

Newspeak: A Secure Approach for Designing Web Applications

Kyle Dent
kdent@seaglass.com
Columbia University

Steven M. Bellovin
smb@cs.columbia.edu
Columbia University

Technical Report CUCS-008-08

Abstract

Internet applications are being used for more and more important business and personal purposes. Despite efforts to lock down web servers and isolate databases, there is an inherent problem in the web application architecture that leaves databases necessarily exposed to possible attack from the Internet. We propose a new design that removes the web server as a trusted component of the architecture and provides an extra layer of protection against database attacks. We have created a prototype system that demonstrates the feasibility of the new design.

1 Introduction

Web applications are pervasive on the Internet. They handle many kinds of transactions and provide important functions to people and business. There are many different approaches to implement a web application, but a typical application employs the now classic 3-tier architecture. A user directs a client, normally a browser, to a web server. The web server passes the request to an application, which in turn makes use of a backend database both to access dynamic information and to store user information. In this scheme, the web server is the external contact to web clients. They are expected to be friendly, but the application is also available to any number of clients that may not be so friendly. The web server sits on the front line of a hostile network and is the most exposed piece in the architecture.

The database is the business-critical piece and typically contains client information as well as inventory and possibly other business-strategic data. It might also have sensitive information such as partner pricing and customer credit cards. A successful attack on the database can be a costly loss to business. A likely attack path to the database server is through the frontend web server which necessarily sits on the hostile network and must have a communication channel to the backend database in order for the application to function. This project proposes a new approach to secure the critical database portion of the web application architecture.

2 Motivation

To understand the issues involved in securing a web application more thoroughly, consider, for example, an ecommerce application selling books. Customers browse the catalog of books for sale, select one for purchase, and complete the financial transaction providing payment and shipping information to the bookstore.

- Browse the catalog
 1. the browser sends requests to the web server providing search criteria

2. the web server passes the search criteria to an application
 3. the application parses the search criteria, generates a query for the database
 4. the database receives the query, and returns a list of hits
 5. the application formats the list to display
 6. the web server sends the formatted list back to the client in reply to its original request
- Select for purchase
 1. the browser identifies a particular book from the database
 2. the web server passes the item id and possibly information about the customer to the application
 3. the application queries the database for customer and item information
 4. the application formats (possibly pre-filled) screens to capture customer data to complete the transaction
 5. the web server sends formatted screens to the browser
 - Complete the transaction
 1. the browser sends all the information necessary for purchasing the book, including information about the book, customer shipping information and customer financial information such as a credit card number
 2. the web server passes the purchasing and financial data to the application
 3. the application, after validating financial information, inserts data for the transaction and deducts the purchased items from inventory
 4. the application formats an appropriate response
 5. the web server sends the formatted response to the browser

Notice from the descriptions that clients send their requests through the web server, but ultimately their business is with the database servers. Applications run on the web server itself or on dedicated application servers, as in Sun Microsystem's J2EE or Microsoft's .NET frameworks. Likewise, the database server software could run on the same machine as the web server, although commonly databases run on their own dedicated boxes.

This point is worth stressing: while web site defacements should be avoided — if nothing else, they generate bad publicity — the fate of a business depends on the security of its databases. Even unauthorized read access can be expensive — California's data breach notification law¹ requires that individuals be informed if their personal data is exposed; write access can be far worse. Imagine the consequences for a bank if account data is modified. Protecting databases is far more important than protecting web sites that access them.

From another perspective, our goal is to contain the effects of bugs. Most security problems are due to buggy code; indeed, a National Academies study [16] found that 85% of CERT advisories issued up to that point described problems that could not be fixed with cryptography. Eliminating bugs from code seems difficult at best [2] and probably impossible. Given that, we must switch our focus to minimizing their effects on security. Small, simple programs are much more likely to be correct and secure [3]. The challenge, then, is this: can we redesign our web applications so that the security of the databases relies on *simple* programs, rather than on large, complex web servers and the applications invoked by them?

¹California Civil Code §1798.82 et seq.

3 Security Issues

Application developers and system administrators already take many precautions to protect systems. Moving databases into an isolated network helps to protect them from the outside. Preferably the database subnet is not reachable in any direct way from the hostile network. (The subnet is generally accessible from an internal corporate network to make administration and maintenance easier.) The web servers are given special access to the database layer for the application to function. Usually special access is provided through dedicated network interfaces between the database systems and the web or application systems such that the web server has both an outside interface for access from the Internet and an inside interface for access to the database. See Figure 1.



Figure 1: Isolating databases into separate subnets.

Isolating the database helps, but the web server is still exposed to the outside. A compromise of the web server can provide an attacker special access to the database.

3.1 Protecting a Web Server

Because of the web server's special position where it straddles a hostile network and an internal private one, it is important to ensure that it is well protected. The article "How we defaced www.apache.org" [9] is an interesting read and a cautionary tale that highlights the importance of carefully maintaining a web server. Below are some of the steps administrators employ to ensure the integrity of their web servers:

- **Limit services running on the web server system** Web servers should be dedicated to their tasks. To avoid the unexpected interplay of multiple services (such as described in [9]), limit web server systems to that single function. Likewise, if the web server allows for selecting modules at compilation or through configuration, limit your build or configuration to just the web server services you need. If your server is not executing CGI scripts, remove that function from the web server, for example.
- **Limit the access given to the web server process** If possible, compartmentalize the web server to limit its access to other parts of the system. If an attacker is able to compromise the server, he will have only limited access to other parts of the system. Unix provides the chroot system call and with some effort a web server can execute within a chrooted environment.
- **Carefully configure the web server and its environment** For example, use a dedicated account to run the web server process. On Unix disallow following symbolic file links from the web server and pay careful attention to file permissions. The web server account must be able to read the files but be sure that there is no chance of the account writing to them. Set the permissions on important files even more strictly. For example, a web server that starts with a privileged account can read its configuration and SSL [15, 6] key files before dropping privileges. Set the permissions on those fields so that the web server account cannot read them.

- **Keep the operating system and web server software up-to-date** Keep the latest patches applied on both the system and the web server software. As vulnerabilities are discovered in software packages, it's important to eliminate them with the latest fixes.
- **Limit network users' ability to execute scripts** Scripts executing in the context of the web server must never be able to run arbitrary commands on the system. Also CGI programs should be able to handle any kind of input (even malicious).

4 The Persistent Problem

Consider these the minimum steps required before exposing a web server to the Internet. But even with these and other precautions, there are several classes of attacks on the database that are still possible against the most tightly secured server.

Following the steps described would have stopped the attack on the `apache.org` site as described. But the database could still be vulnerable. In the first place, applications themselves are not immune to attack. Improper input validation and other careless coding practices can expose the application and network to attackers. If an application accepts input that goes into database queries, then SQL injection attacks can occur. See [4] and [5] for two examples of this type of attack. There are many more.

Locking down an application and plugging vulnerabilities one at a time as they are discovered is not an effective approach to securing a site. A piecemeal approach provides security only until the next accidental misconfiguration or zero-day attack. There have been attempts to protect database systems more systematically from injection-style attacks; see, for example, [1, 7] and many others. (SQL injection attacks have even reached popular culture; see <http://xkcd.com/327/>.) Whatever success they have provided, they are not meant to address the inherent problem that the web server is a trusted component of the overall architecture.

Note that built-in database access controls do not solve the problem. Today's web applications effectively run in "system high mode" [17], where the web server has full access to the database. More precisely, the login to the database is done by the web server, independent of which user's transactions it is processing. Thus, any action that could be performed on behalf of a legitimate user could be performed by the web server itself.

5 A New Approach

We have considered a new approach wherein we remove the web server itself as a trusted component of the system. The web server is the front line and exposed to the outside. Our approach was to devise a scheme that would provide no way for the web server to subvert the database server. In other words, even if an attacker has completely compromised the web server, he or she should gain no particular advantage for launching an attack on the database.

As noted earlier, in general a web application does not manage the database itself. Instead, it contacts a dedicated database server process via some sort of communications channel. The nature of this channel is crucial; if it is unconstrained, an attacker with control of the web server can send commands such as

```
select *
```

If the channel is strongly limited — in particular, if it does not accept such broad constructs — we can achieve a significant increase in database security.

Our model is Orwell's *Newspeak*, a language in which it was not possible to construct disloyal thoughts.

The purpose of Newspeak was not only to provide a medium of expression for the [proper] world-view ... but to make all other modes of thought impossible.

...

There would be many crimes and errors which it would be beyond [a person's] power to commit, simply because they were nameless and therefore unimaginable. [11]

For example, there was no word meaning “freedom”.

Is it possible to turn the idea of Newspeak into something beneficial to application security? Could access to the database occur only through a new “language” that doesn't allow an attack? Could there be a language that could remove unintended side-effects that open security holes? That idea provides the basis for our new approach.

5.1 Requirements

Roughly the requirements for a scheme to eliminate the web server as a trusted component are

1. Authentication is established from the web browser to the database server without the web server brokering the request. We could possibly allow different levels of authorization too but authorization is determined by the database.
2. When the browser sends sensitive data like credit card numbers and passwords to the database server, there is no opportunity for the web server to read the sensitive data.
3. No query coming through the web server can obtain information that it is not authorized to see.
4. No query from the web server can alter information that it is not authorized to change.

Even if an attacker is able to compromise the web server, it should be impossible to use it to attack the database. In other words there is no advantage in having full control of the web server.

6 Design

We are proposing a new module that sits between a web application and a database. To determine if such an approach is practical, we have created a prototype called Propylaeum. The name comes from Greek architecture and describes a monumental entrance to a sacred enclosure, the most famous of which is the Acropolis of Athens. Entering the Acropolis was permitted or denied at the Propylaeum. Our Propylaeum is an access control daemon designed to sit between a web application and the database server it uses for runtime data lookup and data persistence. The daemon determines what data operations are allowed based on its access control configuration. It also provides decryption of private values so that they are not accessible by any processes running on the web server. Web clients use client-side code to encrypt private values, such as passwords or credit card numbers. Data values that are needed by the web application for providing its function remain in plain text.

When a request is sent to the database server, the Propylaeum daemon determines the actions to perform. Query statements additionally go to a decryption module, which changes all of the encrypted strings into the correct plain text values.

The overall security scheme is composed of three main modules.

1. Client-side encryption

2. Database access control
3. Database decryption

Propylaeum thus provides three separate functions:

- It protects sensitive data from possibly-compromised web servers
- It provides sanitization of requests to the database engine
- It blocks direct communication from the web server to the database engine

The web application delivers to the client the JavaScript code and the Propylaeum public key that will be used to encrypt the private information. This makes up the first module of the scheme and sits on the web client. This module encrypts individual pieces of private data before submitting the contents of a form to the web application. A more sophisticated design could use S-HTTP [14, 13]; for our prototype, we used an ad hoc scheme, since no production web browsers implement S-HTTP and doing it in Javascript appeared infeasible.

Any fields that do not affect the application's function can be encrypted so that the application itself cannot read the values since it does not have the Propylaeum private key. The web application is not concerned with those values, it only needs to pass them to the database.

The web application connects only to the Propylaeum daemon. The database server must be configured not to accept connections directly from the web server. This can be accomplished either by binding it to the loopback interface or by configuring firewall rules on the server to block the web server from connecting to it. The web application makes its connection to the Propylaeum daemon, which in turn passes the request to the database server after it determines that the requested query should be carried out. Direct access to the database could be permitted from trusted systems on a different interface or through firewall rules.

7 Implementation

The Propylaeum proof-of-concept is written as a Perl daemon. It contains the connection information it needs to send queries to the database server. It also contains private and public keys for communicating with web clients.

7.1 Configuration

At start up it reads its configuration file. The configuration file is a simple XML structure that defines legal values for variables, named actions and then one or more database queries that should occur for each action. The DTD defining the XML is as follows:

```
<!ELEMENT propylaeum (variables*,action+)>

<!ELEMENT variables (allowed-value+)>
<!ELEMENT allowed-value EMPTY>
<!ATTLIST allowed-value varname CDATA #REQUIRED>
<!ATTLIST allowed-value regex CDATA #REQUIRED>

<!ELEMENT action (query+)>
<!ATTLIST action name CDATA #REQUIRED>

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

The first section defines query template variables, which will be used within query templates later in the file. Variable definitions are not required, but when they are used, a regular expression specifies the valid values permitted for the variable. For example, a field that should contain only digits can have the regular expression `[0-9]+`. Any attempt to include other characters causes the action to be rejected.

This field, of course, implements the sanitization function. Clearly, regular expressions cannot describe all possible legal inputs; still, they can describe very many. More importantly, *forcing* programmers to think about legal syntax improves the odds that they will do some sanitization. (To be sure, nothing will stop a lazy programmer from specifying `".*"` as legal.)

The second section of the file defines one or more actions. Each action can contain one or more queries to be executed when the action is called by the application. Figure 2 shows a sample configuration file.

When an application requests the action `BUY_BOOK`, the assigned SQL commands are executed. Note also that the SQL contains variable names enclosed in double brackets. When a web application makes a request for an action, it also provides a set of name-value pairs. Those values are used to expand the double-bracket variables when executing the queries. Additionally, during variable expansion, Propylaeum checks the provided value against the regular expression for that variable. If the value contains any characters disallowed by the regular expression, the request is rejected with an error message.

The actions provide the isolation layer between the web applications and the database. Note carefully the difference between these actions and SQL stored procedures. Stored procedures are a better way for applications to do database queries, in that they can largely eliminate SQL injection attacks. Our goal, though, is protecting the database against *any* web server compromise. We thus limit the web application so that it can talk only to Propylaeum, and further limit what Propylaeum should say to the database. We could use stored procedures here, too, as an additional layer of protection; the threat here is inadequate sanitization.

7.2 Application-to-Propylaeum Communications

The web application establishes a connection with the Propylaeum daemon. It specifies an action followed by a carriage return/linefeed (CRLF) combination followed by one or more lines of name-value pairs. The name-value pairs are used to expand the interpolated variables in the action definition in the Propylaeum configuration file.

The grammar for communications between a web application and the Propylaeum daemon is as follows:

```
REQUEST → ACTION PAIRS CRLF
PAIRS → PAIR | PAIR PAIRS
PAIR → NAME = VALUE CRLF
NAME → <any series of ASCII characters, except '='>
VALUE → <any series of ASCII characters>
```

The Propylaeum daemon replies with a single line that contains either the string `'OKAY'` or `'ERROR'`. The error reply is followed by a short message describing the problem.

An example of a Propylaeum request is shown in Figure 3. Notice that certain information has been encrypted by the client and is not available in plain text even to the web application. The `'BUY_BOOK'` action corresponds to the action described in the configuration file in Figure 2. Each of the actions in the file will be executed against the database using the values supplied with the request. Values that are in plain text are prefaced with the string `'PLAIN:'` to indicate that those should not be decrypted.

Figure 2: Example Propylaeum configuration file

```
<?xml version="1.0" ?>
<propylaeum>

<variables>
<allowed-value varname="ISBN" regex="[0-9-]">
<allowed-value varname="NAME" regex="[A-Za-z0-9-]">
<allowed-value varname="CC" regex="[0-9 ]">
</variables>

<action name="LIST_BOOKS">
<query>
SELECT ISBN, TITLE, AUTHOR FROM CATALOG
</query>
</action>

<action name="BOOK_DETAIL">
<query>
SELECT ISBN, TITLE, AUTHOR, IMAGEURL FROM CATALOG WHERE ISBN = '[[ISBN]]'
</query>
</action>

<action name="BUY_BOOK">
<query>
INSERT INTO BOOKORDER VALUES('[[ISBN]]', '[[NAME]]', '[[ADDRESS]]',
    '[[CC]]', NULL)
</query>
<query>
UPDATE INVENTORY SET INSTOCK = INSTOCK-1 WHERE ISBN = '[[ISBN]]'
</query>
</action>
</propylaeum>
```

Figure 3: Example Propylaeum request

```
BUY_BOOK
PLAIN:ISBN = 9780060589462
NAME = B4bN5ahVryxCrocj_YY50hdmhql1EjadfBD1jdsfe
ADDRESS = aK2FtRde0hVB4bN5ahVryxCrocjdmhqgyxLaLrLQh
CC = qyxLaLrLQh9DXPIaK2FtRde0hVJvLktB4bN5ahVry

OKAY
```

Figure 4: Example HTML file with crypto.js and a public key

```
<html>
<head>
<title>Book Details</title>
<script src="/crypto.js"></script>
<script>
var key = [76009799,192309627,43776032,73342735,24371]
var mod = [17]
function submit_form() {
  document.forms[0].name.value =
    rsaEncode(key,mod,document.forms[0].cc.name)
window.alert("submitting CC" + document.forms[0].cc.value)
return true
}
</script>
</head>
<body>
<h1>Book Details</h1>
...

```

7.3 Encryption

The encryption on the client side of our current implementation is accomplished using the JavaScript Crypto library [8]. The web application delivers the JavaScript and the Propylaeum public key through HTML pages. Figure 4 illustrates the beginning of an HTML file that includes the `crypto.js` library file and the public key as well as a JavaScript routine to encrypt the credit card number. As shown, a web application has only to include the code to pull in the library and a function that encrypts the sensitive form elements. Propylaeum contains a decryption module with the private key. This key is used to decrypt the protected values.

8 Evaluation

8.1 Security

Obviously we cannot say that the design described here is a foolproof security plan that will eliminate attacks on databases. In fact, a SQL injection attack is not impossible. However, the architecture does provide several advantages to provide protection and mitigate the consequences of an attack. Simplicity goes a long way towards improving security and Propylaeum enforces some structure and modularization of functions simplifying web applications. Any coding mistake that would allow an SQL injection attack, for example, would be limited to one section of the Propylaeum application. The key point is that developers do not have to write their own database access code, where each time they risk making a fatal mistake. Errors in web applications *cannot* result in SQL injection attacks nor in other direct database access.

A particular implementation will require a careful security review to ensure that the cryptography is done correctly, and that the code does not contain security flaws.

8.2 Ease of use

Once a Propylaeum service is installed, it actually makes applications development easier. Developers do not need to worry about database drivers and connections since those details are handled within Propylaeum. It also makes for a clean division of tasks where the database access routines can be written as configuration items that are separate from the application code. The application developers need only specify a simple action and all of the details are handled by Propylaeum. Similarly changes to the database are isolated to a single location making maintenance easier.

8.3 Performance

We have not measured the performance of even a quasi-production implementation of Propylaeum. Our server-side code is in Perl; our client-side code is in JavaScript. Furthermore, the key lengths used in our prototype are too small to be effective in a true security-sensitive application. The response time is not noticeably longer using Propylaeum. The algorithms from [8] encode a random session key, which is used to encrypt individual values. According to the site, encryption of a typical online shopping order occurs within 1 to 2 seconds using a 1024-bit key. Decrypting takes longer and may go as long as 5 to 30 seconds.

A better analysis can be done by assuming use of S-HTTP [14, 13]. S-HTTP is, in essence, a Cryptographic Message Syntax (CMS) encoding of encrypted text [10]. CMS uses traditional hybrid public/symmetric cryptography: a random symmetric traffic key is encrypted with a public key; the traffic key is then used to encrypt the actual data.

Because of the limited use of public keys in CMS, and because of the expense of using them, S-HTTP implements several optimizations. For one thing, key caching is possible. That is, parties communicating via S-HTTP can cache the result of a public key encryption or decryption, and reuse the traffic key without redoing the expensive public key operation. Furthermore, an S-HTTP server — in our case, the Propylaeum daemon, not the web application — could generate a new traffic key and send it to the client; this key can be used for subsequent messages, thus avoiding the expense of public key operations.

Similar optimizations are currently used for SSL. Because of that, we conclude that the performance of S-HTTP (and hence the performance of the cryptographic layer of Propylaeum) will be roughly equal to that of SSL, if implemented in production-quality code. If a separate SSL layer is not used, the CPU cost of the cryptography should be roughly equal.

There is likely to be more of a latency and cost penalty from the extra communications hop to reach the Propylaeum server. We expect that this will be modest. Indeed, extra hops are frequently used today, for load balancers, SSL accelerators [12], etc. We consider the cost modest, especially when balanced against the programming savings in the web application.

That said, some CPU savings are possible if applications make careful use of the encryption and leave non-private information in plain text. It is not clear that this is actually worthwhile. Applications could still run over SSL to protect from external eavesdropping and then encrypt only those values requiring the added protection all the way through to the database; this would provide protection against some forms of traffic analysis.

A production-quality Propylaeum daemon would also want to provide database connection pooling, so that new database connections are not required for every application request. There are other database access optimizations possible, including caching of common queries that can offset the cost of the additional security overhead.

9 A Variant Implementation

Propylaeum is not a traditional “defense in depth” scheme. To understand the difference, consider a variant implementation. Assume that there is no cryptography at the Propylaeum layer. Instead, all database transactions — that is, all requests from the client through the web server to the database — contain a per-client authenticator. If the web server or its applications are compromised, accounts that are active during that period are indeed at risk. In typical e-commerce sites, however, most accounts are not active most of the time. A portion of the database is at risk, but only a very small portion. The bulk of the database — that is, the vast bulk of the company’s assets — will remain intact.

In this version, the purpose of Propylaeum is to implement Newspeak. That is, it prevents all potentially-dangerous SQL queries. For example, there is no command that will list the entire database. Similarly, Propylaeum enforces the per-user authentication requirement, sometimes in non-obvious ways. Consider a web storefront. Obviously, access to customer records must be authenticated. Less obviously, purchase requests — adding an item to a shopping cart — must be authenticated. After all, web *servers* do not buy things; *users* do. The database operation that debits the stockroom by one item is thus authenticated. By contrast, such an operation could be a denial of service attack, if done maliciously by a compromised web server.

Similar design considerations can be applied to payment. Today, traditional web applications can perform billing by retrieving credit card numbers from a database and sending them to a payment server. In a Newspeak-based design, there is no verb for “give me a credit card number”. Instead, the web application would say “charge N zorkmids to credit card #3”. The Propylaeum daemon would do the card number retrieval and billing request, but only if the request were properly authenticated. The web application, though, could request a precis of the credit card (the last four digits, the card type, the user’s name for that card, etc.), that being a permitted verb in Newspeak.

10 Conclusions

Buggy web applications are and will remain a serious security threat. Indeed, buggy web servers are and will remain a problem. Several years ago, Microsoft’s Internet Information Server (IIS) had so many security holes that a major consulting company warned its clients not to use it. (That advisory was lifted a few years later.) Apache 2.2 has had at least 13 security advisories against it. The challenge for an organization is protecting what matters most: its databases.

Traditional approaches have not worked very well. In particular, the goal of bug-free code has (not surprisingly) proved infeasible. We assert that a better direction for web site security — and indeed, for security architecture in general — is to assume that certain components will be faulty. Accordingly, we should design systems that minimize the resulting damage.

Propylaeum and Newspeak are examples of this design paradigm. We assert that this is a better path to protecting what really matters.

References

- [1] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, volume 3089, pages 292–304. Notes in Computer Science, Springer-Verlag, 2004.
- [2] Frederick P. Brooks. *Mythical Man-Month*. Addison-Wesley, first edition, 1975.

- [3] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, first edition, 1994.
- [4] Security Focus. Demarc PureSecure authentication check SQL injection vulnerability. <http://www.securityfocus.com/bid/4520>.
- [5] Security Focus. LogiSense Hawk-i login SQL injection vulnerability.
- [6] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Netscape Communications Corporation, November 1996.
- [7] William G. J. Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.
- [8] John M. Hanna. JavaScript shopping & crypto. <http://shop-js.sourceforge.net/crypto2.htm>.
- [9] Hardbeat and {}. How we defaced www.apache.org. <http://www.dataloss.nl/papers/how.defaced.apache.org.txt>.
- [10] R. Housley. Cryptographic message syntax (CMS). RFC 3852, Internet Engineering Task Force, July 2004.
- [11] George Orwell. *Nineteen Eighty-four*. Harcourt Brace Jovanovich, Inc., 1949.
- [12] E. Rescorla, A. Cain, and B. Korver. SSLACC: A clustered SSL accelerator. In *Proceedings of the 11th USENIX Security Conference*, August 2002.
- [13] E. Rescorla and A. Schiffman. The secure HyperText transfer protocol. RFC 2660, Internet Engineering Task Force, August 1999.
- [14] E. Rescorla and A. Schiffman. Security extensions for HTML. RFC 2659, Internet Engineering Task Force, August 1999.
- [15] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.
- [16] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.
- [17] R. Shirey. Internet security glossary, version 2. RFC 4949, Internet Engineering Task Force, August 2007.