Combining Provenance and Security Policies in a Web-based Document Management System

Brian J. Corcoran Nikhil Swamy Michael Hicks

{bjc, nswamy, mwh}@cs.umd.edu

1. Introduction

Provenance and security are intimately related. Cheney et al. [3] show that the dependencies underlying provenance information also underly information flow security policies. Provenance information can also play a role in history-based access control policies [1]. Many real applications have the need to combine a variety of security policies with provenance tracking. For instance, an online stock trading website might restrict access to certain premium features it offers using an access control policy, while at the same time using an information flow policy to ensure that a user's sensitive trading information is not leaked to other users. Similarly, the application might need to track the provenance of transaction information to support an annual financial audit while also using provenance to attest to the reliability of stock analyses that it presents to its users.

We have been exploring the interaction between provenance and security policies while developing a document management system we call the *Collaborative Planning Application* (CPA). The CPA is written in SELINKS, our language for supporting user-defined, label-based security policies [7]. SELINKS is an extension of the Links web-programming language [4] with means to express label-based security policies. Labels are associated with the data they protect by using dependent types which, along with some syntactic restrictions, suffice to ensure that user-defined policies enjoy *complete mediation* and cannot be circumvented [6]. Our interest in provenance and security policies is thus part of a broader exploration of how security policies can be encoded, composed, and reasoned about within SELINKS. In this paper, we describe the architecture of the CPA and its approach to label-based provenance and security policies (Section 2) and we sketch directions for further exploration on the interaction between the two (Section 3).

2. Current Status: Provenance in the CPA

The goal of the CPA is to help users organize information, potentially at a variety of security levels, in the style of a blog/wiki *a la* Intellipedia [5]. Documents in the wiki are composed of blocks, which can range from an entire page to a single word, and a *label* can be associated with any block. Labels have the general form seen in Fig. 1 (using Links' notation for declaring a recursive type). Labels consists of access control labels (`Acl`), which consist of lists of groups of users with read access and write access, and provenance labels (`Prov`) which are comprised of a list of `ProvAction` labels that have affected the labeled data. We allow labels to be combined through the use of a `Composite` label; this permits the composition of policies. Labels can be modified (given sufficient privilege), e.g., to change the security policy. Care must be taken to ensure that labels associated with different policies do not interact badly. For instance, provenance information can itself be sensitive and may need to be protected with its own security policy. We invite the reader to experiment with the CPA at `http://macdonald.cs.umd.edu:8002/`.

```
typename Label = mu label .
[| Composite: [label]
 | Acl: (read: [Group], write: [Group])
 | Prov: [ProvAction]
|];
```

Figure 1. The type of a document's labels.

Data provenance is a natural fit to the CPA. As with most wiki software, we are concerned with keeping a log of how, when, and by whom pages are edited. To implement data provenance, we wrap every "public" function that modifies the state of the wiki so that it will correctly create or update the appropriate label. Provenance policies consider creating, modifying, deleting, restoring, and relabeling blocks. `ProvAction` labels, defined in Fig. 2, associate this operation type with assorted provenance data. We store the username and IP address of the current user, the modification date and time, and the original source of the data (e.g., a URL or another block). We imagine provenance information being used to audit data, or to assign blame for information leaks. Provenance information can itself be protected from tampering; e.g., cover-ups of information leaks can be avoided by preventing permanent deletion of provenance meta-data.

Fig. 3 shows a snippet of code that is ensured to mediate all operations that modify a page. As such, we think of this snippet as our policy for propagating provenance labels that track document modification events (i.e., the `Modify` operation) and we annotate the function with the `policy` qualifier. The argument `cred` is a credential representing the identity of the user currently logged in; `doc` is the document that is to be modified; `path` represents a path in the document tree from the root to the node that is to be replaced by `block`. The function constructs a provenance label recording the `Modify` action by the user and it adds this label to the set of labels associated with the `block`.

The higher-order function `applyWriteToBlock` takes as an argument a function that will modify part of the document, applying this function only if the `Acl` labels on the document allow. The `replace` function expects to receive the old label and old block from the specified path. In this case, we are entirely replacing the

```
typename ProvOp =
 [|Create|Relabel|Modify|Copy|Delete|Restore|];
typename ProvAction = (
  oper : ProvOp, user : String,
  date : String, time : String,
  ipaddr : String, source : String
);
```

Figure 2. The type of provenance actions.

```
fun replaceBlockProv(cred, page, path, block)
policy {
  fun replace (li, _) {
    var lProv = mkProvLabel(Modify, cred);
    var l = joinLabels(li, lProv);
    labelBlock(l, block)
  }
  applyWriteToBlock(cred, replace, path, page)
}
```

Figure 3. A policy that tracks `Modify` actions.

block, so `replace` ignores the value the second argument and replaces it with with the new block `block`, after updating the associated labels.

The code for `applyWriteToBlock` is given in Fig. 4. It first calls `getBlock` to retrieve a specific block and label from a page. This pair (`l`, `oldBlk_l`) is given the type of a dependent pair in which the first component is a label that labels the block in the second component. The CPA doesn't require all blocks to be labeled. However, an unlabeled block in the document tree must inherit the access control label of its closest labeled parent. The function `getBlock` traverses the tree as necessary to locate both the requested block and its associated label, returning a default label if none is found. Once we have the old block, the `applyWrite` function applies the function `f` to convert it to the new block. The new block is then saved to the database using `dbreplaceBlock` and the updated page is returned.

Access control and provenance policies interact in other situations as well. For example, when the access control labels of a particular page are modified using the CPA interface, this must be done through the corresponding `relabelBlock` policy. In addition to modifying the labels, this policy adds a provenance label of type `Relabel`, recording the standard meta-data as well as the details of the new access control label. This ensures that all relabeling actions are logged, and that it is possible to view the complete security history of a document, not just the current set of access controls.

The CPA permits copying a block from one page to another, based on Buneman et al's tracking of copy/paste operations in curated databases [2]. The copy/paste policy ensures that the provenance information associated with the original document will be available at the destination. Since the data is copied, it allows derivative pages that have increased or decreased levels of access control. This could allow users with read-only access to "fork off" their own branch to edit, or allow a group to copy a version such that the original authors cannot see it (e.g., to use in a classified report). Due to the provenance system, all the prior history of the forked document remains available.

3. Directions for Research

We are currently exploring several extensions to this basic model.

The combination of access control with provenance in the CPA is a very simple instance of policy composition. While both policies are enforced in the program, the labels of one policy do not influence the decisions made by the other policy. We are currently implementing a history-based access control policy in the CPA where the provenance of a document influences the access control policy that applies to that document. We are also implementing a model for protecting the provenance labels themselves (rather than the data) with access control policies. For example, we could restrict viewing provenance labels to users in an *auditors* group.

While SELINKS permits arbitrary forms of policy composition, reasoning about non-trivial compositions is difficult. We are considering a variety of information flow analysis techniques to ensure

```
fun applyWriteToBlock(cred, f, path, page) policy {
  var (l, oldBlk_l) = getBlock(cred, page, path);
  var newBlk_l = applyWrite(cred, f, l, oldBlk_l);
  dbreplaceBlock(cred, oldBlk_l, newBlk_l)
}
```

Figure 4. A policy that controls write-access.

that policies do no interact badly. Ultimately, we hope to partially automate reasoning about the correctness of policy encodings.

We have not yet implemented support for arbitrary queries over the provenance data; we expect such queries will better illustrate interactions between provenance and other policies. For example, if users search for all files they have modified, they will not find any documents where their access has been removed since the modification. Arbitrary querying is a prerequisite to any kind of provenance-based access control mentioned above.

The provenance labels in the CPA only track events on documents and direct data flows to and from other documents. More generally, data can also be influenced by indirect dependences on other data—i.e. implicit flows. We have shown how such policies can be encoded in SELINKS. However, programming while tracking (and perhaps restricting) implicit flows can be tricky. To ease this burden, we are investigating an approach based on program rewriting that, given a policy specification, automatically transforms a program to insert the appropriate label manipulations.

Finally, since Links is a multi-tier language, we are keen on combining traditional techniques of provenance tracking through database operations with our language-based approach. Our current approach to storing provenance labels is fairly inefficient, as all labels are stored as marshaled data structures. One possible enhancement is using a provenance-enhanced database to perform all queries, thus storing the provenance information as meta-data at the database level, and converting it to provenance labels when the data is accessed at the language level.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *NDSS*, 2003.

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

[3] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis, 2007.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. `http://groups.inf.ed.ac.uk/links/papers/links-esop06.pdf`, 2006.

[5] All news, all about intellipedia! `http://www.esenai.com/blog/intellipedia/`.

[6] N. Swamy and M. Hicks. Fable: A language for enforcing user-defined security policies (extended version). University of Maryland, Technical Report, CS-TR-4876; `http://www.cs.umd.edu/projects/PL/fable/tr.pdf`.

[7] N. Swamy, M. Hicks, and S. Tsang. Verified enforcement of security policies for cross-domain information flows. In *Proceedings of the 2007 Military Communications Conference (MILCOM)*, Oct. 2007. To appear.