedForC2
  $checkedForC1   < $unchecked
  $checkedForC2   < $unchecked
}
```

Figure 6 shows the graphic representation of the lattice. Function `f` should expect the parameter to be of type `checkedForC1C2`.

Figure 7 gives an example of a controlled operation requiring multiple authorizations identified by the runtime analysis tool [6]. Three security checks are necessary for the controlled operation `unlink()` on a directory inode, namely, permission to traverse the inode, permission to write the inode, and permission to unlink file

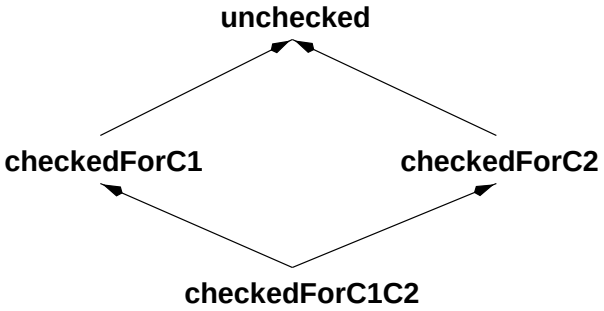

Figure 6: A four-node type qualifier lattice.

in the directory. In the function prototype definition of `unlink()`, we specify the authorization requirement `checkedforExecWriteDirunlink`. After the security checks, a new variable `Cdir` that possesses the right authorization requirements replaces the old variable `dir`, and is passed to the controlling function.

4 Results

We ran the experiments on Linux version 2.4.9 with the September 4th, 2001 LSM patch. We used GCC version 3.0.2 and CQUAL version 0.9 for our static analysis.

We analyzed four subsystems of Linux: the file system (including ext2 physical file system), virtual memory management, networking, and IPC. The analysis generated 524 *type errors* (CQUAL inference conflicts). Below we give a detailed analysis of the type errors and discuss techniques in coping with false positives.

4.1 Type Error Categorization

We categorize the unique type errors into three groups that we examine below.

4.1.1 Category 1: Inconsistent Checking and Usage of Controlled Object Variables

In this category, the variable that is checked is not the variable that is used subsequently. There is, however, some sort of mapping between the checked variable and the used variable (e.g. the used variable is a field of the checked variable). Therefore, it is easy to obtain the checked variable from the passed variable and vice versa.

```
/* inserted by our tool */
struct inode *
  $checkedForExecWriteDirunlink Cdir;

/* code from include/linux/fs.h */
struct inode_operations {
  ...
  int (*unlink) (struct inode *
    $checkedForExecWriteDirunlink,
    struct dentry *);
  ...
}

/* code from fs/namei.c */
int vfs_unlink(struct inode *dir,
  struct dentry *dentry)
{
  ...
  /* check for EXEC and WRITE */
  may_delete(dir, dentry, 0);
  ...
  /* check for UNLINK */
  security_ops->inode_ops
    ->unlink(dir, dentry);
  ...
  /* controlled operation */
  dir->i_op->unlink(Cdir, dentry);
  ...
}
```

Figure 7: An example of controlled operation requiring multiple authorizations. Note that error checking code is removed to make the code easier to follow.

These type errors are subject to TOCTTOU [2] attacks, because the mapping between the checked variable and the used variable might change during the course of execution. Whether the vulnerability is exploitable depends on whether the user can manipulate the mapping without special privilege. At least one of the type errors that we found is exploitable, as we demonstrate below.

Figure 8 shows the code path that contains the type error. The code sequence shows Linux implementation of file locking via the `fcntl` system call. In function `sys_fcntl()`, the variable `filp`, which is a pointer to the `file` structure and is retrieved via the file descriptor `fd`, is checked by the `security_ops->file_ops->fcntl(filp,...)` hook. However, after the check, the file descriptor `fd`, instead of the checked variable `filp`, is passed to the intermediate function `do_fcntl(fd,...)` and eventually to the worker function `fcntl_getlk(fd,...)` , where the `filp` is retrieved *again* with the given `fd`.

This double retrieval of the `file` pointer creates a race condition and can be exploited as follows. A user can have the `security_ops->file_ops->fcntl(filp)` authorization performed on a different file to the one that is eventually locked. Figure 9 shows the exploit.

Note that although step (7) is written as a whole system call, there is actually only one line of C (an assignment) in step (7) that needs to come between (6) and (8). Since step (6) does a `get_user`, the attacker can cause their own program to page fault which enables step (7) to be performed before (8).

Also note that non-LSM Linux is not vulnerable since the validation in `fcntl_setlk` is done after the second lookup. LSM is vulnerable because the only authorization that protects the operation is performed before the second lookup.

As an example of how dangerous this can be, `login` and `su` (PAM'd versions) both try to lock the file `/var/run/utmp` (world readable). `insmod` locks any modules it loads.

A patch that fixes this problem was posted to the LSM mailing list [5].

The remaining type errors in this category involve kernel data structures that cannot be easily modified by users via system calls. As a result, it is unclear whether these type errors can lead to exploits. However, it certainly complicates the code unnecessarily and increases the

```
/* 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 8: Code path from Linux 2.4.9 containing an exploitable type error.

```
THREAD-A:
(1) fd1 = open("myfile", O_RDWR);
(2) fd2 = open("target_file", O_RDONLY);
(3) fcntl(fd1, F_SETLK, F_WRLOCK);

      KERNEL-A (do_fcntl):
      (4) filp = fget(fd1);
      (5) security_ops->file_ops
            ->fcntl (fd1);
      (6) fcntl_setlk(fd1,cmd)

THREAD-B:
/* this closes fd1, dups fd2,
 * and assigns it to fd1.
 */
(7) dup2( fd2, fd1 );

      KERNEL-A (fcntl_setlk)
      /* this filp is for the target
       * file due to (7).
       */
      (8) filp = fget (fd1)
      (9) lock file
```

Figure 9: An example exploit.

chance of race conditions when the data structures are not properly synchronized, which may result in potential exploits.

Here we present a type error of this kind. Many security checks that intend to protect the inode structure are performed on the dentry data structure. For example, the following code does the permission check on the `dentry` structure, but does the "set attribute" operation on the `inode` structure.

```
/* from fs/attr.c */
...
security_ops->inode_ops
  ->setattr(dentry, attr);
...
inode = dentry->d_inode;
inode_setattr(inode, attr);
...
```

It is also quite common in Linux to check on the `file` data structure and operate on the `inode` data structure.

```
/* from mm/filemap.c */
struct page * filemap_nopage(
  struct vm_area_struct * area, ...)
{
   struct file * $unchecked file
     = area->vm_file;
   ...
   page_cache_read(file, ...);
   ...
}

static inline int page_cache_read(
  struct file * file, ...)
{
  struct inode * $unchecked inode =
    file->f_dentry->d_inode;
  struct address_space *mapping =
    inode->i_mapping;
  ...
  mapping->a_ops->readpage(file, page);
  ...
}
```

Figure 10: An example of unauthorized access.

4.1.2 Category 2: Controlled Objects Modified Without Security Checks

This category includes functions that modify controlled objects without any security checks. The code segment in Figure 10 shows an example of such cases. The function `filemap_nopage()` is called when a page fault occurs within an m'mapped region. Since there is no check on the `file` object within the function, its type is `unchecked`. It is then passed to function `page_cache_read()`, which in turn calls `mapping->a_ops->readpage()`, which expects a checked `file` object. This code path shows that once a file is mapped into a process address space, the process can access the file even after security attributes of the file have changed.

Since there is an LSM authorization hook to verify read access to a file on each `read` call, this is inconsistent with the current hooks. A discussion with the LSM community revealed that enforcement on each `read` is optional and will only be used for files that are not m'mapped. This hooks, as well as the one for checking access on `write` have been documented to clarify this inconsistency.

In other cases, for example function `iput()`, it seems that checks are not necessary, as the function is used for reference counting. In other cases, such as initialization function `clean_inode()` for the `inode` data structure, there is no need for security protection, as modification of the data structure is restricted to zeroing and initialization of the contents. We call these functions "safe" functions and consider type errors induced by these functions as false positives.

4.1.3 Category 3: Kernel-Initiated Operations Bypassing Security Checks

This category includes operations that are initiated inside the kernel, instead of going through system call interfaces. As such, they do not go through the normal security checks that system calls go through. As the kernel developers have added some limitations on the kernel's use of these commands, it is clear that they are security-sensitive.

One example is the `do_coredump()` function, which creates a core file containing in-memory image of the running process, when certain signals are caught that end the process. A check is done when the core file is created, however, subsequent seeks and writes to the file are performed without security checks. This deviates from the user case, where every `lseek()` or `write()` system call requires a check.

Another example is the kswap daemon. The kswapd daemon calls `prune_icache()`, which tries to sync the inodes that are to be released. The inodes are reached via a global variable `super_blocks`, which contains heads for various inode lists.

4.2 Type Error Rates

CQUAL type errors can be examined in two ways: source type errors and path type errors. A *source type error* is a variable that is used in such a way that a type error is generated. That is, the variable is used in an `unchecked` state in at least one function that expects the variable to be `checked`. A *path type error* is a unique call path that leads to a type error. Figure 11 shows an example path type error. Note that for each source type error there may be multiple path type errors.

Table 1 shows both the source and path type error counts for Linux kernel subsystems. For source type errors, we also display the *source type error rate*, defined to be the percentage of controlled variables that are involved in type errors.

Subsystems	Path Type Error Counts	Source Type Error Counts	Source Type Error Rate (%)
File System	73	57	10%
Memory Management	18	17	9%
Networking	431	308	22%
IPC	2	2	3%

Table 1: Path and source type errors.

Table 1 shows two interesting facts: (1) over 500 path type errors are present in the kernel and (2) most of the type errors occur on one path. Fortunately for the LSM community, most the type errors identified by the analysis are false positives. However, examining this many type errors to find a few exploitable errors is not practical. Therefore, we need secondary analyses to remove obvious false positives. Second, since most types errors associate one source with one error path, so it may be that some of the sinks of the analysis (i.e., the functions with controlled operations) may not really require authorization.

4.3 Reducing False Positives

Given that the tools generated about 500 type errors, one may conclude that the false positive rate is unmanageable, but we do not find this to be the case. A significant number of the errors are in functions in which it is easy to verify that no security compromises are present, such as those caused by "safe" functions described in Section 4.1.2. "Safe" functions are falsely marked as controlling functions because they modify field members of controlled data structures. However, since the modification is for the purpose of reference counting or initialization, the modification does not require security authorizations.

To identify what these functions are, we (slightly) modified CQUAL to print the inferencing path that leads to a type error. Figure 11 shows an example error path involving a "safe" function `iput()`. `iput()` decreases the usage count for the given inode and releases it if the usage count hits zero.

We then report the list of controlling functions that are the sinks of the error paths. Because *hot* controlling functions often contribute to multiple type errors, the number of controlling functions are much smaller than the number of type errors. We then manually go through the list and identify "safe" functions, which are removed from the list of controlling functions. Appendix A lists the "safe" functions we identified. The CQUAL analysis process is then restarted.

```
inode.ii:8383 $unchecked <= inode_p
inode.ii:8387 inode_p    <= iput_arg0
inode.ii:8831 iput_arg0  <= $checked
```

Figure 11: An example error path ending in function iput. Each line represents an inference according to the CQUAL rules, e.g. the first line means that inode_p is a super class of the unchecked qualifier type. The first column shows the source file and line number where the inference occurs.

It is painful to manually identify "safe" functions. But two reasons make it a manageable task. First, there are only a few such functions, even though they accounted for a significant portion of the type errors (Table 2). Secondly, these functions are relatively stable across kernel releases. So with a high probability this task only needs to be done once and the results can be reused in future kernel releases. After the "safe" functions are identified, we only need to verify that they do not change in new kernel releases, or that the changes do not affect their intended functionality.

Table 2 shows the reduction in terms of both path and source type errors after removing the "safe" functions for the four kernel subsystems we tested. This reduces the number of type errors by around 75% for both path and source type errors.

While this is a significant improvement, other means for removing false positives are being examined. First, there may be a significant number of other "safe" functions. Second, there are several cases where a variable is assigned from another variable that is `checked`. In the file system, often the `dentry` is authorized, then the `inode` is assigned from the `dentry->d_inode`. Unfortunately, CQUAL cannot yet reason that a field extracted from a `checked` structure is also `checked` (see Section 5.2). Third, we have not yet fully examined kernel-initiated paths that lead to type errors.

Subsystems	Path Type Errors			Source Type Errors		
	With "Safe" Functions	Without "Safe" Functions	% Reduction	With "Safe" Functions	Without "Safe" Functions	% Reduction
File System	73	37	49%	57	31	45%
Memory Management	18	14	22%	17	13	24%
Networking	431	73	83%	308	55	82%
IPC	2	2	0%	2	2	0%

Table 2: Error reduction after eliminating "safe" functions.

5 Discussion

Here we examine the effectiveness of our approach and a possible extension to CQUAL that may improve its utility.

5.1 Effectiveness of Our Approach

Given the extensive nature of static analysis, we are somewhat surprised that we have only found a couple of exploitable CQUAL type errors in our analysis. Some of the analyses are fairly new, so we may find more errors, but this is a bit of a surprise.

We are encouraged by one of the exploits that we did find. The Category 1 TOCTTOU exploit is one that would be difficult to find via runtime analysis. Typically, the association between the file descriptor and the file would not change, so benchmarks consisting of benign programs would not uncover this error. With static analysis, the inconsistency was clear.

Another aspect of the effectiveness of our approach is its ease of use, since most of the analysis process is automated. It is straightforward to apply the process to a modified kernel or new releases of the kernel. We tested this by running the tool against Linux version 2.4.18. After the kernel source tree is downloaded, and a few small changes are applied to the Makefile and two source files (see Section 3.2.1), the rest of the process requires little manual effort (except for identifying false positives). The time it takes to complete the process is also quite reasonable. As a matter of fact, most of the time is spent on kernel builds - our modified version of GCC collects information on controlled types while compiling the source code.

Here we present the times for the major steps. These numbers are only intended for a ballpark measure of the effort needed to perform analysis, so they should not be interpreted as representing the optimized performance of the tools. The test platform was a 667 MHz Pentium III machine with 128MB of memory. It took about 30 minutes to do the three clean kernel builds using our extended GCC to generate the annotation information. It should be possible to perform all this analysis in one kernel build, however. Most of that time is contributed by the GCC backend that generates machine code (whereas our GCC analysis code only works on the AST tree). We compared normal kernel build time with the build time that has our GCC analysis code enabled, and the difference is negligible. Annotation of the source by the Perl scripts took about 1 minute, And finally, it took about 10 minutes for CQUAL to perform the analysis. With the additional analysis overhead of a 15 minutes or less, we expect that an optimized process can be done sufficiently quickly for these tools to be useful for kernel programmers.

5.2 Possible CQUAL Extension

A possible extension to CQUAL would enable us to correctly verify mediation between the controlled operations and all security-sensitive operations. The CQUAL team has an interim solution and are looking into a general solution [8]. We describe the problem here.

Currently, structures in CQUAL are treated as a collection of fields, so there is no relationship between a structure and its member fields. For example, in the code below, `var->bar` would not have type `checked` even though `var` does. Since structures are used extensively in the kernel, we believe it would greatly enhance the tool if CQUAL supports user-defined rules for inferring the types of member fields from the types of structures.

```
struct foo {
  int bar;
};

$checked struct foo *var;
```

For instance, for case 3 in Section 3.2.5, we would want the inode that is extracted from a `checked` dentry to be `checked` as well. In the case that a dentry is unchecked, the inode of the dentry is implicitly unchecked as well.

In addition, with current version of CQUAL, all instances of a structure type share the same qualifer type. For example, if `bar` is qualified as a `checked` type, all instances of `foo` would have a `checked` field for `bar`. What we want is to assign qualifier types to members on a per-instance basis.

For verifying that the controlled operations mediate the security-sensitive operations, we would also want any structure field accessed through a `checked` type to be `checked` as well. This would enable us to propagate authorizations through the structure completely. Then, we could find any members of a security-sensitive data type that is not accessed through a controlled data type.

Note that this approach is not always applicable depending on the semantics of the qualifications. This would not be appropriate for the type of qualifiers used by Wagner et. al. [14].

6 Related Work

We are unaware of any other research work on static verification of LSM. However, a number of static analysis tools that were successfully applied to the security domain. Here, we compare their work to ours.

Wagner et. al. [14] used CQUAL to identify format string vulnerabilities. Their work motivated us to apply CQUAL to the more complicated problem of LSM verification. The main difference between our usage of CQUAL and theirs lies in the annotation process. In their work, the target code for annotations is well-defined and has a limited number of occurrences. Therefore, the annotations are done by hand. In our case, the scope of annotated code is much larger, and thus we employ GCC to automatically detect the code to be annotated. We automate the process of marking as well.

Engler *et al* enables extension of GCC, called *xgcc*, to do source analyses, which they refer to as *meta-compilation* [7, 1]. 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. While it appears that *xgcc* could be used for the static analysis we perform, *xgcc* is not available at this time, so we are unable to evaluate it. A key difference may be that *metal* rule expressions will have to be extended to reference GCC AST structures rather than the source directly.

Larochelle et. al. [11] enhanced their LCLint tool to detect likely buffer overflows in C programs. The LCLint tool bases static analysis on annotations of the programs (or the libraries) that restrict the range of values a reference can have. The strength of LCLint is that the analysis is flow-sensitive, and thus more accurate. The downside of the LCLint tool is its inflexibility. The current LCLint tool is customized to deal with a set of predefined software bugs. It appears that extending LCLint for LSM verification would require a significant amount of effort (i.e. adding new annotation types). CQUAL, on the other hand, is more extensible by employing user-defined type qualifier lattices.

Necula et. al. [12] define the CCured type system. CCured leverages the fact that most C source is written in a type-safe manner to perform a variety of static checks on the source during compilation for things like buffer overflows. For things that cannot be checked statically, CCured introduces runtime checks into the code. This enables certain kinds of errors to be caught regardless of whether they can be found statically or dynamically. While we agree with this approach to verification, as yet the types of errors that CCured can find do not include authorization hook placement.

Koved et. al. [10] presented a technique for computing the access rights requirements of Java applications. Their approach uses more powerful programming analysis techniques: a context-sensitive interprocedural data flow analysis is employed. Although the analysis is performed on Java code, it is conceivable that such techniques can be applied to our problem domain as well.

7 Conclusion

This paper presented a novel approach to the verification of LSM authorization hook placement using CQUAL, a type-based static analysis tool. With a simple CQUAL lattice configuration and some simple GCC analysis, we were able to verify complete mediation of operations on key kernel data structures. Our results revealed some potential security vulnerabilities in the cur-

rent LSM framework, one of which we demonstrated to be exploitable. We further showed that given authorization requirements, CQUAL could be used to verify complete authorization as well. Our results demonstrate that combinations of conceptually simple tools can be powerful enough to carry out fairly complex analyses.

Our main problem is the elimination of false positives. Static analysis generally errs on the conservative side, so we initially had a large number of type errors. However, we have identified techniques for secondary analyses that can eliminate many of those false positives. Extensions to CQUAL are necessary to eliminate some types of false positives, but this is ongoing work.

8 Acknowledgments

We would like to thank Jeff Foster from UC Berkeley for his timely responses to our numerous questions on CQUAL and for his suggestions and advices on the early draft of this paper. We also thank the anonymous reviewers for their valuable comments.

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] LSM Community. Linux Security Module. Available at http://lsm.immunix.org.

[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. Personal communication, January 2002.

[9] 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.

[10] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, November 2002. Accepted for publication.

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

[12] 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.

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

[14] 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.

A “Safe” Functions List

Subsystems	”Safe” Functions	Source Files
File System	__put_super	fs/super.c
	kill_super	fs/super.c
	clean_inode	fs/inode.c
	iput	fs/inode.c
	file_operations.poll	include/linux/fs.h
	super_operations.write_super	include/linux/fs.h
	super_operations.read_inode	include/linux/fs.h
	super_operations.read_inode2	include/linux/fs.h
	super_operations.put_inode	include/linux/fs.h
	super_operations.clear_inode	include/linux/fs.h
	super_operations.put_super	include/linux/fs.h
	block_device_operations.release	include/linux/fs.h
	file_operations.release	include/linux/fs.h
Memory Management	shmem_recalc_inode	mm/shmem.c
	shmem_get_inode	mm/shmem.c
	oom_kill_task	mm/oom_kill.c
Networking	__skb_unlink	include/linux/skbuff.h
	__skb_insert	include/linux/skbuff.h
	skb_reserve	include/linux/skbuff.h