Simplifying Client-Server Application Development with Secure Reusable Components

Yngve Espelid, Lars-Helge Netland, Khalid Mughal, Kjell Jørgen Hole
Department of Informatics, University of Bergen
{yngvee,larshn,khalid,kjellh}@ii.uib.no

Abstract

Internet-related crime is increasing at a rapid rate. Attackers exploit weaknesses in the Web's underlying communication protocols, which were originally designed for a non-hostile environment. Meanwhile, society is facing a large deficit of security trained people that can remedy the situation. The development of secure software is simply too complex for most of today's IT professionals. In this paper, we present a communication component that reduces the complexity involved in engineering secure client-server applications, thereby enabling software practitioners to develop more secure systems. Specifically, we propose a solution based on a Public Key Infrastructure, that encapsulates communication, allowing programmers to focus on high-level application development issues.

1. Introduction

The production of secure software often involves ad-hoc development methods. Unlike the established field of software engineering, there do not exist well-established processes that can be applied when developing a secure software system [1]. Currently, developers rely on best practices and the experience of individual team members more than on structured procedures when it comes to secure programming. In addition, the development of secure software necessitates a mindset that encompasses the eminent threat of powerful adversaries. Web-application programmers have discovered the power of the dark side the hard way. Recently, a number of vulnerabilities have surfaced, all related to the same problem: improper *input validation*. The problem and solutions are discussed in detail in *Innocent Code—A Security Wake-Up Call for Web Programmers* [2].

The design and implementation of security-critical applications lie in the intersection between cryptography, computer security, and software engineering, requiring development teams in this segment to have strong skills within all areas. The coupling of these three fields introduces complexity due to conflicting interests. An example is usability versus authentication mechanisms. Also, in developing client-server applications, the choice of thread models, network programming, and session management add to the complexity.

Presently, there is a shortage of people with security background and training [3]. As a consequence, a large number of unqualified technicians are responsible for developing secure software systems. The situation in today's software industry bears much resemblance to that of the British banking industry in the 80's and early 90's, where implementations by untrained personnel were the cause for the majority of security breaches [4].

Due to the complexities involved in designing secure systems, application developers and end users cannot be expected to fully understand a security-critical program's inner workings. They will need to *trust* software provided by third parties. Designers of secure components face many challenges when it comes to building trustworthy systems. Trust is influenced by such diverse properties as a person's inclination to trust, ease of use, and availability of informational content [5]. For secure programming to succeed, security professionals must find ways to gain trust from consumers.

Client-server applications are important in offering services throughout the Internet. Citizens of Canada can log into government web servers to renew passports, file tax returns, and change home addresses [6]. As more and more authorities move services online, the same basic security building blocks will be required. Software reuse can prevent people from having to re-invent the same security functionality. Moreover, a joint effort could lead to better quality software if it can attract the attention of security experts.

In this paper, we present a communication component addressing security needs in client-server application development. The problem of input validation is tackled through definition of regular expressions. Complexity is addressed through abstraction, reducing some of the tedious network programming to a matter of configuration. The shortage of security trained professionals in the indus-

try can be overcome through software reuse. Also, it is important that users of security software can trust the system. Key ingredients in earning user trust are openness and simplicity.

The rest of the paper is organized as follows: Section 2 outlines the current state of secure software engineering and describes characteristics for good quality code, Section 3 gives an overview of the architecture and discusses the component in detail, Section 4 shows how application developers can use the component to implement an HTTP server, Section 5 emphasizes the importance of trust, Section 6 presents related work, and Section 7 provides a summary of our approach.

2. Security engineering

The production of secure software is a relatively new research field. Initial texts defining the discipline include *Building Secure Software* [7] and *Security Engineering* [8]. Current trends emphasis the importance of considering and dealing with security throughout the Software Development Life Cycle (SDLC). Retrofitting security into an existing application has proved to be more costly and less successful than including it in all development stages. A mixture of best practices, security knowledge, and security tools can be applied at various steps in the SDLC to improve software security. In the article series "Building Security In" [9], security expert G. McGraw gives a detailed presentation of security touchpoints throughout the software engineering process.

A central issue for vendors and users of secure software is the definition of security. Network security people often associate security with systems such as firewalls and intrusion detection systems. Developers of web applications tend to think of the Secure Sockets Layer (SSL) protocol as a silver bullet for web security. However, security is not a feature. It is an emergent system property. Security is the sum of a set of non-functional goals, which include procedures for prevention, traceability, auditing, monitoring, privacy, confidentiality, anonymity, authentication, and integrity [7]. Experts are needed to perform an extensive security evaluation of software, but such professionals are a rare breed. An initiative to collect and catalog the knowledge of experienced practitioners has been launched [10], but it is too early to say whether or not it will be successful. Lacking established guidelines in security engineering, there is an urgent need for key players in the field to distinguish between good and bad software. This distinction can be very hard to make prior to deployment, as most bugs and flaws are discovered during usage. A few important characteristics that can help decide the quality of code include

Documentation An undocumented system has no value in a security setting. Documentation is a necessity in understanding what a system does and does not do. Without such an understanding people cannot trust the software.

Development process The SDLC used in developing the software must incorporate security activities such as creating abuse cases, establishing security requirements, risk analysis, external review, risk-based security tests, static analysis, and penetration testing [9].

Cryptography The design of new cryptographic algorithms should only be done by experts in cryptography. In choosing key sizes, recommendations from local authorities or "Selecting Cryptographic Key Sizes" [11] can be helpful.

History As the software is put into production, a recording of security related events such as system crashes, bug fixes, and security breaches give indications of the system's quality. Especially interesting is the software's ability to cope with these events, as it is an indicator of what to expect in the future.

3. The communication component

The communication component is intended to be utilized in the development of client-server applications. The goal is to properly secure transmissions of data between communicating peers. This is achieved through core security services offered by a Public Key Infrastructure (PKI) [12]. The *authentication service* identifies the participating entities and offers assurance of data origin, the *integrity service* ensures the receiver that the data has not been altered in transit and the *confidentiality service* assures the sender that only the intended receiver is able to read the transmitted data. The purpose of the communication component is to enable developers of secure software systems to use these services in conjunction with the applications' communication protocols.

3.1. Security infrastructure

Our communication component relies on a well-functioning PKI. In the following discussion it is assumed that such an infrastructure is in place, and that it is successfully managing certificates and keys for users of the component. Both server and client side authentication will be employed. More information on PKIs can be found in *Understanding PKI* [12]. In building and managing a PKI, the OpenSSL project [13] and The Legion of the Bouncy Castle [14] can be very helpful.

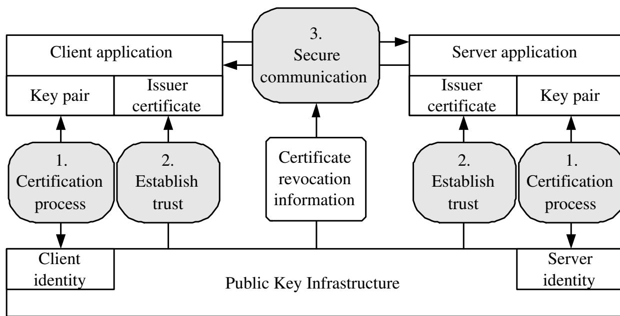

Figure 1. PKI relations

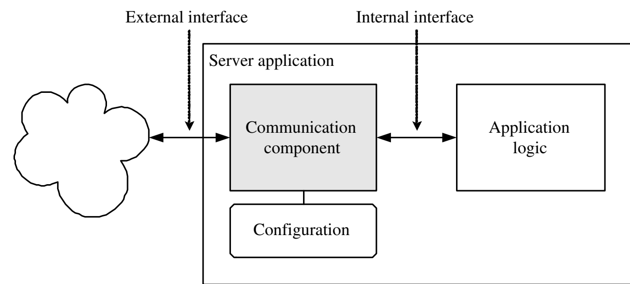

Figure 2. Server application

Figure 1 illustrates the necessary interactions between the communicating peers and the security infrastructure, needed to ensure secure communication. The certification process (1) binds a key pair to the entity requesting certification. The binding is represented by a signed certificate holding the entity's name and public key. These credentials can be confirmed by the entity's possession of the corresponding private key.

Before communicating, the validity of the peers' certificates must be verified. The verification is based on the common trust in the certificate issuer. In order to verify certificate signatures, the entities must obtain the issuer's certificate (2). In addition, the expiration and revocation information must be checked prior to establishing secure communication (3).

3.2. Utilizing the component

Figure 2 shows a typical context in which the communication component is deployed. The component reflects the fundamental concept of secure network communication in client-server applications. In a software engineering context, such reusable components could increase system reliability, lower development costs and reduce process risk [15]. Sensitive information is often communicated over the Internet and information assurance is an essential goal in such situations. The Internet is a hostile environment and there is a risk that sensitive information traversing the network is eavesdropped, recorded, modified or replaced. In a software development scenario, developers may choose to use the communication component to facilitate secure communication. Developers using the communication component are relieved of programming network communication, and can focus on the application logic and configuration of the component.

3.2.1. Input validation. Consider a scenario where the goal is to develop a server application, as depicted in Figure 2. The external interface comprises the port to the service and the application's communication protocol. When servicing the port, the application introduces an entry point for external input and the issue of trust in client input arises. The majority of literature devoted to the challenge of developing secure software includes discussions and guidelines on this topic [2, 16, 17]. Software with too much trust in client input gives rise to a large amount of today's software vulnerabilities. The common principle is to consider all input from clients guilty of malicious intent until proven otherwise. The communication component forces developers to define regular expressions used to validate client input, a practice known as *white listing*. Section 4 describes in depth the use of regular expressions for this purpose.

3.2.2. Control model. The communication component fits in a centralized control model where the application logic has overall responsibility for initializing, starting and stopping the component. Figure 3 illustrates this centralized control. The application logic passes a configuration file to a factory responsible for creating a *communicating peer*. The communicating peer is then requested to start executing the communication logic, for instance servicing a port. The component is an independent executable entity which is executed in parallel to the application logic. This control model makes it easier to develop applications following the principle of graceful degradation [18]. The application logic can simply stop the communication component and try to recover if an error occurs.

3.3. Managing complexity

The concepts presented so far do not require a specific programming language. In the following section, Java will be used to implement the communication component. In a security context, Java is an attractive choice because of features such as type safety, array bounds checking, and memory management. The widely deployed SSL protocol is a natural choice in protecting network data.

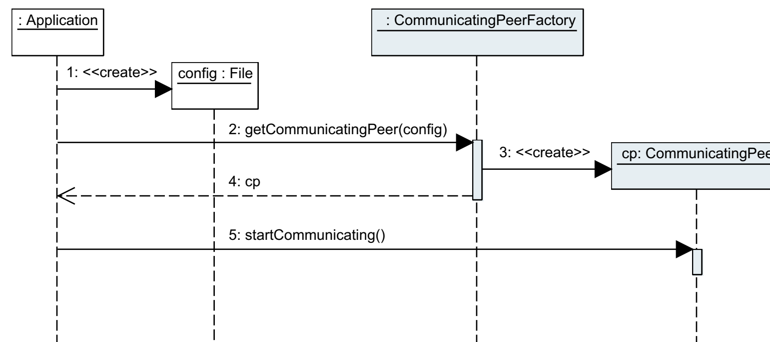

Figure 3. Centralized control

3.3.1. Advanced Java API. In version 5.0 of the Java 2 Standard Edition (J2SE) Platform, developers are given the possibility to combine SSL and enhanced I/O functionality. Prior to the latest release of Java, SSL and the New I/O (NIO) API could not be used in conjunction. The SSL API required a stream-based socket transport mechanism. In version 5.0, a new abstraction is introduced, giving programmers the freedom to choose any suitable transport mechanism. In addition, the API does not dictate a specific threading model, allowing developers to choose freely among different approaches. On the downside, the new flexibility comes with a price of added complexity. Setting up an SSL connection using the old stream-based approach was easy. First, an `SSLContext` had to be initialized with trusted certificates, the end user's own certificate, and the corresponding private key. The next and last step was simply to create an SSL client or server from the `SSLContext`. In total, setting up the connection called for about 10 lines of code on both sides of the communication. In addition, very little knowledge of the SSL protocol was needed to set up the secure channel. The only part developers had to take care of was providing the underlying PKI material, meaning the certificates to be used as part of the protocol.

Setting up SSL connections in J2SE version 5.0 necessitates a more thorough understanding of I/O, threading models and the SSL protocol. In terms of I/O, programmers must choose a specific transport model. Key ingredients in getting the I/O right include `Selectors`, `Channels` and various `Buffers`. Threading issues can be addressed using the `java.util.concurrent` package. Handling SSL requires an understanding of the protocol. In particular, developers must know the SSL Handshake Protocol. The actual processing of SSL data is taken care of by an `SSLEngine`. This engine is essentially a state machine, keeping track of the current stage in the SSL protocol. The programmer's responsibility is to send SSL data to and from the engine. The bottom-line is that in order to set up an SSL connection in the new paradigm, a lot more effort is required. It is not surprising that Sun describes the new library as an advanced API [19]. On top of the intricacies of setting up the communication, the code for the new setup entails a greater chance of making programming errors.

3.3.2. Hiding complexity in configuration. The complexity of implementing secure network communication is reduced to a trivial configuration task when utilizing the communication component. The most obvious configurations are whether the communicating peer should operate in server or client mode, and which port the peer should service or connect to. In addition, developers can configure the threading model handling communication, either choosing a single thread, a fixed number of threads or an extensible pool of threads. Also, the component's transport mechanism can be specified, leaving it up to developers to choose from the old stream-based model or the NIO model. To use the core security services, the location of key and trust material must be specified. The most challenging task when configuring the communication component is the construction and subsequent use of regular expressions applied in validating external input.

When implementing a server application, the task of handling a substantial number of client connections is non-trivial. The communication component provides a much simpler mental model in form of a packet queue of client actions, as illustrated in Figure 4. The component's communication engine continuously handles client interactions and produces packets representing the following events:

1. Acceptance of new client connections.
2. Completion of handshakes in the SSL Handshake Protocol.
3. Conformance of client requests to the communication protocol.

The packet hierarchy representing the above events is shown in Figure 5. The `Communicator` entity represents

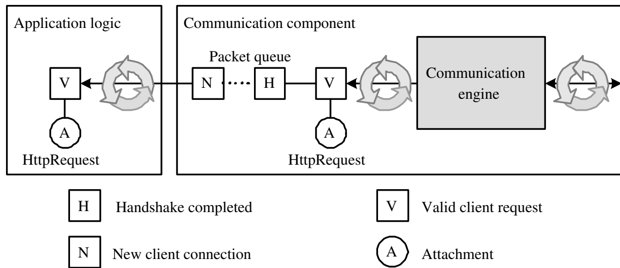

Figure 4. Packet queuing of client actions

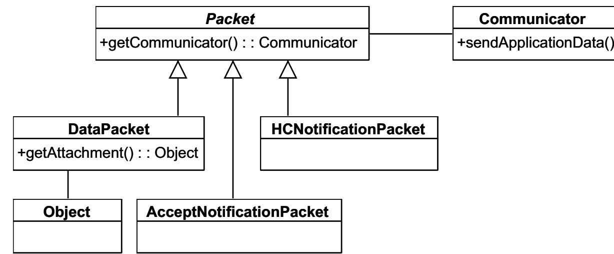

Figure 5. Packet structure

the state of a client connection. Each `Packet` holds a reference to the `Communicator` it originated from, thereby enabling the application logic to use the `Communicator` to send application data to the client. The `Packet` entity is a generalization of the three packet categories mentioned earlier. An `Object` attached to the `DataPacket` represents application data. The next section provides details on how the communication component is deployed in a concrete application.

4. Developing an HTTP server

Figure 6 illustrates that developers utilizing the communication component can focus on high-level development issues when implementing the application logic. Network communication programming is reduced to component configuration. This section illustrates these aspects in the context of implementing an HTTP server. The goal is to facilitate secure HTTP communication.

4.1. Component configuration

The task of configuring the communication component entails selecting appropriate options for the component's parameters. These options were mentioned in Section 3.3.2. We can formulate the following criteria for implementing the HTTP server:

- The application should run in server mode, servicing port 443.
- NIO should be used to facilitate scalable network communication. The HTTP server is expected to handle a large number of concurrent connections.
- An extensible pool of threads should be used to handle the communication logic. The hardware architecture on which the HTTP server is to be deployed has multiple processors, and a multi-threaded approach is expected to boost performance significantly.
- The Transport Layer Security (TLS) version of SSL and client authentication should be employed.
- Key and trust materials exist in Java Key Stores (JKSs), using SunX509 certificate encoding.

A crucial step in the configuration is to construct regular expressions representing the application's communication protocol, i.e. HTTP. In this simple scenario only a single regular expression is required

```
(?m)(GET)(.+\r\n)+(^\r\n)
```

The regular expression above matches HTTP requests using the GET method. The method line can be followed by zero or more header lines and a request is ended with an empty line. Only the structure of a request is validated by the regular expression. To facilitate validation of the content, appropriate application classes for wrapping matched character sequences must be implemented. The class names of such wrapper classes are mapped to the corresponding regular expressions. In this scenario, the `example.server.HttpRequest` class is mapped to the regular expression given above. Client input failing to match the regular expression is discarded. This white listing of client input prevents processing of bogus data.

4.2. Application logic

Configuration information discussed in Section 4.1 is provided in a configuration file, and developers need to implement application logic based on the centralized control model illustrated in Figure 3, to initialize and start a `CommunicatingPeer`. The application logic continuously services the packet queue provided by the `CommunicatingPeer`.

As illustrated in Figure 4, when the communication engine matches a character sequence with the given regular expression, it creates a `DataPacket` with an `HttpRequest` attached and adds it to the packet queue.

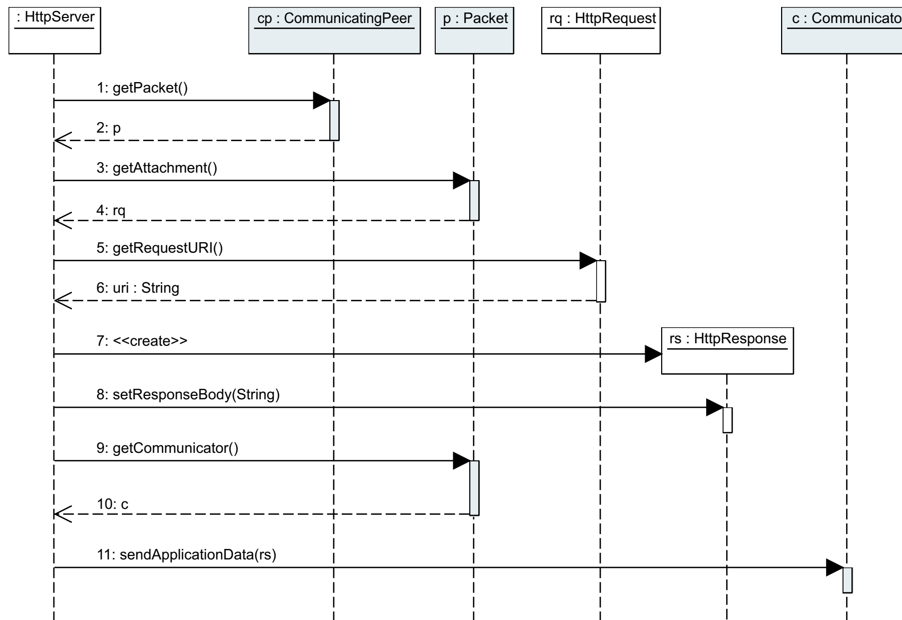

Figure 6. Handling an HTTP request

Figure 6 shows the sequence of actions in the server's application logic when handling such a request, where the numbers in the left-hand column below refer to the steps in the figure:

1–2	A packet, p, is retrieved from the packet queue.
3–4	The packet p is identified as a `DataPacket` and the packet's attachment is requested.
5–6	The attachment is identified as an `HttpRequest` (rq) and the content can be queried. In this sequence, the Unified Resource Identifier (URI) is retrieved to identify the resource targeted by the request.
7	Developers have implemented an `HttpResponse` class to facilitate the task of constructing responses to client requests. An object of this class is instantiated.
8	In case of a static resource targeted by the request, the content of the resource is set as the body of the `HttpResponse`.
9–10	To send this response back to the client, the application logic fetches the `Communicator` (c) representing the client connection.
11	The `Communicator` (c) facilitates the sending of the response to the client.

The application logic also retrieves packets representing new client connections or completion of the SSL Handshake Protocol. If no specific actions are planned for these events, these packets can safely be ignored. Otherwise, some application logic handling these packets must be implemented. For instance, when a client completes the SSL Handshake Protocol and the notification packet is retrieved, the application logic can initiate a client session. The name in the client's certificate can in such scenarios be used to distinguish client sessions.

5. Trust

The success of any piece of security software is closely coupled with its trustworthiness. If potential customers cannot trust your product, they will look for other alternatives. Trust is an active area of research. An overview of the current status quo can be found in *Security and Usability* [5]. Patrick defines software agent success in terms of trust and perceived risk [20]. Factors such as experience with a given product, a user's inclination to trust, system predictability, and interface design determine how much a person is willing to trust the software. On the opposing end, an increasing amount of risk will refrain users from using a software system. Risk is, among other factors, influenced by how well-documented a system is, how much personal information users have to divulge, and the degree of application autonomy.

Our approach must build trust towards two groups; de-

velopers and end users. The former is a direct relationship, where programmers must be satisfied with the level of security facilitated by the communication component. The latter is of indirect nature, an example of propagation of trust, where developers must extend their trust to potential customers. In this regard, our task is to make the transfer of trust possible. The authors believe that two factors are important in achieving trust:

Openness Comprehensive information about the project will be made available to mitigate feelings of risk. We strongly believe that documented source code alone is not sufficient documentation of a secure system. A detailed overview of the software development process and deliverables from activities throughout the SDLC should be made available to the involved parties. With better access to information, consumers should feel more comfortable in making trust decisions.

Simplicity Interface design and ease of use are addressed through a simple mental model that intuitively explains the workings of the component. Our packet queue model serves as an example of such simplification.

6. Related work

OpenSSL [13] is a very popular open-source library used to implement secure network communication. The approach presented in this paper could be used in developing a similar component based on the OpenSSL API. We do not know of any such initiative. From a security standpoint, the OpenSSL library is unattractive due to its incomplete documentation. In contrast, Sun's Java API is fully documented. The OpenSSL community is currently working on improving the project's documentation.

Unvalidated client input, also known as *tainted input*, has received much attention recently. Haldar et al. [21] describe an approach to tag and track tainted data throughout its lifetime in Java. The proposed framework reports cases where input is passed directly to security-sensitive methods without validation. It requires some instrumentation of core string classes in Java. Similar to our approach, developers are forced to think about sanitation of input. However, a tainted string will become untainted by a successful call, i.e. returning true, to any of its string checking or matching methods. This is regardless of the developers intended use. In contrast, the communication component presented in this paper forces application developers to write regular expressions and wrapper classes to validate the structure and content of network data. While the communication component is intended to be used in new development and re-factoring, the tainted-input approach is independent of source code, enabling use in already deployed software.

Probst et al. argue the need for reusable high-level security constructs [22]. They describe a framework called Generic Authorization Mechanisms for Multi-Tier Applications (GAMMA) that offers authentication, access control, and auditing mechanisms. GAMMA separates security functionality from business logic through dedicated security components, residing between the application and backend layers.

7. Summary

There is a shortage of professionals trained to build secure applications. The situation necessitates a transfer of knowledge from security professionals to software engineers. In particular, incorporating explicit security activities in the SDLC and an understanding of security best practices are important factors in building secure software. Our contribution is a client-server communication component that reduces the complexities of programming secure networking code to a configuration task. As an example, we have shown how an HTTP server can be implemented using the communication component. We intend to deploy the communication component in other client-server applications to further test the validity of our approach.

We thank the anonymous reviewers for their comments, which greatly improved this paper.

References

[1] J. Whittaker, "Why Secure Applications are Difficult to Write," *IEEE Security & Privacy*, vol. 1, no. 2, 2003, pp. 81–83.

[2] S.H. Huseby, *Innocent Code—A Security Wake-Up Call for Web Programmers*. John Wiley & Sons, first edition 2004.

[3] President's Information Technology Advisory Committee, "Cyber Security: A Crisis of Prioritization," February 2005.

[4] R. Anderson, "Why Cryptosystems Fail," ACM 1st Conference on Computer and Communication Security 1993.

[5] Editors L. Cranor and S. Garfinkel, *Security and Usability—Designing Secure Systems that People Can Use*. O'Reilly, first edition 2005.

[6] Government of Canada Official Web Site. Retrieved January 2006 from `http://canada.gc.ca/main_e.html`

[7] J. Viega and G. McGraw, *Building Secure Software—How to Avoid Security Problems the Right Way.* Addison-Wesley, first edition 2002.

[8] R. Anderson, *Security Engineering—A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., first edition 2001.

[9] G. McGraw, "Software Security," *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80–83.

[10] S. Barnum and G. McGraw, "Knowledge for Software Security," *IEEE Security & Privacy*, vol. 3, no. 2, 2005, pp. 74–78. Portal website:

https://buildingsecurityin.us-cert.gov/portal/

[11] A.K. Lenstra and E.R. Verheul, "Selecting Cryptographic Key Sizes," Practice and Theory in Public Key Cryptography, PKCS 2000.

[12] C. Adams and S. Lloyd, *Understanding PKI—Concepts, Standards, and Deployment Considerations*. Addison-Wesley, second edition 2003.

[13] OpenSSL. Retrieved September 2005 from http://www.openssl.org.

[14] The Legion of the Bouncy Castle. Retrieved September 2005 from http://www.bouncycastle.org.

[15] I. Sommerville, *Software Engineering*. Addison-Wesley, sixth edition 2001.

[16] M. Howard and D. LeBlanc, *Writing Secure Code—Practical Strategies and Techniques for Secure Application Coding in a Networked World*. Microsoft Press, second edition 2003.

[17] G. Hoglund and G. McGraw, *Exploiting Software—How to Break Code*. Addison-Wesley, first edition 2004.

[18] M.G. Graff and K.R. van Wyk, *Secure Coding—Principles & Practices*. O'Reilly, first edition 2003.

[19] Java Secure Socket Extension Reference Guide. Retrieved September 2005 from http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html

[20] A.S. Patrick, "Building Trustworthy Software Agents," *IEEE Internet Computing*, November-December 2002.

[21] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," 21st Annual Computer Security Applications Conference 2005.

[22] S. Probst, W. Essmayr, and E. Weippl, "Reusable Components for Developing Security-Aware Applications," 18th Annual Computer Security Applications Conference 2002.