Developing Secure Web Applications

Security is too critical to leave up to individual coders. The Secure Web Applications Project enforces centralized security policies across entire Web sites.

David Scott and Richard Sharp
University of Cambridge

Although traditional firewalls have effectively prevented network-level attacks, most future attacks will be at the application level, where current security mechanisms are woefully inadequate.[1] Application-level security vulnerabilities are inherent in a Web application's code, regardless of the technology in which the application is implemented or the security of the Web server and backend database on which it is built. A recent advisory published by Internet Security Systems (see the "Internet Resources" sidebar, p. 44) claims that 11 widely deployed shopping cart applications are vulnerable to a simple attack that lets hackers purchase goods for much less than their listed price. Worryingly, the attack does not require particular technical skill; it suffices to save the shopping cart's HTML confirmation form to disk, use a text editor to modify the price of the goods (stored in a hidden form field), and load the HTML form back into the browser.

Application-level security vulnerabilities are well known, and many articles discuss ways to avoid them. Fixing a single occurrence of a vulnerability is usually easy. However, the massive number of interactions between different components of a dynamic Web site makes application-level security challenging in general. Despite numerous efforts to tighten application-level security through code review and other software engineering practices, many professionally designed Web sites still suffer from serious application-level security holes. This evidence suggests a need for higher-level tools and techniques to address the problem.

The Secure Web Applications Project (SWAP) is an interdisciplinary research initiative between the Laboratory for Communications Engineering and the

 1089-7801/02/$17.00

Related Work on Application-Level Security

Traditionally, the task of preventing unauthorized activity at the application protocol level has been left to network firewalls. Many companies provide application-level firewalls as commercial products. Typical services provided by such firewalls include virus protection and access control. However, we are not aware of any application-level firewalls that are flexible enough to apply fine-grained user-specified security policies—most will either completely allow or completely deny a particular service.

RBAC-Based Systems

Damiani and colleagues[1] describe a method for enforcing role-based access control (RBAC) policies for remote method invocations via the simple object access protocol (SOAP). These policies are very different from ours: They consider access-control issues, whereas we try to prevent application-level attacks in general. The similarity between their system and Spectre lies in the use of a firewall to enforce restrictions at the HTTP level.

Park, Sandhu, and Ahn[2] describe architectures for implementing RBAC for Web applications via HTTP. In one design, they discuss using cookies to cache authentication information securely on the client side, ensuring integrity through a Message Authentication Code (MAC) (this technique is also discussed elsewhere[3]). Our security gateway uses the same method to ensure the integrity of arbitrary application cookies. The main differences between their work and ours is that we wanted to integrate pre-existing components for which no code is available and to express a wide range of security policies.

Language-Based Systems

The <bigwig> project (see the "Internet Resources" sidebar, p. 44) consists of domain-specific languages and tools for developing Web services. A part of the <bigwig> project, PowerForms, allows developers to attach constraints (expressed as regular expressions) to form fields.[4] A compiler generates both client-side JavaScript and code for server-side checks. <bigwig> and our SWAP differ in two important ways: SWAP includes a general-purpose validation language, which <bigwig> does not; and Spectre/SPDL allows developers to check validation constraints on both the client and server irrespective of the language the original application was developed in, while PowerForms does not.

Hanus created a sophisticated CGI library using the multiparadigm declarative language Curry.[5] Using an HTML abstraction layer, logical variables to reference user input, and an event-handling model, it is possible to concisely program complex sequences of interactions. However, security was not a primary goal of the project. Although the high-level approach makes writing secure software much easier, the system offers no explicit support for countering possible application-level vulnerabilities.

Commercial System

The software development company Sanctum sells a product called AppShield that, like Spectre, inspects HTTP messages in an attempt to prevent application-level attacks. Despite this apparent similarity, however, there are significant differences between the two systems: We prefer the programmatic approach of specifying a security policy explicitly, whereas AppShield has no SPDL and attempts to infer a security policy entirely dynamically. While this allows AppShield to be installed quickly, it limits its functionality. In particular, because AppShield has no policy description language for describing validation or transformation rules, it knows very little about what constitutes valid parameter values in HTTP requests and can perform only simple checks on data returned from clients.

References

1. E. Damiani et al., "Fine Grained Access Control for SOAP E-Services," *Proc. 10th Int'l World Wide Web Conf.*, ACM Press, New York, 2001, pp. 504-513.
2. J.S. Park, R. Sandhu, and G.J. Ahn, "Role-Based Access Control on the Web," *ACM Trans. Information and System Security*, vol. 4, no. 1, Feb. 2001, pp. 37-71.
3. J.S. Park and R.S. Sandhu, "Secure Cookies on the Web," *IEEE Internet Computing*, vol. 4, no. 4, July/August 2000, pp. 36-44.
4. C. Brabrand et al., "PowerForms: Declarative Client-Side Form Field Validation," *World Wide Web J.*, vol. 3, no. 4, 2000, pp. 205-214.
5. M. Hanus, "High-Level Server Side Web Scripting in Curry," in *Practical Aspects of Declarative Languages: Proc. 3rd Int'l Workshop, PADL 01*, I.V. Ramakrishnan, ed., Springer-Verlag, Heidelberg, Germany, 2001.

Computer Laboratory of the University of Cambridge that seeks to address the problem of application-level Web security at a higher level. The project reduces application development time and protects against a large class of application-level attacks more effectively than existing Web development methodologies (see the sidebar "Related Work on Application-Level Security").

Application-Level Vulnerabilities

One factor that contributes to the prevalence of application-level vulnerabilities in practice is that it is difficult to abstract security-related code from a large Web application in a structured manner using existing languages and tools.

- *The Web application might be written in a variety of (noninteroperating) languages.* In this case there is no easy way to abstract security-related code behind a clean API. As a consequence, security-related code is scattered throughout the application. This lack of structure makes fixing vulnerabilities difficult, because the same security hole can occur multiple times.
- *The languages used for Web development are not always conducive to writing security-related code.* In particular, it is difficult to give any

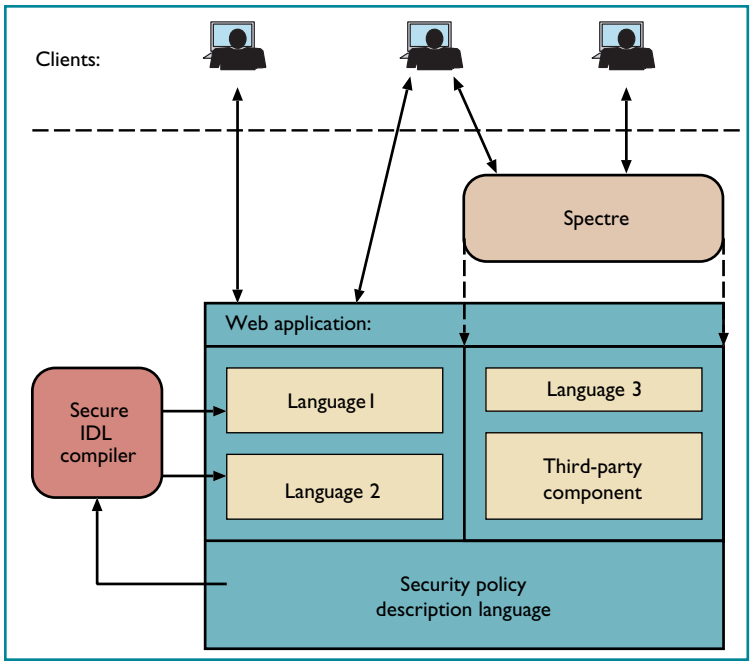

Figure 1. A Web application secured using SWAP tools. Using these tools, developers can abstract security-related code from a Web service.

compile-time guarantees about untyped scripting languages such as PHP: Hypertext Preprocessor (PHP) and Microsoft's Visual Basic Scripting Edition (VBScript).

- *Web applications often contain third-party components*. Because modifying the source of third-party components might not be viable (either because the code was shipped in binary form or because the license agreement is prohibitive), determining how to fix security vulnerabilities is difficult. In reality, a developer is often at the mercy of the company that wrote the component.

In previous work, we proposed a framework to alleviate these problems.[2] Our system used a specialized Security Policy Description Language (SPDL) to program an application-level firewall, which we call a *security gateway*. Security policies written in SPDL are compiled for execution on the security gateway. The security gateway dynamically analyzes and transforms HTTP requests and responses to enforce the specified policy.

Our current work extends this research in a number of ways. First, we have enhanced our existing framework for securing dynamic Web applications, christening the resulting suite of tools Spectre (Security Policy Enforcement through Runtime Checks). Second, we have developed a Secure IDL compiler that, in cases where the source code is available, allows SPDL specifications to be folded directly into the application. We have also designed an object-oriented framework for developing secure Web applications.

SWAP Overview

Because it is almost impossible to abstract security-related code from complex Web applications using existing languages and tools, ensuring a Web application's security requires every programmer involved in the project to be intimately aware of all common application-level security vulnerabilities and to manually write the necessary code to protect against them. Mistakes or oversights are inevitable, however, and they invariably lead to security holes.

Our goal is to develop tools and techniques that remove as many security-related responsibilities as possible from regular coders. Figure 1 shows a typical Web application secured using the SWAP tools. The application contains source code written in three different languages, as well as some third-party components, for which we assume that the source is inaccessible. Spectre protects the rightmost part of the application by dynamically transforming HTTP requests and responses in accordance with the SPDL security policy written for the application.

As you can see, Spectre does not protect the parts of the application written in Language 1 and Language 2. Instead, our secure interface definition language (IDL) compiler has folded the relevant parts of the SPDL policy directly into the source code. The IDL compiler provides multilanguage support, allowing us to translate the SPDL into both Language 1 and Language 2.

Of course, a developer writing a Web application from scratch could limit himself to using only one language, and thus could abstract much of the security-related code into a support library. For cases such as this, we have developed an object-oriented API that, unlike existing APIs for Web development (such as ASP and Java servlets), is designed specifically with security in mind. Besides providing a structured framework for dealing with access control, our API provides a secure database abstraction layer. Our security-aware API can be combined with SPDL-generated stubs to effectively enforce both access control and parameter validation.

The Spectre Tools

Spectre has four components:

- SPDL, which can express validation constraints and transformation rules on a per-URL basis;
- a policy compiler that automatically translates SPDL into code that checks the validation con-

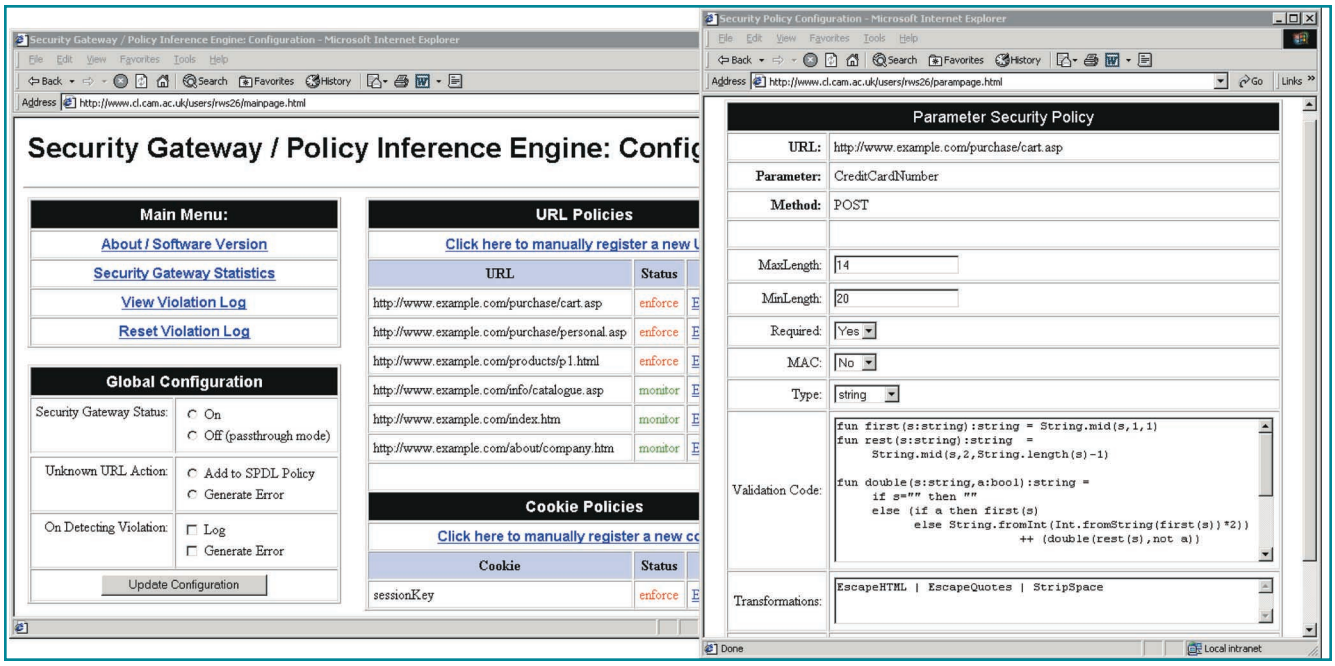

Figure 2. The Spectre user interface. Using the parameter security policy form, developers specify validation constraints and transformation rules for cookies and form and URL parameters.

straints and applies transformation rules;
- a security gateway that dynamically enforces security policies; and
- a user interface that lets developers dynamically create and modify policies.

SPDL facilitates the definition of validation constraints and transformation rules. Validation constraints place restrictions on the interaction between clients and Web applications (for example, "the value of this cookie must never be modified" or "this form-field must contain a valid credit-card number"). Transformation rules specify various transformations that will be applied to user input (for example, "pass data from all fields on this form through a function to escape HTML metacharacters").

Figure 2 shows a screenshot of the Spectre user interface. The leftmost window (main configuration settings) provides global configuration options and lists the cookies and URLs to be secured. The rightmost window shows the parameter security policy form. Using this interface, a designer can specify validation constraints and transformation rules for individual form parameters, URL parameters, and cookies.

The screenshot specifies the security policy for the form parameter `CreditCardNumber`. Validation constraints include bounds on the length of data passed through the parameter, the type of data expected (such as `string`, `int`, `float`, or `bool`), and whether the user must supply the parameter. The designer can specify validation code in a general-purpose programming language that, in our current implementation, resembles a simply typed subset of ML. In the screenshot, the validation code implements the Luhn formula, a commonly used validation check for credit-card numbers.

The transformation control on the parameter security policy form lets the designer specify transformations to be applied to incoming data. Transformations are selected from a user-extensible library (currently implemented in Objective Caml, or OCaml[3]). In this example, we apply transformations that escape HTML metacharacters, preventing cross-site scripting attacks; escape quotes, preventing a class of Structured Query Language (SQL) attacks; and strip out spaces, removing superfluous formatting from credit-card numbers.

The contention between the statelessness of HTTP and the statefulness of many Web applications leaves application designers to manage state explicitly on an ad hoc basis. A common technique — albeit an insecure one — is to thread state through client requests and responses, thus alleviating the overhead of storing state centrally on the server side. Cookies, URL parameters, and hidden form-fields are often used for this purpose. We do not describe it in detail here, but a special validation constraint in Spectre allows data to be threaded through clients securely. Message authentication codes (MACs) are generated using the keyed-hash MAC (HMAC)[4] algorithm and checked dynamically to ensure that clients have not maliciously modified security-critical data.[2] This pro-

```
(* Script parameters are checked and bound here *)
let spec = get_args
     [ ("item_id", Int {Min=0;Max=999999}, Required);
       ("customer_name", String {MinLength=1;MaxLength=30},
       Required) ]

module Parameters = struct
     let item_id =
       int_of_string (valof (lookup_arg spec "item_id"))
     let customer_name =
       valof (lookup_arg spec "customer_name")
end

(* Pages are derived from this skeleton class   *)
class skeleton =
     object (self)
       ...
       method item_id       = Parameters.item_id
       method customer_name
       = Parameters.customer_name
       ...
end
```

Figure 3. OCaml code generated for a shopping cart application. The secure IDL compiler generates a rule structure, code to apply the rules, and a skeleton class from which to derive the code.

tects against attacks such as the "price-changing attack" described previously.

The policy compiler translates validation and transformation rules into code to perform server-side checking and manipulation. If a malicious client violates any of the validation constraints at runtime, Spectre returns a descriptive error page to the client. In addition to generating code for server-side checks, the policy compiler emits JavaScript for client-side validation, which the security gateway dynamically inserts into HTML forms. In this way, validation checks are performed on both the client side (to improve observed latency between form submission and receiving validation errors) and the server side (for security). The key benefit here is that both client- and server-side code is derived from the same specification. Note that we insert JavaScript into forms dynamically, rather than inserting it statically into files in the Web repository, because many applications use server-side code to generate forms on-the-fly.

The Spectre tool is useful in several circumstances. It provides developers with a structured methodology for fixing security vulnerabilities in a deployed Web site, even if the vulnerability lies in a component for which the source is not available (such as a third-party component shipped in binary form). In addition, Spectre allows security-critical code to be removed from a Web application and stored in a centralized repository. Thus, even if the application is written in a variety of noninteroperating languages, we can still abstract much of the security policy.

Its stateless design means that Spectre scales well — if the performance of the security gateway is insufficient for a very popular Web site, a developer can easily add an identical security gateway and load-balance across them. There is, however, a performance cost associated with Spectre: applying the security policy dynamically as pages are fetched decreases the Web server's perceived throughput. Although this performance hit is acceptable for many real-world applications,[2] for very heavily loaded sites it might be better to encode the security policy directly into the application, rather than applying it dynamically at each page request.

Secure IDL Compiler

IDLs are traditionally used in middleware systems (such as COM and Corba), where they allow the definition of type signatures for remote procedure calls (RPCs). An IDL compiler translates IDL specifications into concrete code to marshal and unmarshal procedure arguments. IDL compilers typically target a number of different languages, allowing developers to use a common IDL across a multilanguage project.

Our policy language, SPDL, is effectively an IDL with a specific emphasis on security constraints. We have created a secure IDL compiler capable of transforming SPDL into concrete skeleton code that is then compiled and linked into a Web application. Although binding to almost any language is possible, we focus on the bindings for OCaml, which allow SPDL policies to be incorporated into OCaml-based applications. While OCaml might seem an unusual choice, we believe it offers numerous features conducive to writing security-critical code. In particular, its strong static type system lets developers catch a class of common coding errors at compile time. (In more weakly typed languages, these errors can manifest themselves as security holes.)

Consider a shopping cart application, in which one URL accepts two parameters: `item_id` (representing the item to be purchased) and `customer_name`. Assume the designers have already written SPDL to constrain field lengths, enforce simple type constraints, and make both `item_id` and `customer_name` required fields.

For each parameter of each page covered by the security policy, our secure IDL compiler generates:

- a structure describing the validation and transformation rules to be applied;
- code to apply the rules, halting with an error page if the validation fails; and

Types of Attacks

Application-level attacks fall into several categories. The vulnerabilities highlighted here are not an exhaustive list, but they demonstrate the need for application-level security tools such as ours.

Form Modification

Form-modification attacks are possible when Web designers implicitly trust assertions checked only on the client side. Examples of such actions include constraints imposed by the HTML itself (such as the MaxLength attribute) and by client-side scripts (usually JavaScript programs). Of course, users can easily modify any client-side validation rules, so you should never trust data received from clients.

Although writing server-side code to handle form input securely might not be cerebrally taxing, it is nevertheless a tedious, time-consuming, and error-prone task that is rarely undertaken correctly (if at all) in practice.

SQL Attacks

Web applications commonly use data read from a client to construct SQL queries. Unfortunately, constructing the query naïvely lets the user execute arbitrary SQL against the back-end database. The attack is best illustrated with a simple example.

Consider an employee directory Web site written in PHP: Hypertext Preprocessor (PHP) that prompts a user to enter an employee's surname in the form-box searchName. The server side code uses this search string (stored in the variable $searchName) to build an SQL query to find the employee's contact information. The query might involve code such as

```
$query = "SELECT
forenames,tel,fax,email
     FROM personal
WHERE surname='$searchName';
";
```

However, if the user enters the following text into the searchName form box

```
'; SELECT
password,tel,fax,email FROM
personal
WHERE surname='Scott
```

the value of `$query` will become

```
SELECT forenames,tel,fax,email
FROM personal
WHERE surname='';
SELECT password,tel,fax,email
FROM personal
WHERE surname='Scott';
```

When executed on some SQL databases, this code will result in Scott's password being returned along with his contact information. (Even if only a hash of the password is leaked, a forward-search attack against a standard dictionary stands a reasonable chance of recovering the actual password.)

Cross-Site Scripting

Cross-site scripting (XSS) refers to a range of attacks in which users submit malicious HTML (possibly including JavaScript code or other scripts) to dynamic Web applications. For other users, the malicious content appears to come from the dynamic Web site itself, a trusted source. The implications of XSS are severe — for example, it subverts the Same Origin Policy, a key part of JavaScript's security model. A CERT advisory, listed in the "Internet Resources" sidebar, next page, outlines a range of serious attacks that come under the general heading of XSS.

The fix for XSS vulnerabilities is well known: You can encode HTML metacharacters explicitly using HTML's #&<n> syntax, where <n> is the numerical representation of the encoded character. (Metacharacters are characters with special meaning in HTML, such as < and >, which are used to delimit tags.) However, the flexibility of HTML makes this a more complicated task than many people realize. Furthermore, for large applications, ensuring that all user input has been appropriately HTML-encoded is a laborious and error-prone task.

- a skeleton class from which user code is derived.

The fragment in Figure 3 shows OCaml code generated for this simple e-commerce example.

The code initially calls our internal library function `get_args` with an SPDL-derived security policy specification. The `get_args` function fetches the HTTP parameters using the CGI protocol and checks that they match the given specification. If the parameters do not match the specification, the `get_args` function returns a descriptive error to the client. The parameter values (returned from the call to `get_args`) are then bound as typed OCaml identifiers within the internal `Parameters` module. Finally a class, `skeleton`, is generated with an accessor method corresponding to each URL parameter. By subclassing this skeleton class, a developer is assured that the parameters submitted by the user have been properly sanitized.

If a Web application's source code is available, the secure IDL compiler provides all the benefits of abstracting the security policy without Spectre's performance cost. Even if a Web application is developed entirely in a single language, there are still advantages to abstracting the security policy in SPDL and using the secure IDL compiler to automatically generate skeleton classes. In particular, a single SPDL specification reduces the amount of security-related code written by each developer — and hence the potential for mistakes.

Security-Aware API

Many frameworks for Web development — Active Server Pages, Java servlets, PHP, and so on — abstract the low-level details of HTTP away from the

Internet Resources

- AppShield 4.0 white paper, Sanctum • www.sanctuminc.com/pdf/AppShield_40_WhitePaperFINAL.pdf
- "Best Practices for Secure Web Development," Razvan Peteanu • www.cert.pl/PDF/secure_webdev-3.0.pdf
- The <bigwig> project • www.brics.dk/bigwig/
- CERT Advisory on Cross-Site Scripting (XSS) • www.cert.org/advisories/CA-2000-02.html
- "Form Tampering Vulnerabilities in Several Web-Based Shopping Cart Applications," Internet Security Systems (ISS) security alert • http://xforce.iss.net/alerts/advise42.php
- "HowTo: Review ASP Code for CSSI Vulnerability," Microsoft Knowledge Base Article Q253119 • http://support.microsoft.com/support/kb/articles/Q253/1/19.asp
- Simple object access protocol (SOAP) 1.1 • www.w3.org/TR/SOAP

designers. However, when it comes to application-level security, existing APIs provide virtually no support whatsoever. Every time developers implement a new Web application, they have to manually code against well-known attacks such as those listed in the "Types of Attacks" sidebar (p. 43). Inevitably, many Web sites suffer from application-level security vulnerabilities as a result.

To address this issue, we designed an object-oriented API for Web development specifically with security in mind. Our security-aware API consists of a library of parameter validation and transformation routines to provide the capabilities of SPDL in an API; an access control model using mixin classes[5] to ensure that security-critical checks are always executed; and a database abstraction layer to safely interface with persistent storage.

Access control. A common application requirement is to associate different privileges with different groups of users (where a privilege can be thought of as an "ability to see this specific set of pages"). For example, an e-commerce application might let anyone browse a product catalog as a "guest," but would expect purchasers to be properly authenticated.

Our strategy for dealing with page-level access control is to associate groups of users with mixin classes.[5] Each mixin class contains a constructor that verifies users' membership in the appropriate privilege group by looking them up in an external database. To ensure that the access control policy is enforced, all the developer has to do is inherit each page class from the corresponding access control mixin. When the developer combines the mixin classes with the SPDL-derived skeleton classes, both the access control and the user-input validation policies are met.

In our system, a piece of application code would be structured as

```
class shopping_cart =
      object(self)
      inherit normal_user
      inherit Shoppingcart_spdl.skeleton
      method body =
      let response = set_body blank_response
         ("Actual content goes here")
         in output_response response
end
```

In this example, the `shopping_cart` class (which is associated with a single application URL) inherits from both the SPDL-derived parameter-checking skeleton and the `normal_user` class. The `normal_user` parent class contains constructor code that verifies that the request originates from an authenticated user in the group "`normal_user`."

Database abstraction. Persistent storage is essential for most Web applications. The industry-standard way of talking to an external database is through SQL. Unfortunately, naïve use of SQL leads to a number of problems, including

- *Portability concerns:* Not all databases support the same SQL feature set.
- *Security concerns:* Failing to properly SQL-escape a piece of text received from a user can lead to data leakage and corruption (see the discussion on SQL attacks in the "Types of Attacks" sidebar, p. 43).

To alleviate these problems, we created a simple database abstraction layer in OCaml that supports a subset of SQL (which has a more regular syntax) and automatically applies the appropriate transformation procedures to prevent SQL attacks.

The following fragment shows a small part of our SQL abstraction layer:

```
(* SELECT to read data from the database *)
type selectrec = {fields : string list;
      from : table list;
      where : expression}
and sql = Select of selectrec
      | Update of ...
and expression =
      Field of string
      | Value of sql_type
      | Equal of expression * expression
and transaction = sql list
```

OCaml's support for first-class data types and higher-order functions allows developers to build up complicated SQL expressions concisely. For

example, you can write a simple SELECT query as

```
Select {fields=["surname";"id"];
     from=["personal"];
     where=Equal(Field "name", Value "sharp")}
```

We defined a number of utility functions, including the function `all: expression list → expression`, which returns the expression corresponding to the conjunction of all the argument expressions. This allows us to code in the readable style of the example:

```
let this_customer    = Equal(Field "name",
                     Value "sharp")
and ordered_today = Equal(Field "day",
                     Value "wednesday")
and this_product = Equal(Field "id",
                     Field "productid")

in Select {fields=["id"];
     from=["customers";"orders"];
     where = all  [this_customer;
            ordered_today;
            this_product]}
```

The key is that whereas most database abstraction layers force designers to construct SQL fragments at the string level, we use a structured data type to manipulate SQL at the abstract syntax level. It is this extra structure that allows us to automatically apply the transformations (such as quote-escaping) that prevent SQL attacks. If we operated on SQL fragments at the string level, we would have no way of knowing which parts of the string to transform.

Using these techniques, we built a multiuser content management system (CMS) that supports role-based access control. A large organization bought the CMS and is currently undergoing user acceptance testing. Looking back at the development process, we believe we saved a lot of time by first building the application framework described here. For example, the API design forced us to specify validation and transformation rules for form, cookie, and URL parameters that we might otherwise have forgotten. Furthermore, the structured SQL support prevented us from having to remember to manually transform user input before incorporating it in SQL strings.

Down the Road

There are several interesting directions for future work. In particular, we intend to increase the expressiveness of the security policies that can be written in SPDL. An important class of attacks (which we call control-flow tampering attacks) arise because developers mistakenly assume that users will only click on URLs explicitly presented to them, when of course users are free to modify and create URLs at will. To protect against this, we are currently implementing a tool that abstracts control-flow information from large Web applications.

We have not yet fully addressed the issue of policy design and maintenance. As a Web site grows and expands, so too will its policy description, which could lead to maintenance difficulties. Better support for more structured, modular policies and tools to assist in debugging and long-term maintenance is a topic of future work.

Acknowledgments

This work was supported by (UK) Engineering and Physical Sciences Research Council grant number 64256 and the Schiff Foundation. We would also like to thank Andy Hopper and Alan Mycroft for their valuable support and feedback.

References

1. J. Pescatore, "Web Services: Application-Level Firewalls Required," report no. SPA-15-5542, Gartner, Stamford, Conn., 7 Mar. 2002; available at www4.gartner.com/DisplayDocument?id=353429.
2. D. Scott and R. Sharp, "Abstracting Application-Level Web Security," *Proc. 11th Int'l World Wide Web Conf.*, ACM Press, New York, May 2002, pp. 396-407.
3. X. Leroy, *The Objective Caml System Release 3.0*, INRIA, Rocquencourt, France, 2000.
4. M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," *Lecture Notes in Computer Science,* Springer-Verlag, Santa Barbara, Calif., vol. 1109, 1996, pp. 1-15.
5. G. Bracha and W. Cook, "Mixin-Based Inheritance," *Proc. Conf. Object-Oriented Programming: Systems, Languages, and Applications*, N. Meyrowitz, ed., ACM Press, New York, 1990, pp. 303-311.

David Scott worked at AT&T Laboratories in Cambridge, U.K., where he helped develop the IDL compiler for omniORB, AT&T's open-source Corba implementation. He then moved to the Laboratory for Communications Engineering in the University of Cambridge to continue his research. His research interests now include enhancing the security and reliability of component-based applications through interface analysis and augmentation.

Richard Sharp is a researcher in the Programming Languages Group at the University of Cambridge Computer Laboratory. His main interest involves developing high-level tools to assist in the construction of complex systems. This principle is embodied both in his work in the field of Web development and his research into hardware synthesis.

Readers can contact the authors at {djs55,rws26}@cam.ac.uk.