CSCI 2390 SafeServer

Archer Wheeler
Brown University

Abstract

I implemented a Haskell toolkit, SafeServer, for statically enforcing privacy policies of a web server. SafeServer uses Haskell's strong type system to enforce data flow and catch accidental data breaches. User's need to only to define a small Policy instance that constructs data views for potential users. As long as the policy code is trusted, and application programer is not actively malicious, then SafeServer enforces that data derived from private information can only be shown to appropriate users.

SafeServer is implemented on top of Servant Server [1], a web framework for Haskell, which statically enforces the *type* of the API constructed. SafeServer uses this to statically describe each API endpoint and the privacy policy used in evaluation. In theory, a non malicious site could publish its privacy policy code to hold itself accountable to any changes.

1 Introduction

There are a number of practical ways to enforce privacy policies in web servers. However, many methods require privileged modifications to the runtime and enforce privacy dynamically at runtime.

SafeServer, however, is purely a standard Haskell library and enforces privacy policies *at compile time* through Haskell's type system. The advantage is that the application will raise problems during development rather than raising runtime exceptions. In addition, SafeServer has negligible overhead as there is no modified runtime which needs to examine and verify untrusted data. However, SafeServer does enforce constraints on implementations which will likely significantly impact performance.

Importantly, SafeServer is not designed to abstract away concerns of user privacy from the application programer. Instead of magically handling privacy behind the scenes, SafeServer breaks programs which do not figure out away to correctly implement the specified privacy policies. Because of this, SafeServer is not really a tool designed to factor out privacy concerns from application design. Instead, it is a tool that provides a mechanism for verifying (and statically enforcing) that privacy is upheld. Additionally, while resistant to abuse, SafeServer does require that the application programmer include necessary boilerplate.

SafeServer does provide helper functions and tries to match common idiomatic patterns in Haskell. Because of this, I hope that using SafeServer is not difficult to use per se, but it does add an additional (and unavoidable) burden on the application designer. There is certainly room for additional abstraction and quality of life, but that has not been my primary focus.

2 Background

Information Flow Control (IFC) is a semantics system which enforces that data in an actor type model will only flow to approved actors via a privacy policy attached to the data. One such implementation is the operating system DStar [4]. One of the benefits DStar gains from using IFC is a strong threat model that protects against malicious processes assuming an uncorrupted kernel. However, full IFC guarantees are quite cumbersome and perhaps unnecessary in settings without the need for an extreme threat model.

One practical use case is to provide privacy policy enforcement for a web server when the application programers are non malicious, but wish to prevent bugs or sloppy code from accidentally leaking private user info. Suppose you are running a small company operating a website. You want to hire an intern, but also don't want them to be able to write sloppy code that could leak customer information.

SafeServer was inspired by two systems, Resin and Jacqueline [2, 3], both of which were designed to solve this kind of problem. In both systems users define a small subset of "policy code" which defines the privacy policy for the framework. This code is expected to be trusted and bug free (e.g. inaccessible to the intern)

Other programmers like the intern can then write normal "application code" which may contain bugs but is non malicious. This paradigm isolates code that explicitly handles privacy from the code that performs application logic.

Resin solves this problem by modifying a python runtime to label runtime data with a source an accompanying privacy policy. Resin then propagates the label to any derived values and checks the corresponding labels of data before sending the server response. Jacqueline works similarly, but attempts to create an environment in which the application programer does not need to be aware of how the privacy policies are implemented. Jacqueline modifies the python runtime so that results of database operations (using a special purpose ORM) are lazily evaluated to different results based on the user token.

SafeServer is written in Haskell and takes advantage of its strong type system to check privacy at compile time rather than at runtime. Haskell has an idiomatic typeclass called "Monad" which as a byproduct of a more general semantics effectively "taints" all information contained within. Haskell uses this construct to encode its “IO” type and enforce that any function whose result is not of the IO type will have no side effects. A pure function cannot access data contained in an IO result without itself becoming tainted by the IO type. SafeServer leverages this power to enforce that private info remains tainted by user defined privacy policies.

3 Design

I implemented both a small library, SafeServer, intended to be a general toolkit and a small example program which makes use of it. As of now, SafeServer itself is only implemented as a Haskell Module rather than a full package. Semantically there should be little difference.

Additionally, I went through quite a few different iterations on ideas before landing on my final implementation. Most of the challenge for this project was figuring out the correct way to build SafeServer rather than on implementing it once I realized how to do it. I detail some of the failed attempts and various problems in Section 4.

3.1 Tracking Taints with Types

SafeServer introduces two types

```
newtype PI p a = Box (Text -> Maybe a)
data Safe p a = SafeBox a | Unsafe
```

to taint and track private data. The `p` types encode the policy definition for that private info. For now, it is safe to ignore `p` and assume a single policy definition.

The first type encodes private info which the programmer may work on. The second, is a type that effectively freezes a result derived from a corresponding `PI` type and marks it as available to be sent to a user.

The type `PI p` is an instance of a Monad and so implements idiomatic combinators that let the application programmer operate directly on the "internal" `a` type. However, it is impossible to derive a result from `a` that is not itself contained inside of the `PI` wrapper type.

Internally, the `PI` type acts as a lazy evaluation that given a user token of type "Text" will optionally return the corresponding value `a`. An optional `Maybe a` is used since the policy definer may wish to disallow access of the data to specific users. Likewise, the corresponding `Unsafe` results encodes a final result that should appear as empty. These internals are inaccessible and abstracted from the user.

The corresponding `Safe` type provides no tools for manipulation since it is intended to be directly serialized and return to the user via Servant.

3.2 Haskell WAI Applications and Types

Ultimately the biggest challenge was enforcing that the final server application was correctly typed. Like C, every Haskell program must include a Main function:

```
main :: IO ()
main = ...
```

The type `IO ()` means that when run, the program will have some side effect and return the "unit" value, or effectively nothing. From Haskell’s perspective `IO ()` is the *only* type that makes sense for a compiled program invoked by the operating system. For instance, if the type were not `IO` than it could not perform necessary side effects such as print to stdout or even return an error code on completion. Additionally, once the Haskell program finishes and the runtime is stopped the value "inside" of the `IO (...)` is meaningless to the OS.

However, with SafeServer we want the program to only define a correctly build privacy preserving server. When I started I assumed there was a simple solution as long as the program conformed to some boiler plate in the form of:

```
server :: SafeServer
safeServe :: SafeServer -> IO ()

main :: IO ()
main = safeServe server
```

I expected it would be easy to build such a wrapper function which enforced the types lined up. However, the input `server` needs to be the final definition of the web server and is not a simple as an ordinary pure Haskell function.

As far as I am aware, every Haskell web server framework is built on top of the WAI (Web Application Interface) library which provides low level abstractions for building a web server. However, since all information return by a web server must be encoded into a bytestring, WAI throws away the type that encodes the "meaning" of the response. For general applications this is totally fine, however, SafeServer needs the `SafeServer` type to encode only servers that return safe data. You don’t want your intern being able to access the byte string encodings of the final result.

The two "obvious" ways to solve this problem is to either redefine the low level WAI application type or to define a new

routing framework that constructs a WAI application of type `SafeServer`. Both, however, would require significant work that is tangental to the privacy goal and limit the programmer to functionality recreated by SafeServer.

Luckily, however, there is an existing and robust framework, Servant Server [1], that solves some of these problems. In Servant, the users define a type for an API that specifies the structure and return type of each endpoint. Then the user implements a server that matches this type.

SafeServer, gives the user the tools to define endpoint whose results derive from a specific privacy `Policy` type. Let's start with a simple example of a servant type.

```
type API =
  "user" :> QueryParam "tok" String
  :> Get '[JSON] User
```

This defines an api endpoint of the form "/user?tok={tok}" where "tok" is a query parameter. It also defines that the response from the server will be a `User` datatype serialized into JSON.

In SafeServer one could instead define a "safe" endpoint:

```
type API =
  "user" :> QueryParam "tok" String
  :> GetJSON MyPolicy User
```

The difference is that this endpoint will return a `User` datatype in JSON and observes the policy defined by the `MyPolicy` instance.

The type `GetJSON` is an alias defined as

```
GetJSON MyPolicy User =
  Get '[PJSON] (Safe MyPolicy User)
```

This means that the response is `Safe MyPolicy User` which must be derived under `PI MyPolicy`. It is encoded into `PJSON` which is equivalent to `JSON`. The dummy type `PJSON` is used because the type is "hidden" in the SafeServer module so the caller will not misuse it to decode private data into JSON.

3.3 Policy Definitions

In order to use SafeServer the user must define a `Policy` instance that governs how `PI` data is generated. The policy code must be trusted and bug free e.g. not written by the intern. Users, for instance, may wish to define a specific policy file and require special privileges in their source control to modify the file.

To define a policy, a user constructs a dummy datatype of the form

```
data MyPolicy
```

The policy datatype itself holds no value and the compiler will ensure that it doesn't actually appear at runtime. However, the *type* is used to differentiate user policies. Users define policies as:

```
instance Policy MyPolicy where
  policy :: Proxy p -> Text -> (Text -> Maybe Text)
```

The value `Proxy p` is required by the compiler, and is simply a dummy value. This is a not uncommon Haskell idiom. Given this definition, SafeServer provides a corresponding `load` function.

```
load :: Proxy p -> FilePath -> IO (PI p Text)
```

Notice, that the policy used in loading the file is captured in the type `PI p Text`. This policy remains fixed even if the `Text` value is operated on. Likewise, the policy is defined by the endpoint representation for the Servant server. Therefore the application programmer must design their program so that they return the correct policy data to the correct endpoint.

Importantly, Haskell enforces that there can only ever be one instance definition per datatype. Therefore, as long as the policy code defines an instance for its policy, another module could not overwrite a new incorrect definition for the same policy. A user could define a new policy and policy instance, but then they would also have to change the corresponding endpoint type.

4 Implementation

I spend a considerable amount of time during the implementation of SafeServer trying different attempts to correctly construct SafeServer. Therefore, while my final implementation is not extensive in length, I went through a number of failed implementations before arriving at my result. For instance, I begin by using the Spock server framework. However, without the API type power of Servant, I found it difficult (if not impossible) to enforce that the user defined type actually returned only safely handled data.

In Spock the route defining combinators require that each endpoint evaluate to a unit. Therefore, I could not wrap the endpoint constructors in a way that required the results were actually safe results.

One approach I tried was to augment the state of the Spock server to encode safe responses with a "side effect" result. However, none of the tools provided by the framework correctly lined up in a way that wasn't easy for an application programmer to overwrite. Servant cleanly solved some of these problems, but it too required some ingenuity and research to correctly build.

5 Evaluation

The accompanying SafeServer repository contains a simple web server proof of concept. The server is capable of returning data through a standard REST api and will correctly show different data per endpoint depending on the user token supplied.

It would be difficult to evaluate the performance of SafeServer since the overhead does not come from internal SafeServer code, but rather from how the user must build the server to correctly compile. Because of this, there is no clear baseline to form a comparison.

For instance, in my SafeServer proof of concept in order to count the number of files in a directory it must load each file individually. A server built without SafeServer could directly query the directory size from the OS which would be much quicker. However, without external information, that implementation would need some way to determine which files a user should be aware of.

6 Conclusion

I present SafeServer, a Haskell toolkit that statically enforces privacy policies. SafeServer is still largely a proof of concept, and would likely need additional ergonomics before it would be useful. Additionally, right now there are likely ways of tricking SafeServer into returning unsafe data. I expect that SafeServer could be improved to be more secure, and I hope that it shows that this approach is at least feasible.

References

[1] Alp Mestanogullari, Sönke Hahn, Julian K Arni, and Andres Löh. Type-level web apis with servant: an exercise in domain-specific generic programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2015.

[2] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. *ACM SIGPLAN Notices*, 51(6):631–647, 2016.

[3] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.

[4] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008.