Santa Clara University
DEPARTMENT of COMPUTER ENGINEERING

Date: February 21, 2003

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPervision BY

Jeff Dice

ENTITLED

The Java SSL applet

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

__________________________________________
THESIS ADVISOR

__________________________________________
DEPARTMENT CHAIR
The Java SSL Applet

by

Jeff Dice

SENIOR DESIGN PROJECT REPORT

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Engineering School of Engineering Santa Clara University

Santa Clara, California

February 21, 2003
The Java SSL Applet

Jeff Dice
Department of Computer Engineering
Santa Clara University

ABSTRACT

The purpose of the Java SSL applet senior design project is to demonstrate the validity of deploying communications protocols through the distributed mechanism of Java (and specifically Java applets). Various applications of this method include (but are not limited to) establishment of e-commerce transactions, virtual private networks, instant messaging sessions, `web services\(^1\)`, `grid services\(^2\)`, and RMI (exchanging data objects and methods).

The specific application chosen for implementation is the establishment of e-commerce transactions, the ability to establish a secure means of transmitting personally identifiable information between parties (consumer and merchant). An e-commerce system and a Java applet that transmits secure information to said e-commerce system are provided for within this project.

Keywords: SSL, cryptography, secure transmission, e-commerce, grid, Java applet

---

\(^1\) See [http://www.webservices.org](http://www.webservices.org)

\(^2\) See [http://www.globus.org](http://www.globus.org)
# Table of Contents

1. Introduction 5
2. The SSL protocol 7
3. The Java applet 14
4. The e-commerce system 18
5. Conclusions 21
6. Appendix A: SSL 3.0 draft standard 22
7. Appendix B: Java applet source 23
8. Appendix C: E-commerce source 27
Table of Figures

1. Overview of SSL transaction 7
2. Typical SSL message block 9
3. SSL Handshake 10
4. Generic Java Applet Structure 14
5. Java Applet Operation 16
6. The e-commerce system 18
7. The order submission process 19
1.0 Introduction

There is an evolving framework of distributed systems, from e-commerce systems to electronic business document interchange to applications (‘web services’). This framework of distributed systems will rely on the establishment of various interchange protocols\(^3\) to exchange information and methods. The mechanisms through which these protocols can be integrated into various systems is still under development. Some standard method for the flexible deployment of these protocols must be devised in order to support the interoperability of different systems.

One possible method for allowing the portable deployment of new protocols from one system to another for information exchange is the use of Java applets. Java itself provides a standard means of implementing custom communication protocols, and is predicated upon the design philosophy of ‘write once, run everywhere’ so that support for Java interpretation is built-in to most systems. Java applets are designed to allow for the delivery of custom code for execution from one system to another, directly addressing the need as outlined above for flexible on-demand deployment of protocol implementations between systems.

Take the example of an existing distributed system that relies on the establishment of an interchange protocol to exchange information, an e-commerce system. One hallmark of an e-commerce system is that it requires a secure, encrypted communication channel for transmitting personally identifiable information between two systems: in this case, the consumer and the merchant. Java applets are a mechanism that will provide for the ‘parent’ system to deploy a custom protocol on demand to the ‘child’ system in order to communicate.

Put another way, Java applets are one way of addressing the need for the evolving framework to deploy various protocols for information exchange as needed. This project will demonstrate this approach through the creation of a Java applet that will deploy a secure communications protocol in a typical e-commerce transaction system. Allowing for protocols to be deployed on the fly from one system to another instead of relying on hard-coded implementations within all parties participating in said system is the key to

\(^3\) Note: Use of ‘protocols’ in this document typically will refer to transport and/or application layer, not network layer or physical (e.g. IP/Ethernet)
achieving a flexible framework of the form described above. While an e-commerce system is chosen as the example for this project, the conclusions may be extended to cover all similar distributed systems. Present examples of these systems include (but are not limited to): instant messaging systems, web-based applications using RMI, `web services’, virtual private networks, etc.

The secure communications protocol that will be implemented through the Java applet will be the Secure Sockets Layer, or SSL. The primary cryptographic methods used in e-commerce today can be broken down into four types: authenticating the identity of individual parties involved in a given transaction, encrypting the data used in the transaction, ensuring that data used in a transaction is not tampered with, and the underlying (or perhaps overlaying) protocols used to manage the transaction as a whole. Authentication is typically done through the use of certificates and asymmetric (or public key) encryption (e.g. RSA). Encryption of `routine’ data is usually carried out by the faster symmetric encryption (e.g. RC4). Data packets are typically run through a one-way hash function to compute a message integrity check, which may be re-computed by other parties to ensure that the data received matches the data sent. SSL provides for all of the methods above and is the most widely used e-commerce protocol for secure transactions.

In order to accomplish this project, the following objectives must be achieved:

1. Understand the SSL protocol
2. The creation of a Java applet which uses the SSL protocol
3. The creation of an e-commerce system which will deploy the SSL protocol between systems and utilize that protocol for communication
2.0 The SSL protocol

The Secure Sockets Layer (SSL) protocol is a security protocol designed to allow client/server applications to communicate in such a way as to prevent eavesdropping, message forgery or tampering. When a network application desires a secured connection to another network application running on a different machine, it can invoke the SSL layer to provide that connection. In essence, for secured connections SSL acts as an interface between the network application and the underlying transport protocol (typically the Transport Control Protocol, or TCP). The network application forms the data it wishes to transmit to the other machine as it normally would, but in the case of a secured session that data is handed to the SSL layer first for encryption, and then passed on to the transport layer for transmission rather than simply handed straight to the transport layer.
SSL is a two-level protocol, consisting at the lower level of the Record Protocol, and at the upper level of various high-level protocols, which are encapsulated by the Record Protocol. The high-level protocols are used to establish SSL session parameters, negotiate settings between machines, and in general deal with the larger issues, rather than with individual packets. The Record Protocol performs the “grunt” work, compressing, encrypting, and otherwise packaging the data from the network application as per the settings defined by the high-level protocols for the transport layer. The data passed down from the application passes through three transitions once it reaches the Record Protocol level. The data is fractured into SSLPlaintext records of size $2^{14}$ bytes, compressed into an SSLCompressed record, encrypted and hashed against message tampering in an SSLCiphertext fragment and then transmitted through either a stream or a block cipher\(^4\). The data is encrypted with a symmetric encryption algorithm as specified in the SSL handshake. The keys used to encrypt data are computed from the master secret; the algorithm(s) used can be found in Appendix A, section 6.2.2. The encrypted data is then run through a hash function (MD5 or SHA) to provide a MAC in order to provide a message-tampering check for the receiving party.

The general form of an SSL message\(^5\) contains information regarding the version of SSL being used, the high-level protocol in question, length of the message, the data, etc.

---

\(^4\) More information on the Record Protocol may be obtained from Appendix A, section 5.2

\(^5\) From the SSL 3.0 draft
**Content Type**: Specifies which high-level protocol (handshake, alert, application data, change cipher spec) is being used.

**Protocol Version**: Defines which version of the SSL protocol is being used.

**Length**: The length (in bytes) of the data in the record block.

**Data**: Contains information or data of the length specified above. The type of data is defined by the content type field, and is typically compressed, encrypted, etc.

**Figure 2.** Typical SSL message block.

There are four different high-level SSL protocols: change cipher spec, alert, application data, and handshake. The change cipher spec protocol provides a means to signal changes in the encryption scheme(s) being used; after negotiating a change in the encryption scheme via the handshake protocol, a change cipher spec message is sent to mark the beginning of the new encryption scheme’s use. The alert protocol acts as the ‘error control’, providing a way for problems in an SSL connection to be communicated to all parties involved. Upon detecting a problem, an SSL client/server may transmit any of the following trouble codes\(^6\) in an alert message: unexpected message, bad record mac, decompression failure, handshake failure, no certificate, bad certificate, unsupported certificate, certificate revoked, certificate expired, certificate unknown. The alert protocol also serves as the means to end an SSL session by sending a close notify message to the other party. The application data protocol is simply the routine passing of application data to the Record Protocol, for packaging and transmission to the transport

\(^6\) A further explanation of the alert protocol and the trouble codes may be found in Appendix A
layer. While the change cipher spec, alert and application data protocols provide crucial capabilities to SSL, the “meat” of SSL is found in the handshake protocol.

![SSL Handshake diagram]

**Figure 3.** SSL Handshake.

The handshake protocol is used to begin an SSL session between a client and a server, to establish cryptographic parameters, to authenticate each other, etc. In essence, an SSL handshake is an introduction between two parties and the establishment of the ground rules of a conversation between those two parties. The rest of the conversation is simply the transmission of the encrypted application data via the application data protocol and Record protocol, and is closed by an alert message with the code for close notify contained therein. All of the groundwork to establish a connection is laid with the handshake protocol.
If a client wishes to initiate an SSL connection with a secured server, it sends a ClientHello message to the server at a specific port (for HTTP the port is typically 443). This message identifies which SSL version the client is using, a session identifier, a list of cipher suites the client supports in order of preference, a list of compression algorithms the client supports and a random number which is used later in the handshake. If the session ID field is empty, this indicates that the client either wishes to resume a previous SSL session or that the client wishes to renegotiate the security parameters of an existing SSL session. After receiving this message, the server must respond with a ServerHello message, or a handshake failure alert message will be sent. The ServerHello message responds with:

- A suggested SSL version to use, based on the highest supported by both the client and the server
- A random number different and independent from the one sent by the client
- A session ID
- A cipher suite selected from the ones supported by the client
- A compression algorithm selected from the ones supported by the client

A cipher suite in SSL defines which asymmetric, symmetric, and hash functions are used to authenticate identities, encrypt SSL packages, and provide message integrity checks, respectively. Cipher suites provide a framework into which new encryption methods may be incorporated without requiring a re-definition of the SSL protocol. In general, cipher suites may be broken into two classes: those requiring authentication of at least one party (and possibly both) through the use of public keys and certificates, and those not requiring authentication of either party. While the latter class is provided by the SSL specification, it allows both parties to remain completely anonymous and vulnerable to man-in-the-middle\(^7\) attacks, and is strongly discouraged. When a cipher suite with public key verification is used, the server is required to authenticate itself through the use of either a certificate or FORTEZZA KEA\(^8\) key exchange. I will only implement a few of the cipher suites supported by the full SSL specification.

---

\(^7\) See glossary under “man-in-the-middle”
\(^8\) See Appendix A, section 5.6.7.2, and Appendix D, section D.5
If the server has a certificate, then after generating the ServerHello message the server sends a ServerCertificate message; if the server does not have a certificate, has a certificate used only for signing, or wishes to use FORTEZZA KEA key exchange, then the server sends a ServerKeyExchange message instead. In the case of the ServerCertificate message, the message simply contains the server’s certificate (typically an X.509.v3\textsuperscript{9} certificate. In the case of a ServerKeyExchange message, the message provides all the necessary information to generate a pair of keys underneath the RSA, Diffie-Hellman, or FORTEZZA public key algorithms. This message is used to provide the client with a public key for the server that can be used to encrypt messages that only the server may decrypt.

Provided that the server authenticated itself, the server may then request that the client prove its own identity. The CertificateRequest message sent by the server in this case provides a list of certificates and certificate authorities (CA) which the server supports. The server then concludes its portion of the hello phase of the handshake by sending a ServerHelloDone message.

Once the client has received a ServerHelloDone message, it verifies that the server’s certificate is valid (if not, the appropriate alert messages are generated) and checks the hello parameters provided by the server. If the server generated a CertificateRequest message during the hello phase, the client must respond with either its certificate in a ClientCertificate message or an alert message indicating that it has no certificate. The server may either accept the client’s certificate or lack of same, or if something is not acceptable it will generate the appropriate alert message. If a public key algorithm was selected by the server in its ServerKeyExchange message, the client uses that algorithm when generating its own ClientKeyExchange message. In the ClientKeyExchange message, the client provides a random 48-byte number, which is designated as the pre-master secret data and is used to generate the master secret data\textsuperscript{10}. The FORTEZZA KEA algorithm demands a more complicated message generation, and may be reviewed in Appendix A in section 5.6.7.2. The pre-master secret is encrypted

---

\textsuperscript{9} The X.509 protocol is part of the ISO authentication framework; more information can be found at http://www.mcg.org.br

\textsuperscript{10} Most symmetric encryption algorithms and hash functions rely on the availability of some shared ‘secret data’ in order to generate keys or compute message authentication codes (MAC), respectively.
using the server’s public key as provided in the ServerKeyExchange message. Upon receipt of the ClientCertificate message, the server verifies the client’s certificate and, if the client certificate has signing capability\(^{11}\) responds with a CertificateVerify message. After the hello and authentication phases have been completed, both parties are satisfied with the other’s identity and the ciphersuite selected. The client sends a change cipher spec message to indicate that it has accepted the parameters as negotiated and is setting them as the parameters to be used in any following transactions. The client then sends a ClientFinished message which contains both an MD5 and SHA hash of certain data\(^{12}\) negotiated during the hello and authentication phases so that the server can also compute the hashes and verify that both the client and server have the same data. The server then generates its own change cipher spec and ServerFinished messages containing approximately the same data as the client messages. The SSL handshake is then completed, all encryption parameters have been computed\(^{13}\) and accepted, and the client and server begin trading data via the application data protocol. Application data messages contain data generated by the applications which are using SSL to communicate with each other, and are encrypted by the Record Protocol as discussed earlier.

---

\(^{11}\) All certificates except those which contain fixed Diffie-Hellman parameters

\(^{12}\) See Appendix A, section 5.6.9

\(^{13}\) The master secret is computed by both parties after the ClientKeyExchange message, according to an algorithm provided in Appendix A, section 6.1
3.0 The Java Applet

Providing for the on-demand deployment of various communication protocols via Java applets requires a brief introduction to Java itself. Specifically, discussing the nature of communication protocol implementation in Java and how that translates into the ability to incorporate within an applet. A brief exploration of the mechanism involved in the SSL implementation will also be provided. The structure of the Java applet incorporating SSL is illustrated below.

![Figure 4. Generic Java Applet Structure](image)

This layer represents the logic within the main applet routine(s). Creating the connection objects, sending/receiving data, passing to application.

This layer is present dependent on the protocol spec. Some protocols incorporate a `session` layer to enable opening multiple physical connections at the network layer.

This layer represents the `meat` of the protocol implementation. It defines how a connection is opened to allow communication, how to format data and transmit/receive data, etc.

**Figure 4. Generic Java Applet Structure**

The Java programming language provides a standard communication protocol implementation, specifically TCP/IP (Transmission Control Protocol/Internet Protocol). This implementation is provided as a set of Socket classes that accomplish the task of opening a communications link between different systems, and permitting the exchange of information. Java also provides the ability to override or otherwise extend these `socket` classes in order to develop additional communication protocols. This is one of the prime reasons that Java (and by extension Java applets) is suited to serve for the on-demand deployment of various communication protocols between distributed systems. Java is a universal distribution environment from the standpoint that Java is a `write-once, run anywhere` development and execution environment. Protocols implemented within Java may be easily transported to different distributed systems.

The latest version of the Java development/execution environment from Sun Microsystems (1.4¹⁴) provides an example implementation of a custom communications protocol, specifically the Secure Sockets Layer. This is provided through an extension to
the Java 1.4 environment called the Java Secure Sockets Extension\textsuperscript{15}. The JSSE provides a set of classes known as SSLSocket classes that extend the standard Socket classes to cover the cryptographic aspects of the SSL protocol. This serves as an illustration of the means through which various communications protocols may be implemented within Java. The Java applet developed in the course of this project utilizes the SSL implementation provided through the JSSE.

The function of the Java applet performs as follows: The client will request a secure web page from a non-secure portion of a merchant’s web site. Upon receiving this request, the server will send the web applet to the client web browser. The applet will execute, establish an SSL session with the secure server, and send and receive data (web pages, forms, etc) through the secure SSL session until the secure transaction has completed. After the SSL session has closed, the applet will close any or all network sockets it has created along with any other structures, and end operations. At that point, the client and server will resume their normal, unsecured transactions.

\textsuperscript{14} http://java.sun.com/j2se/1.4/
\textsuperscript{15} http://java.sun.com/products/jsse/index-1.4.html
The key criteria to establish during development of the Java applet were transparency to the user, on-demand deployment of the communication protocol (SSL) to the client system from the server system, and the accurate exchange of information between systems. Transparency to the user calls for all of the Java applet’s operations to take place in the background without presenting a UI to the user to establish an SSL connection. On-demand deployment of the SSL protocol from the server to the client dictates that the connection is only established as a result of specific user action. The accurate exchange of information is validated by the successful submission of an order from the client to the server.

To achieve transparency to the user, the Java applet is loaded into the background of the order page. It remains dormant in memory until such time as the user takes an action, such as clicking on the submit button on the order page, that dictates a connection.
needs to be opened to the server. The data from the order form is then passed to the Java applet, the applet opens a connection to the server and submits the data to the server, reads the response from the server, and then closes the connection. Navigating away from the order page after submission of the order destroys the context in which the applet lives, and the applet exits gracefully from memory.
4.0 The e-commerce system

The creation of a typical e-commerce system is necessary in order to demonstrate the mechanism by which the Java applet will deploy on-demand the communication protocol (SSL). The e-commerce system simulates the functionality of a typical merchant store: the ability to retrieve a list of items for sale, add/remove items to a shopping cart, and placing an order for the items in the shopping cart. This last function, the ability to submit an order to the merchant system, is accomplished by calling the Java applet to establish a secure communication channel on-demand.

Figure 6. The e-commerce system

The e-commerce system consists, in the most basic description, of a series of HTML pages that are connected to ASP (Active Server Pages) pages. ASP is a server-side technology from Microsoft that allows for the processing of dynamic pages through various functions embedded in web page source code. In the case of the e-commerce system, a user selects the catalog page to see a list of items for sale. The catalog page ASP page retrieves a set of items for sale from the database (Microsoft Access). As part of constructing the page that lists the items for sale, the catalog ASP page embeds a set of links that allow the user to add individual items to the shopping cart. If a user clicks on
an add item link, the add item ASP page is called with a parameter of the item number. The add item ASP page runs a SQL query to place the item number in the 'shopping cart', which is a table within the Access database. The user is then taken to the view shopping cart ASP page, which retrieves all item numbers (and the item descriptions) from the shopping cart table. If the user chooses to place an order, they are then taken into the order ASP page, which leads into the submission process that involves the Java applet deploying the SSL protocol on-demand.

![Diagram of the order submission process]

**Figure 7. The order submission process**

The order page consists of three key elements which are involved in the successful on-demand deployment of the SSL protocol for the exchange of information between the two distributed systems (client and server). The first element, the document page, consists of an HTML form that comprises/contains the data entered by the user that must be submitted to the server. The second element, the Javascript, serves as the glue that connects together the HTML form with the Java applet deployed to establish an SSL connection with the server. The third element, the Java applet, will establish an SSL connection with the server and exchange information between the client and the server.

The HTML form is populated with the appropriate information by the user, consisting of personally identifiable information such as name, credit card number, etc.
When the user hits the Submit button on the form, the Javascript is called and passed as a parameter the data entered by the user into the form. The Javascript takes the data, formats it, and then calls the applet, passing along the formatted data. When the Java applet is called by the Javascript it establishes an SSL connection with the server, submits the formatted user data via POST, reads any server response, and then closes the connection to the server. This process ensures the successful on-demand deployment of a communication protocol (SSL) to allow for the exchange of information between distributed systems.
5.0 Conclusions

This project established the validity of using Java (and specifically Java applets) as a means of providing for the on-demand deployment of various communication protocols between distributed systems to allow for the exchange of information and data. The e-commerce system and SSL Java applet as constructed meet the key criteria outlined, namely the ability to transparently establish a communications channel between two systems on the fly through a protocol implementation which is deployed from one system to the other. This addresses the core objective of demonstrating a valid way of meeting the needs of the evolving distributed framework for a mechanism allowing the flexible deployment of various communication protocols without requiring all systems involved to implement all communication protocols.

The Java applet solution to the issue of flexible deployment of communication protocols between distributed systems is currently constrained from the standpoint that Java applets are largely a browser-based technology. This is not inherently limited, however the current view of the industry is that Java applets are only to be deployed on the client-side through web browsers, which limits the utility insofar as server-to-server communication is concerned. There are other similar mechanisms within Java that provide the same functionality and in largely the same manner, such as EJBs and portable servlets. Applets are easier to deploy and are able to serve both browser-based clients as well as server-to-server systems, however the model currently employed for Java objects suggests that extending beyond browser-based clients is problematic.

Extending the use of Java and its various mechanisms (applets, servlets, EJBs) to address the deployment of communication protocols between distributed systems is ongoing. There are several initiatives underway to examine the most appropriate solution that involves Java as the method of choice. Among these is the Java for OGSI initiative, or the Java for Open Grid Standard Infrastructure16.

---

16 http://www.globus.org/ogsa/java/index.html
Appendix A

The SSL 3.0 draft specification document is available at http://sitesearch.netscape.com/eng/ssl3/.
 Appendix B

import java.applet.*;
import java.net.*;
import java.io.*;
import javax.net.ssl.*;

/* The purpose of this applet is to enable the secure transfer
of data between a web browser and a server through the use of
the Java 1.4 SSL implementation. Specifically, passing
confidential user information from a form to a back-end database
in as secure a manner as possible. */

public class SecureTransfer extends Applet
{

    /* Declare the host and port to open a socket to */
    private String host = "127.0.0.1";
    private int port = 443;

    /* Declare the request string that will be passed from
    the form to the applet, and from the applet via SSL
to the processor on the server */
    private String requeststring;

    /* Initialization routine for the applet */
    public void init()
    {
        try
{ 
    System.out.println("Applet initialized correctly 
"); 
}

try 
{
    /* Create an SSL socket factory */
    SSLSocketFactory sslFact = (SSLSocketFactory)SSLSocketFactory.getDefault();

    /* Create an SSL socket (client) */
    SSLSocket socket = (SSLSocket)sslFact.createSocket(host, port);

    /* Initiate the SSL Handshake that begins the SSL session.
       This is the process that initiates the exchange of
       public keys for verifying identity, establishing a session
       encryption key, etc */
socket.startHandshake();

System.out.println("Connected to: "+ host + 
"\n");

/* Create a buffered output stream */
PrintWriter out = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            socket.getOutputStream())));

/* Create a buffered input stream */
BufferedReader in = new BufferedReader(
    new InputStreamReader(
        socket.getInputStream()));

/* Send HTTP messages to initiate a POST of the confidential user data to the server */

out.println("POST /SSLProjectSecure/Enter_Order.asp HTTP/1.0");
    out.println("Content-type: application/x-www-form-urlencoded");
    out.println("Content-length: " + requeststring.length());
    out.println(requeststring);
    out.println();
    out.flush();

/* Read any relevant response from the server. Currently driven to the Java console. */

String inputLine;
String response = "";
while ((inputLine = in.readLine()) != null)
    response = inputLine;

    /* Data exchange between the applet and server is complete.
     Shut down buffered in/out streams, and the SSL socket. */

    in.close();
    out.close();
    socket.close();

}  
catch (Exception e)  
{  
    System.out.println (e);
}  
}  
}  
}
Appendix C