VERIFIED ENFORCEMENT OF SECURITY POLICIES FOR CROSS-DOMAIN INFORMATION FLOWS

Nikhil Swamy Michael Hicks
University of Maryland, College Park
{nswamy, mwh}@cs.umd.edu

Simon Tsang
Telcordia Technologies
stsang@telcordia.com

Abstract

We describe work in progress that uses program analysis to show that security-critical programs, such as cross-domain guards, correctly enforce cross-domain security policies. We are enhancing existing techniques from the field of Security-oriented Programming Languages to construct a new language for the construction of secure networked applications, SELINKS. *In order to specify and enforce expressive and fine-grained policies, we advocate dynamically associating security labels with sensitive entities. Programs written in* SELINKS *are statically guaranteed to correctly manipulate an entity's security labels and to ensure that the appropriate policy checks mediate all operations that are performed on the entity. We discuss the design of our main case study : a web-based Collaborative Planning Application that will permit a collection of users, with varying security requirements and clearances, to access sensitive data sources and collaboratively create documents based on these sources.*

1 INTRODUCTION

Cross-domain security problems arise frequently in common military operations. For instance, every time there is a mission need to share information with coalition partners at varying levels of trust (e.g. U.K., Canada or non-NATO countries); with US Government entities (e.g. FBI, IRS); with deployed troops; or, say, with academic institutions, care must be taken to ensure that shared information is cleared at the appropriate classification level. In general, difficulties arise whenever the parties sharing information operate on different classification levels and further may be on different networks that are cleared at different classification levels. The problem is further complicated when considering messages with varying levels of data sensitivity.

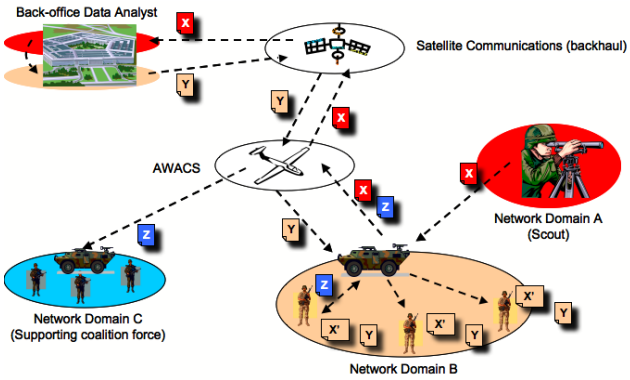

Figure 1. A multi-domain military network.

Figure 1 illustrates a hypothetical, but typical, scenario in which enclaves of entities with varying clearance levels and responsibilities collaborate in a networked environment to achieve a common mission objective. Each network domain connects to the others via a cross-domain guard entity (CDG) at the edge of each domain — for instance, the armored vehicles at the edge of domains B and C. From a security perspective, the CDG is the element tasked with deciding whether or not packets routed between domains are to be transmitted. Information collected by soldiers in Network Domain B (packet Z) is communicated via the CDG to members of a coalition force in Network Domain C, and may be filtered by the CDG if it con-

This document is prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

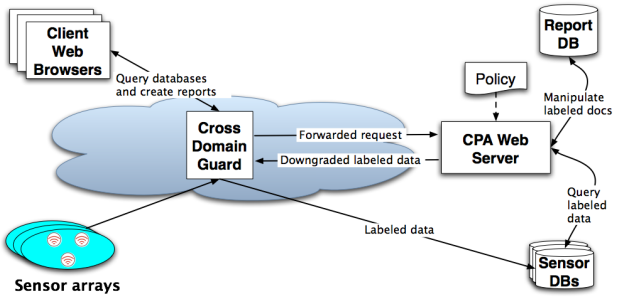

Figure 2. Overview of the CPA.

tains inappropriately sensitive information.

The current approach to dealing with this problem is centered around the data content of the packet. That is, policies at the CDG are defined in terms of filters that scan packets for particular data patterns (e.g. troop location information) and block packets that match the patterns. Our position is that content-based filtering of packets, while flexible, is ad hoc; difficult to reason about formally; and can both be overly conservative in filtering critical data that needs to be shared, and unexpectedly permissive in allowing sensitive data to pass through. In this paper, we propose a framework for maintaining and tracking labels on sensitive information objects in a manner that permits the construction of networked, collaborative applications that are provably compliant with a range of formal, yet practical, security goals.

The remainder of this paper is structured as follows. In Section 2 we describe the functionality and security goals of a web-based application that integrates with the network of Figure 1. In Section 3 we present a short review of existing techniques for the verified enforcement of policies specified using object labeling. In Section 4 we discuss a model for security labels. In Section 5 we describe some of the features of a new programming language we are building that allows cross-domain information flows to be secured. Section 6 concludes with a discussion of the current status of our project and future directions.

2 A COLLABORATIVE PLANNING APPLICATION

We are building a web-based collaborative planning application (CPA) as a test bed in which to experiment with a variety of security policies and enforcement mechanisms. This CPA is designed to allow back-office analysts, field commanders and intelligence agents to collaboratively create documents. These documents could be situation reports, intelligence estimates, or independent assessments by principals at a combat site, each backed up by data gathered from an array of sources, such as video sensors or troop location information. Together, principals with different skill sets and responsibilities collaborate to construct actionable intelligence items for consumption by decision makers.

Figure 2 illustrates, at a high-level, the structure of the CPA. The application is hosted by a web server shown toward the right of the figure. We expect the application to be integrated into the scenario from Figure 1 — as with all other cross-domain communication, requests to the CPA domain are dispatched through a CDG element. The figure shows several domains of sensor arrays that stream data to a collection of sensor databases via the CDG. The data in the sensor DBs are labeled with security labels. The form and content of these labels are discussed in detail in subsequent sections, but, for now, one might think of these as provenance labels — i.e. labels that identify the source of the data.

The basic architecture of the CPA is typical of multi-tiered web applications, found both in the military and in the public sector. The three tiers in this application consist of a client tier (at the top left) consisting of dynamic HTML running in a client's web browser; the web server; and server-side databases (the disks at the right). There are three main security critical tasks that the CPA will perform in order to allow users of various security levels to collaboratively author reports.

Accurate tracking of labels on data sources. In order to author reports, users can issue queries via the CPA to the sensor DBs to extract curated data from the raw sensor data [2]. For two main reasons, it is important for the curated data to accurately reflect the security policies and provenance information of the raw data from which it is derived. First, accurately tracking data sources increases the credibility of the resulting report and permits auditing of a report should errors be discovered. Second, if the security policies of the raw data is not correctly associated with the curated data, then adversaries can easily mount attacks on the data sources by analysis of the curated data.

Tagging documents with dynamic policies. Se-

curity labels need not always originate from outside the CPA. Authors may introduce labels by associating them with various parts of a document. These labels can be used to indicate a security policy, the document's provenance as well as semantic tags such as document keywords that will assist other users with search. We will support a very fine-grained document structure in which even individual characters in a document can potentially be labeled. Importantly, security policies are decentralized in that principals can specify labels independently of other principals. Similar to labels on data sources, document labels will also be tracked through all functions of the application — e.g. database queries that extract fragments of documents will have to accurately reflect the labels of fragments in the query results.

Complete mediation of policy authorization checks. The CPA must ensure that all sensitive operations on labeled objects are mediated by policy checks that are consistent with the intended meaning of the label. For instance, if the label on a datum is an MLS label, the CPA must make sure that the datum never flows to location with a lower MLS label.

We face four main technical challenges in constructing the CPA. First, we must design a language of labels that is sufficiently expressive to model a variety of common security policies. Second, we must be able to precisely track label flows through a variety of program operations in multiple tiers of an application (e.g. through database queries). Next, we must ensure that policy authorization checks that take place in the various tiers of the application are all mutually consistent (e.g. policy checks that occur in DB stored procedures have to have the same semantics as checks that occur in the web application). Finally, we must ensure that when labeled objects are communicated across the network or are persisted to the database that the relationship between object and label is correctly represented by the receiver of the object or in the database.

In the remainder of the paper, we discuss how we intend to perform each of the three main security critical tasks and address the four challenges of the CPA. We begin, however, with a short tutorial on SOPLs. For a more complete treatment, refer to Sabelfeld and Myers' excellent survey paper [5].

3 A PRIMER ON SOPLs

Security-oriented programming languages (SOPLs) utilize automated static and dynamic program analyses to prove that programs enjoy certain security properties. Typically, the main goal is to permit an application to manipulate objects having varying multilevel security *information flow* policies, while *proving* that the data and control flow of the application respects these policies by not permitting information about an object at a given security level flow to a context at another incompatible security level.

A SOPL makes policies about the confidentiality and integrity of data manifest in the program so they can be mechanically checked. Typically, these policies are expressed as labels on the *types* in the program. During the process of type checking, the compiler verifies that the labeled program does not violate the policy described by these types. Various models for security labels have been proposed in the literature. To illustrate the basic features of an SOPL we present a very simple label model here. In subsequent sections we propose more complex label models that are better suited to practical SOPLs for use in a dynamic, distributed environment.

In the simple model, labels are a pair (L_C, L_I), where L_C stands for the confidentiality component and L_I stands for the integrity component of label. Each component can be thought of as the name of a principal in the system. The security policy of a program consists of the label annotations that appear on types as well as a policy describing an acts-for hierarchy among principals. For instance, if the acts-for hierarchy specifies that $Alice \leq Bob$ (meaning that Bob acts for Alice), then *Bob* is permitted to observe all data that *Alice* is permitted to observe, and dually, *Alice* trusts the integrity of all data that *Bob* trusts. In this case, it is permissible for data with a label annotation $(Alice, Bob)$ to flow to a location labeled $(Bob, Alice)$. In this manner, the acts-for relation $\leq$ on principals can be extended naturally to a lattice ordering relation $\sqsubseteq$ on labels (L_C, L_I), where data with label lower in the lattice is permitted to flow to locations higher in the lattice.

The goal of a SOPL is to ensure that a strong security property holds for all well-formed programs. Traditionally, this property has been *noninterference*, which means that values whose labels are higher in the

lattice are independent of computations that produce values observable by principals whose privilege-level is lower in the lattice. To enforce this property, a security type system permits the programmer to add security label annotations on types. The compiler tracks information flows through the program, and rejects programs that violate the security policy described by the type annotations and the label lattice.

For example, the following program declares an integer variable a readable only by *Alice* (and those principals that act for her) and trusted only by *Bob* (and the principals the he acts for), while the contents of the variable b is readable by *Bob* and trusted by *Alice*.

```
1  int{Alice, Bob} a;
2  Point{Bob, Alice} b;
3  a = b.getXCoord();
4  if (b.getXCoord() > 0)
5    a = 0;
```

Assuming the acts-for ordering $Alice \leq Bob$ the compiler rejects line 3 above as potentially insecure: getting the x coordinate of the Point object reveals data to Alice that should only be observable by Bob and corrupts the contents of a (trusted by Bob) with data that is only trusted by Alice. It is also necessary to track *implicit* information flows that arise due to the control structure of the program. Thus, the compiler rejects lines 4–5 as being potentially insecure because the contents of a reveal partial information about b's x coordinate and also corrupts the contents of a since the assignment to a depends on a value (b's x coordinate) that is not trusted by Bob.

Research over the last several years has scaled these basic ideas to many common language constructs, including method invocation, exceptions, constructor invocation, loop guards, and conditionals. However, there is relatively little work exploring practical models of information-flow security in distributed programs, such as the CPA.

4 A MODEL FOR SECURITY LABELS

Most existing security-typed languages use the *lattice model of information flow* [8] in which an information flow policy Π is defined by a lattice $(\mathscr{L}, \sqsubseteq)$, where $\ell \in \mathscr{L}$ is a *label* (or *security level*), and labels are ordered by the relation $\sqsubseteq$. While this simple label model is sufficient in a formal setting, in practice more expressive labels are required for several reasons. First, the lattice of labels may need to be changed during a long-running program's execution. Second, while MLS information flow labels are useful in enforcing noninterference-like properties, other security policies such as downgrading, access control and data provenance tracking also need to be supported. In such cases, no natural lattice ordering on labels exists. In this section, we discuss a label model that supports dynamic policies, and can be enriched to support more than just information-flow policies.

4.1 DYNAMIC MLS LABELS

If policy updates are to be supported, a reasonable administrative model should be able to provide answers to the following questions. (1) *Who* is allowed to make changes to the security policy? (2) *What* parts of the policy are permitted to change? (3) *How* should those changes be reflected in the running program?

Rather than develop an administrative model for existing label models, we looked instead to the body of work on formal policy languages for which administrative models already exist. *Role-based* policy languages suggest a natural label model. In particular, a *role*, which is a name that represents a set of principals, can be treated as a label, and the ordering between labels can be defined in terms of subset on the contents of roles according to the policy. The language RT_0 provides a particularly convenient source of labels. RT_0 is the simplest member of the role-based policy language framework *RT* [4]. The following features of RT_0 roles provides answers to each three questions posed previously.

Ownership. An *RT* role is defined as having an *owner* responsible for the role's definition; a given principal can own many roles. Only a role's owner is allowed to change the definition of that role.

Membership and Delegation. An *RT* policy permits delegation at the granularity of roles, in which one role may be defined in part by the contents of another role. This provides better control than the standard lattice model, which only permits delegation between principals. The result is that in the standard model, principals either delegate all their privileges to another principal or none. By contrast, role membership and role delegation in *RT* are separate concepts. Roles have an owner, and membership is strictly under the owner's control: the owner can either include a principal in a

role directly, or delegate (part of) the definition of a role to another role. Membership does not imply delegation.

Indirection. Defining labels as roles provides a useful level of indirection because the membership of a role may change while the label on data stays the same. That is, a security policy of some data can be modified without requiring the data to be relabeled.

Role ordering. Roles can conveniently be arranged on a lattice using the subset relation, since they can be interpreted as sets of principals. Alternatively, it is also possible to use the logical structure of the policy to define a partial order on roles. While the tradeoffs between the choices are beyond the scope of this paper, we note that both choices admit the specification of multi-level information flow policies.

We have formalized the use of roles as security labels in an SOPL that supports dynamic policy updates [6].

4.2 OTHER FORMS OF LABELS

MLS information flow policies allow strong noninterference-like security properties to be expressed, but such a security goal is not always necessary or appropriate. One relaxation of noninterference that has been studied extensively is downgrading or declassification. This is of particular relevance to the cross-domain setting and policies for downgrading is the focus of another paper at this conference [7]. In this section, we briefly sketch the form of labels that can be used to express access control policies and data provenance policies.

Access control labels. While MLS policies are data-centric (i.e., they control data flow) access control policies can be used more naturally to control *actions* as well as data. However, access control typically does not place constraints on the modes of usage of a sensitive object once access has been granted. For instance, under an access control policy, once a principal is granted access to a file, then she is free to copy contents of the file to another file governed by a weaker policy. This weakness of access control distinguishes itself from MLS security and makes it a viable security policy where MLS security is too strong. We will derive a model for access control labels in a manner that integrates with an information flow policy following a technique proposed recently by Abadi [1]. The core of this approach is to use a specialized SOPL to specify an access control policy. The result is that it is possible to formally prove that a principal's access control policy cannot be tampered with by other unauthorized principals.

Provenance labels. Tracking data provenance in curated databases has been the subject of recent study [2]. Here, a set of labels is associated with each subterm of an expression e that is computed as a function of a labeled data set. For instance, e might be a term that computes statistical information from an array of sensor data by running an SQL query over the database. Each entry in the database is tagged with the identity of the sensor that produced the value. The labels on the subterms of e then reflect which sensor readings contributed to the computation of each subterm. It is possible to prove a noninterference-like theorem based on the dependences between the labeled term and the data sources. One could also derive other security policies from the provenance labels, stating, for instance, that only data from an authorized set of sources may participate in a particular computation.

5 SECURE PROGRAMMING IN SELINKS

The multi-tier structure of the CPA, as well as the coordination between its multiple component applications presents several challenges for controlling the flow of sensitive data. The research we are conducting aims to provide a coherent treatment of flows across the entire application by scaling up the basic ideas of security-oriented programming languages. By using these techniques, we will have effectively constructed a lightweight, static proof of end-to-end security.

We are extending the LINKS [3] web programming language with support for enforcing fine-grained, security policies specified using object labeling. LINKS is novel in that it permits writing a single, multi-threaded program that is automatically split into client, server, and database components, and translated to Javascript, LINKS, and SQL, respectively. The main advantage of using LINKS is that one can analyze a single program, before it is split into and compiled to its multi-lingual component parts. This makes the analysis task much simpler (and more trustworthy). Though space precludes a complete discussion, we attempt to provide the reader with a feel for LINKS programming by discussing a small example program. We

```
1  sig getPage : (Label::MLS as cred, Int) →[Page{cred}]
2  fun getPage(cred, pageid) server {
3      var pages = table "pages" with (
4          pglabel : Label::MLS,
5          pageid : Int{pglabel},
6          title : String{pglabel},
7          wikibody : String{pglabel}
8      ) from database "test";
9      for (var p ←pages)
10       where ((p.pglabel ⊑cred) ∧
11              (p.pageid == pageid))
12       [p]
13 }
14 fun showWikiPage(cred, pageid) client {
15     formatDisplay(getPage(cred, pageid))
16 }
```

Figure 3. Dynamic MLS labels in SELINKS.

call our extension of LINKS, *Security Enhanced Links*, or SELINKS.

Figure 3 shows a small program with an emphasis on server-database interactions. A LINKS program consists of a series of function definitions followed by some initialization code to start the application. Each function is labeled with either `client` or `server` annotation to indicate where it is supposed to run. Function calls may traverse the client/server gap, and the compiler automatically translates such calls into synchronous remote procedure calls (RPCs) — in the example, showWikiPage is a client function that makes a remote call to the server to fetch the page contents and then draws it to the screen.

Interactions with a database are primitive operations in LINKS. Any server function may connect to a database to perform queries and updates. The advantage here as shown in the function `getPage`, is that the database's data model is made evident in the LINKS program. This means that one can translate LINKS values to database rows and back again without the unwieldy usage of quoted SQL strings (as found in Java and JDBC) and and unsafe coercion functions that translate database result set types to types in the application language. In LINKS, database queries are expressed as *list comprehensions*. The example query selects all rows from the table `pages` where the page identifier is as requested. The function also performs some additional security operations which we discuss next.

At line 1, the signature of the function getPage describes its security behavior using a limited form of dependent typing. The function expects two formal parameters, the first of type *Label* and the second an *Int*. The *Label* type is an addition in SELINKS and stands for the type of label values; in this case, the *kind* of the *Label* is *MLS*, stating that this function only deals with MLS security (rather than access control, downgrading etc.). The function returns a list of Page elements, where each Page is at a security level not higher than the level specified by the cred parameter.

At line 4 in the body of the function, the database schema is updated to reflect the storage of *Label* values in the database, where previously only a user identifier was stored. Each field in each row of the `"pages"` table is at a security level determined by the *Label* stored in the same row. At line 9, we define an expression that selects documents from the table, but we must be careful to ensure that in line 10 the correct authorization check is performed in the `where` clause. In this case, we check in line 10 that the privilege level specified in cred is no less than the pglabel field of the row. Since we only select rows that satisfy this conditions, we can safely use subsumption to treat each selected record as being protected at level cred, which is the requirement stated in the type signature. Note that if the kind of the label was not *MLS* then some other form of authorization check would have to be performed at line 10. Notice that unlike the example from Section 3, the actual value of a security label is known only at runtime.

Ensuring complete mediation of the security policy requires three steps. First,a static analysis of SELINKS programs to ensure that all authorization checks, such as the label ordering check on line 11 of Figure 3, are present. Second, complete mediation requires that object-label relationships represented in the type language of the SELINKS program (e.g Page{cred}) be consistent with the label-data relationship as maintained in the database. In the example in Figure 3, the mapping between the database's view of objects and labels and the application's view is particularly straightforward — the object and the label are stored as a tuple in the database and in the application. However, much more complex relationship are possible, and necessary, for practical programs. When persisting or querying labeled data to/from the database, we must ensure that the object-label relationships are preserved. Finally, we must ensure that all policy checks

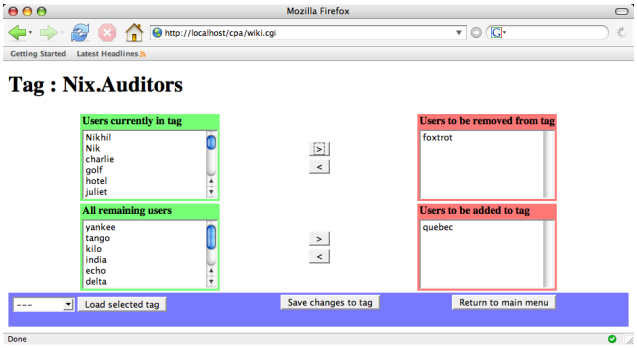

Figure 4. Managing a user's roles.

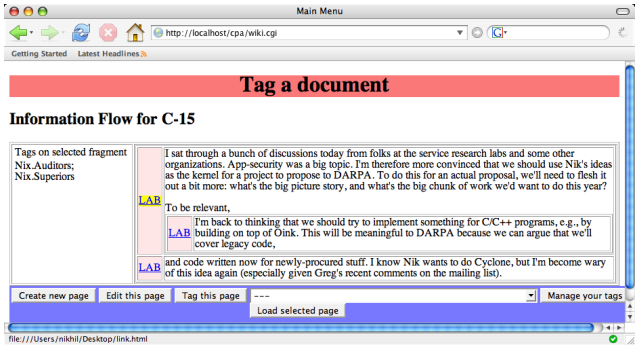

Figure 5. Viewing labeled documents.

that occur with SQL queries that run at the database are consistent with the corresponding policy checks that occur within the SELINKS program.

6 CONCLUSIONS

We conclude by showing two screenshots from an prototype of the CPA that is under development. Figure 4 shows the interface that allows a user to manage his role labels. The top of the window shows the name of the role (in this case, the *Auditors* role belonging to the user *Nix*). The application provides a GUI to allow the user to add new users to the role, or to remove existing users from the role. Other operations such as delegating role management will also be supported. Figure 5 shows the interface that allows a user to view existing labels on a document and to possibly add new labels to the document. The contents of the page is structured into a tree-like structure of tables, with nested tables showing the labels associated with each labeled fragment of the document. A mouse over the label loads the set of labels in the pane toward the left of the frame. Designing an intuitive UI that is well-suited to a security critical application is also a major challenge in this project and is an area which we have only just begun to explore.

Throughout this paper, we have attempted to give a flavor of our approach to address each of the key tasks and challenges of the CPA identified in Section 2. In review, we are enhancing techniques proposed by Buneman et al. [2] in order to accurately track provenance labels on data sources; we are using roles derived from the RT_0 language as dynamic labels to tag documents; and we have illustrated by example how the type system in SELINKS ensures complete mediation of policy authorization checks. Our challenges remain to integrate multiple models of security policies into a single label model. We address some key issues in the design of cross-domain security policies in an accompanying paper [7]. Finally, we address the problems of label consistency, tracking and enforcement across the multiple tiers of an application by performing a cross-tier analysis of SELINKS programs.

References

[1] M. Abadi. Access control in a core calculus of dependency. In *ICFP*, 2006.

[2] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[3] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. `http://groups.inf.ed.ac.uk/links`, 2006.

[4] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *Security and Privacy*, 2002.

[5] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[6] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW*, 2006.

[7] S. Tsang, M. Hicks, and N. Swamy. Cross-domain information flow policy languages, properties and analysis. In *Submission to MILCOM '07*, 2007.

[8] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *JCS*, 4(3), 1996.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.