SPECTRE: A Tool for Inferring, Specifying and Enforcing Web-Security Policies

David Scott and Richard Sharp

Computer Laboratory, University of Cambridge,
William Gates Building, JJ Thompson Avenue, Cambridge CB3 0FD, UK
djs55@eng.cam.ac.uk, rws26@cl.cam.ac.uk

Abstract. Implementing web-applications securely is a laborious and error-prone task; as a result a large number of (professionally designed) websites suffer from serious application-level security vulnerabilities. In this paper we describe SPECTRE, a tool which helps to secure dynamic web-applications. As well as aiding in the development process of new applications SPECTRE can also be used to fix vulnerabilities in existing web-based components, even when the source of these components is not available.

1 Introduction and Motivation

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). Such vulnerabilities are well-known and a number of articles have been published advising developers on how they can be avoided. However, despite efforts to tighten application-level security through code-review and other software-engineering practices, the fact remains that a large number of professionally designed websites still suffer from serious application-level security holes.

Examples of common vulnerabilities include *Cross-Site Scripting* (where sites such as bulletin boards can be subverted through the submission of malicious content), *SQL-vulnerabilities* (which allow arbitrary SQL to be executed against an application's back-end database) and *Form-Manipulation* (where maliciously modifying HTML-forms with a text-editor can lead to unexpected data being posted to an application) [3]. The consequences of application-level vulnerabilities are severe: hackers have tricked e-commerce sites into shipping goods for no charge [2], usernames and passwords have been harvested and confidential information (such as addresses and credit-card numbers) has been leaked [1].

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

– The web-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. This lack of structure makes fixing vulnerabilities difficult: the same security hole may occur multiple times throughout the code.

– 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 previous work we propose a framework to alleviate these problems [3]. 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.

In this paper we describe our implementation of these techniques in the form of SPECTRE[1]: a tool for securing dynamic web-applications.

2 The SPECTRE Tool

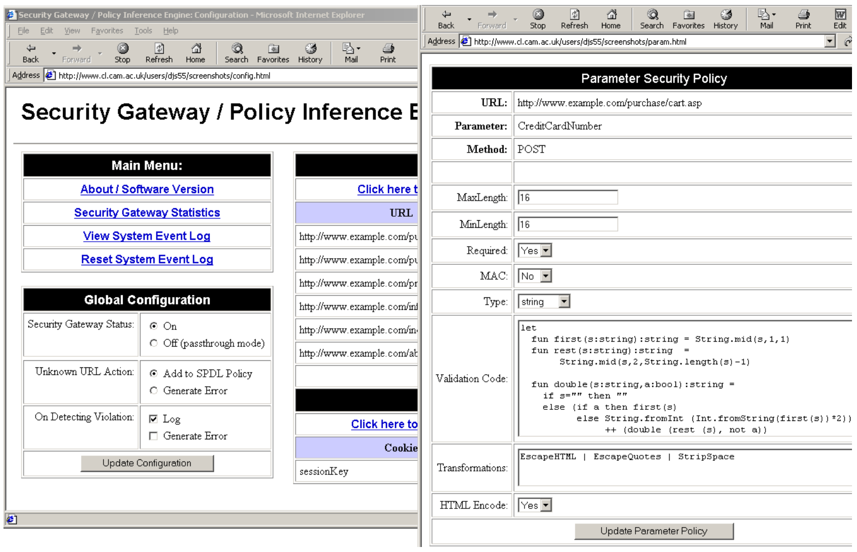
The SPECTRE tool consists of three components: (*i*) a *policy compiler* which automatically translates SPDL into code which checks validation constraints and applies transformation rules; (*ii*) a *security-gateway* which dynamically enforces security policies and (*iii*) a *security-policy inference engine* which analyses interactions between users and web-applications in order to automatically generate (SPDL) security-policies. Once deployed, the SPECTRE tool can be programmed and configured using a standard HTML interface.

Security-Policy Specification and Enforcement: The SPDL language facilitates the definition of *validation constraints* and *transformation rules*. Validation constraints place restrictions on the interaction between clients and web-applications (e.g. “the value of this cookie must never be modified” or “this form-field must contain a valid credit-card number”). Transformation rules specify various transformations which will be applied to user-input (e.g. pass data from all fields on form, f, through a function to escape HTML meta-characters). The details of SPDL are described fully in [3].

Figure 1 shows a screenshot of the SPECTRE User-Interface displayed using Microsoft Internet Explorer. The leftmost window (Main Configuration Settings) provides global configuration options and lists the Cookies and URLs to be secured. The rightmost window shows the Parameter Security-Policy Form. Using this interface a designer can specify validation constraints and transformation rules for individual form-parameters, URL-parameters and cookies.

On close inspection one can see that the screenshot specifies the security policy for the form-parameter `CreditCardNumber`. Validation constraints include bounds on the length of data passed via the parameter, the type of data expected (e.g. string, int,

[1] SPECTRE stands for: Security Policy EnforCement Through Run-timE checks.


Security Gateway / Policy Inference Engine: Configuration - Microsoft Internet Explorer

Address http://www.cl.cam.ac.uk/users/djs55/screenshots/config.html

Security Gateway / Policy Inference E

Main Menu:
About / Software Version
Security Gateway Statistics
View System Event Log
Reset System Event Log

Global Configuration	
Security Gateway Status:	[✓ On] Off (passthrough mode)
Unknown URL Action:	[✓ Add to SPDL Policy] Generate Error
On Detecting Violation:	[✓ Log] Generate Error
Update Configuration	

Click here t

URL
http://www.example.com/pu
http://www.example.com/pu
http://www.example.com/pr
http://www.example.com/inf
http://www.example.com/in
http://www.example.com/ab

Click here to

Cookie
sessionKey

Address http://www.cl.cam.ac.uk/users/djs55/screenshots/param.html

Parameter Security Policy	
URL:	http://www.example.com/purchase/cart.asp
Parameter:	CreditCardNumber
Method:	POST
MaxLength:	16
MinLength:	16
Required:	Yes
MAC:	No
Type:	string

Validation Code:
```
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))
```

Transformations:
```
EscapeHTML | EscapeQuotes | StripSpace
```

HTML Encode: Yes

Update Parameter Policy

Fig. 1. Using SPECTRE to secure an e-Commerce system

float, bool) and whether the parameter is required (required parameters *must* be supplied by the user). Facility is provided for the designer to specify validation code in a general purpose programming language which, in our current implementation, resembles a simply-typed subset of ML. In the screenshot the validation-code implements the Luhn-formula, a commonly used validation check for credit-card numbers.

The transformation control (on the Parameter Security-Policy Form) allows the designer to specify transformations to be applied to data received via the `CreditCardNumber` parameter. Transformations are selected from a user-extensible library (currently implemented in OCAML). In this example we apply transformations which (*i*) escape HTML meta-characters (preventing cross-site scripting attacks); (*ii*) escape quotes (preventing a class of SQL attacks); and (*iii*) strip spaces (removing superfluous formatting from credit-card numbers).

The contention between the stateless nature of HTTP and the stateful nature of many web-applications leaves application designers with the task of managing state explicitly on an *ad-hoc* basis. A common technique (albeit an insecure one) is to thread state through client requests and responses thus alleviating the overhead of storing state centrally on the server-side. Cookies, URL-parameters and hidden form-fields are often used for this purpose. Although not described in detail here, the MAC validation constraint allows data to be threaded through clients securely. Message Authentication Codes are generated and checked dynamically to ensure that security-critical data has not been maliciously modified by clients [3]. This protects against attacks such as the infamous *price-changing attack* [2].

The Policy Compiler translates validation/transformation rules into code to perform server-side checking/manipulation; this code is dynamically linked into the Security Gateway. If any of the validation constraints are violated at run-time then a descriptive error page is returned to the client. As well as generating code for server-side checks, the Policy Compiler also emits JavaScript for client-side validation; the Security Gateway dynamically inserts the JavaScript validation code into HTML-forms. In this way validation checks are performed on *both* the client-side (to improve observed latency between form-submission and receiving validation errors) and the server-side (for security). The key benefit here is that both client- and server-side code is derived from the same specification. Note that 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.

As well as aiding the development of *new* web-applications, SPECTRE can be used as a tool to secure *existing* web-applications. SPECTRE operates completely independently of the original application source code and is therefore useful regardless of whether the code is available or not.

Security-Policy Inference: We acknowledge that writing SPDL for large web-applications with complex interactions between components can be a time consuming and tedious task. To allieviate this problem we have incorporated an automatic security-policy inference feature. When in "inference-mode" SPECTRE dynamically analyses the interactions between a web-application and its clients in order to generate a simple SPDL policy automatically. By analysing HTTP-requests, SPECTRE builds up a database of URLs annotated with their associated parameters and cookies. For each parameter, the inference engine keeps track of the type of data passed, maintains upper and lower bounds on the length of data and records whether or not the parameter is *always* present in requests for a particular URL. This information is used to construct an SPDL skeleton which can be further refined by the designer.

References

1. 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.
2. INTERNET SECURITY SYSTEMS (ISS). Form tampering vulnerabilities in several web-based shopping cart applications. ISS alert. `http://xforce.iss.net/alerts/advise42.php`.
3. SCOTT, D., AND SHARP, R. Abstracting application-level web security. Tech. Rep. 2001.11, AT&T Laboratories Cambridge, November 2001. Also submitted for publication.