FriendStar: Extensible Web Applications with Information Flow Control

Amit Levy *Ali Mashtizadeh*
Stanford University

1 Introduction

Web applications are increasingly becoming the primary curators of personal and corporate data. Social media applications like Facebook, LinkedIn and Twitter have transformed how users communicate with each other, while online document suites like Google Docs or Docs.com have made online collaboration the norm. Much of the success of such Web applications is due to the flexibility in allowing third-party vendors to extend user experiences. For example, Mint.com has been able to leverage the availability of banking data online to provide users with better ways of managing and visualizing their money.

However, today's web application APIs provide too little extensibility at too high a privacy cost for the next generation of web application extensions. Commonly, web application platforms find a middle ground between restricting access to data by third-party applications and requiring users to disclose private information: sacrificing both extensibility and privacy. This is a fundamental tradeoff in current architectures since third-party vendors must be trusted completely with data they have access to. For example, it is impossible for users to use an external photo editing application like Picnik while having the guarantee that their private Facebook photos won't be misused. In the social web, data belonging to one user might be accessible to an application installed by another user, introducing even more complexity.

In this paper we describe FriendStar, a web platform built in Haskell that uses information flow control [1] (IFC) to enforce policies on untrusted code. IFC allows users to specify policies in terms of where data can flow instead of what code is privileged to access it. For example, Alice can allow Bob to access her photo albums, while preventing any of Bob's applications from leaking her photos to other users or external servers. To enforce these policies, every object is labeled, allowing the system to verify information flow is not being violated at the boundaries, e.g. the file system, network or database.

We use the Labeled IO (LIO) library for Haskell [2]. This packages provides a framework for information flow control in Haskell at the library level, with no modifications to the compiler. Using LIO we can build a trusted authentication mechanism, leaving actual application implementation to untrusted code, including any third party extensions.

In order to demonstrate our system we built an extension called Pleaserobme that attempts to steal a users location and leak it to an external server. FriendStar's enforcement of IFC policies guarantees that this extension is prevented from leaking data. Information leakage is only permitted for those users who explicitly allow it.

2 The FriendStar Application

We used a prototype application platform called FriendStar to help guide our design of an IFC system for the Web. FriendStar is similar in principle to Facebook [3]. End-users have *profiles* containing personal information such as their name, picture and current city. Some of the information on a user's profile is public (e.g., their name and picture), while some is private (e.g., current city) and thus should only be shared with certain other users and should not be leaked to the network. Users also have a ''wall" on which they, or other users, can post messages. Users can connect to each other by claiming each other as ''friends" – a symmetric relationship implying they would like to share, e.g., private information.

FriendStar allows developers to enhance it by writing extensions. This code is compiled into the FriendStar binary, but is obviously untrusted. The main contribution of FriendStar is demonstrating that such extensions can be given full access to user data while ensuring that they do not violate the users privacy policies.

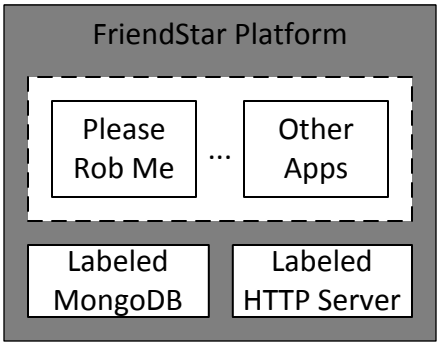

Figure 1: The FriendStar platform consists of a labeled HTTP library and labeled MongoDB database for persistent storage. The users privacy policy is enforced even on the untrusted third party applications such as please rob me.

3 Design

Our design focuses on a small trusted computing base built on top of several existing Haskell libraries. Figure 1 shows the general architecture diagram. We have a Labeled MongoDB database and a labeled HTTP server that form the foundation of our social networking platform. Our implementation uses IterIO for reading and writing to sockets and for parsing HTTP requests, LIO as the basis for enforcing information flow control, blaze for building HTML views and MongoDB bindings to communicate with the database.

3.1 DC Labels

For our labeling scheme we chose DC Labels [4] for their flexibility in expressing IFC policies, and because an implementation already exists for the LIO library. A DC Label is composed of secrecy and integrity components. Each component, in turn, is a conjunction of disjunctions of basic types (e.g. Haskell strings).

With disjunctive labels we found we were able to easily express the policies we wanted for our system in terms of labels. For example, denoting that a user's friends are able to read her current location is as simple as creating a disjunction of the usernames of those friends.

In FriendStar we chose canonical, human-readable names for the labels of entities. For example, each subcomponent of the application has a corresponding label named by it's URL prefix and users have corresponding labels named by their usernames.

3.2 HTTP Server

We use the IterIO HTTP library to build a trusted HTTP server. At the core, the HTTP server parses incoming requests and passes them to the untrusted components, enforcing IFC at the boundary between untrusted code and the network. To this end, our library adds user authentication and integration with LIO on top of the existing IterIO library.

3.2.1 User Authentication

Authentication must be handled by a trusted component in our system, as the label of the TCP connection to the client browser is set based on the users credentials. We chose the Basic Access Authentication scheme. With every request, the client sets the HTTP header "WWW-Authenticate" to a base-64 encoded authentication string of the form "username:password". Typically web servers authenticate users using a database or *.htpasswd* file. In FriendStar we chose the latter approach. If the header is not present in the HTTP request, the server responds with status code 401 (unauthorized), signaling to the user they must authenticate. The actual interface exposed to the user is browser specific, but typically browsers present a dialog box prompting the user to for a username and password.

We chose HTTP Basic Authentication for simplicity, but our library could trivially be expanded to support Digest Authentication, or even session-based authentication through HTML forms. For session based authentication it may be useful to allow untrusted components of the site to serve and parse the HTML forms – for example, to allow for a variety of login interfaces, or simply to minimize the trusted base. This could be accomplished by assigning a unique label to every HTTP request from unauthenticated users, effectively preventing untrusted code from writing any data from the request (e.g. the password) to the database or file system, or leaking it to the network.

3.2.2 Integrating with LIO

With IterIO it is trivial to construct an *Iter* and *Onum* pair corresponding to the input and output of a socket, respectively. However, the types of these are:

- *Iter L.ByteString IO ()*
- *Onum L.ByteString IO L.ByteString*

This means that they are allowed to perform operations in the *IO* monad, which is unacceptable for untrusted code. Instead, we want an *Iter* and *Onum* that are allowed to perform operations in the *LIO* monad. We accomplish this by wrapping both with trusted code. For

example, we consumes output from the *Onum*, execute any IO actions with *ioTCB* (e.g. reading from the underlying socket), and use the output to construct an *Onum* that artificially wraps the *LIO* monad.

We use the trusted code wrapper as an opportunity to set the clearance, privilege, and initial label of the untrusted code. We use the logged in user's label for the clearance, a prefix of the HTTP request path for the privilege (e.g. an application that lives under */profiles/* will get a privilege labeled "profiles"), and set the initial label to *public*.

Once the LIO environment is set up, we can safely execute the untrusted code. Untrusted handlers have the type signature *HttpReq () → Iter L.ByteString DC (HttpResp DC)*, so we can pass it the parsed HTTP request (after scrubbing any sensitive authentication data) and fuse it with the request body to get back an *HttpResp DC*. We check that the label after executing the untrusted code can flow to the user's browser, and either send the response to the client, or respond with an error status (500) if information flow would be violated.

3.3 Database

Our database layer is trusted, and wraps the Haskell MongoDB library with an easy interface for accessing deserialized data types and enforces IFC policies on user data. In particular, the database layer exports a set of queries such as *findProfileByUsername* that fetches BSON from the database, deserializes it into application data types and applies labels to individual columns in the data type based on a set of pre-specified policies.

We do not yet protect writes to the database, however our design supports it. Database write protection can be implemented by allowing the application to raise it's integrity such that it could write to a user's data only under certain conditions (for example, only if the logged in user is making the request).

3.4 Network

Some components of the application may need to communicate with external network services, but cannot be given direct access to *IO*, we implemented a labeled HTTP library. Our library allows untrusted code to execute arbitrary HTTP requests so long as they have not viewed labeled data. The implementation checks the current label before executing any HTTP action, and aborts if it cannot flow to the public label. Therefore, untrusted code can only leak sensitive information to the network if it has privileges to declassify it.

While we do not currently support such functionality, it would be possible to let users specify particular HTTP endpoints (e.g. a URL or domain name) to which untrusted code should be allowed to leak sensitive data. For example, users might want to allow untrusted code to leak their current city to the Google Maps service, located at *maps.google.com*. This could be enabled by adding the domain or URL to the disjunction of the label of the sensitive data, and modifying our network library to verify the flow satisfies this new, more lenient constraint. For example, the following flow is valid: $jdoe \vee maps.google.com \sqsubseteq maps.google.com$

3.5 REST Controller

We structured FriendStar into a set of RESTful style controller modeled after Ruby on Rails. This structure is implemented entirely in untrusted code – new components of FriendStar need not follow this pattern. Our RESTFul controllers are instances of a class that requires a method for each RESTful action (index, show, create, destroy, update, edit and new). These methods return a *RestControllerContainer* monad, which is a transformation of the *State* and *LIO* monads, which holds in it's state the Http request and response.

Figure 2 shows an example of what an instance of this class might look like. Notice that the internals of IterIO are abstracted away. *ToyController* uses methods such as *redirect* and *params* to get or transform the state of the Monad. We have baked in such methods that we found useful, but developers can easily extend their applications with more. Our *RestController* module also exports methods that add the appropriate routes to an IterIO route map. We implemented the entire FriendStar web application using instances of this class, and found that developing within this model was much faster than writing raw IterIO *HttpRequestHandler*s.

4 Results

We built an example untrusted extension called *Pleaserobme*, which we loosely based on a real Twitter application of the same name [5]. This extension allows users find the current city of other users (or themselves) on FriendStar. Given a valid FriendStar username, Pleaserobme will display that user's current city if the client is allowed to see it (otherwise the extension will timeout as it will have violated information flow).

In addition to displaying this sensitive information to the client, Pleaserobme also attempts to leak it to an external server. Since it cannot execute *IO* actions, its only option is to use the labeled networking library. However, because it has to raise its label in order to read the current city value, the networking library will reject the request to communicate with an external server, thus thwarting the attempt to leak information to the network.

```
newtype ToyController = ToyController Int

instance RestController ToyController where
  restIndex self = do
    toys <- lift $ allToys
    render "text/html" $ toJson toys

  restShow self toyId = do
    toy <- lift $ findToyById toyId
    render "text/html" $ toJson toy

  restUpdate self = respondWith 500 $
                      "Updates not allowed"

  restCreate self = do
    let newToy = toyFromParamList params
    toy <- lift $ createToy toy
    redirect "/toys"
```

Figure 2: Example RESTful Controller. Imports are excluded for brevity.

Users are able to grant permission to the extension to leak their information. We accomplish this by allowing users to add the extension to the disjunction forming the label for their current city. This allows Pleaserobme to declassify that user's current city using the privilege given to it by the HTTP library. Once the information has been declassified it can successfully be leaked to the network.

5 Related

There are countless examples of extensible web application. However, to our knowledge, all such systems take a fundamentally different approach than ours. As a representative example, the Facebook Developer Platform allows external servers to retrieve private user data given explicit permission from the user (or sometimes one of their *friends* or *friend-of-friends*). This approach forces the developer extensions to be totally trusted as they gain complete control over the data. In FriendStar we choose to execute untrusted code on trusted servers, allowing us to enforce security policies throughout the lifetime of the data.

Jif [6] provides language level decentralized information flow control. In Haskell though we are able to enforce the same programmatic behavior by just extending the language through a library (LIO). Jif also has the limitation that it assumes only static labels and no dynamic labels.

Asbestos [7] and HiStar [8] are two operating systems that implement decentralized information flow control to enforce policies on applications. Unlike our implementation that relays on LIO it can only enforce policies at a higher granularity. An application that has a user's data would have to fork and exit when it is done, otherwise it would contain tainted data. Our system typically does not require processes to fork and exit, as it works at the language level.

6 Conclusion

We presented FriendStar, an extensible web application made up of untrusted components. FriendStar shows that information flow control can be leveraged to allow secure execution of untrusted code over sensitive data on a web platform. We demonstrated this property with an example untrusted extension which attempts to leak sensitive user data to the network. Finally, we've argued that the security policies in such a system map naturally to users actual privacy concerns. Our code is available for download at http://github.com/alevy/friendstar

7 Acknowledgements

We thank Deian Stefan for his help understanding the LIO library and designing the database model, David Mazières for writing the IterIO and LIO library, David Mazières and Bryan O'Sullivan for teaching a wonderfully informative class, and James Blake for fueling late night writing with monotonous music.

References

[1] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *In Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pp. 129–142, 1997.

[2] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, (New York, NY, USA), pp. 95–106, ACM, 2011.

[3] M. Zuckerburg, C. Highes, D. Moskovitz, and E. Saverin, "Facebook.com."

[4] D. Stefan, A. Russo, and J. C. Mitchell, "Disjunction category labels," in *Proceedings of NordSec 2011*, NordSec '11, 2011.

[5] B. Borsboom, B. v. Amstel, and F. Groeneveld, "Please rob me."

[6] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, p. 2000, 2000.

[7] P. Efstathopoulos, M. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazires, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *In Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, pp. 17–30, 2005.

[8] N. Zeldovich, S. Boyd-wickizer, E. Kohler, and D. Mazires, "Making information flow explicit in histar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 263–278, USENIX Association, 2006.