Building Secure Systems with LIO (Demo)

Deian Stefan and David Maziéres

Stanford University

{deian,⊥}@cs.stanford.edu

Abstract

LIO is an information flow control (IFC) system. In this demo, we give an overview of the Haskell LIO library and show how LIO can be used to build secure systems. In particular, we show how to build secure web applications with high-level data-security policies and describe how LIO automatically enforces these policies.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: Protection Mechanisms

General Terms Security, Languages

Keywords Security, LIO, DCLabels, Hails, IFC, Web

1. Introduction

Building software systems is a challenging, error-prone task. As the number of recent vulnerabilities in SSL libraries and the severity of bugs—where a single line change can introduce a vulnerability [1]—show, building secure systems is harder still. Unfortunately, as we have learned over again (e.g., recently by the popular bug in GitHub's form-submission code [2]) such security-related bugs are inevitable even if developers are careful and use safe, high-level languages. Indeed, this is because only a small fraction of programmers are equipped to write secure code. How then can we expect the average developer to build secure systems?

One approach to bridging this security gap is to use information-flow control (IFC) [2]. IFC tracks and controls the flow of information through a system, according to a security policy. By ensuring that the policy associated with a piece data is always enforced, most code in a IFC system can be considered untrustworthy. Indeed, a typical IFC application is composed of two parts: a small trustworthy component, where the security policy is specified, and the rest of the application logic, which can be untrusted.

Consider, for instance, a conference review system where reviewers are expected to be anonymous and users in conflict with a paper are prohibited from reading specific committee comments. Here, the site administrator is trusted to specify such policies on data. However, the rest of the application, which may fetch data from the network, read reviews, etc., can be built by untrustworthy developers (even a conflicting reviewer)—the underlying runtime system ensures that the confidentiality and integrity of a user's reviews will be preserved. In an IFC system, bugs that appear in this part of the application are simply *that*—bugs—not vulnerabilities.

In this demo, we describe one such IFC system, called LIO [3, 4]. LIO is implemented as a Haskell library, and leverages Haskell's monadic approach to encoding side-effects as a way to control how (and if) information enters/exists the system. Specifically, LIO provides an `LIO` monad in which all side-effects are mediated according to IFC. Hence, an `LIO` computation can perform arbitrary, complex effects, as long as it does not violate the confidentiality or integrity policy on data.

2. Overview

Like other IFC systems, LIO tracks and controls the propagation of information by associating a *label* with every piece of data. A label encodes a security policy as a pair of positive boolean formulas over *principals* specifying who may read or write data. For example, a review labeled `"alice" \/ "bob" %% "bob"` specifies that the review can be read by user `"alice"` or `"bob"`, but may only be modified by `"bob"`. Indeed, such a label may be associated with `"bob"`'s review, for a paper that both `"bob"` and `"alice"` are reviewing.

Our library associates labels with various language constructs, including references, channels, and files. Moreover, we provide a type, `Labeled`, which can be used to explicitly label individual Haskell terms; since LIO is a library, terms that are not explicitly `Labeled` are associated with the label of the current computation (described below). We use such `Labeled` values to protect reviews.

[1] `https://www.imperialviolet.org/2014/02/22/applebug.html`

[2] `https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation`

PLAS '14, July 29, 2014, Uppsala, Sweden.
Copyright © 2014 ACM 978-1-4503-2862-3/14/07...$15.00.

Labels on objects are partially ordered according to a *can flow to* relation $\sqsubseteq$: for any labels L_A and L_B, if $L_A \sqsubseteq L_B$ then the policy encoded by L_A is *upheld* by that of L_B. For example, data labeled $L_A =$ `"alice" \/ "bob" %% "bob"` can be written to a file labeled $L_B =$ `"bob" %% "bob"` since L_B preserves the secrecy of L_A. In fact, L_B is *more* restrictive, as only `"bob"`—not both `"alice"` and `"bob"`—can read the file. Conversely, $L_B \not\sqsubseteq L_A$, and thus data labeled L_B cannot be written to an object labeled L_A.

It is precisely this relation that LIO uses to restrict the effects of computations executing in the `LIO` monad. The `LIO` monad encapsulates a computation that executes in Haskell's "default" `IO` monad, associating with it a label—the *current label*—that tracks the sensitivity of all the data that the computation has observed. To illustrate the role of the current label, consider the code below that reads `"bob"`'s private review and tries to leak it into `"alice"`'s reference.

```
-- Current label: public == True %% True
bobReview <- readFile "/reviews/bob/5.txt"
-- Current label: "bob" %% True
writeLIORef aliceRef bobReview
-- Fail: "bob" %% True ⋢ "alice" %% "alice"
```

Here, the current label is first raised by `readFile`, reflexing that `"bob"`'s sensitive information was incorporated into the context. Importantly, however, this label is also used to subsequently restrict writes; in this case, the `writeLIORef` action throws an exception, since the write is unsafe.

In general, IFC enforcement in LIO follows this approach of exposing functions (e.g., `writeLIORef`), which inspect the current label and the label of the object they are about to read/write as to uphold the *can flow to* relation. We solely rely on Haskell's monad support as a way to define a sublanguage for which we can enforce IFC. And, by ensuring that untrusted code is written in this sublanguage, i.e., it cannot execute arbitrary `IO` actions, we can incorporate arbitrary untrusted code to compute on sensitive data. For example, as we will show, our conference review system can incorporate code provided by users of the system without fear of leaking reviews or reviewer identities, all while allowing the code to interact with the external world (e.g., using the HTTP client).

3. Automatic data labeling

LIO guarantees that code executing in the `LIO` monad cannot violate the confidentiality and integrity restrictions imposed by labels. Thus the untrusted parts of an application can be implement almost carelessly—LIO ensures that bugs do not escalate to vulnerabilities. Unfortunately, the trusted part of an application, that of assigning appropriate labels to data, is still a challenge. And, while using a simple label model such as DCLabels can address certain pitfalls, a non-expert approach to setting labels is desirable.

In the context of web applications, we present a declarative policy language, similar to that used in Hails [1], which makes this task more tractable. Specifically, since data models for web applications are typically specified in a declarative form, and, in many applications, the authoritative source for who should access the data resides in the data itself, our policy language leverages and extends these ideas directly. In our system, labels are specified as read and write clauses alongside the data model, and in terms of it.

Consider the definition of the `Review` data type used in our conference review system:

```
data Review = Review { _id    :: ID
                     , paper :: ID
                     , owner :: User, ... }
```

To associate a label with a review we can leverage the information present in the record type. Specifically, we can specify that the only user allowed to modify such a review is the owner of the review, and that the only users allowed to read such a review are the owner and other reviewers of the same paper. The latter declaration requires that we perform a lookup, using the paper id of the current review, to find the other reviewers. Below is the policy code specifying this.

```
policy :: HailsDB m => Review -> m DCLabel
policy rev = do
  let me = owner rev
  reviewers <- findReviewersOf $ paper rev
  makePolicy $ do readers ==> me \/ reviewers
                  writers ==> me
```

This function is self-explanatory; we only remark that the function takes a `Review` and returns a `DCLabel` in a monad `m` that allows code to perform database actions (in this case the `findReviewersOf` action), a change from the original pure policies of Hails.

While some care must be taken to ensure that the specified policy is correct, the extend of understanding a security policy in such LIO/Hails applications is limited to such functions. It is these policy functions that our database system uses to label reviews when a fetch, insert, or update is performed. Indeed, as we will see in the demo, the core of the conference review system does not manipulate labels; high-level APIs leveraging this automatic labeling approach make most of the IFC details transparent.

References

[1] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th OSDI*, pages 47–60. USENIX, 2012.

[2] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1):5–19, 2003.

[3] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, 2011.

[4] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ICFP*, pages 201–214. ACM SIGPLAN, 2012.