Abstracting Application-Level Web Security

David Scott
Laboratory For Communications Engineering
Engineering Department
Trumpington Street
Cambridge
CB2 1PZ
djs55@eng.cam.ac.uk

Richard Sharp
Computer Laboratory
William Gates Building
JJ Thompson Avenue
Cambridge
CB3 0FD
rws26@cl.cam.ac.uk

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 (*i*) describe a scalable structuring mechanism facilitating the abstraction of security policies from large web-applications developed in heterogenous multi-platform environments; (*ii*) present a tool which assists programmers develop secure applications which are resilient to a wide range of common attacks; and (*iii*) report results and experience arising from our implementation of these techniques.

1. INTRODUCTION

On the 25th January, 2001 an article appeared in a respected British newspaper entitled *Security Hole Threatens British E-tailers* [13]. 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 article published in ZD-Net [17] suggests that between 30% and 40% of e-commerce sites throughout the world are vulnerable to this simple attack. Internet Security Systems (ISS) identified eleven widely deployed commercial shopping-cart applications which suffer from the vulnerability [14].

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 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, 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 [22, 23, 28]. Fixing a single occurrence of a vulnerability is usually easy. However, the massive number of interactions between different components of a dynamic website makes application-level security challenging in general. Despite numerous efforts to tighten application-level security through code-review and other software-engineering practices [18] the fact remains that a large number of professionally designed websites 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 specialised Security-Policy Description Language (SPDL) which is used to program an application-level firewall (referred to as a *security gateway*). Security policies are written in SPDL and compiled for execution on the security gateway. The security gateway dynamically analyses and transforms HTTP requests/responses to enforce the specified policy.

The remainder of the paper is structured as follows: Section 2 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

[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.

technical details of our system for abstracting application-level web security. Our methodology is illustrated with an extended example in Section 4 and we discuss how the ideas in this paper may be generalised in Section 5. We have implemented the techniques discussed in this paper. The performance of our implementation is evaluated in Section 6. Related work is discussed in Section 7; finally, Section 8 concludes.

2. APPLICATION-LEVEL SECURITY

We start by briefly categorising 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.

Form Modification

HTML forms are an application-level security minefield. Our own experiments indicate that a significant percentage of web forms are vulnerable to application-level attacks. The main reason for this is that web designers implicitly trust validation rules which are enforced only on the client-side. Examples of client-side form validation include both constraints imposed by the HTML itself (e.g. the `MaxLength` attribute) and scripts (usually JavaScript programs) which are executed on the client. Of course, in practice users can easily modify client-side validation rules so they should never be trusted.

The stateless nature of the HTTP protocol leaves designers with the task of managing application state across multiple requests. It is often easier to thread state through a series of request/responses using hidden form fields than it is to store data in a back-end database. Unfortunately using hidden form fields in this way enables the client to modify internal application state, leading to vulnerabilities such as the *price changing attack* described in the Introduction. It is interesting to note that a respected textbook on HTML [21] recommends this dangerous practice without any mention of security issues.

Form modification is often used in conjunction with other attacks. For example, changing `MaxLength` constraints on the client may expose buffer overruns and SQL errors on the server side. Information gleaned from such failures provides insights into the internal structure of the site possibly highlighting areas where it is particularly vulnerable.

Although writing server-side code to handle form input securely may not be cerebrally taxing, it is a tedious, time-consuming and error-prone task which 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 leads to a vulnerability where the user can execute arbitrary SQL against the back-end database. The attack is best illustrated with a simple example:

Consider an Employee Directory Website (written in the popular scripting language PHP [3]) which prompts a user to enter the surname of an employee to search for by means of a form-box called `searchName`. On the server-side this search string (stored in the variable `$searchName`) is used to build an SQL query. This may involve code such as:

```
$query = "SELECT forenname,tel,fax,email
          FROM personal
          WHERE surname=‘$searchName’; ";
```

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

```
’; SELECT password,tel,fax,email FROM personal
WHERE surname=‘Sharp
```

then the value of variable, `$query` will become:

```
SELECT forenames,tel,fax,email FROM personal
    WHERE surname=‘’;
SELECT password,tel,fax,email FROM personal
    WHERE surname=‘Sharp’;
```

When executed on some SQL databases, this will result in Sharp's password being returned instead of his forename. (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 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 website itself, a trusted source. The implications of XSS are severe; for example, the *Same Origin Policy*, a key part of JavaScript's security model [12], is subverted.

A CERT advisory (CA-2000-02) [7] outlines a range of serious attacks which come under the general heading of XSS. The list of attacks include stealing confidential information (e.g. usernames, passwords, credit-card numbers), altering the behaviour of forms (e.g. posting data to a cracker's machine) and exposing SSL-Encrypted connections. Clayton *et al* [9] describe the details of a Java/JavaScript XSS attack which reveals the IP-addresses of clients using a (supposedly) anonymous dating service.

It is well known that XSS vulnerabilities can be fixed by encoding HTML meta-characters[2] explicitly using HTML's `#&`$\langle n\rangle$ syntax, where $\langle n\rangle$ is the numerical representation of the encoded character. However, the flexibility of HTML makes this a more complicated task than many people realise [8]. Furthermore, for large applications, it is a laborious and error-prone task to ensure that *all* input from the user has been appropriately HTML encoded.

[2] Meta-characters are those which have special meaning within HTML. For example `<` and `>` are used to delimit tags.

2.1 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 [3] and VB-Script [24]. 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 run-time 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 in 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 non-technical one, is a lack of concern for security in the web-development community. Although we realise that this is a generalisation, 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[3]; 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 websites from application-level attacks. Whilst we recognise 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) is used to specify a set of validation constraints and transformation rules.

2. A *policy compiler* automatically translates the SPDL into code for checking validation constraints.

3. An application-level *security gateway* is positioned between the web-server and client machines.

Figure 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 could be executed as a separate process on the existing web-server or, to achieve better performance, integrated into the web-server directly.

3.1 System Overview

A designer codes a set of *validation constraints* and *transformation rules* in SPDL. 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 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 detailed description of SPDL is given in Section 3.2.

The *policy compiler* translates SPDL 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, analyse and transform whole HTTP messages (see Section 3.4). As well as checking HTTP requests, the security gateway also rewrites the HTML in HTTP responses, annotating it with Message Authentication Codes (MACs) [27] to protect state which may have been malliciously modified by clients (see Section 3.4.2).

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 specification. The security gateway analyses 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

[3] We hasten to add that this was not our sponsors, AT&T!

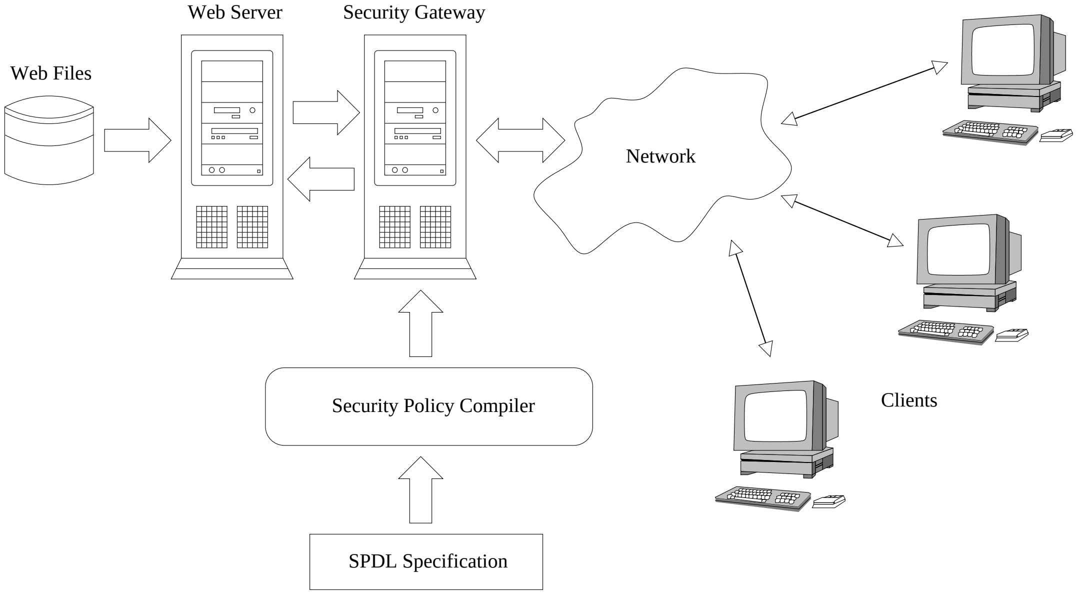

Figure 1: A diagrammatic view of our system for abstracting application-level web security

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

At the top level an SPDL specification is an XML document. The DTD corresponding to SPDL is shown in Figure 2. A `policy` element contains a series of `URL` and `cookie` 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.
- Setting `required` to “`Y`” specifies that p must always contain a (non-zero length) value;
- Setting `MAC` to “`Y`” specifies that the value of p must be accompanied by a Message Authentication Code (MAC) [27] generated by the server. This prevents the user from changing the value of the parameter to arbitrary values (see Section 3.4.2).
- 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 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 security policy description:

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

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 `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 [20]. (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.)

```
<!ELEMENT policy (URL*, cookie*)>


  <!ELEMENT URL (parameter*)>

    <!ATTLIST URL prefix CDATA #REQUIRED>


    <!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 MAC        (Y | N) "Y">
    <!ATTLIST cookie type       (int | float | bool | string) #REQUIRED>


  <!ELEMENT validation (#CDATA)>


  <!ELEMENT transformation (#CDATA)>

    <!ATTLIST transformation htmlencode (Y | N) "Y">
```

Figure 2: The XML DTD for the Security Policy Description Language

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

Figure 3: The Abstract Syntax of the Validation Language

The abstract syntax of the validation language is shown in Figure 3. A well-formed validation expression has type boolean. 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 3.3). 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 cast 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 are much simpler than validation expressions and are delimited by the `<transformation>` tag. The contents of a `transformation` element nested within a `parameter` element, p, specifies a pipeline of transformations to be applied to data received via p. For example, if we always wanted to apply transformation `t1` followed by `t2` to parameter `p` passed via a given URL then our SPDL specification would contain:

```
<URL prefix="...">
   <parameter name="p" ...>
      <transformation> t1 | t2 </transformation>
   </parameter>
</URL>
```

Transformations are selected from a pre-defined library. In our current implementation we have defined the following transformations:

EscapeSingleQuotes Replace single quotes with their HTML character encoding.

EscapeDoubleQuotes Replace double quotes with their HTML character encoding.

HTMLEncode HTML-encode the data. Replace meta-characters 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>`.

Facility is provided for the user to define other transformations and include them in the library.

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:

```
 ...
   <transformation htmlencode="N">
      PartialHTMLEncode | EscapeSingleQuotes
   </transformation>
 ...
```

Recall from Figure 2 that, at the top-level, an SPDL description consists 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. In this presentation we make the simplifying assumption that cookies are global across the whole site (i.e. the *path attributes* of all `Set-Cookie` headers in HTTP responses are set to "/"). Under these circumstances the client sends the values of *all* the application's cookies with each HTTP-request. Since all client-side state is sent to the server in each request we can generate MACs securely without requiring server-side state in the security gateway (see Section 3.4.2).

3.3 Policy Compiler

The *policy compiler* takes an SPDL 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 ready to be embedded into forms and executed on clients.

Compilation is performed in two passes. In the first pass the declared parameters and their types are enumerated; in the second pass the contents of the validation and transformation elements are compiled. Using a two-pass architecture allows the use of forward parameter references. For example, consider a `URL` element, u, which contains declarations of parameters p_1 and p_2, where p_1 is declared before p_2. It is perfectly acceptable for the validation code of parameter p_1 to refer to p_2 (and *vice-versa*).

Validation expressions are type-checked at compile-time, helping to eliminate errors from SPDL validation code. In the current incarnation of the system, validation 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 [19]) could not be employed in future versions.

3.4 The Security Gateway

Figure 4 shows the algorithm executed by the Security Gateway on receipt of an HTTP request. First, the URL is extracted from the HTTP header. This is used to select the appropriate validation rules and transformations to apply. If the URL does not match any of those specified in the security policy then the request is not propagated to the web-server and an error page is returned to the user. By forbidding all URLs that do not match those explicitly in our database we prevent a cracker using obscure, non-standard URL encoding techniques to circumvent the security gateway (thus avoiding attacks of the kind recently used on Cisco's Intrusion Detection System [11]). Rejecting unspecified URLs also provides an engineering benefit: since each URL requires a corresponding SPDL definition engineers are *forced* to keep the security policy in sync with the application.

Having identified a valid URL, the security gateway proceeds to check the names of all parameters and cookies passed in the HTTP request. Errors are generated if (*i*) any of the parameters present are not declared in the SPDL policy; (*ii*) any of the required parameters are missing; or (*iii*) the cookies present do not precisely match those specified in the SPDL specification. Once we are sure that the HTTP message contains a valid combination of cookies and GET/POST parameters, type and length constraints are checked. If any violations occur at this stage then a descriptive error message is returned to the client. The security gateway then checks that the message authentication code is valid. Section 3.4.2 describes this process in detail.

Next the transformations specified in the security policy are applied. Transformations are *total* functions on strings—well written transformation code should not generate exceptions. However, if a badly written transformation function does generate a run-time exception then the process is aborted and an error message is returned to the client. Finally all validation expressions are evaluated. If all of the validation expressions evaluate to *true* then the HTTP request is forwarded to the web-server and the page is fetched.

The security gateway processes HTTP responses returned from the web-server: JavaScript validation code (generated by the SPDL compiler) is inserted into HTML forms (see Section 3.4.1), `MaxLength` attributes on form elements are set according to the SPDL specification and, finally, message authentication codes are generated for form fields and URL-parameters (see Section 3.4.2) if required.

3.4.1 Client-side Form Validation

For each HTML-form in the HTTP response the security gateway inserts JavaScript code to perform validation checks on the client's machine. (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 process of inserting validation JavaScript on forms involves the following steps:

1. The security gateway scans the HTML for `<form>` tags and extracts the destination URL from the form's `action` attribute.

2. From this destination URL the validation constraints to apply to form fields are determined. (Note that we also have to look at the form's `method` attribute to determine whether the fields will be sent as GET or POST parameters).

3. A JavaScript validation program is constructed by concatenating pre-compiled JavaScript fragments (one for each of the fields on the form). Recall that the JavaScript code has already been generated by the policy compiler—see Section 3.3.

4. The JavaScript validation program is inserted into the `onSubmit` attribute of the `form` tag.

If there is already an `onSubmit` attribute present then the security gateway does not insert the validation program. We take the view that if a form already has an `onSubmit` attribute then this over-rides the automatically generated validation JavaScript.

As well as performing the checks specified in the `validation` elements of SPDL specifications the validation JavaScript also enforces type-checking rules, enforces min-length constraints and ensures that all required fields contain data. (Note that we do not have to worry about max-length constraints since these are inserted directly as HTML attributes).

3.4.2 Message Authentication Codes

We have already seen that an SPDL specification can declare that certain URL parameters must only contain data accompanied by a *Message Authentication Code* (MAC) [27] 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).

Consider an ordered list of values, l. We write, $mac(l)$ to denote the message authentication code corresponding to l. In our current implementation the value of $mac(l)$ is calculated as the MD5-hash [25] of a string containing the values of l concatenated together along with a time-stamp and a *secret*. The *secret* is a value which is not known by the client; since clients do not know the secret they cannot construct their own MACs.

Before describing the algorithm used to annotate the HTML with MACs we make a few auxiliary definitions. Consider a list of pairs, $l = [(k_1, v_1), \ldots, (k_n, v_n)]$. We define $sort(l)$ to be l sorted by k-values and $vals(l)$ to be the list $[v_1, \ldots, v_n]$. Function *sortVals* is the composition of *sort* and *vals* (i.e.

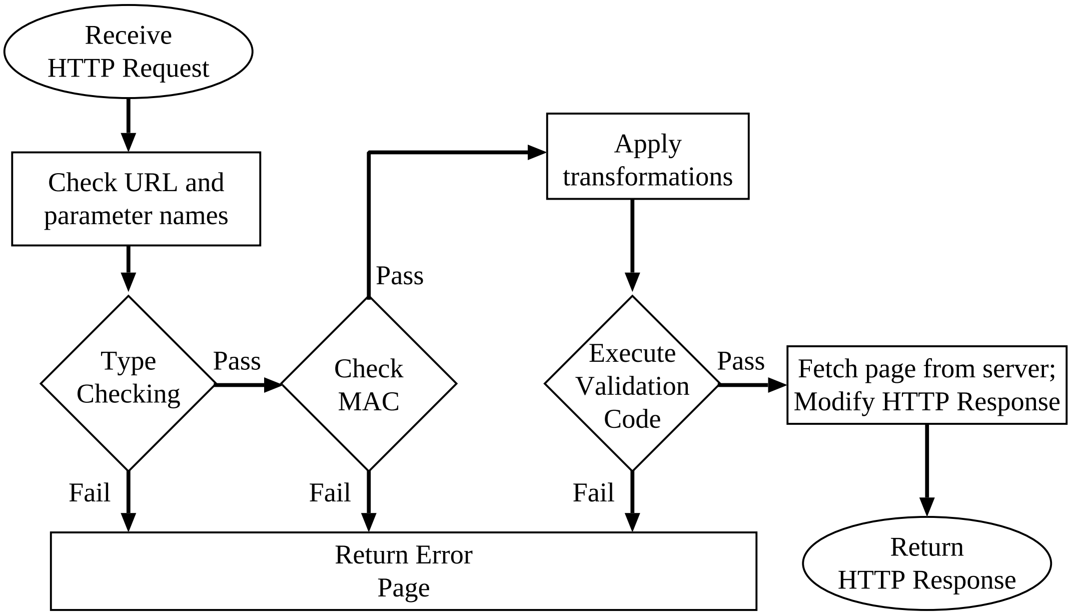

Figure 4: The tasks performed by the security gateway

$sortVals(l) = vals(sort(l)))$. Appending of lists is performed by the binary infix operator '@'. Now consider an HTTP request, H_{req}, which triggers response H_{res}. The algorithm for annotating the body of H_{res} with MACs is as follows:

1. Construct a list, l_c, of the name/value pairs of cookies in H_{req}. (Recall our assumption that clients return all cookies with each HTTP-request—see Section 3.2).
2. Remove the entries from l_c corresponding to cookies which do not have the `MAC` attribute set to "`Y`" in the SPDL specification.
3. For each URL, u, (which is not a form action) in the body of H_{res}:
   (a) Construct a list, l_g, containing all the parameter/value pairs contained in URL u.
   (b) Remove the entries from list l_g corresponding to parameters which do not have the `MAC` attribute set to "`Y`" in the SPDL specification.
   (c) Generate a time-stamp to be included in the MAC.
   (d) Add a new parameter to u which records the value of the time-stamp.
   (e) Add a new parameter to u to record the MAC. The value of the MAC is given by:
   $$mac(sortVals(l_c) \text{ @ } sortVals(l_g))$$
4. For each form, f, with action-URL, u, in the body of H_{res}:
   (a) Construct a list, l_g, of all the parameter/values pairs contained in URL u.
   (b) Construct a list, l_p, of all the field-name/value pairs contained in form f.
   (c) Remove the entries from lists l_g and l_p corresponding to parameters which do not have the `MAC` attribute set to "`Y`" in the SPDL specification.
   (d) Generate a time-stamp to be included in the MAC.
   (e) Add a new parameter to u which records the value of the time-stamp.
   (f) Add a new parameter to u to record the MAC. The value of the MAC is given by:
   $$mac(sortVals(l_c) \text{ @ } sortVals(l_g) \text{ @ } sortVals(l_p))$$

For example consider the URL:

```
http://example/a.asp?p1=4&p2=5
```

In the case where both `p1` and `p2` require a MAC then this URL will be re-written, taking the form:

```
http://example/a.asp?p1=4&p2=5
                     &mac=3a53fe1d995a23
                     &time=13eaf49b
```

where the parameter `mac` stores the message authentication code corresponding to `p1` = 4, `p2` = 5 with the time-stamp stored in parameter `time`.

When data is received from a client via GET/POST parameters then the values of those parameters which have their `MAC` attribute set to "`Y`" are fed back into the MAC generation algorithm described above. Note that the original time-stamp (returned from the client as a GET parameter) is also required to recompute the MAC. We compare the recalculated MAC with the MAC returned from the client in order to determine whether any parameters were tampered with.

When designing the MAC algorithm one of our major concerns was to avoid *replay attacks* [29] 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 key is that the MAC prevents the `Price` and `productID` being modified independently.

3.5 Extensions

As well as applying the validation and transformation rules of SPDL specifications, our security gateway performs a number of other tasks. Two of these are described in this section.

3.5.1 Restricting Values of Select Parameters

Select parameters (delimited using the `<select>` tag in HTML forms) invite users to choose options from a pre-specified list. Although web designers often make the assumption that clients will only select values present in the list, a simple form-modification attack allows clients to submit arbitrary data in select parameters.

The security gateway protects against such an attack, preventing clients from submitting values for select parameters that were not present in the original HTML form. Our implementation involves the use of a *control field* which encodes dynamically generated lists of valid values for select parameters. The control-field is a hidden form-field generated automatically by the security-gateway and inserted into forms which contain `<select>`s. When form parameters are returned to the server, the value of the control field is decoded and used to validate values of select parameters. To prevent the control field being maliciously modified by clients, its value is included in the calculation of the form's message authentication code.

3.5.2 Protecting against Server Misconfiguration

Web applications often consist of a number of files containing embedded code (e.g. PHP or VBscript) which is executed on the server-side in order to generate dynamic responses to client requests. A number of attacks on web-servers (or indeed badly configured web-servers) can result in this embedded code being transmitted to the client in source form[4]. This can be potentially devastating since it gives crackers detailed information about the inner-workings of the application. For example, in some (badly written) applications the code contains plaintext passwords used to authenticate against back-end database servers.

We have demonstrated that our security gateway can protect against such attacks by searching for sequences of characters which delimit embedded code in HTTP responses (e.g. `<%`, `%>` for ASP or `<?php`, `?>` for PHP). Detection of such delimiters implies (with reasonable probability) that server-side code is about to leaked. Hence, if any delimiters are found the security gateway filters the HTTP response and returns a suitable error message to the client informing them that the page they requested is unavailable.

4. CASE STUDY

To illustrate our methodology we consider using our system to secure a simple e-commerce system. Consider the following scenario:

As a final step in a purchasing transaction, 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>
```

A single cookie, `sessionKey`, is used to uniquely identify clients' sessions. 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.

For the purposes of this example let us assume that the system is vulnerable in the following ways:

1. The "modifying values of hidden form-fields" attack (as described in the Introduction) can be used to reduce the price.

[4] It is amusing to observe that entering a few VBscript keywords into a search engine (we tried '`dim "<%" set con`' on Google) results in a number of matches, some of which contain ASP code unwittingly returned from misconfigured servers. Even if the problem was subsequently fixed, Google keeps a cached copy of the code!

2. JavaScript can be embedded in the `surname` field which, when viewed on the company's intranet, leads to cross-site scripting vulnerabilities.

3. SQL can be entered into the `surname` field using the attack described in Section 2.

4. The session key 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 `sessionKey` cookie.

The SPDL specification corresponding to the form's action URL is presented in Figure 5. Each of the parameters shown in the form above are declared and a number of validation and transformation rules specified. Most of the SPDL 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.

Using the policy description of Figure 5 we are able to fix all of the system's vulnerabilities (described above) without modifying any of the code:

1. The form- and cookie-manipulation attacks can no longer be used since the `price` and `productID` fields along with the `sessionKey` cookie all have their `MAC` attributes (implicitly) 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 is HTML-encoded, preventing XSS attacks.

3. SQL attacks are prevented by applying transformations to escape quotes in the `surname` field.

As the purchasing form is sent to clients, the security gateway inserts JavaScript (derived from the SPDL specification of Figure 5) 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 non-zero-length value etc. Note that if extra validation constraints are required, they can simply be added once to the SPDL 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. GENERALISING OUR SYSTEM

Whilst we advocate the use of a specialised Security Policy Description Language in the majority of cases, we recognise that there are circumstances where the increased flexibility of a general purpose programming language may be desirable.

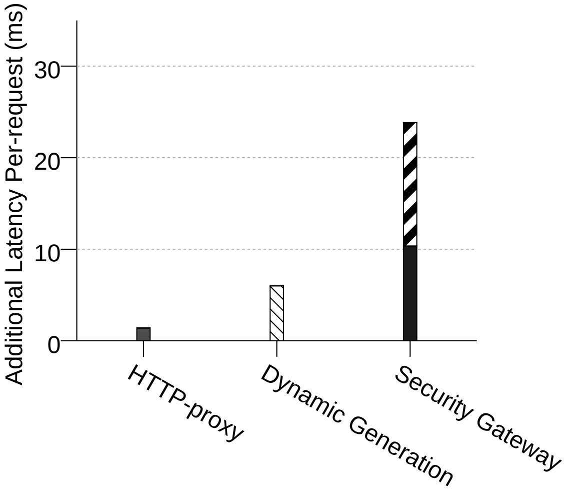

Figure 6: A comparison of the latency of our system with latencies incurred in common types of HTTP processing

In such cases we argue that using a security gateway to abstract security-related code is still a useful technique. Instead of generating code for the security gateway via the SPDL compiler, we observe that the security policy can be encoded in a programming language of choice and compiled directly for execution on the security gateway. Although one loses the specialised features of the SPDL, using a general purpose programming language provides designers with a greater degree of freedom. (Of course with great power comes great responsibility [15]. Programmers must take extra care to structure their security policy enforcement code carefully).

We experimented with this idea by programming our security gateway directly in OCAML [16], using a comprehensive HTTP library to process HTTP requests and responses. We found that, even when using a general purpose programming language to express the security policy, using a security gateway to structure an application still provides a number of advantages. In particular:

- we found the features of OCAML (notably its strict type system) more conducive to writing security critical code than other languages more commonly used for web application development;

- by abstracting the security policy cleanly we gained the usual advantages of maintainability, clarity, increased code re-use and reduced code size.

6. SYSTEM PERFORMANCE

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

Figure 6 shows the 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 Communications Engineering

```
<policy>
  <url prefix="http://www.example/buy.asp">
     <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> EscapeSingleQuotes | EscapeDoubleQuotes </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>
  <cookie name="sessionKey" maxlength="15" minlength="15" type="int" />
</policy>
```

Figure 5: SPDL specification for case study

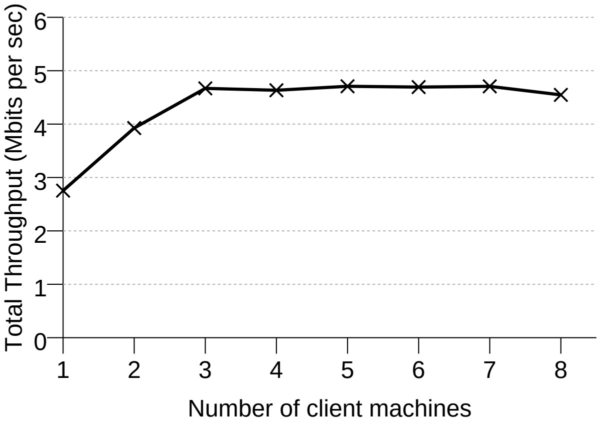

Figure 7: Total throughput of a single security-gateway as the number of concurrently connected clients varies

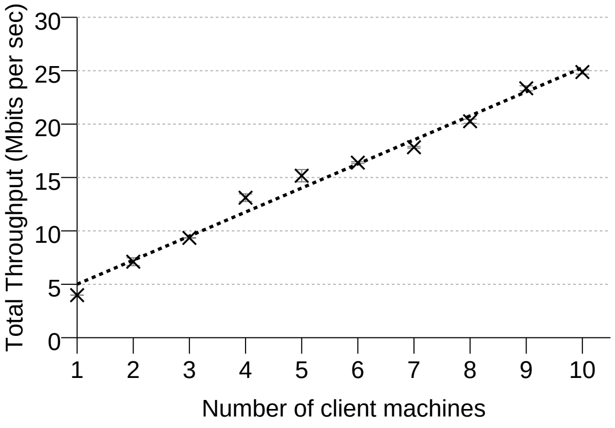

Figure 8: Total throughput of a cluster of security-gateways as the number of concurrently connected clients varies

(University of Cambridge)[5] augmented with the web-form described in our case-study of Section 4. The leftmost bar shows the latency added by a Squid [4] 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 [2] backend; the rightmost bar shows the latency of using the security gateway to enforce the security policy of Figure 5. 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.

The latency of our system is large compared with the latencies incurred in proxy caching and dynamic page generation. To some extent this is due to the fact that our naïve implementation is completely unoptimised. However, we recognise 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 concentrate on performance. Potential optimisations include (*i*) using a specialised 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); (*ii*) reducing latency by streaming the HTTP messages and processing them on-the-fly whenever possible; (*iii*) writing speed critical parts of the security gateway directly in C.

Figure 7 shows how the total throughput of a single security gateway varies as the number of concurrently connected clients increases. The measurements were taken running the security gateway on a dual P-III 500 MHz. The throughput quickly reaches a maximum value as the CPUs become saturated. Again, we are confident that optimising 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 backend database. Since the security gateway is stateless, one may increase throughput linearly simply by deploying multiple security gateways and using a load balancing scheme to distribute work between them (see Figure 8). (Note that stateful systems do not scale linearly in this way since, ultimately, the centralised state becomes a bottleneck across the cluster.)

The measurements presented here are *worst case* in the sense that the HTML used to test the system was long and complicated, containing both URLs and form parameters. In reality we believe that many HTML pages would be simpler for the security gateway to process (i.e. shorter, without form parameters), leading to better *average case* performance. Note that many of the HTTP messages would contain graphics and hence would not require any processing at all. A performance-optimised security gateway could examine the content-type header of HTTP responses, using streaming instead of buffering if no HTML processing is required.

Furthermore, note that most of the overhead of the security gateway is due to annotating HTML with MACs. If the SPDL policy does not require the use of MACs then only HTTP-request parameters need to be checked; the security gateway can stream HTTP-responses directly.

We believe that the performance figures presented in this section demonstrate that our techniques are applicable in practice.

7. RELATED WORK

The idea of using firewalls to prevent unauthorised 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* [10] describe a method for enforcing rôle-based access control policies for remote method invocations via the SOAP protocol [5]. The type of policies described are very different to 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 [1] consists of domain-specific languages and tools for the development of web services. A part of the `<bigwig>` project, *PowerForms* [6], 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 main difference between this and our work is that PowerForms can only be used for web-applications developed using the `<bigwig>` languages and tools. In contrast our security-gateway works at the HTTP-level and secures web-applications written in all languages.

Sanctum Inc. provide a product called *AppShield* [26] 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 SPDL or compiler and attempts to infer a security policy dynamically. Whilst 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* websites 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.

[5] `http://www-lce.eng.cam.ac.uk/`

8. CONCLUSIONS AND FURTHER WORK

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

- The application may be written in a variety of (non-interoperating) 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.
- 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 was described and illustrated with a realistic example.

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, centralised set of assertions laid out in the SPDL security specification. As well as reducing the amount of code written by each developer we hope that the project's SPDL 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 high priority for future work.

Another direction for future work is to augment the security gateway with a library of security-related services (e.g. authentication and generation of secure session IDs). These services could be called from the web-application using protocols such as XML-RPC or SOAP [5].

On their website Sanctum claim that their "AppShield software secures your site by blocking *any* type of application manipulation through the web". Clearly this is false: if it were possible to solve all application-level security problems with a black-box tool then there would be no need for further security research.[6] In contrast, we do not claim that we have found a automatic fix for all application-level security problems: although our tool helps 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 significant and immediate impact to the many websites which currently suffer from application-level security vulnerabilities.

[6] If you don't find this argument convincing then consider the following counter-example: deleting JavaScript validation code on the client-side is an "application manipulation through the web" which AppShield will not detect.

Acknowledgement

This work was supported by (UK) EPSRC GR/N64256 and The Schiff Foundation. Both authors are sponsored by AT&T Laboratories, Cambridge.

The authors wish to thank Alan Mycroft, Andrei Serjantov and Richard Clayton for their valuable comments and suggestions.

9. REFERENCES

[1] The `<bigwig>` project. `http://www.brics.dk/bigwig/`.

[2] *MySQL database server*. `http://www.mysql.com/`.

[3] *PHP hypertext preprocessor*. `http://www.php.net/`.

[4] *Squid web proxy cache*. `http://www.squid-cache.org/`.

[5] Box, D. Simple object access protocol (SOAP) 1.1. world wide web consortium (W3C). May 2000. `http://www.w3.org/TR/SOAP`.

[6] Brabrand, C., Mller, A., Ricky, M., and Schwartzbach, M. Powerforms: Declarative client-side form field validation. *World Wide Web Journal 3*, 4 (2000).

[7] CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests. `http://www.cert.org/advisories/CA-2000-02.html`.

[8] CERT. Understanding malicious content mitigation for web developers. `http://www.cert.org/tech_tips/malicious_code_mitigation.html/`.

[9] Clayton, R., Danezis, G., and Kuhn, M. Real world patterns of failure in anonymity systems. In *Proceedings of the Workshop on Information Hiding* (2001), vol. 2137, Springer-Verlag, LNCS.

[10] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. Fine grained access control for soap e-services. In *Proceedings of the 10th International World Wide Web Conference* (May 2001), ACM, pp. 504–513.

[11] eEye Digital Security. %u-encoding IDS bypass vulnerability. Advisory AD20010705, `http://www.eeye.com/html/Research/Advisories/AD20010705.html`.

[12] Flanagan, D. *JavaScript: The Definitive Guide*. O'Reilly. ISBN: 1565923928.

[13] Goodley, S. Security hole threatens british e-tailers. The Daily Telegraph Newspaper (UK). 25th January, 2001. `http://www.telegraph.co.uk/et?pg=/et/01/1/25/ecnsecu2.html`.

[14] Internet Security Systems (ISS). Form tampering vulnerabilities in several web-based shopping cart applications. ISS alert. `http://xforce.iss.net/alerts/advise42.php`.

[15] Jemas, B., Bendis, B. M., and Bagley, M. *Ultimate Spider-man: Power and Responsibility.* Marvel Books. ISBN: 078510786X.

[16] Leroy, X. *The Objective Caml System Release 3.0.* INRIA, Rocquencourt, France, 2000.

[17] Lorek, L. New e-rip-off maneuver: Swapping price tags. ZD-Net. 5th March, 2001. `http://www.zdnet.com/intweek/stories/news/0,4164,2692337,00.html`.

[18] Microsoft. HOWTO: Review ASP code for CSSI vulnerability. `http://support.microsoft.com/support/kb/articles/Q253/1/19.ASP`.

[19] Milner, R. A theory of type-polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978).

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

[21] Musciano, C., and Kennedy, B. *HTML & XHTML: The Definitive Guide (4th Edition).* O'Reilly, 2000. ISBN: 0-596-00026-X. See page 328.

[22] Peteanu, R. Best practices for secure web development. Security portal. `http://securityportal.com/cover/coverstory20001030.html`.

[23] Peteanu, R. Best practices for secure web development: Technical details. Security portal. `http://securityportal.com/articles/webdev20001103.html`.

[24] Petrusha, R., Lomax, P., and Childs, M. *VBscript in a nutshell: a desktop quick reference.* O'Reilly. ISBN: 1565927206.

[25] Rivest, R. The MD5 message digest algorithm. Internet Request For Comments. April 1992. RFC 1321.

[26] Sanctum Inc. AppShield™ white paper. March 2001. Available from `http://www.sanctuminc.com/`.

[27] Schneier, B. *Applied cryptography: protocols, algorithms, and sourcecode in C.* John Wiley & Sons, New York, 1994.

[28] Stein, L. D. Referer refresher. `http://www.webtechniques.com/archives/1998/09/webm/`.

[29] Syverson, P. A taxonomy of replay attacks. In *Computer Security Foundations Workshop VII* (1994), IEEE Computer Society Press.