Preventing Common Vulnerabilities in Web Applications using Programming Language Abstractions (Invited Paper)

Dachuan Yu, Ajay Chander, and Liang Fang

DOCOMO USA Labs
3240 Hillview Avenue, Palo Alto, CA, 94304, USA
{yu,chander,fang}@docomolabs-usa.com

ABSTRACT

The Web of today is commonly used as a platform for hosting sophisticated applications. A key component of such web-based applications is the programs running on the web servers. With the growing popularity of web applications, especially those which manage sensitive data and perform critical functionalities (*e.g.,* online banking), the security of server programs has become a major concern. The BASS project at DOCOMO USA Labs aims to understand what a suitable model is for programming web applications, and prevent common vulnerabilities using proper programming language abstractions. In this paper, we give a status update on BASS, including a review of previous results, an introduction to a new prototype, and a discussion on ongoing efforts.

Keywords: Cloud computing, programming language abstraction, web application security.

1 Introduction

Cloud computing gives users access to powerful computing resources through relatively lightweight client-side software (e.g., a browser). Among other use cases, it is very helpful in the context of mobile computing—users may leverage the computing power on the cloud to compensate for the physical limitations (e.g., hardware) of mobile devices.

Web applications, as an important component of cloud-based services, are software applications running on the cloud. They alleviate the burden of software maintenance, because code is installed and maintained on web servers. Users access these applications typically through the use of web browsers.

Unfortunately, it is notoriously difficult to program a solid web application. Besides addressing web interactions, state maintenance, and whimsical user navigation behaviors, programmers must also avoid a minefield of security vulnerabilities. The problem is twofold. On the one hand, we lack a clear understanding of the new computation model underlying web applications. On the other hand, we lack proper abstractions for hiding common and subtle coding details that are orthogonal to the functional logic of specific web applications.

The BASS project at DOCOMO USA Labs aims to address these issues using declarative server-side scripting. It allows programmers to work in an ideal world equipped with new abstractions to tackle problematic aspects of web programming. Early results on BASS have been published at WWW'08.[1] In this paper, we review basic BASS ideas, present a new BASS prototype deployed as a Python library, and discuss ongoing efforts on a deployment on Google App Engine.

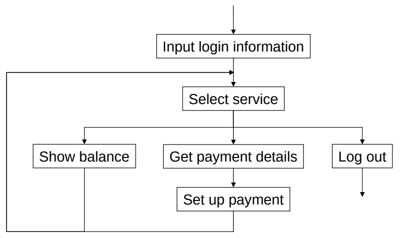

Figure 1: Idealistic work flow of online banking

2 Background

2.1 Server-Side Scripting

Web applications face more security threats than desktop applications [7], [18]. Some representative ones include command injection [20], cross-site scripting (XSS) [15], cross-site request forgery (CSRF) [3], and session fixation [14]. Any of these could cause serious consequences: sensitive user information could be stolen, data and associated belongings could be damaged, or service availability could be compromised.

We use an online-banking example to explain what is involved in secure server-side scripting, and how proper abstractions can help. This application provides two services: showing the account balance (the "balance" service) and setting up a payment (the "payment" service). A user must be logged in to access the services.

Although serving multiple users, this web application logically deals with one client at a time. In an ideal view, there are multiple orthogonal instances of the server program running, each taking care of a single client. Every single instance of the program can be viewed as a sequential program of a conventional application. A work flow graph following this ideal view is shown in Figure 1.

The above ideal view cannot be directly implemented, because of some limitations of the underlying HTTP mechanism for web interactions. In particular, there is no persistent channel for a server program to obtain input from a client. Instead, HTTP supports a one-shot request-response model where a client requests resource identified by a URL, and a server responds with the resource if the request is accepted.

[1]Sections 2, 3, 6, and 7 are adapted from the authors' paper titled Better Abstractions for Secure Server-Side Scripting at WWW'08.

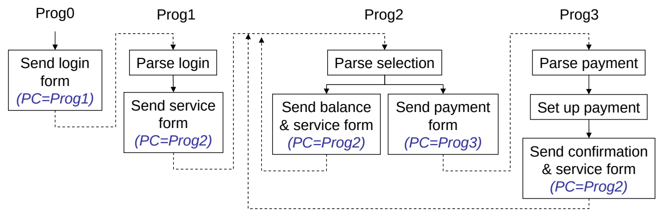

Figure 2: Actual work flow of online banking

Using HTTP, web interactions are typically carried out as a sequence of requests (form submissions providing user input) and responses (web pages presenting information):

$$\texttt{HTTP 0} \rightarrow \texttt{HTML 0} \rightarrow \texttt{HTTP 1} \rightarrow \texttt{HTML 1} \rightarrow \texttt{HTTP 2} \rightarrow \texttt{HTML 2} \ldots$$

Using this model, a server program is split into multiple fragments, each taking an HTTP request and producing an HTML response. In the response, there can be a web form with an embedded URL pointing to the next fragment, so that the next request is targeted correctly. Therefore, the server work flow of our banking application is more accurately described as in Figure 2. There are 4 program fragments connected with URL embeddings, as indicated by the dashed lines. In particular, because the payment service requires user input, the structure of the service loop in the ideal view can no longer be coded as an explicit loop. Instead, a goto-style structure is exhibited through URL embeddings. Such fragmentation and low-level control structures obscure the control flow.

Besides obscurity, there is a bigger issue: since HTTP is stateless, server programs must maintain program states on their own. In the example, the user name obtained from the login input must be saved and restored explicitly across the later web interactions. In addition, one must somehow correlate incoming requests with specific clients, since multiple clients may be interacting with the server at the same time, although in logically separate transactions.

In general, a web application needs to encode states at a level above HTTP. Typically, a virtual concept of "a session" is used to refer to a logical transaction. Every session is associated with a unique ID, called SID. Saved program states and incoming client requests are both identified by the SID. As a result, much code in server programs is dedicated to managing sessions. Before generating a response, a server program must save state and embed the SID in the response. Upon receiving a request, the server program must obtain an SID from the request and load state. Based on the application, some parts of the state should be saved on the server, whereas others should be on the client via cookie or URL embedding. These routine manipulations increase program complexity, reduce productivity, and extend the chances of programming errors.

2.2 Security

Assuming a programmer has taken care of the above issues correctly, the result program may still not be ready for deployment. The problem is security: clients in the real world may be malicious, or attackers may trick innocent clients into making mistakes. Indeed, there have been many common vulnerabilities identified. Secure programming solutions exist, but a programmer must be aware of all the issues involved and implement the related defenses. Most of the defenses are orthogonal to the functional logic of specific web applications. Their handling further complicates server-side scripting. We now briefly overview some representative security issues.

CSRF: An attacker may forge a request as if it were intended by a user. This is applicable when SIDs are stored in cookies. Given a vulnerable banking program, CSRF can be launched if a user, while logged in, opens a malicious email containing a crafted image link. Trying to load the image, the user's browser may follow the link and send a request asking for a payment to be set up to the attacker.

XSS: An attacker may inject code into the web page that a server program sends to a user. For example, an attacker sends to a user a crafted link with JavaScript code embedded; when the user loads the link, a vulnerable server program may propagate the code into the HTML response. The code, now coming from the server, gains the privilege of the server domain. It may then read the cookie set by the server and send it to the attacker. There are also second-order attacks [17] that do not require the use of forged requests.

Session fixation: An attacker may trick a user into interacting with the server using a fixed SID. This is applicable if SIDs are embedded in URLs. A user may follow a link in an email which claims to be from our banking site. The link takes the user to our site, but using an SID fixed by an attacker. If the server programs use the same SID for later interactions after the user logs in, the knowledge of the SID will grant the attacker the user's privileges.

Others: Many other aspects affect security [18]. Since a web application is implemented as multiple program fragments, each fragment is open as a service interface. An attacker could make up requests to the interfaces without following the links in server responses. Using crafted requests, they could poison program states (*e.g.,* by modifying naïve implementations of client-side states), inject malformed input (*e.g.,* by exploiting insufficient input validation), or circumvent work flows (*e.g.,* by using the "back" button).

Programmers need to be aware of all these issues and follow the relevant security practices. In the result code, functional logic is intertwined with security manipulations. Consequently, secure web programming is difficult, and web programs are hard to maintain.

3 BASS

In response to such security threats, existing languages and frameworks for server-side scripting and the web application security community largely promote secure coding practices (*e.g.,* input validation) and provide useful libraries in support (*e.g.,* filter functions). However, there is no guarantee that programmers will follow the recommendations correctly, if they followed them at all. Furthermore, even if all programs are written with the security practices strictly enforced, the extra care that programmers spend on preventing vulnerabilities much distracts from the functional logic. Take the implementation of online payments as an example. To securely code a web interaction of "obtaining payment details," one must correctly perform input validation, maintain program states across the interaction, and prevent CSRF.

Observing that many of the security issues are orthogonal to the functional logic of specific web applications, we propose some new abstractions for writing secure server programs. These abstractions provide an ideal view of key aspects of web programming (*e.g.,* "to obtain a web input"), and hide the common security handling (*e.g.,* input validation, state maintenance, CSRF prevention). Using these abstractions, a language for server-side scripting can be given a high-level syntax and semantics that reflect secure web operations, with the enforcement of the semantics taken care of by the language implementation following established security practices once and for all. As a result, all programs written in the language, although not directly dealing with low-level security constructs, are guaranteed to be free of certain common vulnerabilities. In addition, now thinking in terms of the high-level semantics, programmers can focus more on the functional logic, which results in better security and higher productivity. To some extent, the new abstractions hide security details in the same way as object creation primitives in OO languages hide low-level memory management details.

In particular, we propose to program web applications using a language that handles common security aspects behind the scene. This language benefits from new abstractions designed for web programming. A program in this language more directly reflects the functional logic of the application; therefore, it is easier to write, to reason about, and to maintain. The language implementation (the compiler) will generate secure target code following established security practices.

Abstracting web interactions From earlier discussions, it is easy to see that much of the complication is due to the need of obtaining user input. Therefore, supporting web interactions is a key. We introduce a dedicated construct `form` for this:

$$\texttt{form}(\texttt{p} : \text{“}\texttt{username}\text{”}, \texttt{q} : \text{“}\texttt{password}\text{”});$$

The intention of this is to present an HTML page to the client and obtain some input. In our prototype, such a construct may take as argument a full-fledged HTML document. In this section, we simply let it take arguments to describe input parameters of a web form for ease of exposition. In the above example, the `form` construct presents to the client a form with two fields: `username` and `password`. After the client fills in the form and submits it, the `form` construct assigns the values of the fields to the two variables `p` and `q`.

A few issues are handled transparently by the implementation. First, the server program is split upon a `form` construct, and URL embedding is used to connect the control flow. Second, input values are parsed from the form submission, and input validation is performed according to declared variable types. Third, security practices are followed to manage sessions, maintain states, and defend against common exploits. From a programmer's point of view, it suffices to understand this construct as an abstract and secure web input operation which does not break the control flow.

Supporting user navigation The `form` construct implicitly opens a service interface for receiving user requests. There would be vulnerabilities if it were not handled properly, or if its handling were not fully understood by the programmer. Previous work (*sans* security) [2], [4], [6] on web interaction abstractions requires the interface be "open" only once—a second request to the interface will be rejected. This much restricts user navigation [11], [12], [22]. In practice, it is common for a user to return to a previous navigation stage using the "back" button. In general, the user could revisit any item in the browser history. The validity of such an operation should be determined by the application.

We allow two modes of web interactions: a *single-use* mode ($\texttt{form}_\texttt{S}$) and a *multi-use* mode ($\texttt{form}_\texttt{M}$). In the former, the interface is open for request only once; revisiting the interface results in an error. In the latter, the interface remains open for future requests. The semantics of BASS articulates the program behavior in both cases; therefore, the programmer can choose the suitable one based on the application. In either case, a request is accepted only if it follows the intended control flow of the server program to the interface. Consider our banking example. It is okay if the user reached the service selection page, bookmarked it, and reused it before logging out. However, it is not okay if the user forged a payment request without first going through the login page.

Maintaining program states Multi-use forms are sufficient to accommodate all client navigation behaviors, because any behavior can be viewed as revisiting a point in the browser history. From a programmer's point of view, the program is virtually forked into multiple parallel "threads" at a multi-use form, together with all appropriate program state. The handling of the program state is nontrivial. Some parts of the state could be local to the individual threads, whereas others could be global to all threads. Careless treatment of the state may result in logical errors [12].

The exact partitioning of the state should be determined by the application. We let programmers declare mutable variables as either *volatile* or *nonvolatile*. In the BASS implementation, volatile state can be stored in a database on the server across web interactions, thus all forked threads refer to the same volatile state. In contrast, nonvolatile state (after proper protection against client modifications) can be embedded in the URLs of web forms upon web interactions, thus every forked thread has its own nonvolatile state.

Manipulating client history Suppose the user tries to reload the service selection page after logging out of our banking application. The server program will receive a request that should not be processed. In general, we need a mechanism to disable some of the entries in the client history. In existing web applications, this is sometimes handled by embedding special tokens into web forms and checking them upon requests. While logging out, the server program expires the

```
string  $user, pass, payee$;
int     $sel, amnt$;
form_S($user$ : "$username$", $pass$ : "$password$");
LoginCheck($user, pass$);
while (true) do {
  form_M($sel$ : "1 : $balance$; 2 : $payment$; $others$ : $logout$");
  if ($sel$ == 1)
  then ShowBalance($user$)
  else if ($sel$ == 2)
       then {form_S($payee$ : "$payee$"; $amnt$ : "$amount$");
             DoPayment($user, payee, amnt$)}
       else {clear; break}
}
```

Figure 3: Simple banking in BASS

corresponding token, thus further requests to the service selection page will no longer be processed.

We do not wish to expose the details of token embedding to the programmer. Instead, we introduce a `clear` command for a similar purpose. From the programmer's point of view, `clear` resets the client history so that all previously forked threads are discarded. This roughly corresponds to the "session timeout" behavior of many web applications. However, instead of thinking in terms of disabling the SID token, BASS encourages programmers to think in terms of the client history. Our previous work [25] discusses more general ways to manipulate client history, which introduce no new difficulties.

Example revisited Figure 3 demonstrates the appeal of these abstractions by revisiting our banking example. The new abstractions provide an ideal world where there is only one client and the client is well behaved. In the code, we obtain login information from the client, perform login check, and proceed with a service loop. In the loop, based on the service selection of the client, we carry out the balance service or the payment service, or log the user out. The service selection input is coded using a multi-use form; therefore, the user may duplicate the corresponding web page and proceed with the two services in parallel. In addition, `clear` is used to disable all service threads when the user logs out. In this example, only the $user$ variable is live across web interactions. Its value is obtained from a single-use form, and will not be updated after the login process. Therefore, it can be declared as either volatile or nonvolatile.

This code corresponds well to the work flow of Figure 1, and is much cleaner than a version written in an existing language. More importantly, it does not sacrifice security, because the BASS implementation will take care of the "plumbings" transparently—it will split the program into fragments, maintain states across web interactions, filter input based on variable types, and carry out relevant security manipulations such as the embedding of secret tokens.

4 BASS for Python

In previous work [25], we studied formal aspects of the BASS programming model and security guarantees, and the theory was supported by a prototype compiler for a core programming language. The prototype provides the exact benefits outlined above, but suffers from its small scale nature when building real-world applications. Specifically, it uses a new syntax, and lacks library and community support.

For more practical deployment, we have recently developed a new prototype in the form of a Python library using mod_python [21] (an Apache module that embeds the Python interpreter within the Apache server). Using this new prototype, programmers enjoy regular Python syntax, but must follow specific BASS programming patterns.

4.1 Basic Coding Patterns

We stress the point that BASS programmers do not deal directly with anything related to the underlying HTTP mechanism or common security protections. To some extent, the programmer may assume that the application only interacts with a single user, and the user is always well-behaved.

Take again the simple banking application in Figure 1 an an example. Now we translate this work flow into a BASS work flow. Note that we are restricted to follow basic coding patterns of mod_python. As a result, we essentially split the work flow wherever there is an interaction with the user. This yields the work flow in Figure 2.

Prog0, Prog1, Prog2, and Prog3 correspond to 4 routines in mod_python, each processing a user request and producing a response HTML page to the user. Now we insert some BASS API calls into each of these routines. At the very beginning of the entire work flow, we add `bass_init`. At the beginning of all other pieces, we add `bass_check`.

```
def prog0(req) :
  bass_init(req)
def prog1(req) :
  try :
    bass_check(req)
  exceptBassError, e :
    // insert error handling code
```

The former sets up the environment for the BASS library, and the latter parses the incoming request, maintains program states, and conducts common security checks. Here `req` is the standard mod_python request object, and it is used in all routines of the BASS API. Although BASS programmers do not need to directly work with `req`, they must keep passing it around for the BASS library. Programmers must follow these coding patterns so as to enjoy the BASS programming model and relevant security protections. Nonetheless, they do not need to understand the implementation of these routines.

When `bass_check` fails, exception `BassError` will be thrown. The parameter `e` of `BassError` contains an error message from the BASS library. BASS programmers may analyse the error message and build a customized error reporting page for the user based on the application logic. Representative error messages and their meanings are given as follows:

- Bad CID: The received request does not have a proper CID. Plausible causes are: (1) user deleted cookie during the session; (2) user is a victim of a CSRF attack.
- Bad SID: The received request does not come with a valid session identifier, indicating either a session-based attack such as session fixation, or a session timeout.
- Bad action: The received request is targeting a program interface unexpectedly, indicating an attempt at circumventing the application logic.
- Bad token: The received request does not come with a valid token, indicating that the web form should not be accepted (*e.g.,* re-submitting a single-use form).
- Bad cipher: The received request is not properly encoded, indicating a forged request.

4.2 Simple User Interfaces

At the end of a routine, mod_python uses `return` to render a web page. Although the design of the web pages is application-specific, the essence is to present information to the user and obtain further input therefrom.

4.2.1 Content

A simple web page can be built in BASS using a `Content` object that encapsulates a string as a web page. This is mainly for building a static page (or the static part of a web page). Although it is possible to build links and forms with regular HTML tags directly using `Content`, it does not enjoy BASS security protections, because the BASS library would not have a chance to embed special entities therein.

```
page = Content(req,'<html><h1> Hi! </h1>')
page.addContent('What a nice day! </html>')
return page.content
```

Note that one must pass the request handle `req` to the `Content` object. A `Content` object can be appended using a `addContent` method or overwritten using a `setContent` method. The result page can be rendered using `return`.

BASS is compatible with psp (Python Server Pages) [21]. For example:

```
<!-- A file named t.html -->
<html>
    <h1> Hello, world! </h1>
    <%= message%>
</html>
```

```
page.setContent('What a nice day!')
return psp.PSP(req, filename =' t.html',
                    vars = {'message' : page.content})
```

4.2.2 Forms

HTML forms can be used to build more useful web pages. BASS provides form API for this purpose.

```
sform = SingleForm(req, [amount, submit],' prog3.py')
```

The `SingleForm` class encapsulates single-use forms—forms that can be submitted by the user at most once. The constructor takes a request handle `req`, a list of form items, and a target action. Form items are entities that make up forms, such as text fields, selections, and buttons. Target action points to the server program that handles the request upon form submission. The above example creates a single-use form to obtain details of a money withdrawal transaction, where `amount` and `submit` are two variables holding an input field and a submit button, respectively (we defer the creation of BASS variables to a later section). When the user clicks on the submit button, `prog3.py` will be invoked to process the request. Since this is a single-use form, the user is only able to submit it once. For instance, if the user accidentally submitted the form a second time (e.g., via the reload button), an error would be raised to the user, avoiding unintended additional withdrawal.

A multi-use form can be created as follows:

```
mform = MultiForm(req, [selection, submit],' prog2.py')
```

The `MultiForm` class encapsulates multi-use forms–forms that can be submitted by the user multiple times. The interface of the constructor is similar to that of `SingleForm`'s. The above example creates a multi-use form to obtain a service selection, where `selection` and `submit` are two variables holding an input selection of service and a submit button, respectively. When the user clicks on the submit button, `prog2.py` will be invoked to process the request.

```
sform.render_form()
mform.render_form()
page.addContent('How much?' + sform.render_form())
```

Both `SingleForm` and `MultiForm` are subclasses of `Content`. Besides encapsulating strings, these two classes maintain various tokens and program state. BASS programmers do not need to be aware of the internal workings of these classes. However, after creating a BASS form, one must call `render_form` to render it. This routine composes an HTML form to be presented to the user; it embeds into the form object special tokens and program state so as to enforce security and pass on session information.

4.2.3 Links

It is also possible to render a form as a link using `render_link`.

```
balanceForm = MultiForm(req, [],' balance.py')
balanceLink = balanceForm.render_link('Show Balance')
page.addContent(balanceLink)
return page.content
```

In this example, multi-form `balanceForm` is rendered as a link with special BASS entities embedded. It is added to a `Content` object `page`, and then rendered as a hyperlink when displayed in a browser. Once the user clicks on the link, the underlying web form will be submitted to the server for processing. A link has the same single-use/multi-use property as the underlying form.

4.2.4 Clear

Recall that single-use forms can be submitted only once, whereas multi-use forms can be submitted multiple times with different values. It may sometimes be desirable to disable all existing forms to prevent further user interactions through them.

This can be achieved using `clear`. This method disables all existing forms on the client side. One use of this operation is to implement logout behavior. After calling `clear`, the client would not be able to resubmit requests using previously received forms. Nonetheless, the application can still interact with the client using forms created afterwards.

4.3 Templates and Variables

One may wish to go beyond basic BASS form API introduced above when building sophisticated web pages. This can be achieved using BASS templates, whose syntax is similar to that of psp (in fact, BASS templates are implemented by adding BASS protections on top of psp templates). Specifically, a BASS template is a regular HTML page (which may include JavaScript) with "holes" (enclosed by <% = ... %>) to be filled in with BASS variables. Here is a sample template for a login page:

```
<! -- login.html -->
<h3> Bank Login </h3>
<p>
<form action = "prog1.py">
<table border = "0">
<tr><td align = "left" colspan = "2">
     <b> Login Please </b>
</td></tr>
<tr><td width = "10%">
     Username :</td><td>
     <input type = "text" name = "username"
       value = "<% = username%> ">
</td></tr>
<tr><td width = "10%">
     Password :</td><td>
     <input type = "password" name = "password"
       value = " <% = password%> ">
</td></tr>
<tr><td>
     <input type = "submit" value = "Submit">
</td></tr>
</table>
</form>
```

This template implements a web page in HTML, except for the two holes (written in bold for ease of reading) left for BASS variables `username` and `password`. Before sending the page to user, the holes will be filled in with values of the corresponding variables. Similarly, once the user completes the HTML form and submits it, the corresponding variables will get their new values from the form.

Here is some sample code on the instantiation of templates:

```
sp = STemplate(req,'login.html', [username, password])
sp.render_template()
mp = MTemplate(req,'login.html', [username, password])
mp.render_template()
```

STemplate is a subclass of `SingleForm`. It creates a single form by plugging the list of variables (the 3rd argument) into the template file (the 2nd argument). MTemplate is a subclass of `MultiForm`, and has a similar interface. As is the case of form objects, objects of these classes must be rendered using `render_template` so as to enjoy BASS protections.

As discussed previously, BASS supports volatile (global) variables and nonvolatile (local) variables. Global variables (implemented as server-side state) are shared by all copies of a form, and changes in one affects the value in all. In contrast, local variables (implemented as client-side state) are specific to each copy of a form, thus changes in one does not affect the value in others. This is supported using class `NewVar`:

$$\texttt{NewVar}(request\_object, global/local, variable\_name, \\ default\_value[optional], type[optional], \\ prompt[optional], follow\_up\_text[optional])$$

Class `NewVar` manages BASS variables and provides getter and setter API. It takes three mandatory arguments on the request object, the kind of the variable (global or local), and the name of the variable. It also takes optional arguments on default value, variable type (*e.g.,* int, string), and text strings to appear before and after the input field. It is up to the programmer to use the appropriate kinds of BASS variables (global/local) based on the specific application.

5 BASS for GAE

We are currently in the process of porting the Python BASS library onto Google App Engine (GAE), a cloud-based platform for hosting web applications on Google's infrastructure. GAE supports Python, but puts nontrivial restrictions on the web applications developed using it. We now discuss some notable aspects of the porting effort.

5.1 Web Application Framework

The BASS library discussed in Section 4 uses mod_python for its performance, versatility, and scalability [23]. A web application deployed on GAE, on the other hand, interacts with the web server using CGI. As a result, we must rewrite the Python BASS library to be CGI-compliant. This can be done using the cgi module from the Python standard library.

Fortunately, the switch from mod_python to the cgi module can be hidden by the BASS API, thus transparent to BASS programmers. Specifically, all BASS routines take the standard mod_python request object `req` as an argument. This object contains rich context information about the HTTP request and server status. When porting to GAE, this parameter can be used to hold an object compatible with the cgi module instead. The handling of this object would be significantly different than that of the request object, but it happens entirely within the BASS library implementation. A BASS programmer simply passes the object around without having to look into it, as is the case of the request object.

In addition, BASS templates are currently implemented using psp, which is part of mod_python. For porting onto GAE, we need to implement a replacement of (a subset of) psp in the BASS library for variable substitution to support templates. This is again transparent to BASS programmers.

5.2 Persistent Data

The Python BASS library uses Python's pickle library to manage all data that persist between requests through local files. In contrast, GAE uses a substantially different method for managing such data. Specifically, GAE applications run in a secure environment with limited access to the underlying operating systems. They use the App Engine Datastore for the persistent storage of data instead.

The Python Datastore API consists of data modeling API and query API. An application defines a data model describing the kind of entities and their properties, and entities are stored by calling a `put()` method. Two interfaces are supported for queries: a query object interface and a query language GQL. The porting of BASS to GAE consists of two main tasks accordingly. One is to design a data model compatible to Datastore. The other is to replace local file operations with Datastore `put()` and query operations. Neither task affects the BASS API; therefore, the changes required remain transparent to BASS programmers.

6 Related Work

MAWL [1], [2] and its descendants (`<bigwig>` [4], JWIG [6]) use domain-specific constructs to program web applications. They view web applications as form-based services, and provide abstractions on some key aspects such as web input and state management. These abstractions hide implementation details (e.g., input validation, embedding of continuation and state in URL), thus preventing certain programming errors. Graunke *et al.* [10] propose the design and implementation of an I/O construct for web interactions. This construct helps to program web applications in a more traditional model of interactions, and avoids the manual saving and restoring of control state.

Although similar in spirit to BASS on these aspects, the above work does not provide a formal semantics with the same security guarantees. However, security should not be overlooked for declarative web programming—now that the details of web interactions are hidden by new abstractions, programmers can no longer carry out the secure coding practices by themselves. As a result, a naïve application of new abstractions could suffer from security vulnerabilities such as CSRF. It is thus crucial that the proposed abstractions and their implementation provide related security guarantees.

On expressiveness, MAWL and descendants enforce a strict control flow where every form is, in the BASS terminology, single-use. For example, users will be redirected to the beginning of a session if they hit the back button. In contrast, BASS leaves the design decision to the programmer, rather than disabling "whimsical navigation" [12] altogether. This flexibility is important [11], [12], [22].

We emphasize that the goal of BASS is to facilitate secure web programming with abstractions more suitable for the domain. Besides having declarative support on web interactions, single-/multi-use forms, state declarations, and history control, it is important that the features are all modeled within an original and self-contained semantic specification. In previous work [25], we have given BASS an intuitive and formal programming model and articulated its meta properties. This allows programmers to fully grasp how BASS programs behave. The common task of following secure coding practices, which is orthogonal to the specific application logic, is carried out by a BASS implementation once and for all.

There has also been work developing domain-specific languages or frameworks for web programming as libraries of existing type-safe languages. Examples include the Curry-based server-side scripting language by Hanus [13], Haskell-based WASH/CGI by Thiemann [22], and Smalltalk-based Seaside by Ducasse *et al.* [9]. These provide useful abstractions in the form of libraries to handle some common aspects of web programming, such as structured HTML generation, session management, and client-server communication. However, there is no stand-alone formal semantics for the new abstractions, although in principle the behaviors could be inferred from the implementations and the semantics of the host languages. In addition, they are tied to the host languages, thus the ideas are not easily applicable to other languages.

Finally, some recent work [8], [19], [5] uses a unified language or framework for web application development, automatically compiling programs into server code and client code. Most of such work does not address security. A notable exception is Swift [5], which ensures confidentiality and integrity of web application information using type annotations which reflect information flow policies. The security guarantee of Swift is largely orthogonal to those of BASS. If programmers use proper annotations, the general information flow guarantees of Swift can help guarding against some common vulnerabilities such as SQL injection and XSS. However, it remains vulnerable to (it is up to the programmer to write secure code against) others such as CSRF.

7 Conclusion and Future Work

Web applications reflect a different computation model than conventional software, and the security issues therein deserve careful study from the perspectives of both language principles and practical implementations. In this paper, we reviewed the basic BASS ideas, presented a BASS prototype deployed as a Python library, and discussed ongoing efforts on porting this library onto Google App Engine. In general, we believe web programming will benefit significantly from the use of domain-specific abstractions, and much can be done in the area.

BASS provides some security guarantees using a few new abstractions. These abstractions are not meant to be "complete," and there are other desirable properties uncovered. It is useful to explore abstractions for other areas, such as dynamic HTML generation [13], [6], [16], [22], privilege management, and dynamic SQL construction [20], [24].

Designed for web programming in general, BASS addresses only common security aspects, rather than issues on the specifics of an application. For example, directory traversal [7] (accessing the parent directory using the path "`..\`") is not prevented by common type-based input validation, and programmers must perform additional filtering. Application-specific security analysis will still be necessary. However, with the new abstractions closing up some common vulnerabilities and clarifying the control flow, such analysis should be easier. In general, the new abstractions should help the analysis, reasoning, and testing of web programs, because they provide an ideal model (*e.g.,* structured control flow, automatic state maintenance, single well-behaved client) that is amenable to established language techniques.

REFERENCES

[1] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proc. 1997 Conference on Domain-Specific Languages*, 1997.

[2] D. L. Atkins, T. Ball, G. Bruns, and K. Cox. MAWL: A domain-specific language for form-based services. *IEEE Trans. on Software Engineering*, 25(3):334–346, 1999.

[3] R. Auger. The Cross-Site Request Forgery FAQ. `http://www.cgisecurity.com/articles`, 2007.

[4] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Trans. on Internet Technology*, 2(2):79–114, 2002.

[5] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. 21st Symposium on Operating Systems Principles*, pages 31–44, Oct. 2007.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Trans. on Programming Languages and Systems*, 25(6):814–875, Nov. 2003.

[7] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cve.mitre.org/docs/vuln-trends`, 2007.

[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, Nov. 2006.

[9] S. Ducasse, A. Lienhard, and L. Renggli. Seaside — a multiple control flow web application framework. In *Proc. 12th International Smalltalk Conference*, pages 231–257, Sept. 2004.

[10] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *Proc. 16th International Conference on Automated Software Engineering*, pages 211–222, Nov. 2001.

[11] P. Graunke and S. Krishnamurthi. Advanced control flows for flexible graphical user interfaces. In *Proc. 2002 International Conference on Software Engineering*, pages 277–296, 2002.

[12] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Modeling web interactions. In *Proc. 2003 European Symposium on Programming*, pages 122–136, 2003.

[13] M. Hanus. High-level server side web scripting in Curry. In *Proc. 3rd International Symposium on Practical Aspects of Delcarative Languages*, pages 76–92, 2001.

[14] J. Kolšek. Session fixation vulnerability in web-based applications. `http://www.acrossecurity.com/papers.htm`, 2002.

[15] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying XSS vulnerabilities in web applications. In *Proc. 6th International Workshop on Web Site Evolution*, pages 71–80, 2004.

[16] K. Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, 2005.

[17] G. Ollmann. Second-order code injection attacks. `http://www.nextgenss.com/papers`, 2004.

[18] OWASP Foundation. The ten most critical web application security vulnerabilities. `http://www.owasp.org`, 2007.

[19] M. Serrano *et al.*. Hop. http://hop.inria.fr, 2006.

[20] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. 33rd Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.

[21] The Apache Software Foundation. Apache/Python integration. `http://www.modpython.org`, 2008.

[22] P. Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. on Internet Technology*, 5(1):1–46, Feb. 2005.

[23] G. Trubetskoy. Introducing mod_python. O'Reilly Media, `http://www.oreillynet.com/pub/a/python/2003/10/02/mod_python.html/`.

[24] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. 2007 Conference on Programming Language Design and Implementation*, pages 32–41, June 2007.

[25] D. Yu, A. Chander, H. Inamura, and I. Serikov. Better abstractions for secure server-side scripting. In *Proc. 17th International World Wide Web Conference*, Apr. 2008.