SafeServe: User-Editable Websites in Haskell

John Hiesey
jhiesey@cs.stanford.edu

December 17, 2011

1 Introduction

Traditionally, websites are written in a scripting language, such as Python, Ruby, or PHP and edited in a standalone text editor before being manually uploaded to the server. The owner of a server must either trust the developers of the site, or carefully use operating system-level permissions to ensure that the site cannot harm the rest of the system. Therefore, it is not very safe to allow untrusted users to edit sites written in traditional web scripting languages from a browser.

As an alternative, Content Management Systems (CMSs) such as WordPress, allow even non-technical users to edit sites through a web browser. Unfortunately, these systems don't allow any changes to be made to the behavior of a site, only to the content. This essentially precludes their use for complex interactive sites without writing or installing third-party plugins, which also compromise the safety of the system, especially if their authors are not trusted.

Using SafeHaskell, however, we can allow users to edit the server-side code while ensuring at the language level that whatever code users write will be safe to execute. This form of language-level safety also limits the attack surface for would-be attackers, compared to other techniques that require much more careful code inspection or reliance on operating system security mechanisms. This is the goal of SafeServe.[1]

2 SafeServe User Interface

SafeServe can be run from the command line as safeserve, and once started, listens (currently on port 3000) for HTTP requests. A site can be created or edited simply by visiting `http://<hostname>/edit/<site_name>` in a browser, where `<site_name>` is the name of the site the user wants to create. Since this will also be the name of the associated Haskell module, it must start with a capital letter and contain no punctuation; otherwise, it will be automatically adjusted if possible or rejected with an error.

The user is then presented with an editor, which uses the CodeMirror[2] JavaScript library to make the editor user-friendly, as well as "Save" and "Run!" buttons. The "Save" button

[1]https://github.com/jhiesey/SafeServer
[2]http://codemirror.net/

saves the code in the editor, while the "Run!" button simply points to `http://<hostname>/run/<site_nam`
which is the root of the plugin's site. From there, the user can interact directly with the generated site.

3 SafeServe Application Interface

A SafeServe application is simply a single Haskell source file that follows the following rules:

1. It must be able to be compiled with -XSafe. Any other programs will be rejected.
2. It must define a module of the same name as the site name.
3. It must define an object called "resource" with the following type, defined in the top-level API module:

```
data Interface = Interface {
  function :: Application
}
```

where Application is defined in module "SafeBase.Framework" as follows:

```
type Application = Env -> RIO Response
```

Env and Response are data types corresponding to the information received in an HTML request and provided back in an HTML response, and RIO is a restricted IO monad, which will be discussed in the next section.

Here is an example of a minimal plugin:

```
{-# LANGUAGE OverloadedStrings #-}

module Simple where

import API
import SafeBase.Framework as F
import SafeBase.RIO
import qualified SafeBase.ByteString.Char8 as B

resource = Interface { function = theApp }

theApp :: Application
theApp =  F.safeserve $ do
  F.get "/" $ do
    F.html "<html><head><title>A Minimal Example</title>
    </head><body>Hello, world</body></html>"
```

Within the plugin, the RIO monad can be used for very limited IO, as provided by the functions in the“SafeBase.RIO” module.

Each plugin is provided with a directory which it can access, and access is limited to only this directory. At this point, reading and writing files, as well as getting the current time, are the only permitted operations. Expanding the capabilities of the RIO monad would be a particularly useful way to extend SafeServe in the future.

Various other functionality, most of which is a subset of that provided by the Miku web framework, is also available. A slightly modified version of the Blaze combinator-based template library is also available through importing the SafeBlaze package provided with SafeServe.

4 Implementation

4.1 Existing functionality

A number of existing libraries and frameworks were crucial in implementing SafeServe. An overview of the most important ones are here, see their documentation for more details.

SafeHaskell SafeHaskell[3] is a Haskell compiler extension that ensures that a Haskell program is actually type-safe. The key idea is to reject any program that uses unsafe functions, such as unsafePerformIO or unsafeCoerce, and any program that imports such a module. This checking is enabled with the -XSafe option in GHC.

If a module uses unsafe modules but only exports a safe interface, the module can be compiled with -XTrustworthy to mark it as safe to import into other safe modules.

Miku In order to provide a reasonable interface to the users of SafeServe, we decided to base it on an existing Haskell web framework. Of the frameworks listed on the Haskell Wiki page, the most popular were judged to be too complex to modify for this project. To make SafeHaskell useful, sites must in particular not be allowed to access the IO monad. Happstack and Snap, both popular frameworks, are complex in terms of code size and rely heavily on constant access to the IO monad. These two also rely significantly on Template Haskell, though less so than the popular framework Yesod does, and as a consequence are even more difficult to make safe.

To avoid these problems, we used Miku[4], a relatively simple Haskell web framework that relies only minimally on the IO monad and Template Haskell in its web interface.

hs-plugins Once the user supplies a program to run, it must be compiled and run. In order to provide an easy interface between the SafeServe server code and the user's code, we decided to load and run the user's code inside the main server process.

[3]http://hackage.haskell.org/trac/ghc/wiki/SafeHaskell
[4]https://github.com/nfjinjing/miku

In order to dynamically compile and load haskell code, we used the hs-plugins[5] module. It provides both an interface to GHC to compile the user's source file and, more importantly, an interface to dynamically load the module into the running Haskell process.

The simplest interface to the loader has the following type signature:

```
load :: FilePath -> [FilePath] -> [PackageConf] -> Symbol -> IO (LoadStatus a)
```

where the input types are all newtypes for String. Unfortunately, this function does no type checking, so even if SafeHaskell is enabled when compiling the plugin, safety can be violated if a plugin provides a function of the wrong type, which the server would then proceed to evaluate.

The hs-plugins module has a few options to solve this problem, but the one most useful here is an alternative loading function with the following type signature:

```
pdynload_ :: FilePath -> [FilePath] -> [PackageConf] -> [Arg] ->
  Type -> Symbol -> IO (LoadStatus a)
```

Although this function is still polymorphic (and hence unsafe) in its return type, it ensures that the called function has the type specified by its Type argument, as a string. It accomplishes this by dynamically creating a Haskell source file that attempts to import the plugin file and call the function with the type supplied by the user. The result of running the Haskell typechecker (through invoking GHC again) is used as an indicator of whether the plugin provides the correctly-typed export–if so, that function is loaded using the unsafe load function previously discussed, and if not, the dynamic load fails. As long as we are certain that we supply the Type argument that our code expects, we are guaranteed that the system as a whole remains type safe.

Unfortunately, hs-plugins does not compile properly with GHC 7.2.2. Included in the source is a patch which should fix this.

4.2 Architecture

The actual SafeServe application is itself a site hosted within an unmodified version of Miku, with its main module located in the file SafeServe.hs. This application handles editing, loading, and running the SafeServe sites that it hosts, and also directly serves the static files belonging to each application.

All source files, as well as all of the static files belonging to the plugins, are located in the `plugins/<plugin_name>` directory. Due to limitations on form uploads in Miku, there is currently no way to upload static files for an application. Until this is remedied, these files can be manually put in place (if a user has access to the server), or they can be generated by a plugin program which writes the correct file as a result of its execution. Submitting a patch to the Miku codebase would be the best way to fix this limitation.

To actually run a plugin, the plugin source is compiled and dynamically linked using hs-plugins, as described above. In order to avoid problems with lazy evaluation and reloading plugins, after each call to a plugin (one per page load), the result of the call is forced with

[5] http://code.haskell.org/ dons/code/hs-plugins/

deepseq and then the plugin is subsequently unloaded. Fortunately, the make function only recompiles plugins when the source has actually changed, which keeps page load time from being consistently excessive.

The other substantial component of SafeServe is the SafeBase module, which provides a safe, but substantially reduced, subset of the functionality from Miku. The entire module is compiled with -XTrustworthy, as its interface has been carefully selected to be safe. Nonetheless, ideally SafeBase would be split into two modules, the smaller of which would contain all unsafe imports and be compiled with -XTrustworthy, and the larger of which would import modules from the the first and be compiled with -XSafe. This would provide better automatic checking of the safety of the interface exported by SafeBase.

SafeBase itself contains a few submodules: SafeBase.ByteString is a version of the Data.ByteString package, with all but a few unsafe functions reexported. (the SafeBlaze module is similar, except that it wraps the Blaze HTML template library). The other two SafeBase submodules, SafeBase.Framework and SafeBase.RIO, will be described in the next subsections.

4.3 SafeBase.Framework

This module provides a sanitized and condensed version of the Miku framework. For simplicity, the entire implementation resides in a single module, which provides both the data types and functions that are needed to build a website. In order to avoid the IO monad entirely, many of the core data structures used by Miku are modified, such as Env (representing an HTTP request) and Response (representing an HTTP response). For simplicity of implementation, their fields are also simplified somewhat. A few safe types are, however, directly imported from Miku.

Like in Miku, much of the code resides in the AppMonad, which in this case has the following type:

```
ReaderT Env (ST.StateT Response RIO) ()
```

The State monad keeps allows the code to build up the response, while the Reader monad gives it access to the request. Finally, the RIO monad provides limited IO capabilities.

In order to actually execute the plugins' web handlers, the top-level SafeServe application converts Miku's requests into SafeServe's request format, and then converts the results back into Miku's native format. In order to allow the returned value to be forced with deepseq before the plugin is unloaded, the Response datatype is made an instance of the NFData typeclass.

Despite the relative simplicity of the Miku interface, Miku's dependence on the author's own "Air" Prelude replacement module, which relies on Template Haskell and other unsafe operations, necessitated significant syntactic desugaring and expansion of many of the functions that were copied into SafeBase.Framework. Restructuring this code turned out to be a significant implementation challenge in implementing SafeServe as a whole.

4.4 SafeBase.RIO

The RIO monad is a safe wrapper around the the IO monad, with a hidden value constructor to ensure that no plugin code can convert an IO action into a RIO action. It is implemented as follows:

```
newtype RIO a = UnsafeRIO { runRIO :: String -> IO a }

instance Monad RIO where
  return x = UnsafeRIO (\_ -> return x)
  m >>= k = UnsafeRIO $ \path -> (runRIO m path >>= \x -> runRIO (k x) path)
```

Keeping the value constructor (UnsafeRIO) hidden prevents running a RIO action except by handing it back to a function that resides in the IO monad, which can then evaluate the runRIO function to execute the RIO action.

The value hidden inside the monad, however, is not just an IO action, but rather a function from a String to an IO action. The string is used for keeping track of permissions: in SafeServe, it contains the name of the plugin being executed. This allows files to be read or written relative to a certain directory. For example, writeFileRIO is defined as follows:

```
writeFileRIO :: String -> String -> RIO ()
writeFileRIO name contents = UnsafeRIO $ \path -> do
  cwd <- getCurrentDirectory
  let absPath = normalise $ cwd ++ "/plugins/" ++ path ++ "/" ++ filterPath name
  writeFile absPath contents
```

where filterPath removes ".." path components. Given this definition, if we execute

```
runRIO (writeFile <path> <contents>) <basepath>
```

then no matter what we choose for `<path>`, it will always be treated as a path relative to `<basepath>`. The only way we could access a different directory would be if we called runRIO with a different basepath, but as runRIO is in the IO monad, it cannot be run from inside a plugin, ensuring safety of the code. Similar tricks could be used in the future to provide, for example, database access to only certain databases or tables.

At this point the interface provided by the RIO monad is very limited, consisting only of file reading and writing, getting the current time, and a few debugging functions.

5 Results

Due to time constraints, only a few simple examples were completed. We recommend that the reader install and play with SafeServe to get a better feel for its capabilities. The plugins directory contains a few examples to get the user started.

6 Future Research

SafeServe is, at this point, very much a proof of concept. To make it a viable framework for web development, its capabilities would need to be expanded significantly.

Most importantly, the RIO monad needs to be more powerful. More functions for file IO should be written, and a safe database interface that resides in the RIO monad would be essential before real applications could be developed. Potentially, this could be developed without significant changes to the RIO monad if the entire database engine were put inside the RIO monad and directly accessed the database files, but this seems like an unnecessary limitation. The per-plugin permissions provided by the RIO monad could provide, or even be expanded further to provide, a much richer database interface.

The editor and site building interface could also use some improvement. The ability to upload static files without writing Haskell code that emits such files is essential, as is the ability to upload complete Haskell packages, or at least multiple source files as part of an application.

Finally, the original vision of the project, as a system that allows modules to call *each other,* has not been achieved. Such a system would allow shared copies of libraries to be run from other plugins, and allow direct function calls between services hosting data and client plugins that present this data to users.

7 Conclusions

As a proof of concept, SafeServe is a success. It has proven that allowing users to edit websites written in Haskell within a browser can be done safely while still allowing the sites to perform essential actions, such as storing state in the file system.

With some additional work, SafeServe could become a practical sandbox for users to write websites in Haskell and see their creations run immediately and safely in the cloud. To make this truly useful, SafeServe would need to keep track of permissions between plugins and perhaps also support Distributed Information Flow Control.