Towards Automated Authorization Policy Enforcement

Vinod Ganapathy
Univ. of Wisconsin-Madison
vg@cs.wisc.edu

Trent Jaeger
Pennsylvania State Univ.
tjaeger@cse.psu.edu

Somesh Jha
Univ. of Wisconsin-Madison
jha@cs.wisc.edu

Abstract

In systems with shared resources, authorization policy enforcement ensures that these resources are accessible only to users who are allowed to do so. Recently, there is growing interest to (i) extend authorization policy enforcement mechanisms provided by the operating system, and (ii) enable user-space servers to enforce authorization policies on their clients. A popular mechanism for authorization policy enforcement retrofits the code to be secured with *hooks* to a reference monitor. This is the basis for the Linux security modules (LSM) framework, and is also the intended usage of the recently-released security-enhanced Linux policy management framework for user-space servers. Unfortunately, reference monitor hooks are currently placed manually in operating system and user-space server code. This approach is tedious, does not scale, and as prior work has shown in the context of LSM, is error-prone. Our research is on techniques to largely automate authorization hook placement. We have devised a technique to do so, and have tested its effectiveness by applying it to determine hook placement for the Linux kernel, and cross-validating it with LSM hook placement. Our initial results are encouraging, and we have extended our technique to work with user-space servers. In particular, we have applied the technique to determine authorization hook placement for the X11 server.

1 Motivation

The goal of an authorization framework is to ensure that security-sensitive operations on system resources are only performed by users who are permitted to do so by the site-specific authorization policy. A popular architecture for constructing an authorization framework uses a reference monitor, which encapsulates the authorization policy to be enforced. The system to be secured poses an authorization query to the reference monitor before it performs a security-sensitive operation—it performs the operation only if the authorization query succeeds.

This architecture has been adopted by Linux security modules (LSM) [7], a flexible framework which allows diverse authorization policies to be enforced by the Linux kernel. LSM hooks are now available as part of the Linux-2.6 kernel, and these hooks have also formed the basis for the implementation of SELinux. In LSM, the reference monitor is implemented as a loadable kernel module, and *authorization hooks* are placed at appropriate locations in the kernel. These hooks define the interface (the API) of the reference monitor, and each hook call poses an authorization query to the reference monitor.

Authorization policy enforcement mechanisms have traditionally been confined to the operating system. However, recently, there is growing interest to retrofit user-space servers with the ability to enforce authorization policies via reference monitoring. The reason is that user-space servers, such as X Windows, web-servers, and middle-ware, offer shared resources, such as buffers and caches, to their clients, and manage multiple clients simultaneously. Thus, it is paramount to protect these shared resources from unauthorized access. For example, in the X server, the cut-buffer is shared between X clients. Suppose the X server runs on a machine capable of enforcing multi-level security (MLS), then the X clients will also have associated security-labels, such as *Top-Secret* and *Unclassified*. To enforce end-to-end security, the X server may wish to enforce an authorization policy on its X clients; for instance, it may wish to ensure that a "cut" operation from a *Top-Secret* window can never be followed by a "paste" operation into an *Unclassified* window. In fact, efforts are underway to secure the X server using a reference monitor-based architecture [5, 9]. The recent release of the SELinux policy management server [8] is intended to enable development of authorization policies in the SELinux policy language for any user-space application that would benefit. As with LSM, this policy management server provides answers to authorization queries, and authorization hooks are to be placed at appropriate locations within the

user-space server.

Unfortunately, there is little work on systematic techniques to place authorization hooks. Instead, placement is often decided manually and informally. This process suffers from two drawbacks:

• *Does not scale.* The process of placing hooks for the Linux kernel (in the context of LSM) was an iterative, time-consuming process. Clearly, it is tedious to repeat this process for each user-space server that needs to be retrofitted for reference monitoring. Automated solutions to determine hook placement are desirable.

• *Is prone to security-holes.* Prior research has shown security holes due to improper hook placement in the Linux kernel. Zhang *et al.* [10] demonstrate that inadequate placement of hooks results in security-sensitive operations being performed without the appropriate authorization query being posed to the reference monitor. Jaeger *et al.* [4] also demonstrate similar bugs by comparing the consistency of hook placements along different program paths. These bugs are potentially exploitable.

Our research is on *techniques to largely automate placement of authorization hooks*. We have developed a program analysis-based technique to do so, and have conducted two case studies. Our first study was to study the effectiveness of our algorithms by reproducing hook placement in LSM [1]. Because the (manual) hook placement in LSM has been extensively-verified, this enables us to evaluate the effectiveness of our technique. As we will show, our results with this study were encouraging. In recent work we have enhanced our technique, and have used it to determine authorization hook placement for the X11 server [2].

2 Benefits to the SELinux Community

We believe that our technique benefits the SELinux community in two ways:

• *Enables hook placement in user-space servers.* Our technique uses a combination of static and dynamic program analysis to determine where a user-space server performs security-sensitive operations. These locations are then retrofitted with hooks to a reference monitor. Because our technique is largely automated, it can significantly reduce the turnaround time of hook placement. For example, it took us about a week to use our technique to reconstruct the placement of file-system and networking hooks for the LSM framework (with fairly good precision). We have further refined the basic technique, and with these refinements, we were able to determine placement of hooks to protect `Window` operations in the X server in a few hours.

• *Can be used for verification.* While the focus of our work is to develop techniques to determine authorization hook placement for user-space servers, our technique can be adapted for verification as well. Thus, for code with authorization hooks placed, such as LSM, our technique can be used to verify existing placement by comparing it against the placement produced by our technique.

3 Overview of our Technique

We present a high-level, informal overview of our technique, and refer the interested reader to [1, 2] for details. Our technique proceeds in six steps, as shown in Fig. 1. Where applicable, we illustrate the technique using an examples from the X server and the Linux kernel. In the discussion below, we will denote the server to be retrofitted with authorization hooks by $\mathcal{X}$ (if the kernel is being retrofitted, then $\mathcal{X}$ refers to the kernel).

Step 1: Find security-sensitive operations to be protected. The first step is to determine the set of resources of the server $\mathcal{X}$ accesses to which must be controlled by an authorization policy. We refer to the operations that can be performed on these resources as *security-sensitive operations*. In our work so far, we have relied on manual identification of security-sensitive operations. In current work, we are investigating heuristics to automatically identify security-sensitive operations as well.

Manual identification of security-sensitive operations proceeds typically by considering a wide range of policies that $\mathcal{X}$ must enforce, and determining the set of security-sensitive operations based upon these policies.

For instance, about 500 security-sensitive operations were manually identified for the Linux kernel [6], and 59 security-sensitive operations were manually identified for the X server [5]. In the rest of this paper, we will represent these operations using sans-serif font as Resource_Operation. For example, in the case of Linux, shared resources included files, directories, sockets, and so on, and the security-sensitive operations identified for Linux included File_Write, File_Read, File_Execute, Dir_Rmdir, Dir_Mkdir, Socket_Create and Socket_Listen, each with their intuitive meanings. Similarly, for the X server, shared resources include `Window`s, `Font`s and `Drawable`s, and include security-sensitive operations such as Window_Create, Window_Map, and Window_Enumerate.

In the case of both the Linux kernel and the X server, a design team (at NSA) manually identified the set of security-sensitive operations. These security-sensitive operations are often only accompanied by an informal English-language description of their meaning (as in [6, 5]), and a precise code-level description is often not given.

One of our main contributions is in formalizing security-sensitive operations using code-level descriptions. That is, we characterize security-sensitive operations by the actual code-templates (also referred to as code-patterns), that are responsible for the security-sensitive operation. This is formalized in step 2.

Step 2: Infer root-cause of security-sensitive operations. The second step is to identify the *root-cause*

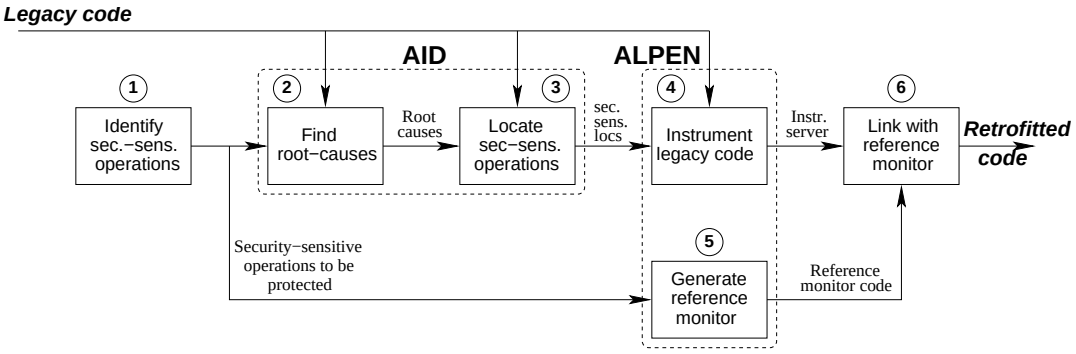

Figure 1: Steps involved in retrofitting a server for authorization policy enforcement.

of each security-sensitive operation. The root-cause of a security-sensitive operation is defined as the code-level constructs that *must* be executed for the security-sensitive operation to be performed. Formally, each root-cause is expressed as a conjunction of several *code-patterns*, which represent code-level constructs in terms of their abstract-syntax-trees (ASTs). Instead of presenting a formal definition of code-patterns and root-causes (which can be found in [2]), we present a two examples—further examples are available in [3]:

• The security-sensitive operation Dir_Write in the Linux kernel denotes a write operation to a directory. Its root-cause is identified as `SET inode->i_ctime` $\wedge$ `CALL address_space_ops->prepare_write()`. The intuition is that writing to a directory usually involves adding new content to the data structures that store directory content (achieved via the call to `prepare_write()`), followed by setting the change time (field `i_ctime` of the directory's inode).

• The operation Window_Map in the X server, corresponding to mapping an X client window to the screen, is characterized by `SET xEvent->union->type TO` $MapRequest \wedge$ `SET xEvent->union->type TO` $MapNotify$. This intuitively corresponds to an X client request to the X server for mapping a window, followed by a notification by the server that the operation was successful.

For our initial case study, that of placing authorization hooks in Linux, we wrote these root-causes manually. While it was fairly easy to write root-causes—we wrote about 100 in a week corresponding to security-sensitive operations related to the file-system and networking subsystem—it was clear that that an automated technique was needed if this approach is to scale for user-space servers. The key challenge is to automatically recover the association between security-sensitive operations, and the code-patterns that are their root-causes.

A key observation helps us achieve this goal. It is that each security-sensitive operation is typically associated with a tangible side-effect. For example, the security-sensitive operation Dir_Write in the Linux kernel typically corresponds to changed directory contents. Similarly, the security-sensitive operation Window_Map i n the X server results in an X client window being mapped to the screen. Thus, if we induce the server to perform the tangible side-effect associated with a security-sensitive operation, and trace the server as we do so, the code-patterns that characterize the security-sensitive operation *must* be in the trace.

However, program traces are typically long, and it is still challenging to identify the code-patterns that characterize a security-sensitive operation from several thousand entries in the program trace. We have developed a technique (the details of which are in [2]) to compare program traces corresponding to different side-effects, and reduce the portion of the trace that must be examined to determine root-causes.

Using this technique identifying root-causes reduces to studying fewer than 10 entries, on average, in a program trace. We have applied this technique to determine root-causes of security-sensitive operations on the `Window` data structure in the X server, and have automatically and precisely identified the root-causes of these operations. For example, the root-cause of `Window_Map`, discussed earlier, was automatically identified by our technique.

Step 3: Find all locations which are security-sensitive. Finding root-causes of security-sensitive operations alone does not suffice—we must also find all locations in the code of the server where these operations may potentially be performed. The third step uses the results of root-cause analysis to statically identify all locations in the server where code-patterns that characterize a security-sensitive operation occurs; each of these locations performs the operation. Consider Fig. 2, which shows a snippet of code from `MapSubWindows`, a function in the X server. It contains writes of `MapRequest` and `MapNotify` to `event.u.u.type`, as well as a traversal of the children of the window pointer `pParent`. A call

to the function `MapSubWindows` performs, in addition to Window_Map, the security-sensitive operation Window_Enumerate, corresponding to enumeration of child windows. We automatically identify the set of security-sensitive operations performed by each function call using a static analysis algorithm, which searches the code of the server for the code-patterns in the root-cause of a security-sensitive operation.

```
MapSubWindows(pParent, pClient) {
  pWin = pParent->firstChild;
  for (;pWin; pWin = pWin->nextSib)
  { event.u.u.type = MapRequest;...
    event.u.u.type = MapNotify;...
} }
```

Figure 2: `MapSubWindows`

In addition to identifying the locations where security-sensitive operations occur, in this step we also use heuristics to identify the subject and object associated with the operation. To do so, we identify the variables corresponding to subject and object data types (such as `Client` and `Window`) in scope. In most cases, this heuristic precisely identifies the subject and the object. In Fig. 2, the subject is the client requesting the operation (`pClient`), and the object is the window whose children are to be mapped (`pParent`), both of which are parameters of `MapSubWindows`, and are in scope.

Steps 2 and 3 together identify all locations where the server performs security-sensitive operations, and at each location, also help identify the subject and object associated with the operation. We have implemented a prototype tool, called AID, that performs these steps.

Step 4: Instrument the server. Having identified all locations where security-sensitive operations are performed, the server can be retrofitted by inserting calls to a reference monitor at these locations, to achieve complete mediation. Note that if AID determines that a statement **`Stmt`** is security-sensitive, it also identifies the *security-event* that it generates. A security-event is a triple $\langle sub, obj, op \rangle$, denoting the subject sub requesting operation op to be performed on object obj. A statement **`Stmt`** that generates the security event $\langle sub, obj, op \rangle$ is instrumented as shown below.

`if (query_refmon(`$\langle sub, obj, op \rangle$`) == False) then` **`handle_failure`**`; else` **`Stmt`**`;`

The statement **`handle_failure`** can be used by the server to take suitable action against the offending client, either by terminating the client, or by auditing the failed request. Authorization policies are typically expressed in terms of security-labels of subjects and objects. Security-labels can be stored in a table within the reference monitor (generated in step 5), or alternately, with data structures used by the server to represent subjects and objects. The latter technique is used by LSM, which adds fields to kernel data structures such as `inodes` and `sockets` to store security-labels. The same technique can also be used with user-space servers. For example, in the X server, extra fields can be added to the `Client` and `Window` data structures to store security-labels. Because we pass both the subject and the object to the reference monitor using `query_refmon`, the reference monitor can lookup the corresponding security-labels, and consult the policy.

Step 5: Generate reference monitor code. This step generates code for the `query_refmon` function. We generate a template for this function, omitting two details that must be filled-in manually by a developer. First, the developer must specify how the policy is to be consulted. We do not constrain the authorization policy language to be used, and the developer can choose a policy language and a policy management framework of his choice. For example, the Tresys SELinux policy management framework can be used [8].

Second, he must specify how the security-labels of subjects and objects change in response to an authorization request. For example, in the X server, when a security-event $\langle$`pClient`, `pWin`, Window_Create$\rangle$ succeeds, corresponding to creation of a new window, the security-label of `pWin`, the newly-created window, must be initialized appropriately. Similarly, a security-event which copies data from $\texttt{pWin}_1$ to $\texttt{pWin}_2$ may entail updating the security-label of $\texttt{pWin}_2$.

Because security-labels are either stored as a table within the reference monitor, or as fields of subject or object data structures, as described earlier, the developer must modify these data structures appropriately to update security-labels. Note that while steps 2-4 are policy independent, step 5 requires knowledge that depends on the specific policy to be enforced. Steps 4 and 5 together ensure complete mediation of security-sensitive operations identified by AID: we have prototyped these steps in a tool called ALPEN (see Fig. 1). While we have not yet designed ALPEN to generate code that can be used with Tresys' SELinux policy management server, we intend to do so in the near future.

Step 6: Link the modified server and reference monitor. The last step involves linking the retrofitted server and the reference monitor code to create an executable that can enforce authorization policies.

A noteworthy feature of our approach is its modularity. In particular, alternate implementations of root-cause analysis and instrumentation can be used in place of AID and ALPEN, respectively. Thus, our technique benefits directly from improved algorithms for these tasks.

4 Case Studies

To understand the effectiveness of our approach, we have conducted two case studies. Our first case study was performed with the Linux-2.4 kernel. Our goal here was to

Hook Cat.	Num. Locs.	False Pos.	False Neg.
inode (26)	40	13	4
socket (12)	12	4	0

Figure 3: Results of hook placement using our technique. False positives count locations where our technique places an extra hook, while false negatives count locations with missing hooks.

reproduce, as closely as possible, the hook placement in LSM. The reason we chose version 2.4 of the kernel (instead of version 2.6) was because hooks are not placed by default in version 2.4, thus allowing us to objectively evaluate the precision of our technique. As mentioned earlier, for this study, we wrote root-causes for security-sensitive operations manually. However, we soon realized that this approach would not scale to user-space applications. Thus, we designed an automated technique to identify root-causes, as discussed in step 3 of §3. Our second case study used automated root-cause finding to determine hook placement for the X11 server. We discuss preliminary results from both case studies below.

4.1 Placing hooks in the Linux kernel

To study the effectiveness of our technique, we evaluated the precision with which it determines hook placement in the file system and the networking subsystem of the Linux-2.4 kernel. Fig. 3 presents the results of our study.

The LSM framework places 149 distinct hooks at 248 locations in the Linux-2.4 kernel. Of these, 26 are inode hooks placed 40 locations in the kernel, while 12 are socket hooks, and are placed at 12 locations in the kernel. Fig. 3 compares the results produced by our automated technique against manually placed hooks in the LSM framework. It shows both *false positives* as well as *false negatives* that were generated by our technique. A false positive corresponds to a location in the kernel where our technique places an extra authorization hook as compared to the LSM hook placement. False positives are undesirable because they result in extra authorization, potentially (wrongly) denying a user access to a resource. False negatives result in missed authorization checks, and are potentially causes of security holes. Our technique produced 13 false positives and 4 false negatives for inode hooks and 4 false positives for socket hooks.

As mentioned earlier, we wrote root-causes manually for security-sensitive operations in the Linux kernel. We are encouraged by the results in Fig. 3 because they were obtained using about 100 root-causes that we wrote in just a week. We have since enhanced our technique to automate the process of writing root-causes.

4.2 Placing hooks in the X server

Our second case study was to place hooks in the X server. Note that others [5, 9] have also made similar efforts. However, our goal was to automate the process as much as possible. We have so far focused on placing hooks for security-sensitive operations on the `Window` data structure. The NSA [5] has identified 59 security-sensitive operations for the X server, of which 22 are related to `Windows`. We were able to precisely identify the root-causes for 18 of the 22 security-sensitive operations using our automated root-cause-finding algorithm.

Using AID and ALPEN we have placed hooks to protect security-sensitive `Window` operations. We have tested the efficacy of our technique by writing policies to prevent a few attacks. For example, we have written a policy to prevent an unauthorized X client from setting properties belonging to another X client. Similarly, we have also written a policy to prevent information leakage via an unauthorized "cut-and-paste" operation.

As mentioned earlier, our policies are not yet written in the SELinux policy language. In future work, we intend to integrate our technique to work with the SELinux policy management server, thus enabling enforcement of policies written in the SELinux policy language.

References

[1] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *12th ACM CCS*, Nov 2005.

[2] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. Technical Report 1544, Univ. of Wisconsin, Nov 2005.

[3] www.cs.wisc.edu/∼vg/papers/ccs2005a/idioms.html.

[4] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM TISSEC*, May 2004.

[5] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window system with SELinux. NAI/TR03-006.

[6] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical report, NAI Labs/TR01-043, December 2001.

[7] C. Wright *et al*. Linux security modules: General security support for the Linux kernel. In *11th USENIX Security*, August 2002.

[8] Tresys technology, SELinux policy management server. http://sepolicy-server.sourceforge.net.

[9] E. Walsh. Integrating XFree86 with security-enhanced Linux, 2004. Manuscript.

[10] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *11th USENIX Security*, August 2002.