Specifying and Enforcing Application-Level Web Security Policies

David Scott and Richard Sharp

Abstract—Application-level Web security refers to vulnerabilities inherent in the code of a Web-application itself (irrespective of the technologies in which it is implemented or the security of the Web-server/back-end database on which it is built). In the last few months, application-level vulnerabilities have been exploited with serious consequences: Hackers have tricked e-commerce sites into shipping goods for no charge, usernames and passwords have been harvested, and confidential information (such as addresses and credit-card numbers) has been leaked. In this paper, we investigate new tools and techniques which address the problem of application-level Web security. We 1) describe a scalable structuring mechanism facilitating the abstraction of security policies from large Web-applications developed in heterogeneous multiplatform environments; 2) present a set of tools which assist programmers in developing secure applications which are resilient to a wide range of common attacks; and 3) report results and experience arising from our implementation of these techniques.

Index Terms—Application-level Web security, security policy description languages, component-based design.

1 Introduction

On 25 January 2001, an article appeared in a respected British newspaper entitled *Security Hole Threatens British E-tailers* [1]. The article described how a journalist hacked a number of e-commerce sites, successfully buying goods for less than their intended prices. The attacks resulted in a number of purchases being made for 10 pence each, including an Internet domain name (ivehadyou.org.uk), a "Wales Direct" calendar, and tickets for a Jimmy Nail pop concert.[1] The author of the article rightly observes that the process "requires no particular technical skill"; the attack merely involves saving the HTML form to disk, modifying the price (stored in a hidden form field) using a text editor, and reloading the HTML form back into the browser. A recent survey article published in ZD-Net [2] suggests that between 30 and 40 percent of e-commerce sites throughout the world are vulnerable to this simple attack. Internet Security Systems (ISS) identified 11 widely deployed commercial shopping-cart applications which suffer from the vulnerability [3].

The price-changing attack is a consequence of an *application-level* security hole. We use the term *application-level Web security* to refer to vulnerabilities inherent in the code of a Web-application itself (irrespective of the technology in which it is implemented or the security of the Web-server/back-end database/operating-system on which it is built). Most application-level security holes arise because Web applications mistakenly trust data returned from a client. For example, in the price-changing attack above, the Web application makes the invalid assumption that a user cannot modify the price because it is stored in a *hidden* field.

Application-level security vulnerabilities are well-known and many articles have been published advising developers on how they can be avoided [4], [5], [6]. 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 [7], the fact remains that a large number of professionally designed Web sites still suffer from serious application-level security holes. This evidence suggests that higher-level tools and techniques are required to address the problem.

In this paper, we present a structuring technique which helps designers abstract security policies from large Web applications. Our system consists of a specialized Security-Policy Description Language (SPDL-2), which is used to program an application-level firewall (referred to as a *security gateway*). Security policies are written in SPDL-2 and compiled for execution on the security gateway. The security gateway dynamically analyzes and transforms HTTP requests/responses to enforce the specified policy. We call our Security-Policy Description Language SPDL-2 because it extends the language SPDL, which we defined in previous work [8].

The remainder of the paper is structured as follows: Section 2 briefly surveys a number of application-level attacks and discusses some of the reasons why application-level vulnerabilities are so prevalent in practice. In Section 3, we describe the technical details of our system for abstracting application-level Web security. Our methodology is illustrated with an extended example in Section 4. We have implemented the techniques discussed in this

1. Some readers may argue that 10p is the true value of tickets to such a concert. A full discussion of this topic is outside the scope of this paper.

- *D. Scott is with the Laboratory for Communication Engineering, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK. E-mail: djs55@eng.cam.ac.uk.*
- *R. Sharp is with Intel Research, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK. E-mail: richard.sharp@intel.com.*

Manuscript received 15 July 2002; revised 15 Dec. 2002; accepted 6 Jan. 2003. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 118230.

paper. The performance of our implementation is evaluated in Section 5. Related work is discussed in Section 6; finally, Section 7 concludes.

2 Application-Level Security

We start by briefly categorizing and surveying a number of common application-level attacks. We make no claims regarding the completeness of this survey; the vulnerabilities highlighted here are a selection of those which we feel are particularly important. A more detailed survey may be found in previous work [8].

2.1 Form Modification

HTML forms are a common source of application-level security problems. The main contributing factor is that Web designers implicitly trust validation rules which are enforced only on the client-side. Even a relatively unskilled attacker is able to save a form to disk and then change or remove the validation constraints. Clearly, client-side validation must never be trusted and all user input must always be revalidated on the server-side.

2.2 SQL Attacks

Web applications commonly use data read from a client to construct SQL queries. Unfortunately, constructing the query naively leads to a vulnerability where the user can execute arbitrary SQL against the back-end database. The vulnerability is present because certain input characters have special meaning in SQL (for example, ' and ;). To prevent these attacks from happening, all user input must be properly escaped before being sent to a back-end database.

2.3 Cross-Site Scripting

Cross-Site Scripting (XSS) refers to a range of attacks in which users submit malicious HTML (possibly including scripts—e.g., JavaScript) to dynamic Web applications. The malicious HTML may be embedded inside URL parameters, form fields, or cookies. When other users view the malicious content, it appears to come from the dynamic Web site itself, a trusted source. The implications of XSS are severe; for example, the *Same Origin Policy*, a key part of JavaScript's security model [9], is subverted.

2.4 Motivation and Contributions

In this section, we discuss a number of factors which contribute to the prevalence of application-level security vulnerabilities. We believe that each of the problems listed below points to the same solution: The security policy should be applied at a higher-level, removing security-related responsibilities from coders whenever possible.

A major cause of application-level security vulnerabilities is a general lack of language-level support in popular untyped scripting languages. For example, consider the languages PHP [10] and VB-Script [11]. When using these languages, it is the job of the programmer to manually verify that all user input is appropriately HTML-encoded. Inadvertently omitting a call to the HTML-encoding function results in a vulnerability being introduced. For large applications written in such languages, it is inevitable that a few such vulnerabilities will creep in. (Note that some technologies provide greater language-level support in this respect: When using typed languages, such as Java, the type-system can be employed to statically verify that all user input has been passed through an HTML-encoding function; Perl's *taint mode* offers similar guarantees but through runtime checks rather than compile-time analysis.)

If Web applications were written in a single programming language by a small number of developers, then one could separate the security policy from the main body of code by abstracting security-related library functions behind a clean API. However, in reality, large Web applications often consist of a large number of interacting components written in different programming languages by separate teams of developers. To complicate the situation further, some of these components may be bought from third-party developers (possibly in binary form). In such an environment, it is difficult to abstract common code-blocks into libraries. The inevitable consequence is that security-critical code is scattered throughout the application in an unstructured way. This lack of structure makes fixing vulnerabilities difficult: The same security hole may have to be fixed several times throughout the application.

Another major issue, albeit a nontechnical one, is a lack of concern for security in the Web development community. Although we realize that this is a generalization, evidence suggests that factors such as time-to-market, graphic design, and usability are generally considered higher priority than application-level security. We recently talked with some Web-developers working for a large telecommunications company;[2] they were surprised to hear of the attacks outlined in Section 2 and had taken no steps to protect against them.

In this paper, we present tools and techniques which protect Web sites from application-level attacks. While we recognize that our proposed methodology is not a panacea, we claim that it does help to protect against a wide-range of common vulnerabilities.

3 Technical Details

Our system consists of a number of components:

1. A *security policy description language* (SPDL-2) is used to specify a set of validation constraints and transformation rules.
2. A *security policy creation tool* assists the policy developer with the task of creating the security policy.
3. A *policy compiler* automatically translates the SPDL-2 into server-side (and, optionally, client-side) code for dynamically enforcing the policy.
4. An application-level *security gateway* is positioned between the Web-server and client machines and filters all the HTTP messages passing between them.

Fig. 1 shows a diagrammatic view of the components of our system and the interactions between them. Note that the security gateway does not have to run on a dedicated machine: It can run as a separate process on the existing

2. We hasten to add that this was not our sponsors, AT&T!

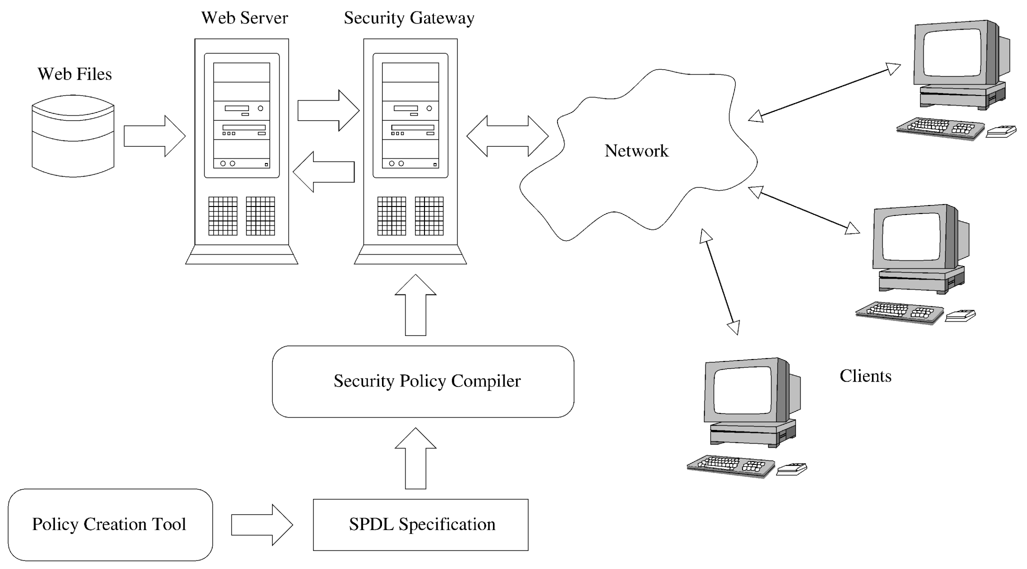

Fig. 1. A diagrammatic view of our system for abstracting application-level Web security.

Web-server or, to achieve better performance, integrated into the Web-server directly.

3.1 System Overview

With assistance from the *security policy creation tool*, the designer codes a set of *validation constraints* and *transformation rules* in SPDL-2. Validation constraints place restrictions on data in cookies, URL parameters, and forms. For example, typical constraints include "the value of this cookie must be an integer between 1 and 3" and "the value of this (hidden) form field must never be modified." The transformation rules of an SPDL-2 specification allow a programmer to specify various transformations on user-input. The kind of transformations which may be specified are "pass data from all fields on form f through an HTML-encoding function" or "escape all single and double quotes in text submitted via this URL parameter." A summary of SPDL-2 is given in Section 3.2.

The *policy compiler* translates SPDL-2 into code which checks validation rules and applies the specified transformations. The generated code is dynamically loaded into the *security gateway* where it is executed in order to enforce the specified policy. The security gateway acts as an *application-level* firewall; its job is to intercept, analyze, and transform whole HTTP messages (see Section 3.5). As well as checking HTTP requests, the security gateway also rewrites the HTML in HTTP responses, annotating it with Message Authentication Codes (MACs) [12] to protect state which may have been maliciously modified by clients (see Section 3.6.2).

Although performing validation checks on the server-side is sufficient for security purposes, user-interface issues sometimes require validation rules to be applied on the client-side. For example, Web-forms often use JavaScript for client-side validation to reduce the observed latency between form submission and receiving validation errors. To address this need, the policy compiler offers the option of generating JavaScript directly from the validation rules of the SPDL-2 specification. The security gateway analyzes forms as they are sent to the client, automatically inserting JavaScript validation rules where appropriate. Since both client-side and server-side validation code is derived from a single specification, designers only have to write the security policy once. Even if the client-side JavaScript is subverted, there are still server-side checks in place.

Note that the reason we insert JavaScript into forms dynamically (rather than inserting it statically into files in the Web repository) is that many applications use server-side code to generate forms on-the-fly. Although there is scope for analysis of Web scripting languages to insert validation code statically, this is a topic for future work.

3.2 Security Policy Description Language Version 2

At the top level, an SPDL-2 specification is an XML document. The DTD corresponding to SPDL-2 is shown in Fig. 2. The document consists of a single `<site>` element which in turn contains a collection of `<policy>` elements. Each `<policy>` element contains a group of related `<URL>` and `<cookie>` elements and optionally `<parameter>` and further nested `<policy>` elements.

For each `<URL>` element, a number of `<parameter>`s are declared. The attributes of a `<parameter>` element with `name` $= p$ place constraints on data passed via p:

- The `maxlength` and `minlength` attributes specify the maximum and minimum length of data passed via p.
- A setting `required` to "`Y`" specifies that p must always contain a (nonzero length) value.
- Setting `MAC` to "`Y`" specifies that the value of p must be accompanied by a Message Authentication Code (MAC) [12] generated by the server. This prevents the user from changing the value of the parameter to arbitrary values (see Section 3.6.2).
- The `type` attribute specifies the data-type of p (either `int`, `float`, `bool`, or `string`).

The `method` attribute determines whether the specified constraints apply to p passed as a GET-parameter (i.e., a URL argument) or a POST-parameter (i.e., returned from a form). Setting `method` to `GETandPOST` means that the constraints within the `<parameter>`

```
<!ELEMENT site (policy*)>
  <!ATTLIST site name                 CDATA #REQUIRED>
  <!ATTLIST site policy_author_name   CDATA #REQUIRED>
  <!ATTLIST site policy_author_email  CDATA #REQUIRED>
  <!ATTLIST site description          CDATA "unspecified">
  <!ATTLIST site base_url             CDATA #REQUIRED>


  <!ELEMENT policy (parameter*, url*, cookie*, policy*)>
    <!ATTLIST policy name             CDATA #REQUIRED>
    <!ATTLIST policy description      CDATA "unspecified">
    <!ATTLIST policy javascript       (Y | N) "N">


    <!ELEMENT parameter (validation*, transformation*)>
      <!ATTLIST parameter method      (GET | POST | GETandPOST) "GETandPOST">
      <!ATTLIST parameter name        CDATA #REQUIRED>
      <!ATTLIST parameter maxlength   CDATA #REQUIRED>
      <!ATTLIST parameter minlength   CDATA   "0">
      <!ATTLIST parameter required    (Y | N) "N">
      <!ATTLIST parameter MAC         (Y | N) "Y">
      <!ATTLIST parameter type        (int | float | bool | string) #REQUIRED>


    <!ELEMENT cookie (validation*, transformation*)>
      <!ATTLIST cookie name           CDATA #REQUIRED>
      <!ATTLIST cookie maxlength      CDATA #REQUIRED>
      <!ATTLIST cookie minlength      CDATA   "0">
      <!ATTLIST cookie required       (Y | N) "N">
      <!ATTLIST cookie MAC            (Y | N) "Y">
      <!ATTLIST cookie type           (int | float | bool | string) #REQUIRED>


    <!ELEMENT url (parameter*)>
      <!ATTLIST url prefix            CDATA #REQUIRED>


      <!ELEMENT transformation        (#PCDATA)>
        <!ATTLIST transformation htmlencode (Y | N) "Y">

      <!ELEMENT validation            (#PCDATA)>
```

Fig. 2. The XML DTD for the Security Policy Description Language v2.

element are applicable to both GET and POST parameters with `name` $= p$. (The `GETandPOST` option is particularly useful if parts of a Web-application are written in a language which does not force a distinction between GET and POST parameters with the same name—e.g., PHP.)

For example, consider the following policy fragment:

```
<policy name="example" description="...">
   <URL prefix="http://example">
     <parameter name="p1" maxlength="4"
                type="int" required="Y"
                MAC="N" />
     <parameter name="p2" method="POST"
               maxlength="3" type="string" />
   </URL>
</policy>
```

This example specifies constraints on parameters passed to URLs with prefix "`http://example`." The first `<parameter>` element defines constraints to be applied to a parameter named `p1` (either GET or POST); the second `<parameter>` element defines constraints to be applied to a POST parameter named `p2`. A larger example of a policy definition can be found in the case study of Section 4. The case study explicitly demonstrates how policies can be nested within each other in order to abstract common parameters (e.g., Session IDs, etc.) from a group of related URLs.

We hope that the attributes of `<parameter>` elements cover the majority of validation constraints that designers require. However, in some circumstances, a greater degree of control is required: This is provided by the `<validation>` element. The `<validation>` element allows complex constraints to be encoded in a general purpose *validation language*. The content of the

$$
\begin{array}{lcll}
e & \leftarrow & x & \text{(variables)} \\
 & | & c & \text{(constants)} \\
 & | & f(e_1,\ldots,e_k) & \text{(function calls)} \\
 & | & \texttt{getparam.}c & \text{(value of GET parameters)} \\
 & | & \texttt{postparam.}c & \text{(value of POST parameters)} \\
 & | & \texttt{this} & \text{(value of this field)} \\
 & | & e_1 \ \langle \texttt{op} \rangle \ e_2 & \text{(binary infix operators)} \\
 & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{(conditionals)} \\
 & | & \texttt{let } d \ldots d \texttt{ in } e \texttt{ end} & \text{(local declarations)} \\
 & & & \\
d & \leftarrow & \texttt{val } x : t = e & \text{(immutable bindings)} \\
 & | & \texttt{fun } f(x_1 : t, \ldots, x_k : t) : t = e & \text{(function definitions)} \\
 & & & \\
t & \leftarrow & \texttt{int} \ | \ \texttt{float} \ | \ \texttt{string} \ | \ \texttt{bool} & \text{(types)}
\end{array}
$$

Fig. 3. The abstract syntax of the validation language.

`<validation>` element is a *validation expression* written in a simple, call-by-value, applicative language which is essentially a simply typed subset of Standard ML [13]. (Note that the precise details of the language are not the main focus of this paper. In principle, any language could be used to express validation constraints. For expository purposes, we choose to make the language as simple as possible.)

The abstract syntax of the validation language is shown in Fig. 3. A well-formed validation expression has type `bool`. If the validation expression of parameter, p, evaluates to *true*, then this signifies that p contains valid data; conversely, evaluating to *false* highlights a validation failure. Badly typed validation programs are rejected by a compile-time type-checking phase (see Section 3.4). Within validation expressions, the value of the field specified in the enclosing parameter element is referred to as `this`. Values of other (declared) GET and POST parameters can be referenced as `getparam.name` and `postparam.name`, respectively. In this way, validation rules can be dependent on the values of multiple parameters.

A number of primitive-defined functions and binary operators are provided. Although we do not list them all here, those of particular importance are outlined below:

- Arithmetic operators `+`, `-`, `*`, and `/` can be applied to both integers and floating point values. String concatenation is represented by the infix operator `++`.
- The function `format(s,regexp)` returns true iff `s` matches the form specified by regular expression, `regexp`.
- We provide the function `mid(s,l,r)` which returns the substring of `s` which starts at character `l` and finishes at character `r` inclusively. (Characters of `s` are numbered from 1).
- Functions are provided to convert between different types. For example, `String.fromInt(i)` returns the string representation of integer `i`.
- Function `isdefined(p)` takes a parameter (e.g., `getparam.p` or `postparam.p`) and returns a Boolean indicating whether `p` is defined (i.e., has been passed to the URL in the HTTP request). Using an undefined parameter as an argument to any other function or operator leads to a dynamically generated error message.

Transformation rules for parameters and cookies are written in the same language as validation expressions, but are often much simpler. The system provides a set of common transformations in a library which we hope cover most common cases. For example, if we wanted to remove all spaces from a parameter `p` and then SQL-escape the result, our specification would look like:

```
<transformation>
 let fun filter(s: string,char: string):
                string =
    let val first:string = String.mid(s, 1, 1)
        val rest:string = String.mid(s, 2,
                String.length(s) - 1)
    in if (fst = char) then filter(rest, char)
                      else first ++
                        (filter(rest, char))
    end
 in SQLEncode( filter( this, " " ) )
 end
</transformation>
```

Our current transformation library contains the following functions:

- *EscapeSingleQuotes*: Replace single quotes with their HTML character encoding.
- *EscapeDoubleQuotes*: Replace double quotes with their HTML character encoding.
- *HTMLEncode*: HTML-encode the data. Replace metacharacters with their numerical representations.
- *PartialHTMLEncode*: HTML-encode the input, but leave a small number of allowed tags untouched (including style tags, `<b>`, `<u>`, and `<i>`, and anchors of the form `<a href="...">  ...  </a>`).
- *SQLEncode*: SQL metacharacters such as ';' are escaped.

We consider the HTML-encoding transformation to be of particular importance since inadvertently forgetting to HTML-encode user-input leads to Cross-Site Scripting vulnerabilities (see Section 2). For this reason, we adopt the convention that *all* parameters are HTML-encoded unless explicitly specified otherwise in the security policy. To turn off HTML-encoding, one must set the `htmlencode` attribute of the `transformation` element to `N`. For example, one may write:

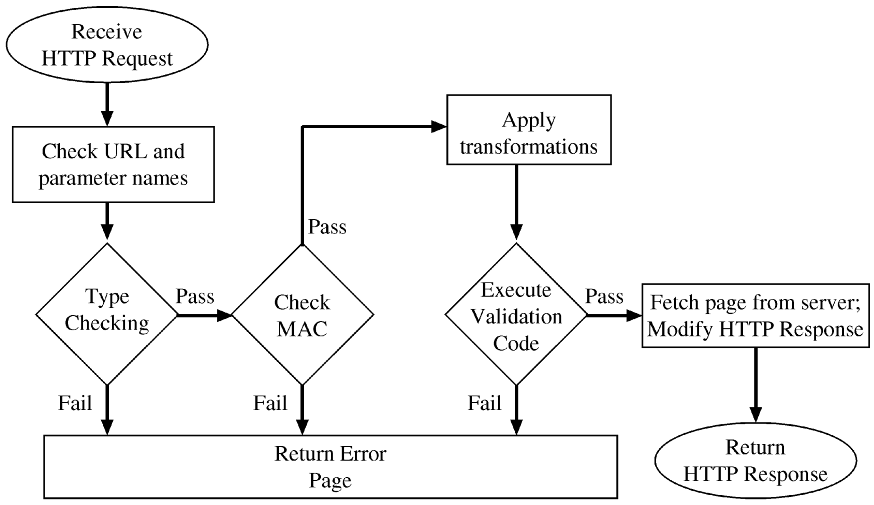

Fig. 4. The tasks performed by the security gateway.

`<transformation htmlencode="N" />`

Recall from Fig. 2 that policies within an SPDL-2 document consist of a series of `<URL>` and `<cookie>` elements. We have already discussed `<URL>` elements in detail; in a similar fashion, `<cookie>` elements allow designers to place validation constraints on cookies returned from clients' machines.

3.3 Differences from SPDL Version 1

A number of enhancements have been made to the original Security Policy Description Language (SPDL Version 1) described in our previous work [8]. In particular, SPDL-2

- allows policies to be defined hierarchically, factoring out common elements and leading to more readable specifications;
- allows fine-grained control over expensive HTML-modification operations, specifically the use of JavaScript and MAC insertion; and
- uses the same language for both validation and transformation expressions for consistency.

3.4 Policy Compiler

The *policy compiler* takes an SPDL-2 specification (as described in Section 3.2) and compiles it for execution on the security gateway. Validation rules and constraints are also compiled into JavaScript which can be embedded into forms and executed on clients. We recognize that generating client-side form validation JavaScript code is not always desirable—many applications already have their own, custom JavaScript for this purpose—and, therefore, we allow JavaScript generation to be turned off on a `<policy>`-wide basis (by setting the `javascript` attribute of the `<policy>` tag to "`N`").

Validation and transformation expressions are type-checked at compile-time, helping to eliminate errors from SPDL-2 code. In the current incarnation of the system, expressions are *simply typed* (that is, we do not allow parametric polymorphism). However, should experience show this to be too inflexible, there is no reason why more sophisticated type-systems (e.g., ML style polymorphism [14]) could not be employed in future versions.

3.5 The Security Gateway

Fig. 4 shows the algorithm executed by the security gateway on receipt of an HTTP request. Note the following points:

- The security gateway is intended to be *failsafe*—if a request does not properly match anything in the policy, then it will be rejected.
- Transformations are *total* functions on strings—well-written transformation code should not generate exceptions. However, if a badly written transformation function does generate a runtime exception, then the process is aborted and an error message is returned to the client.
- If the SPDL-2 document requires the insertion of client-side JavaScript or if the policy indicates that it might contain a link to a MAC-protected URL, then the security gateway must process the HTTP response returned from the Web-server. The HTML is parsed and rewritten in order for the appropriate validation code (pregenerated by the SPDL-2 compiler) to be added to forms (see Section 3.6.1). Message authentication codes are also inserted to protect form fields and URL-parameters from malicious client-side tampering (see Section 3.6.2).

3.6 Security Policy Creation Tool

The aim of the *security policy creation tool* (SPCT) is to simplify the task of generating SPDL-2 for a large Web site. A complete policy must associate *every* application URL with a policy, even if most URLs have no complicated security requirements. Writing out all the trivial cases by hand is very tedious.[3] The policy creation tool is able to automatically generate all the trivial cases, leaving the more complicated part of the specification to the human policy designer.

The tool operates in a similar way to the security gateway (Section 4) except that, rather than *enforcing* policy, it *creates* it. The tool sits between the original application and the clients, monitoring HTTP messages sent between them. The system maintains a database of URLs observed,

3. Especially if written out in XML!

together with information on their associated parameters and cookies. The algorithm used by the tool is as follows:

For each HTTP message observed, let u be the target URL and $p_1 \ldots p_n$ be the associated parameters.[4]

Is u already in the database?

- **No**: For each parameter p_i
  - assign a specialized type (`bool`, `int`, `string`...)
  - set the p_i's `required` attribute to "`Y`"
  - create a new database record for u
- **Yes**: For each parameter p_i
  - Is p_i in the database record for u?
    - **No**:
      - assign a specialized type (`bool`, `int`, `string`...),
      - set the p_i's `required` attribute to "`N`",
      - add p_i to the database record for u.
    - **Yes**:
      - Generalize the type associated with p_i to be consistent with all values seen so far (e.g., if previous values were `ints` and this value is a `bool`, then generalize to a `string`.)

Once the system has observed a sufficiently large and representative amount of traffic, a useful SPDL-2 document can be created. Rather than naively generating a single `<policy>` element with a flat list of URLs, the policy creation tool first applies simple heuristics to group probably related URLs together. The heuristics include:

- URLs with a parameter in common (e.g., `SessionID`) are grouped together if possible to avoid having to declare the shared parameter multiple times.
- URLs with prefixes in common (e.g., `/Component/ShoppingCart/`) are automatically grouped together into a single `<policy>` element. This reflects the fact that URLs with common prefixes are often related; keeping them together enhances the readability of the generated policy.

The resulting skeleton SPDL-2 document is then written out to disk and may be edited by hand. Note that the SPCT should be run under secure test conditions as the tool assumes all observed requests are valid and creates a policy to permanently allow them all. If the SPCT observes an application-level attack, it will assume the attack is a legitimate use of the Web site and extend the policy appropriately.

3.6.1 Client-Side Form Validation

Whenever requested by the policy document, the security gateway inserts JavaScript code to perform client-side validation checks. (Recall that the insertion of JavaScript is merely to enhance usability—the generated JavaScript is not considered a substitute for server-side validation checking). The inserted code checks most of the SPDL-2 constraints: types, lengths, and all custom constraints written in the validation language. The resulting program is inserted into the `onSubmit` attribute of a `<form>` tag unless such an attribute is already present—we take the view that, if a form already has an `onSubmit` handler, then this takes priority over our generated code.

3.6.2 Message Authentication Codes

We have already seen that an SPDL-2 specification can declare that certain URL parameters must only contain data accompanied by a *Message Authentication Code* (MAC) [12] generated by the security gateway. As data is sent to the client, the security gateway annotates it with MACs; as data is returned from clients, the MACs are checked. In this way, we prevent users from modifying data which should not be changed on the client-side (e.g., security-critical hidden form-fields).

Our implementation uses the HMAC [15] algorithm to generate MACs by securely combining the protected data with a secret key and a timestamp. In this way, an attacker is unable to generate new MACs without first finding out the secret key. One of our major concerns is to avoid *replay attacks* [16] where clients replay messages already annotated with MACs in unexpected contexts. We take two steps to avoid such attacks:

1. We include a time-stamp in the MAC and do not accept MACs which are more than a few minutes old.
2. Rather than generating separate MACs for each individual protected field, we generate a single MAC for all protected client-side state. This protects against cut-and-splice attacks (in which MAC-annotated fields are swapped into other messages).

Despite these preventative measures, the responsibility for ensuring that replay attacks are not damaging ultimately rests with the security policy designer. For example, in the case study of Section 4, a MAC is generated for *both* the `productID` *and* `Price` fields. Although users can replay such messages, this results in multiple purchases of the same product for the *correct* price. The intention is that the MAC prevents the `Price` and `productID` being modified independently.

SPDL-2 requires that HTML pages fetched from URLs that are contained in a single `<policy>` block may only contain links to MAC-protected URLs[5] which are found in the same `<policy>` block (or in a nested `<policy>` block). By forcing the application's URLs to be partitioned in this way, we facilitate an important performance optimization which is discussed in Section 5.

4 Case Study

To illustrate our methodology, we consider using our system to secure a simple e-Commerce system. The hypothetical Web site is first partitioned into groups of URLs (see Fig. 5) corresponding to:

4. For simplicity and without loss of generality, consider cookies as additional URL parameters.

5. A MAC-protected URL has at least one `<parameter>` with its `MAC` attribute set to "`Y`."

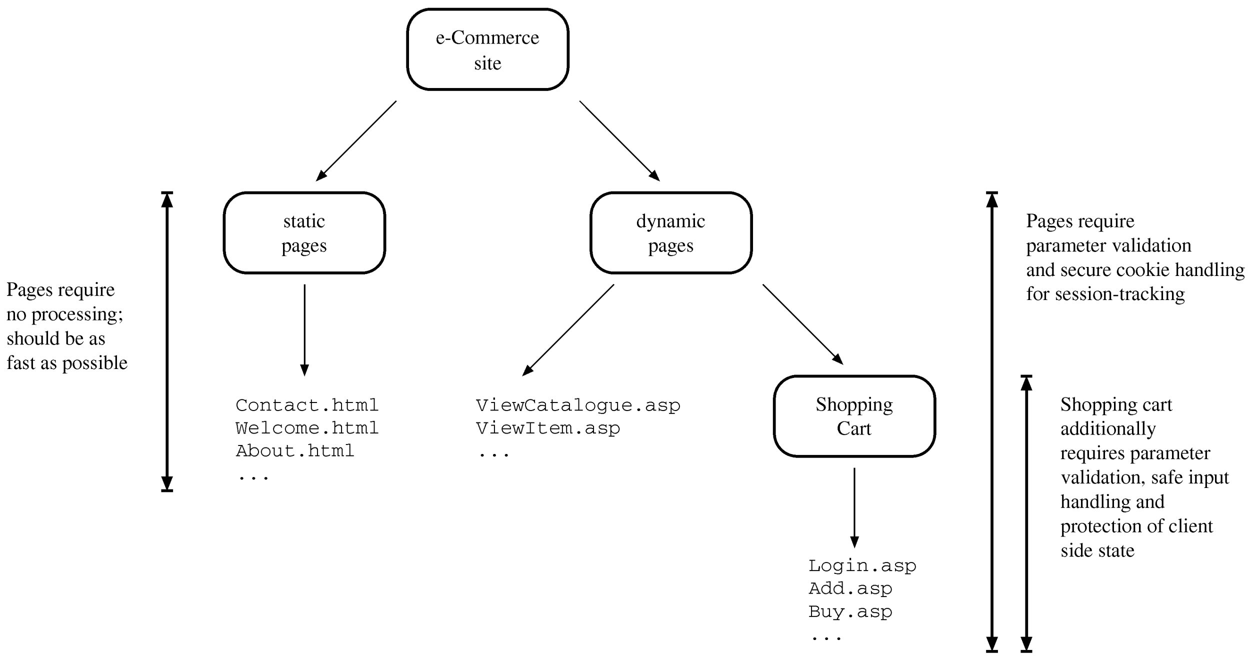
e-Commerce site
static pages
dynamic pages
Shopping Cart
Pages require no processing; should be as fast as possible
Contact.html
Welcome.html
About.html
...
ViewCatalogue.asp
ViewItem.asp
...
Login.asp
Add.asp
Buy.asp
...
Pages require parameter validation and secure cookie handling for session-tracking
Shopping cart additionally requires parameter validation, safe input handling and protection of client side state

Fig. 5. High-level structure of a simple e-Commerce Web site.

- *static content*, including "welcome" pages, an "about" page, and all static multimedia content (e.g., images, videos, music);
- custom *dynamic content* specific to this site, including code to browse the online product catalog; and
- an off-the-shelf *shopping cart* component capable of credit card transactions.

For the sake of a more interesting scenario, let us assume that:

- The custom dynamic content and the shopping cart component are both configured to set a cookie, `sessionKey`, to monitor user movement through the site for marketing research purposes; and
- The off-the-shelf shopping cart component is supplied in a binary-only form and, therefore, cannot be modified.

Let us further assume that our site is vulnerable in the following ways:

1. The custom code to view the online catalog has missing input validation code and can be used to execute arbitrary SQL against the backend database using the attack described in Section 2.
2. The `sessionKey` cookie is predictable since it is created using a time-seeded random number generator; clients can spoof other active sessions by modifying the value of the cookie in their browser.
3. JavaScript can be embedded in the `surname` field of the shopping cart login page which, when viewed on the company's intranet, leads to cross-site scripting vulnerabilities.
4. The "modifying values of hidden form-fields" attack (as described in the introduction) can be used to reduce the price of items in the shopping cart component.

4.1 Designing the Security Policy

We need to design an appropriate SPDL-2 description to protect the site.

Clearly, the static content can be described simply as a single `<policy>` block combining a straightforward list of URLs; no processing is required for these pages. The dynamic content, on the other hand, necessitates a more complex policy description. In the example application described above, the code is comprised of two separate components: the custom site-specific code and the shopping cart. Each component naturally maps onto its own `<policy>` block. Since the `sessionKey` cookie is common to both components (both the custom code and the off-the-shelf shopping cart), it is desirable to nest both the policy specification for custom code and the shopping cart within a parent `<policy>` block which declares this shared cookie. The resulting SPDL-2 document has the following high-level structure:

```
<site name="e-Commerce Web site" ...>
  <policy name="Static pages" javascript="N">
    <url prefix="About.html" />
    <url prefix="Contact.html" />
    ...
  </policy>
  <policy name="Dynamic content" ...>
    <cookie name="sessionKey" MAC="Y"
               maxlength="16" type="int"/>

    <policy name="our custom-written code">
      <url prefix="ShowCatalogue.asp" />
      ...
      </policy>
      <policy name="off-the-shelf shopping
                    cart component" ...>
```

```
            <url prefix="Login.asp"       />
            <url prefix="Buy.asp"         />
            ...
        </policy>
    </policy
</site>
```

All that remains is to fill in the individual <url> elements by writing the appropriate validation and transformation code. To see how this is done, consider the final step in the purchasing process in the shopping cart. Imagine that users are sent an HTML form requesting their surname, credit-card number, and its expiry date. The price and product-ID are stored in hidden form fields on the form. For example, when purchasing a product with productID=144264, the form sent to the client is as follows:

```
 <form method="POST" action="http:
              //www.example/Buy.asp">
  <input type="hidden" name="price"
         value="423.54">
  <input type="hidden" name="productID"
         value="144264">
  <input type="text" name="surname">
  <input type="text" name="CCnumber">
  <input type="text" name="expires">
 </form>
```

Once purchases have been made, an order record is entered into the company's back-end database which can be subsequently viewed on their local intranet.

The SPDL-2 fragment corresponding to the form's action URL is presented in Fig. 6. Each of the parameters shown in the form above are declared and a number of validation and transformation rules specified. Most of the SPDL-2 specification is self-explanatory, although a few points are worth noting. The <validation> element for the price field simply states that negative prices are not allowed; the more complicated validation expression for the CCnumber field is an implementation of the Luhn-formula commonly used as a simple validation check for credit-card numbers; the validation expression for the expires field ensures that it is of the form *mm/yy* and also checks that the month is in the range 1-12.

Through repeating this process for all <url> elements, we are able to fix all of the system's vulnerabilities (described above) without modifying any of the application code:

1. The form and cookie-manipulation attacks can no longer be used since the price and productID fields, along with the sessionKey cookie, can all be protected by having their MAC attributes set to "Y." This forces the security gateway to generate and check message authentication codes in order to prevent their values being tampered with.
2. The surname field can be HTML-encoded, preventing XSS attacks.
3. SQL attacks are prevented by applying the transformation, SQLEncode (see Section 3.2), to escape quotes in all relevant fields.

If specified in the SPDL-2 specification, the security gateway inserts JavaScript code (derived from the SPDL-2 specification of Fig. 6) to check validation rules on the client-side. In this example, JavaScript is generated to ensure that credit-card numbers satisfy the Luhn-formula, that expiry dates are of the form *mm/yy*, that the surname field contains a nonzero-length value, etc. Note that, if extra validation constraints are required, they can simply be added once to the SPDL-2 specification. Using conventional tools and techniques, the addition of extra validation constraints may require them to be coded multiple times (once in JavaScript for client-side validation and at least once in the Web application's source code).

5 System Performance

In this section, we discuss performance issues and present experimental results derived from our implementation of the security gateway.

Fig. 7 shows the worst-case latency of the security gateway and compares it to the latency of other common types of HTTP processing. The results were measured by fetching the homepage of the Laboratory for Communication Engineering (University of Cambridge)[6] augmented with the Web-form described in our case-study of Section 4. The leftmost bar shows the latency added by a Squid [17] proxy cache when fetching a statically compiled version of the page; the middle bar shows the added latency of dynamically generating the page using PHP and a MySQL [18] backend; the rightmost bar shows the latency of using the security gateway to enforce the security policy of Fig. 6. The final bar is divided into two sections: The (lower) solid black section represents the latency due to buffering the HTTP messages; the (upper) striped section shows the latency due to parsing the HTTP messages and annotating the HTML with MACs. Fig. 8 shows the relative cost of processing HTTP requests and HTTP responses in the case of our example Web-form. Note that the total processing time is dominated by the HTML parsing stage.

The latency of our system appears large compared with the latencies incurred in proxy caching and dynamic page generation. To some extent, this is due to the fact that our naive implementation is a prototype which is completely unoptimized, based upon an inefficient non-pipelined HTTP 1.0 library. However, we recognize that the complexity of the application-level tasks performed by the security gateway will necessarily incur more latency than the lower-level manipulation performed by proxies such as Squid. We regard our current implementation as a proof-of-concept. In future work, we intend to emphasize performance. Potential optimizations include 1) using a specialized HTML parser to concentrate only on relevant parts of HTML syntax (we currently use a general HTML parser which performs a great deal of unnecessary work); 2) reducing latency by streaming the HTTP messages and processing them on-the-fly whenever possible; and 3) writing speed critical parts of the security gateway directly in C.

Fig. 9 shows how the total throughput of a single security gateway varies as the number of concurrently connected clients increases. Each client is running a single threaded

6. http://www-lce.eng.cam.ac.uk/.

```
...
<policy ...>

  <url name="Buy.asp" description="Do CreditCard transaction" javascript="Y">

     <parameter name="price" method="POST" maxlength="10" minlength="1" required="Y"
                type="float" >
       <validation> this isGreaterThan 0.0 </validation>
     </parameter>

     <parameter name="productID" method="POST" maxlength="10" minlength="1"
                                 required="Y" type="int" />


     <parameter name="surname" method="POST" maxlength="30" minlength="2"
                               required="Y" MAC="N" type="string">
         <transformation>
            Transform.EscapeSingleQuotes(Transform.EscapeDoubleQuotes(this))
         </transformation>
     </parameter>

     <parameter name="CCnumber" method="POST" maxlength="16" minlength="16"
                               MAC="N" required="Y" type="int">
         <validation>
            let
                fun first(s:string):string = String.mid(s,1,1)
                fun rest(s:string):string  = String.mid(s,2,String.length(s)-1)

                fun double(s:string,a:bool):string =
                   if s="" then ""
                     else (if a then first(s)
                           else String.fromInt ( Int.fromString( first(s) ) * 2 ))
                          ++ (double (rest (s), not a))

                fun sum(s:string):int =
                   if s="" then 0
                           else (Int.fromString (first(s))) + (sum (rest(s)))

             in sum(double(this,false)) % 10 = 0
            end
         </validation>
     </parameter>

     <parameter name="expires" method="POST" maxlength="5" minlength="5"
                               MAC="N" required="Y" type="string">
         <validation> format(this,"\d\d/\d\d") and
                      Int.fromString( mid(s,1,2) ) <= 12 and
                      Int.fromString( mid(s,1,2) ) >= 0
         </validation>
     </parameter>
  </url>
</policy>
```

Fig. 6. SPDL-2 specification fragment for case study.

application continuously requesting the test URL. The machines are all connected on a fast 100Mbit/s switched network and the measurements were taken running the security gateway on a dual Intel P-III 500 MHz during an offpeak time when the network was idle. The throughput quickly reaches a maximum value as the CPUs become saturated. Note that it took three clients to saturate two CPUs, probably because of lack of support for persistent connections in our HTTP library. Again, we are confident that optimizing our code for performance and running the filter on a higher spec machine would yield a significantly higher maximum throughput.

We designed the security gateway in a *stateless* manner, choosing to annotate URL parameters, form fields, and cookies with MACs rather than storing session state in a back-end database. Since the security gateway is stateless, one may increase throughput linearly simply by deploying multiple security gateways and using a load balancing

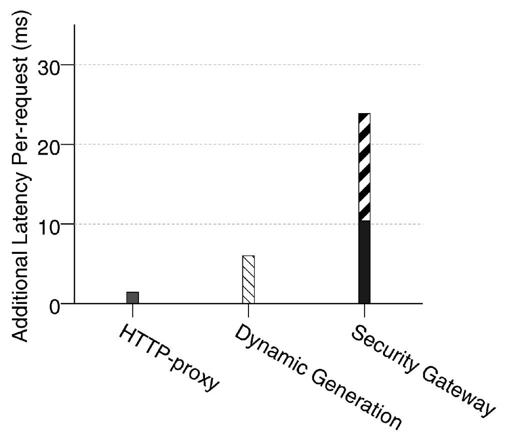

Fig. 7. A comparison of the latency of our system with latencies incurred in common types of HTTP processing.

scheme[7] to distribute work between them (see Fig. 10). (Note that stateful systems do not scale linearly in this way since, ultimately, the centralized state becomes a bottleneck across the cluster.)

The measurements presented here are *worst case* in the sense that the test policy and the test HTML were both complicated and required extensive processing within the security gateway. We designed SPDL-2 to allow more fine-grained use of expensive features like MACs and client-side JavaScript. In the first version of SPDL [8], turning on the MAC facility for *any* URL would have necessitated processing the HTML output of *every* page. By requiring that URLs within a single `<policy>` block do not reference MAC-protected URLs outside that block (see Section 3.6.2), the new system allows the developer to specify the set of application URLs which make use of MAC-protected client-side state and, critically, *those which do not*. URLs with no MAC or JavaScript requirements can be processed much more efficiently by the security gateway as the HTTP response from the Web-server can be streamed to the client directly: It does not have to be parsed or rewritten. Given that processing the HTTP response is by far the dominant performance cost incurred by the security gateway (see Fig. 8) significant speedups are achieved.

Furthermore, we believe that many HTML pages are simpler than our contrived example and have fewer security constraints (i.e., shorter pages without form parameters), leading to better *average case* performance. For example, consider that many of the HTTP messages would contain graphics and, hence, would not require any processing at all. A performance-optimized security gateway could examine the content-type header of HTTP responses, using streaming instead of buffering if no HTML processing is required.

In summary, the performance figures presented in this section relate to our unoptimized prototype under worst-case conditions. The system is designed to be scalable and optimizable and, therefore, we believe that our techniques are applicable in practice.

7. This assumes that load balancing can be achieved cheaply, e.g., by round-robin DNS.

6 Related Work

The idea of using firewalls to prevent unauthorized activity at the application-protocol level is not new. A large number of 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 which apply user-specified validation and transformation rules.

Damiani et al. [19] describe a method for enforcing rôle-based access control policies for remote method invocations via the SOAP protocol [20]. The type of policies described are very different from ours: They consider access control issues, whereas we try to prevent application-level attacks in general. However, the similarity between the two systems lies in the use of a firewall to enforce restrictions at the HTTP-level.

The `<bigwig>` project [21] consists of domain-specific languages and tools for the development of Web services. A part of the `<bigwig>` project, *PowerForms* [22], allows constraints (expressed as regular expressions) to be attached to form fields. A compiler generates both client-side JavaScript and code for server-side checks. Apart from the lack of a general purpose validation language, the other main difference between this and our work is that our system allows the developer to attach validation constraints on both the client and server side, irrespective of the language the original application was written in.

Sanctum Inc., provide a product called *AppShield* [23] which, like our security gateway, inspects HTTP messages in an attempt to prevent application-level attacks. However, despite this apparent similarity, there are significant differences between the two systems: We take the *programmatic approach* of specifying a security policy explicitly; in contrast, AppShield has no policy description language or compiler and attempts to infer a security policy dynamically. While this allows AppShield to be installed quickly, it limits the tasks it can perform. In particular, since there is no policy description language for describing validation or transformation rules, AppShield knows very little about what constitutes valid parameter values in HTTP-requests and can only perform simple checks on data returned from clients. AppShield is intended as a plug-and-play tool which provides a limited degree of protection for *existing* Web sites with application-level security problems. In contrast, we see our approach as a suite of development tools and methodologies which aid in the *design-process* of secure applications. Furthermore, we believe that the combination of our Security Policy Creation Tool and the Security Policy Description Language allows us to get the best of both worlds—a reasonably quick to deploy (thanks to the semiautomatic policy generation) and yet flexible system able to express a wide variety of useful application-specific security policies.

7 Conclusions and Further Work

Enforcing a security policy across a large Web-application is difficult because:

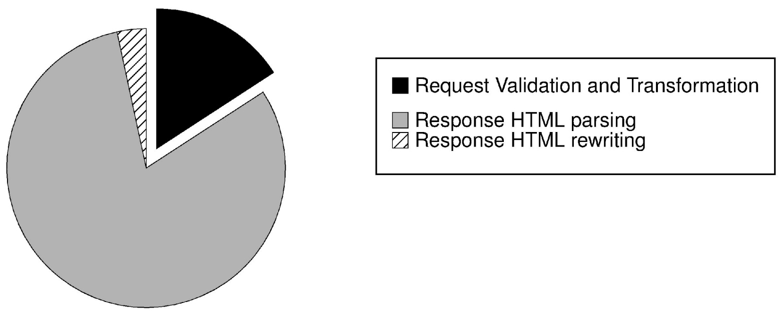

Fig. 8. A breakdown showing the relative cost of HTTP processing stages within the security gateway.

- The application may 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 will be scattered throughout the application.
- The languages used for Web development are not always conducive to writing security-related code. In particular, it is difficult to give any compile-time guarantees about untyped scripting languages such as PHP and VBScript.
- A large Web site may consist of hundreds, if not thousands, of URLs. It must be possible to easily manage the complexity of the security policy as the application scales.
- Performance is a critical factor. It is important to be able to avoid excessive processing in the common case, while being able to impose sufficient checking where necessary to ensure application safety.
- Web applications often contain third-party components. Since it may not be viable to modify the source of such components (either because the code was shipped in binary form or because the license agreement is prohibitive), then it is not obvious how security vulnerabilities should be fixed. (In reality, one is often at the mercy of the company who wrote the component.)

In this paper, we have presented a method for abstracting security-critical code from large Web applications which addresses the problems outlined above. A specification language for describing application-level security policies and a tool to help create policies was described and illustrated with a realistic example.

As well as hoping to secure existing Web sites, we hope that the tools and techniques described in this paper will be useful in the *development process* of new Web applications. By abstracting the security policy from the outset, programmers have the advantage of a well-defined, centralized set of assertions laid out in the SPDL-2 security specification. As well as reducing the amount of code written by each developer, we hope that the project's SPDL-2 specification would act as a useful document, aiding communication between teams of developers and speeding up code-review processes. Justifying these claims with reference to real-life case studies is a high priority for future work.

Another direction for future work is to augment the security gateway with routines for checking digital signatures. It is not possible to leave this to the backend application since any data transformation will cause the signature to become invalid. Therefore, this step must be performed in the security gateway. Similarly, in order to support encryption, the endpoint must be in the security gateway—it is impossible for the gateway to check data it cannot read. Investigating these issues is left for future work.

We do not claim that we have found a automatic fix for all application-level security problems: Although our tools help to secure a Web application, it still requires a competent, security-aware engineer to write/check the security policies by hand.

Based on the research reported in this paper, we claim that our methodology provides a stronger foundation for secure Web applications than conventional tools and development techniques. In addition, we believe that applying this methodology in practice would make a

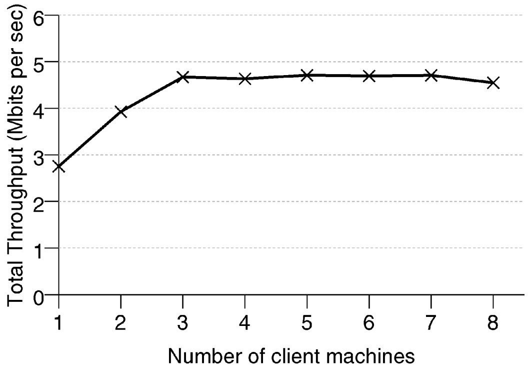

Fig. 9. Total throughput of a single security gateway as the number of concurrently connected clients varies.

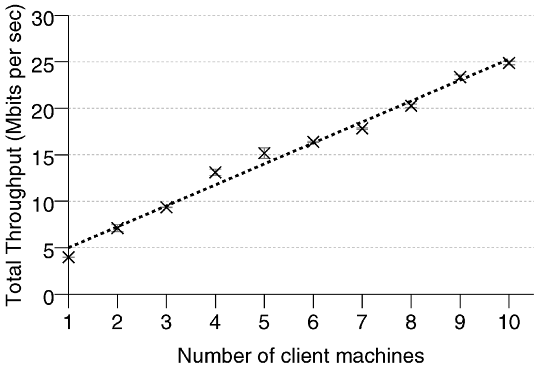

Fig. 10. Total throughput of a cluster of security gateways as the number of concurrently connected clients varies.

significant and immediate impact to the many Web sites which currently suffer from application-level security vulnerabilities.

ACKNOWLEDGMENTS

This work was supported by (UK) EPSRC (Grant Number: GR/N64256) and The Schiff Foundation. Both authors were sponsored by AT&T Laboratories, Cambridge. The authors wish to thank Alan Mycroft and Andy Hopper for their support, encouragement, and patience.

REFERENCES

[1] S. Mu, S. Goodley, "Security Hole Threatens British E-Tailers," The Daily Telegraph Newspaper (UK), available at http://www.telegraph.co.uk/et?pg=/et/01/1/25/ecnsecu2.html, 25 Jan. 2001.

[2] L. Lorek , "New E-Rip-Off Maneuver: Swapping Price Tags," ZD-Net, available at http://www.zdnet.com/intweek/stories/news/0,4164,2692337,00.html, 5 Mar. 2001.

[3] Internet Security Systems (ISS), "Form Tampering Vulnerabilities in Several Web-Based Shopping Cart Applications,"ISS alert, available at http://xforce.iss.net/alerts/advise42.php, 2003.

[4] R. Peteanu, "Best Practices for Secure Web Development," Security Portal, http://securityportal.com/cover/coverstory20001030.html, 2002.

[5] R. Peteanu, "Best Practices for Secure Web Development: Technical Details," Security Portal, available at http://securityportal.com/articles/Webdev20001103.html, 2002.

[6] L.D. Stein, "Referer Refresher," http://www.Webtechniques.com/archives/1998/09/Webm/, 2003.

[7] Microsoft, "HOWTO: Review ASP Code for CSSI Vulnerability," http://support.microsoft.com/support/kb/articles/Q253/1/19.ASP, 2003.

[8] D. Scott and R. Sharp, "Abstracting Application-Level Web Security," *Proc. 11th Int'l World Wide Web Conf.*, pp. 396-407, May 2002.

[9] D. Flanagan, *JavaScript: The Definitive Guide,* third ed. O'Reilly, 1998.

[10] *PHP Hypertext Preprocessor,* available at http://www.php.net/, 2003.

[11] R. Petrusha, P. Lomax, and M. Childs, *VBscript in a Nutshell: A Desktop Quick Reference,* first ed. O'Reilly, 2000.

[12] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Sourcecode in C.* New York: John Wiley & Sons, 1994.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised).* MIT Press, 1997.

[14] R. Milner, "A Theory of Type-Polymorphism in Programming," *J. Computer and System Sciences,* vol. 17, no. 3, 1978.

[15] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," *Advances of Cryptology—Crypto '96 Proc.,* 1996.

[16] P. Syverson, "A Taxonomy of Replay Attacks," *Proc. Computer Security Foundations Workshop VII,* 1994.

[17] *Squid Web Proxy Cache,* available at http://www.squid-cache.org/, 2003.

[18] *MySQL Database Server,* available at http://www.mysql.com/, 2003.

[19] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "Fine Grained Access Control for Soap E-Services," *Proc. 10th Int'l World Wide Web Conf.,* pp. 504-513, May 2001.

[20] D. Box, "Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium (W3C)," http://www.w3.org/TR/SOAP, May 2000.

[21] "The `<Bigwig>` Project," http://www.brics.dk/bigwig/, 2003.

[22] C. Brabrand, A. Mller, M. Ricky, and M. Schwartzbach, "Powerforms: Declarative Client-Side Form Field Validation," *World Wide Web J.,* vol. 3, no. 4, 2000.

[23] Sanctum Inc, "AppShield™ White Paper," Mar. 2001. Available from http://www.sanctuminc.com/, 2003.

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

Richard Sharp currently works in Intel Research having recently completed the PhD degree at the University of Cambridge. 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.