WEBLOG: A Declarative Language for Secure Web Development

Timothy L. Hinrichs, Daniele Rossetti, Gabriele Petronella, V. N. Venkatakrishnan,
A. Prasad Sistla, Lenore D. Zuck

University of Illinois at Chicago
Computer Science Department

hinrichs@uic.edu, daniele.rossetti@me.com, gabriele.petronella@gmail.com, venkat@uic.edu,
sistla@cs.uic.edu, lenore@cs.uic.edu

Abstract

WEBLOG is a declarative language for web application development designed to automatically eliminate several security vulnerabilities common to today's web applications. In this paper, we introduce WEBLOG, detail the security vulnerabilities it eliminates, and discuss how those vulnerabilities are eliminated. We then evaluate WEBLOG's ability to build and secure real-world applications by comparing traditional implementations of 3 existing small- to medium-size web applications to WEBLOG implementations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Security and Protection*]: Access Controls; K.6.5 [*Security and Protection*]: Unauthorized access

General Terms Languages, Security, Verification, Design

Keywords web security; correct-by-construction; declarative language

1. Introduction

While many programming languages and frameworks have been designed specifically to develop software for the World Wide Web, they widely differ in addressing security concerns. Some languages simply provide a library of countermeasures that a developer may apply at will, such as taint-tracking (e.g. Perl) and auto-sanitization (in Google Closure Templates) of user-provided data. Such languages provide no real guarantees for the developer but simply reduce her manual effort. Other languages are more ambitious and are designed to eliminate security vulnerabilities automatically, *e.g.*, by ensuring that a type-safe program is not vulnerable to a specific class of attacks [12, 19]. Such languages typically eliminate one or two classes of vulnerabilities; extending such approaches to address a wide range of Web vulnerabilities is hard because of fundamental computational limits on how much analysis can be performed on the underlying language (*e.g.*, analyzing arbitrary loops is undecidable).

The limitations of current-day, language-based web security are rooted in the fact that today's web development languages are designed primarily for rapid prototyping — for a developer to achieve a certain functionality quickly. In contrast, the web development language introduced in this paper, WEBLOG, was designed primarily for security—for protecting applications from today's most prevalent vulnerabilities. In fact, we began with a list of prevalent web security vulnerabilities (the OWASP Top-10 [21]) and designed a language just expressive enough that the compiler and runtime can automatically eliminate those vulnerabilities correctly, comprehensively, and automatically. Not surprisingly, we discovered that the language must be significantly less expressive than traditional programming languages, meaning that not all possible web applications can be expressed. One of the main questions we are trying to answer in this paper is whether such a language is expressive *enough* to be used for real-world applications.

WEBLOG supports two conceptually different kinds of defenses. It includes automatic defenses for some of the most critical web application security risks as ranked in the OWASP Top-10. These defenses are cutting-edge, leveraging the recent advances in web security. Because they are incorporated into the compiler, developers can always leverage the most recent countermeasures by simply updating their compiler and recompiling their applications.

A second, more compelling, class of defenses are those that require weaving a separately written security policy throughout the application. For example, controlling which users can access resources in which ways (access control) is something that can only be implemented by changing all of the application fragments that access sensitive resources. For these defenses, the developer writes a separate security policy, and the compiler generates code that simultaneously implements the application's functionality while respecting the developer's policy.

We view this work as a first step in a long-term research project aimed at developing a multi-layer web development programming language. WEBLOG serves as the core of the language; any application expressible in WEBLOG is automatically protected against a plethora of security vulnerabilities. Each layer built on top of WEBLOG could add additional expressiveness at the cost of some number of security guarantees. At the top level is a fully-expressive, traditional programming language that provides no absolute security guarantees but is outfitted with a library of common countermeasures. A developer can then weigh the use of additional expressiveness either for functionality or convenience against the loss of automatic security.

PLAS'13, June 20, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2144-0/13/06. . . $15.00

This paper makes the following technical contributions:

- WEBLOG *design* We present the design rationale and conceptual basis for WEBLOG[1]. We show how WEBLOG is a minimal, yet powerful framework for web development.
- *Practical Defenses for Common Attack Vectors* WEBLOG includes cutting edge defenses for most of the OWASP security errors. An important characteristic of WEBLOG is that the security of the application is less reliant on the programmer. Except for problems with the logic of the application, the burden of eliminating security weaknesses lies with the framework.
- *Evaluation* We show that WEBLOG is sufficiently expressive to implement an entire, real world web application. We also discuss how common web application features can be realized efficiently and securely through WEBLOG. Finally, we report on development experiences and the security of two implementations of the same application: one in WEBLOG and one in Sinatra (a Ruby-based web development framework).

This paper is organized as follows. Section 2 provides a running example and illustrates the motivation behind the design of WEBLOG. Section 3 gives an overview of WEBLOG. Section 4 describes how WEBLOG automatically eliminates a number of prevalent security vulnerabilities by leveraging recent results from web security research. Section 5 discusses our evaluation of WEBLOG. In Section 6, we discuss related work. Section 7 concludes.

2. Background

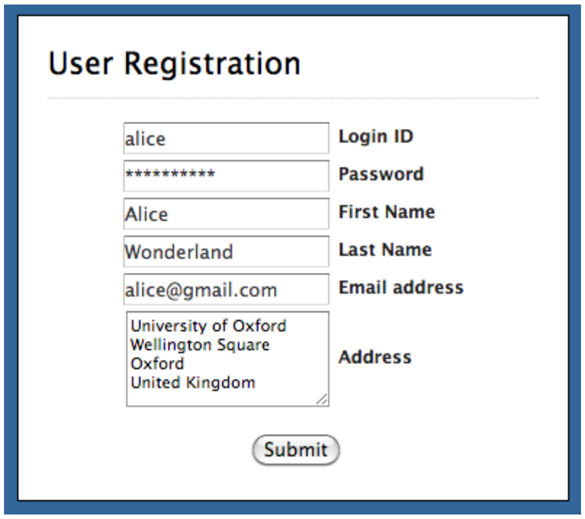
Web Applications Web applications are based on the client-server architecture, where the client and server communicate using the Hypertext transfer protocol (HTTP) or its encrypted variant (HTTPS). The client must be written in one of a handful of programming languages the web browser understands, *e.g.*, usually some combination of HTML, JavaScript, Flash, and Silverlight. Servers can be written in any programming language, though most applications today are written using web-specific languages, such as PHP, or web-development frameworks built on top of general-purpose languages, such as Java Servlets, Ruby-on-Rails, or Django (a framework for Python).

Running Example The example we use throughout the paper is a real-world online auction application called WeBid [3]. WeBid allows people to create new auctions, search through a myriad of existing auctions, and place bids on merchandise of interest. Figure 1 shows a simplified version of the user registration page where a new user chooses a login ID and password and provides some basic personal information like first and last name, address, and email.

Once the user fills out the form, the browser encodes the form's data as a collection of key-value pairs in an HTTP request, and sends that request to the server. The server accepts the HTTP request, checks the data for errors, updates its backend relational database to store the user's profile data, and responds with another HTTP message whose payload is the client code describing whether or not the registration succeeded. Sometimes the browser will send auxiliary information to the server by embedding *cookies* within that initial HTTP request—information that typically tells the server something about the user sending the data. The server may also maintain additional information in an in-memory *session* object to improve performance, so named because it is typically used to group several distinct HTTP requests into a single user session.

Web Security An attacker could attempt to compromise this web application in numerous ways. If the attacker includes SQL commands within the Last Name field, she might fool the application into executing those commands on the server's backend database, an attack referred to as SQL Injection (SQLi). If the attacker includes malicious JavaScript code within one of the fields, she might convince the application to create a client page that executes that code when loaded by another user, an attacked referred to as cross-site scripting (XSS). If the attacker changes the value of a hidden field called *admin* on the form, she might convince the application to grant her administrative rights, an attack called parameter tampering.

[1] Implemented as part of the CLICL library available at `https://code.google.com/p/clicl/`


User Registration

alice	Login ID
**********	Password
Alice	First Name
Wonderland	Last Name
alice@gmail.com	Email address
University of Oxford Wellington Square Oxford United Kingdom	Address

Figure 1. Running example: WeBid user registration

The attacker could also simply type URLs into the browser that the web developer did not expect and in so doing circumvent the application's intended access control policy. For example, in WeBid, there are "premier" auctions that are only visible to registered users. All of WeBid's search functions delete premier auctions from their results unless the user performing the search is logged in. But if an unregistered user asks the application for the details of a premier auction by providing that auction's ID directly, she gains access to that auction, thereby violating the application's intended access control policy.

Motivation Securing an application like WeBid is difficult for several reasons. First, the developer must understand the security defenses to put in place to defend against each kind of attack. Second, defenses must be deployed comprehensively and consistently across the application. Third, because applications are always undergoing bug fixes and feature improvements, the task of hardening the application against attack must be revisited each time the application is updated. Finally, security is still a secondary goal of many web developers (and managers) and hence in practice devoting the necessary time and energy to protecting applications is difficult.

The above challenges are not easy to meet as illustrated by the vast numbers of SQL injection and XSS attacks on applications (still the top two as reported by Common Weaknesses Enumeration (CWE) [2]). A web development language where cutting-edge security defenses are installed by the compiler would address all of the concerns above. The developer would not need to keep up to date with the intricacies of web security defenses; the defenses would be deployed completely and automatically; maintenance of web applications would require no additional hardening phase; and resource-limited organizations could still build secure, functional web applications.

Scope In this paper we focus on developing a language for programming the server component of a web application and leave client-side programming to future work (though we discuss a placeholder client-side language for the sake of completeness). This means that our testing and development of WEBLOG has focused on Web 1.0 applications; however, WEBLOG has been designed with Web 2.0 applications in mind (where the client asynchronously communicates with the server). We expect that the development of a complementary client-side programming language will require few changes to WEBLOG. One final note on scope: WEBLOG makes no attempt to address noSQL-based web applications.

3. WEBLOG

Our main thesis is that many of today's web applications are conceptually simple but are written using programming language constructs that are unnecessarily difficult for the compiler to analyze. WEBLOG is a web development language that eliminates the hard-to-analyze programming language features while including domain-specific constructs that help the programmer overcome that lack of expressiveness. WEBLOG is an investigation into whether it is possible to design a domain-specific language that is expressive enough to build real-world web applications while simultaneously being *inexpressive* enough that the language itself can both soundly and completely eliminate a plethora of security vulnerabilities (possibly even those we have yet to identify).

WEBLOG is a server-centric language similar to PHP, Java servlets, and Ruby-on-Rails. By *server-centric*, we mean that a developer focuses on writing the code that runs on the server and treats the client as simply a mechanism that renders the server's output in human-readable form and solicit's additional input.

WEBLOG's attempt to balance expressiveness with inexpressiveness is based on two key ideas. Instead of giving programmers the ability to choose from a wide variety of data structures (*e.g.*, binary trees, arrays, lists), it supports a single data structure that is both necessary and sufficient for typical web applications. Second, it ensures that the language for manipulating that data structure is written using purely declarative constructs that can be analyzed by state-of-the-art automated reasoning tools.

Data-centric Programming Model WEBLOG supports a single data structure: the relational database. Conceptually, a programmer writes code that takes as input one relational database and outputs another relational database. This database-centric programming model is a natural fit for the web domain because all the main data structures manipulated by typical web application can be construed as relational databases. Obviously the backend relational database of the web application fits the paradigm. The form data a client submits to the server in an HTTP request is a collection of key-value pairs, which we can represent as a database table with two columns. Even the auxiliary data structures a web application commonly manipulates (the *cookies* for the client and the *session* for the server) are sets of key value pairs. The output of a web application is also often just a relational database, but one rendered in HTML.

Purely Declarative Because the only data structure in WEBLOG is the relational database, the obvious language for describing the input/output functionality of a web application is the most prevalent one for managing relational databases: SQL. Besides being a purely declarative, well-known language that is well-suited for the task, large fragments of SQL can be deeply analyzed using the theorem provers, model builders, and SMT solvers developed by the automated reasoning community over the last 50 years (see [5, 11] for the relationship between SQL and first-order logic).

3.1 Syntax

At the top-level, a WEBLOG application is a collection of *servlets*. Each servlet is a self-contained piece of web server functionality that takes an HTTP request as an input, modifies the server's session and RDBMS, and returns an HTML page to the client.

Definition 1 (Servlet)**.** *A servlet has the following fields.*

- ***transducer****: a collection of SQL update statements that describe how data sent to the servlet changes the RDBMS, cookies, and session, and how to compute the output.*
- ***guard****: a collection of SQL integrity constraints dictating the conditions under which the servlet can be executed.*
- ***renderer****: a description for how the output data is displayed on the client.*

The input to a servlet is a relational database, some of whose tables represent all the usual sources of information for a servlet[2]. The user data contained within an HTTP request is stored in the two-column table `HttpInput`. The cookie data is stored as the two-column table `Cookies`. The session data is stored as the two-column table `Session`. The remaining tables in the servlet's input are the backend RDBMS tables.

A servlet's guards are written as SQL integrity constraints over the relational database provided as input to the servlet. If any of those guards fail, no changes are made to the cookies, session, or RDBMS, and WEBLOG automatically sends a response to the client describing what the failure was.

If all of the guards succeed, the servlet's transducer is executed on the input database. Any changes made to the RDBMS tables are sent to the RDBMS. Any changes to the `Cookies` table are reflected in the HTTP cookies sent back to the browser. Any changes to the `Session` table are carried over to the server's actual session object (and if there is any session data stored, a WEBLOG-managed session ID cookie is set automatically).

Once the transducer execution completes, the resulting database is given to the servlet's renderer, which creates the HTML page to send to the browser. Conceptually, any query results that the renderer needs has already been computed by the transducer and stored in a table agreed upon by the transducer and the renderer. The renderer is free to display any data present in the relational database however it likes, though the compiler checks to ensure the renderer never releases information it should not.

In our running example application, the web page for creating a new user sends an HTTP request to the server with the user's first and last name, an email address, a login ID, and a password, which causes the server to insert that data into the RDBMS and log the user into the application. To implement this functionality in WEBLOG, a developer creates a servlet whose transducer is the following two SQL update statements. The first inserts the user's data into the database, and the second sets the session's username to her login ID, thereby logging her into the application. This is shown in Listing 1.

To require that the user's selected login ID has not already been chosen by another user, a WEBLOG developer would include a guard with SQL integrity constraint shown in Listing 2 guaranteeing that the login ID in the HttpInput table does not already exist in the User table of the RDBMS.

While this example requires no real data be sent back to the client, consider a web page that returns all auctions whose titles include a query string chosen by the user. To implement the search servlet, a WEBLOG developer writes a SQL update command that inserts all the auctions from the RDBMS table `Auctions` whose titles match the user's search string (stored in the `HttpInput`

[2] We use the term *relational database* to mean a collection of named *tables*. A *table* is a collection of rows, all of which have the same number of columns of atomic data.

Listing 1.

```
INSERT INTO Users (login, first, last, email) SELECT login, first, last, email FROM HttpInput
INSERT INTO Session (user) SELECT login FROM HttpInput
```

Listing 2.

```
CHECK (NOT EXISTS
   (SELECT * FROM Users U, HttpInput H WHERE U.login = H.login))
```

table) into a temporary table, here called `SearchResults`, as shown in Listing 3. Those search results are then rendered to the client.

Because the same SQL statements can be used in many different servlets, WEBLOG enables a developer to organize SQL code into modules and to combine modules using a simple form of inheritance. Each SQL module is either an *update* module, meaning that it includes just SQL update statements (*i.e.*, INSERT, DELETE, or UPDATE), or it is a *guard* module, meaning that it includes just integrity constraints. SQL update modules are defined with the `defupdate` command; SQL guard modules are defined with `defguard`. A servlet's transducer is a *sequence* of SQL update modules. A servlet's guard is a *sequence* of guard modules. When a servlet is invoked, the guard modules are checked in the order they appear in the servlet definition; the first module in which an error appears (*i.e.*, a constraint is violated) halts the execution of the servlet, and WEBLOG returns all errors in that module to the user. If no errors occur, the update modules are executed in the order they appear, and the servlet's output is rendered and returned to the user. Listing 4 is the WEBLOG code for the user-registration servlet described above.

From the perspective of basic functionality, the renderer for a servlet can be any piece of code written in any language that takes a database as input and outputs HTML. But for the sake of security, WEBLOG requires that the compiler be able to perform three tasks. First, the compiler must be able to analyze the rendering code for each servlet to compute exactly that fragment of the database the renderer requires in order to produce the client. Said differently, what fragment of the database can WEBLOG hide from the renderer without altering its output? This requirement is not much of a restriction in practice because it is often a simple matter to require each renderer declare which tables it will read (and enforce that declaration by only providing the renderer with exactly those tables).

Second, the compiler must be able to analyze the rendering code to compute the HTML context (or the lineage of HTML tags) in which each piece of user data is to be embedded. For example, if the renderer takes a user's profile and displays it on the screen in an HTML table, then the context for each of the profile's data elements might be `<html><body><table><tr><td>`. This requirement is far more difficult to attain because all looping and recursion constructs must be analyzable, and dead code must always be identifiable.

Third, the compiler must be able to control the behavior of all the web forms that are produced. At the very least, the compiler must be able to supply code for each form's event handlers without modifying the client's overall behavior, and it must be able to reference and modify the form's values.

Currently, WEBLOG is outfitted with a simple rendering mechanism that satisfies the properties above. It transforms the results of a servlet's execution into one of several languages: XML, JSON, or HTML with cookies. Rendering data as XML and JSON is straightforward and can be carried out by WEBLOG autonomously, but rendering HTML requires additional developer involvement. In WEBLOG, we view an HTML page as a way of displaying data to the user or requesting data from the user. We call the specification for how data is to be displayed in HTML a *table*, and the specification for a request for data a *form*, reflecting common use cases for the HTML elements of the same name.

The renderer operates similar to HTML templates in traditional development frameworks. The developer writes an HTML page along with several table and form specifications. At runtime, WEBLOG inserts into that HTML page renderings of data and renderings of requests for data, as per the table and form specs.

Definition 2 (Renderer)**.** *A renderer has the following fields.*

- ***HTML****: an HTML page*
- ***forms****: a set of substitutions of the form* htmlID/form
- ***tables****: a set of substitutions of the form* htmlID/table

The substitution htmlID/form *means that the HTML element with ID* htmlID *should be replaced by an HTML representation of* formname. *Similarly for tables.*

A form specification is a database schema (a collection of table names and column names) along with guards that dictate the permitted combinations of table values and the servlet to which data should be submitted. A table specification is just a database schema combined with the name of one of several algorithms built into WEBLOG for rendering a database table in HTML.

Definition 3 (Form, Table)**.** *A form has the following fields.*

- ***schema****: a database schema*
- ***guard****: a list of guards*
- ***target****: the servlet to submit data to*

A table has a schema and the name of one of several algorithms built into WEBLOG *for rendering data.*

Forms are created with the `defform` command; tables are created with `deftable`; schemas are created with `defschema`; renderers are created with `defrender`. Listing 5 shows the renderer for the new-user registration page, which includes a form with a guard requiring that the two passwords the user provides are equal. The HTML page WEBLOG generates automatically ensures that the user's data satisfies any guards on the embedded forms before the user is allowed to submit that form to the server.

Notice that the user-registration renderer supplies the argument `:lang HTML` to `defrender`, which dictates that the renderer is one that outputs HTML. Writing a renderer that outputs XML or JSON requires simply changing the `:language` and ignoring the `:html`, `:forms`, and `:tables` arguments.

Admittedly, the renderer infrastructure for outputting HTML is not as expressive as we would like because it requires the developer to choose from one of several pre-built rendering options for each table and form, and clients are limited to static HTML pages. A more ambitious approach to rendering is studied in [20], where the basic rendering language mixes HTML with arbitrary code from a

Listing 3.

```
INSERT INTO SearchResults (id) SELECT A.ID FROM Auctions A, HttpInput H WHERE A.Title LIKE H.query
```

Listing 4.

```
defservlet new-registration ()
   :guard (unique-login)
   :transducer (save-registration
                login)
   :renderer reg-renderer

defupdate save-registration ()
   INSERT INTO Users (login, first, last, email) SELECT login, first, last, email FROM HttpInput

defupdate login ()
   INSERT INTO Session (user) SELECT login FROM HttpInput

defguard unique-login ()
   (NOT EXISTS (SELECT * FROM Users U, HttpInput H WHERE U.login = H.login))
```

Listing 5.

```
defrender user-registration (:lang HTML)
   :html "path/to/HTML-file.htm"
   :forms ("userdata"/new-user)
   :tables ()

defform new-user ()
   :schema userreg
   :guard (passwordsEq)
   :target new-registration

defschema userreg ()
   CREATE TABLE Userinfo (login, password, password2, firstname, lastname, email)

defguard passwordsEq ()
   NOT EXISTS (SELECT * FROM Userinfo WHERE password != password2)
```

traditional language like Ruby. This work is attractive because it builds on the common paradigm of HTML templates but achieves the analysis required by WEBLOG using a type system to detect HTML contexts. In the future, we plan to develop a richer client-side development language that complements the server-side programming support within WEBLOG. In the remainder of the paper, the only thing about the rendering language we assume are the three requirements described earlier in this section.

3.2 Semantics

The semantics of WEBLOG is basically the same as any traditional web development framework. The main difference is that WEBLOG provides primitives for the developer to write the entire application by simply manipulating a collection of database tables. That collection, which we denote with Δ, includes tables representing all of the traditional data sources in a web application: the client's cookies, the server's session, the backend RDBMS, the payload of the HTTP request, and any data output to the client. Algorithm 1 defines the functional semantics of WEBLOG in terms of how Δ is constructed from the traditional data sources, how a servlet changes Δ, and how changes to Δ are carried back to the original data sources.

Algorithm 1 SERVE(URL, HTTPdata, Cookies)

```
 1: if the servlet $S$ referenced by URL does not exist then throw a 404 error
 2: Set $\Delta$ to be the RDBMS data
 3: if Cookies includes a session token then
 4:     if a session $E$ for that token exists then
 5:         $\Delta = \Delta \cup E$
 6:     else
 7:         throw a Lost-session error
 8: $\Delta = \Delta \cup$ Cookies
 9: $\Delta = \Delta \cup$ HTTPdata
10: /*** $\Delta$ is now fully constructed ***/
11: for all $g$ in SERVLET-GUARDS(S) do
12:     if $\Delta$ violates $g$ then throw a Guard-failure error
13: for all $t$ in SERVLET-TRANSDUCER(S) do
14:     $\delta$ = COMPUTE-UPDATES($t, \Delta$)
15:     $\Delta$ = APPLY-UPDATES($\delta, \Delta$)
16: /*** $\Delta$ is now fully modified ***/
17: Update session and RDBMS data to match $\Delta$
18: Extract output data $O$ and cookie data $C$ from $\Delta$
19: return APPLY(SERVLET-RENDERER(S), O) and $C$
```

The main semantic question about the WEBLOG implementation is how the servlet's transducer is applied. Recall that the servlet's transducer is a sequence of SQL update modules, each of which is a collection of SQL Insert, Delete, and Update commands. The update modules are applied in the order they appear. Applying a module means computing all those tuples that the module dictates should be inserted, deleted, and updated based on the current Δ (COMPUTE-UPDATES above) and once all the updates are computed, applying them to alter Δ (APPLY-UPDATES above). Said another way, if an update contains one insert and one delete command, WEBLOG first computes all those tuples that are to be inserted, all those tuples that are to be deleted, and then inserts and deletes those tuples from Δ. (Any tuple that a module both inserts and deletes is inserted into Δ by WEBLOG.)

4. WEBLOG Security Defenses

Every year the Open Web Application Security Project (OWASP) collects data on web vulnerabilities and ranks them by their prevalence, calling the resulting list the OWASP Top 10. In this section we detail the defenses we have designed for the WEBLOG compiler. Each of WEBLOG's defenses satisfy two main goals: *automation* and *transparency*.

The automation goal requires that, once the developer has written a functional description of the application, the compiler installs its cadre of defenses soundly and completely. Thus, were there formal definitions for each of the OWASP Top 10 vulnerabilities, we could have formally proved that the compiler guarantees that no WEBLOG application is vulnerable to any of the OWASP Top 10. We do allow for the possibility that the functional description does not contain enough information for the compiler to infer how the application ought to behave in response to an attack, in which case the developer provides that information as a security policy (*e.g.*, access control). But that policy is written independently of the application's functionality, and it is the compiler's job, not the developer's, to see that the policy is obeyed by automatically weaving together the application's functional description and its security policies.

The transparency goal requires that for non-malicious users, the functional description of the web application operates the same whether or not the compiler has automatically installed any defenses. Thus the web developer need not even be aware that the defenses are put in place or have any knowledge of how they operate. Transparency is convenient for developers and enables new security defenses to be installed in an existing WEBLOG application by simply re-compiling the application using an updated version of the WEBLOG compiler.

Defenses that have been implemented are marked with ✓. Those that have only been designed are marked with ✗.

4.1 SQL Injection Attacks (SQLIAs) ✓

Of the OWASP Top-10 web application vulnerabilities SQL injection (SQLi) consistently ranks as one of the most prevalent. A SQLi attack is one in which the attacker includes SQL commands inside a text string that the application typically treats as an atomic entity, *e.g.*, a last name or an email address. SQLi attacks cause the application to do things the developer did not intend if those embedded SQL commands are turned into actual commands when the application interacts with its backend RDBMS.

For example, consider the running example where the application retrieves the list of auctions that match the user provided auction title. There is a single input `query`, and we want to retrieve the titles of auctions that match the query. In PHP, we might write the following code to find all the appropriate books.

```
sql.execute("SELECT * FROM Auctions WHERE
    title LIKE " + $_POST['query'])
```

The problem, of course, is that the user can supply SQL commands in the string `query`, which, when concatenated with the SQL fragment above, results in a string that manipulates the database in a way the developer did not intend.

WEBLOG prevents SQLi attacks by requiring developers to write all of their code in a single programming language (SQL) instead of two separate languages (e.g., PHP and SQL). This important design choice allows us to deal with defending SQLi in a natural way: SQLi is only possible because code written in a language such as PHP generates SQL code at run-time by concatenating SQL commands together with user input. In contrast, WEBLOG developers write all the SQL commands directly, without trying to combine the query "code" with input. User input affects the results of the queries and updates in much the same way, but those commands are expressed in a safer syntax that ensures that user inputs are never treated as SQL keywords.

4.2 Cross-site Scripting (XSS) ✓

A cross-site scripting attack (ranked by OWASP as the 2nd most prevalent) is one in which the attacker includes a command (*e.g.*, in JavaScript) that is meaningful for the web browser within a text string that the application does not expect to contain a command, *e.g.*, a text string representing an email address. When that string is sent back to another user's browser, commands can be executed and compromise that user's security or privacy. Just like for SQLi, commands supplied by the user are being executed as they were issued by the web application itself.

The standard defense from cross-site scripting is to sanitize every piece of user data before sending it to the browser (where the sanitization performed depends on the HTML context in which that data is to appear). Sanitization ensures commands embedded in user data are not interpreted as commands by the web browser. For example, the string

```
<img src = "javascript:alert()"></img>
```

is sanitized to

```
&lt;img src = &quot;javascript:alert()&quot;&gt;
```

thereby ensuring that the HTML command embedded within the string (the `<img>` tag) are not executed by the browser but are rather treated as data.

This defense, while thorough, is impractical for those web applications that actually want their users to submit HTML commands within their data, *e.g.*, so as to format blog posts or descriptions of items for sale. Thus for many applications, there are two kinds of data elements: those that are allowed to include HTML tags and those that are not. And even data elements permitted to include HTML are not typically allowed to include arbitrary HTML because of security vulnerabilities; rather, they are allowed to contain only certain HTML tags.

WEBLOG protects against XSS attacks by sanitizing all user-supplied data (depending on the HTML context in which that data is embedded) when that data is sent to the user, unless the developer marks the data element as not needing sanitization, and WEBLOG ensures that the values assigned to data elements not needing sanitization cannot produce XSS attacks. To mark an element as not needing sanitization, the developer declares a data element to be of type `HTML`, `integer`, or `boolean`. The `integer` and `boolean` types can only be assigned the obvious values; the `HTML` type can only be assigned strings from a fragment of HTML that is parsed consistently across all browsers and is (strongly believed to be) devoid of XSS attacks [22]. All undeclared data elements are of type `string` and are sanitized when output to the user.

These types are enforced in two ways. Whenever data is submitted to a WEBLOG application from a user, the types of that user data are checked, and an error is thrown if user data fails to be of the proper type. (Implementationally, the compiler adds a new guard to each servlet that checks types.) Furthermore, WEBLOG performs type inference and type checking at compile time to ensure that a developer does not, by omission or by commission, copy a data value of type `string` into a data element declared to be of type `integer`, `boolean`, or `HTML`, thereby circumventing the sanitization that ought to have been performed on the data. This requires ensuring that no implicit casting is performed and that there are no unsafe builtins that perform type casting. WEBLOG's type system ensures that a data element's type follows that data element around and hence when it is sent to the renderer can be properly sanitized according to its context as well as its content.

The reader might have observed that WEBLOG performs *output sanitization* (sanitization is performed only when data is sent to the renderer) as opposed to the more conservative *input sanitization* (sanitization occurs when the data is received by the application). Input sanitization fails to meet our design goal of transparency: the code that the developer writes is less likely to behave the same if the XSS defense were turned off. For example, assume the developer wishes to check the length of a stored string value. With input sanitization (which may convert "<" to "&lt") the length of a string value is different after sanitization, whereas output sanitization only changes values after the web application has finished processing them.

4.3 Parameter Tampering ✓

Similar to SQLi and XSS, parameter tampering attacks involve submitting data to a web server that the developer did not expect. A parameter tampering attack is one where the attacker circumvents the client's user interface to submit data to the web server that the client would have rejected. Parameter tampering attacks are often successful on web forms that include a drop-down list with a fixed set of choices for the user. If a malicious user chooses a value not in the list (*e.g.*, by manually sending an HTTP request to the web server), and the server neglects to check that the submitted value is one of the permitted choices, an attack can compromise the integrity of the application [6].

For example, WeBid asks a user to select one of her pre-registered financial accounts to pay for any new merchandise she wins at an auction. If instead of selecting one of her pre-registered accounts from the drop-down list WeBid provides, she submits someone else's account number, the server might charge the other person for her merchandise.

The cause of parameter tampering vulnerabilities is that application developers must write and maintain separate code bases for the client and server, which over time can become unsynchronized. If the client performs data validation that the server fails to perform, no amount of client testing will reveal the unintended behavior of the application under parameter tampering attacks because testers will always submit data that passes the client validation checks.

WEBLOG addresses parameter tampering attacks by automatically replicating any data validation checks performed by the client onto the server. This ensures that any data that the client would reject is also rejected by the server; hence, all parameter tampering attacks are eliminated. This replication process requires three steps. First the WEBLOG compiler reformulates the application so that each servlet processes exactly one form, *i.e.*, that every form is submitted to a distinct servlet. If k forms submit to the same servlet, the compiler makes k copies of that servlet and changes the targets of the k forms to point to different copies. Second, the WEBLOG compiler walks over all forms and adds each form's guards to the guards of the corresponding servlet. Finally, the compiler generates a new guard for each servlet that ensures the only form fields submitted is one of those that appears in the form. This last guard prevents negative tampering attacks, which are described in [7].

4.4 Access Control ✓

Access control is the problem of ensuring that only certain users can manipulate sensitive resources in permitted ways. It is not mentioned directly in the OWASP Top 10 but addresses two of those vulnerabilities nevertheless. One kind of Session Management vulnerability (OWASP 3) is when the application places sensitive information in the user's Cookie instead of the Session or RDBMS. Access control protection ensures that all the data sent to the client, whether in the cookies or in the HTML page, obeys the access control policy. The other vulnerability addressed by access control is called Insecure Direct Object References (OWASP 4). These can arise when the user requests data by issuing a URL that the developer did not anticipate. Proper access control protection ensures that no user is sent data she is not authorized for, regardless of how or when it is requested.

Typically, access control is enforced implicitly by a web application's code. In WEBLOG, however, a developer writes an access control policy separately from the application's functional description and WEBLOG weaves the two together to implement the functionality of the application while simultaneously obeying the access control policy. WEBLOG allows a developer to express such a policy using SQL statements that describe the fragment of the database (the RDBMS, Session, Cookie) that the application is allowed to read and write, conditioned on the current database.

For example, in WeBid, to require that a user profile is only writable by the user herself, the developer would state that the tuples of the `User` table in the RDBMS that are writable are those where the `username` is the same as the `username` of the Session as shown in Listing 6. These statements are syntactically similar to SQL's GRANT/REVOKE statements.

Currently, WEBLOG only supports *write* policies, since they are easier to enforce than *read* policies. *read* policies are more difficult to enforce because the compiler must ensure that given the output of the servlet and the source code for the application, the user can only *infer* information that she is allowed to *read*.

To enforce a `write` policy on a servlet, WEBLOG analyzes the servlet code that inserts tuples and the servlet code that deletes tuples. Because of SQL's syntax and WEBLOG's support for only a fragment of SQL, it is straightforward to extract all the conditions (written in SQL) on the HTTP inputs, session, cookies, and RDBMS under which the servlet inserts data into a given table. It is convenient to think of these conditions as a (possibly complex) SQL integrity constraint. Likewise, it is easy to extract all the conditions (written in SQL) under which data is deleted from a given table, and it is easy to extract all the conditions under which the access control policy allows a given table to be written. We use ι_t to denote the conditions under which the servlet inserts tuples into table t, δ_t to denote the conditions under which the servlet deletes tuples from table t, and ζ to denote the conditions under which the access control policy allows table t to be written.

Conceptually, a servlet obeys an AC write policy if for every table t and for all possible HTTP inputs, sessions, cookies, and RDBMS databases, all the tuples inserted and deleted are allowed to be written, *i.e.*, that the following is always true: $\iota_t \vee \delta_t \Rightarrow \zeta$.

This analysis can be delegated to off-the-shelf, first-order automated reasoning systems because WEBLOG supports only that fragment of SQL that can be easily translated into first-order logic. Thus the formula above can be viewed as a formula in first-order logic, where if that formula is valid (*i.e.*, true for all HTTP inputs, sessions, cookies, and RDBMS databases), then the servlet obeys the access control policy. For lack of space, we give no details about

Listing 6.

```
ALLOW write ON SELECT User.* FROM User, Session WHERE User.username = Session.username
```

converting SQL to and from first-order logic but refer the reader to well-known results on the topic [5].

If the formula above fails to hold (which we discover because the automated reasoning system invoked by the compiler provides a counterexample or the system fails to find a proof in a reasonable amount of time), it is a simple matter to modify the servlet in question to enforce the access control policy. The compiler adds a new guard to the servlet that checks if the combination of the HTTP request data, cookies, session, and RDBMS data violates the formula above, and if so rejects the data. By adding such a guard, the compiler guarantees that the servlet will never be executed on inputs that will violate the access control policy. The guard that the compiler constructs is simply the formula above converted from first-order logic back into SQL. This is a rather direct implementation of the access control policy; the real difficulty lies in simplifying that guard and eliminating checks that the servlet's other guards perform. The check that the formula above is true for *all* databases can be seen as an extreme attempt at simplification: one in which the guard is eliminated altogether.

4.5 Cross Site Request Forgery (CSRF) ✗

CSRF attacks (ranked at OWASP 5) arise when one web application fools a user into executing functionality on a different web application. For example, a third-party application could produce a web page with a link labeled as "Home" that when clicked sends an HTTP request to WeBid that places a new bid on an existing auction. Such attacks are only possible because the WeBid server cannot tell whether the request originated from a legitimate WeBid client page or from another application.

The typical defense against CSRF attacks is to embed a unique token in every client web page that is sent to the server whenever sensitive operations are performed. The server only performs those sensitive operations when the right token is included in the HTTP request. Since a WEBLOG application declares what forms / links are present on each page, the compiler can automatically add the required tokens along with guards that ensure the proper token is received. Moreover, the compiler can also automatically detect which servlets cause side effects (by inspecting whether they make updates to the RDBMS/cookie/session), and hence can add tokens for only those forms / links whose targets will cause side effects. This alleviates the problem of *overprotection* (protecting requests that are do not change state at the server), as tokens will only be checked for requests that can make state changes to the server.

4.6 Session Management Vulnerabilities ✗

Session Mangement Vulnerabilities (OWASP 3) are of two sorts. The first arise because the developer made simple mistakes about the mechanics of creating a cookie representing the session ID for a given request. Similar to many modern web development frameworks, WEBLOG directly implements session management and hence addresses mechanical session management vulnerabilities. A WEBLOG developer need manipulate the session for each request, and WEBLOG automatically includes the necessary session ID cookie in the response (whenever the session is non-empty). Similarly, each incoming request with a session ID cookie causes WEBLOG to load the corresponding session data automatically. The second kind of vulnerability arises because the developer mistakenly sends sensitive information to the user through cookies. This type of session management vulnerability is addressed by WEBLOG's access control protection, described in Section 4.4.

4.7 Cryptographic storage ✗

Cryptographic storage vulnerabilities (OWASP 7) arise when sensitive data stored in the RDBMS (or in files) is not encrypted to protect against attackers who gain access to the database itself. We envision the developer writing a separate cryptographic security policy dictating which fragments of the database ought to be encrypted upon storage. Given such a policy, the compiler automatically (a) adds a final update to each servlet that encrypts data just before saving it to the backend RDBMS and (b) modifies each servlet so that the data is decrypted before any guard is checked or update performed on that data. The compiler could improve the performance of this basic approach by only decrypting that data that the servlet needs when it needs it. For example, if a servlet only checks the equality of two sensitive pieces of data and the encryption algorithm is deterministic, decryption is unnecessary and the compiler could avoid inserting the decryption code.

4.8 Transport layer ✗

Transport layer vulnerabilities (OWASP 9) arise when sensitive data is not encrypted in messages sent between client and server (*e.g.*, the application fails to use HTTPS when requesting credit card numbers). Similar to cryptographic storage vulnerabilities, WEBLOG could allow the developer to write a transport layer policy dictating which fragments of the database ought to be protected in transit, and then the compiler could choose HTTPS for any servlet sending or receiving information that is derived from or saved to a sensitive fragment of the database.

4.9 Forced browsing ✗

A forced browsing attack (OWASP 8 & 10) occurs when the attacker requests URLs in an order that the application did not anticipate. For example, an attacker could try to skip crucial pages in an e-commerce shopping cart application to manipulate the total price of her purchases. WEBLOG's underlying access control model ensures that no single servlet invocation will leak sensitive information, but does not ensure that multiple servlets will interact as they were intended (*e.g.*, redirects and forwards). We are currently working on adding a workflow policy language that enables developers to describe how servlets ought to interact so that the compiler can automatically enforce that policy.

5. Evaluation

To evaluate WEBLOG, we addressed the following questions. Which common web application features is WEBLOG expressive enough to implement? And, can WEBLOG be used to implement a real-world application in its entirety, and how does that implementation's security compare to an implementation in a traditional web development framework? When evaluating security, we concentrated on the four defenses our implementation currently supports (implementations for the remaining defenses are underway): SQLi, XSS, parameter tampering, and access control.

5.1 Web Application Features

To understand which features are important in today's web applications, we surveyed two existing ones: WeBid (an online auction website used as the running example in this paper) and B2Evolution (a blogging platform) [1]. We analyzed WeBid closely and found that the features it uses are limited to CRUD (creating, retrieving, updating, and deleting information from the database), sending emails, basic file IO (*e.g.*, uploading file from disk), processing credit cards/paypal payments, captcha support, basic computation (*e.g.*, image manipulation for creating thumbnails), AJAX support, and HTTP redirects. The following table summarizes the prevalence of these features in WeBid in terms of the number of PHP files concerned primarily with each feature. For B2Evolution, we made a cursory analysis looking for features not appearing in WeBid but found nothing noteworthy.

74%	CRUD + HTTP redirects
8%	Email support
7%	File IO
3%	Payment processing
3%	Captcha
3%	Computation
1%	AJAX

CRUD (Create, Retrieve, Update, Delete) CRUD operations are the actions necessary to manipulate the data contained in the back-end relational data management system (RDBMS) of an application: create new elements, retrieve existing elements, update existing elements, and delete unwanted elements. In both WeBid and B2Evolution, for example, user profiles must be created, the details of a profile auction must be retrieved from the database so as to be displayed to the user, profiles must be updated if the user wants to change her email address, and profiles must be deleted if the user decides to terminate her use of the application.

Both WeBid and B2Evolution are implemented in PHP, where all of the CRUD operations (which are implemented manually) follow the same basic outline. Check that the user-provided data is well-formed and either reject malformed data or sanitize it. Dynamically construct SQL statements by concatenating developer-written SQL fragments with user-provided data. Execute the SQL statement. Iterate over the results of the execution and output HTML tags to mark up each of the results. Typically, each CRUD operation is implemented by a separate piece of PHP code, at the end of which a different SQL command is executed. Listing 7 contains examples of SQL commands for CRUD operations.

When developing CRUD operations with WEBLOG, we can often combine the code for CUD into a single SQL update module. The servlets for creation, updating, and deletion can then all use this same module. The only reason the Retrieve functionality cannot be included is that instead of moving data from the user to the RDBMS, the R operation moves data from the RDBMS to temporary tables for output to the user. This is shown in Listing 8.

Search While analyzing WeBid, we discovered that half of the CRUD-only PHP files are limited to simply reading from the database, but not writing to it. These files can therefore be viewed as providing search functionality; here we focus on implementing an advanced search, a common feature of today's applications. The advanced search mechanism allows a user to search for information by specifying conditions on different attributes. For example, WeBid allows users to search for an auction by putting constraints on the auction title, description, and maximum bid price.

The PHP implementation of WeBid's advanced search dynamically constructs the appropriate SQL query by iteratively combining developer-written SQL fragments with user-data. The difficult part is ensuring that only those attributes that the user gave conditions for are used in the resulting query. For example, if the user selected a maximum bid price and included a query string for the title but imposed no restrictions on the description, the SQL query must include the bid price and the title in the WHERE clause but not the description. The following PHP snippet gives the flavor of how the PHP code constructs such a query.

```
$query = "SELECT * FROM auctions";
if ($title != "" || $desc} != "")
   $query .= " WHERE ";
if ($title != "")
   $query .= "title = ".$title;
if ($desc != "")
   $query .= "OR description = ".$desc;
```

Implementing this feature in WEBLOG requires care because SQL queries cannot be generated dynamically. Instead of dynamic generation, we can think of the construction of the search results as a sequence of database updates, each of which filters the auctions found so far by a different search attribute. In our example, the first update creates a temporary table with all of the available auctions. The second update eliminates all those auctions that fail to satisfy any `title` search criteria or is a noop if no search criteria are imposed. The third update eliminates all those auctions that fail to satisfy the `description` search criteria, and the fourth update handles the `bid` critera. See Listing 9.

Web Form Validation Today's web applications typically include web forms for soliciting user data that validate the user's data before sending it to the server. For example, both WeBid and B2Evolution check that when someone creates a new user, the email address has the right format. Because client-side validation cannot be trusted by the server (an attacker can disable JavaScript or manually send an arbitrary HTTP request), the same validation must be duplicated on the server. Writing and maintaining two different code bases (often in different languages) implementing the same functionality is a notoriously difficult problem.

While PHP includes some support for validation (`http://www.php.net/manual/en/book.filter.php`), the developer typically writes the JavaScript validation code, the PHP validation code, and the HTML for displaying the form. Ruby on Rails (ROR) has built-in support for server-side validation but still requires manually written JavaScript code for client-side validation. The Google Web Toolkit (GWT) is the forerunner in this category because it allows the developer to dictate which server-side code should be duplicated on the client.

In WEBLOG, a developer wanting to build a web form dictates what kind of data the form is supposed to solicit from the user and allows WEBLOG to do the rest. For a basic form without client-side validation, the developer gives just a database schema for the user to fill out, and WEBLOG generates HTML for building that form. To include client-side validation, the developer adds SQL guard modules, and WEBLOG then automatically generates the JavaScript for checking those constraints. WEBLOG also checks those constraints on the server when data is submitted. Thus, the developer only writes a single validation code base, and WEBLOG duplicates that validation code wherever it is needed.

Limitations One of known limitations of WEBLOG is lack of support for systems operations, such as sending emails, writing files, caching, Apache configuration. We will only add such support if doing so does not diminish the compiler's ability to eliminate security vulnerabilities. For example, file manipulation is likely to make deep analysis far more difficult and will likely not be supported. Another class of limitations is that of administrative control: when significant customization or sophisticated administrative interfaces are used. For example, a page that accepts an arbitrary SQL query and returns the results cannot be built in WEBLOG. A third class of limitations arises when significant computation is necessary, *e.g.*, image manipulation. Such algorithms can be written in traditional languages and then "called" by referencing special database tables

Listing 7.

```
sql.execute("INSERT INTO users VALUES ($username,$pwd,$email,$bio)");
sql.execute("SELECT * FROM users WHERE username = $username")
sql.execute("UPDATE users SET email = $email, biography = $bio WHERE username = $username");
sql.execute("DELETE FROM users WHERE username = $username");
```

Listing 8.

```
defupdate user-cud
   DELETE FROM Users WHERE EXISTS (SELECT * FROM HttpInput WHERE HttpInput.login = Users.login)
   INSERT INTO Users (login, first, last, email) SELECT login, first, last, email FROM HttpInput
```

Listing 9.

```
defservlet display-search-results
   :transducer (upd0 upd1 upd2 upd3)

defupdate  upd0
  INSERT INTO Search * SELECT * FROM Auction

defupdate upd1
  DELETE FROM Search WHERE EXISTS (SELECT * FROM HttpInput H WHERE H.title != "" AND H.title LIKE Search
      .title)

defupdate upd2
  DELETE FROM Search WHERE EXISTS (SELECT * FROM HttpInput H WHERE H.desc != "" AND H.title LIKE Search.
      desc)

defupdate upd3
  DELETE FROM Search WHERE EXISTS (SELECT * FROM HttpInput H WHERE H.bid != "" AND H.bid >= Search.
      maxbid)
```

in WEBLOG, but we must investigate how the addition of such algorithms affects WEBLOG's security guarantees.

5.2 Expressiveness and Security

Next we evaluated whether or not WEBLOG is sufficiently expressive to build an entire web application and how the resulting application compares in terms of security to the same app written by someone of roughly the same skill in a more traditional web development framework. We experimented on the existing application WebSubRev [4], an open source application for submitting papers to a conference that has been used at several dozen venues, and asked one student to re-implement it in WEBLOG and another to re-implement it in Sinatra (a Ruby-based framework).

Implementations The most important result is that it is possible to implement WebSubRev in its entirety using WEBLOG. WebSubRev is mainly a collection of CRUD operations (saving submitted papers in a database), a task for which WEBLOG excels. The WEBLOG and Sinatra implementations also turned out to be quite similar syntactically and required roughly the same amount of time to write, about an hour. Their similarity can be attributed to the correspondences between language constructs in Sinatra and WEBLOG (see the below table for a summary).

RUBY/SINATRA	WEBLOG
`get '/submit'`	`defservlet new-submit`
`post '/submit'`	`defservlet submit`
`.erb files`	`defrenderer`
`DataMapper models`	`defschema`

Security The second interesting result was that despite their similar syntax and development times, the two versions differed in terms of security/usability. Both applications are free of SQL injection attacks. For Sinatra, this was a byproduct of using an object-relational mapper (ORM), which safely handles the common functions like writing an object to the database. However SQL injection attacks would be possible if extending the experiment to include functionality not covered by the ORM, such as advanced search. In contrast, the WEBLOG application developer could not introduce a SQL injection attack even if she tried.

For cross-site scripting (XSS), the Sinatra implementation of WebSubRev was vulnerable to attack. Any HTML code submitted via the form is blindly stored in the database and sent to the browser unsanitized. The developer relied too heavily on the ORM's built-in functionality and neglected to consider the problem of XSS. Had we used Ruby on Rails, an XSS defense is built-in, but that defense must be disabled to allow users to format their data using HTML, thereby making it vulnerable to XSS attack.

For parameter tampering, the Sinatra implementation was not vulnerable to attack because it included all the proper data validation on the server; however, it included no client-side validation since doing so required writing validators in JavaScript. The WEBLOG implementation was also immune from attack but included both client-side and server-side validation, despite the developer having only written client-side validation. The automatic security of WEBLOG enabled the developer to build a more user-friendly application that was just as secure as the Sinatra application.

Finally, for access control neither implementation had any access control policy violations, mainly because WebSubRev had so few features that there was little opportunity.

6. Related Work

There have been numerous approaches to web development based on the declarative paradigm and SQL, *e.g.*, Hilda [23], SAFE [18], FORWARD [14], Strudel [13], Spicey [15], and SVC [25]. Typically this work is concerned with providing a language where the compiler can automate the implementation of conceptually simple but technically tedious or delicate functionality. For example, FORWARD allows a web developer to build clients that display a fragment of the application's backend database where updates are automatically propagated via AJAX. To use FORWARD, the developer writes the SQL query describing the fragment of the database that should be displayed, and the compiler automatically generates the necessary AJAX and server code. In contrast to languages aimed at providing additional functionality, WEBLOG was designed to provide security guarantees—to completely eliminate those security vulnerabilities that are most prevalent in today's web applications.

Within the security community, the work most related to WEBLOG falls under the category of *by-construction security*, which [17] describes comprehensively. The most relevant work is is SELinks [12], SIF [10], Swift [9], Resin [24], UrFlow [8], [19], [20], and [16]. All these works are based on fully expressive, traditional programming languages (such as Java and Haskell) and are carefully designed to eliminate one or two known security vulnerabilities from web applications. But because their underlying languages are fully expressive, extending these approaches to eliminate additional vulnerabilities requires redesigning the language, if it is possible at all. In contrast, because WEBLOG is less expressive it is far more likely that additional security guarantees can be implemented by simply improving the compiler. When WEBLOG was designed, we only considered protections against SQLi, XSS, and parameter tampering. All of the other defenses (including access control) were added on after the design was complete. In short, this paper reports on the benefits and drawbacks of languages with reduced expressiveness for securing applications deployed on the World Wide Web.

7. Conclusion

WEBLOG is a declarative web development language designed to eliminate today's most prevalent security vulnerabilities. WEBLOG's compiler takes a description of an application's basic functionality together with independently authored security policies and generates code that simultaneously implements the desired functionality while respecting the developer's policies. We evaluated WEBLOG by implementing real-world applications, analyzing their security properties, and investigating the WEBLOG development process. Overall, we found that the reduced expressiveness of WEBLOG enables many strong security guarantees while enabling the development of web applications common in small businesses.

8. Acknowledgements

This work was supported in part by the National Science Foundation under awards CNS-1228697, CNS-1141863, CNS-1035914, CCF-0916438, DGE-1069311, CNS-1065537 and CNS-0845894.

References

[1] B2evolution app. `http://b2evolution.net/`.

[2] Common weaknesses enumeration. `http://cwe.mitre.org/`.

[3] Webid app. `http://sourceforge.net/projects/simpleauction/`,.

[4] Websubrev app. `http://people.csail.mit.edu/shaih/websubrev/`,.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2010.

[7] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2011.

[8] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 31–44, 2007.

[10] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the Usenix Security Symposium*, 2007.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[12] B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIG for the Management of Data*, 2009.

[13] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Declarative specification of web sites with strudel. *The VLDB Journal*, 2000.

[14] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The SQL-based all-declarative FORWARD web application development framework. In *Proceedings of the Conference on Innovative Data Systems Research*, 2011.

[15] M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. In *the Symposium on Practical Aspects of Declarative Languages*, pages 201–216, 2010.

[16] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *Proceedings of the International World Wide Web Conference*, 2010.

[17] X. Li and Y. Xue. A survey on web application security. Technical report, Vanderbilt University, 2011. URL `http://www.truststc.org/pubs/814.html`.

[18] R. M. Reischuk, M. Backes, and J. Gehrke. SAFE extensibility for data-driven web applications. In *Proceedings of the World Wide Web*, 2012.

[19] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the Usenix Security Symposium*, 2009.

[20] M. Samuel, P. Saxena, and D. Song. Context-sensitive autosanitization in web templating languages using type qualifiers. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 587–600, 2011.

[21] The Open Web Application Security Project. The Ten Most Critical Web Application Security Vulnerabilities. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[22] E. Z. Yang. HTML Purifier. `http://htmlpurifier.org`.

[23] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *the International Conference on Data Engineering*, 2006.

[24] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertion. In *the ACM Symposium on Operating Systems Principles*, 2009.

[25] W. P. Zeller and E. W. Felten. Svc: Selector-based view composition for web frameworks. In *Proceedings of the USENIX Conference on Web Application Development*, 2010.