

ABSTRACT

TIM LOWMAN. Secure Computer Applications in an Enterprise Environment. (Under the direction of Dr. Vicki Jones.)

Sophisticated computing environments support many of the complex tasks which arise in modern enterprises. An enterprise environment is a collective of the organization's software, hardware, networking, and data systems. Typically, many user workstations communicate with shared servers, balancing computer processing throughout the organization. In a "secure" modern enterprise issues of authentication, private communication, and protected, shared data space must be addressed. In this thesis we present a general model for adding security to the currently popular enterprise architecture: the World Wide Web (WWW).

The results of our investigation into adding security to the general WWW architecture are reported in this document. We focus on authenticating users (Kerberos), establishing a secure communication link for private data exchange (SSL), protected space to store shared data (AFS filesystem), and an enhanced server (Apache) to integrate these components. After presenting our secure model, we describe a prototype application, built using our approach, which addresses a common problem of secure online submission of homework assignments in a university environment.

**SECURE COMPUTER APPLICATIONS IN AN
ENTERPRISE ENVIRONMENT**

by

Tim Lowman

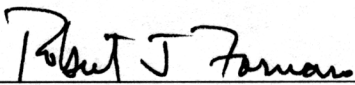
A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

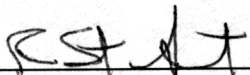
COMPUTER SCIENCE

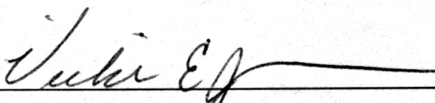
Raleigh

1998

APPROVED BY:







Chair of Advisory Committee

BIOGRAPHY

Tim Lowman is the Systems Programmer II for North Carolina State University. He earned his Bachelors of Science in 1990. He also teaches at NC State, CSC255: String Processing with the Perl Language.

ACKNOWLEDGEMENTS

I would like to thank all of the people who gave me support. My mother (Carolyn Lowman) and my father (Jimmy Lowman) who always told me I would go far. Dr. Fornaro and Carol Miller, two instructors who were there from start to end of my undergraduate and graduate career. Dr. Vicki Jones for accepting the position as chair of the committee. Dr. Rob St. Amant for comments about the user interface. The entire AD&D crew (Dan Deter, Dana and Charles Brabec, Jeff Webster, Lee Gray, Dwayne Sorrell, and James Deal) for repeated kicks in the rear to get this work finished. I would like to thank Lou Harrison for being so understanding when I was down to the final wire. Lastly, I would like to thank my hero Larry Wall (creator of Perl and many other tools I use on a daily basis).

Contents

List of Figures	vi
1 Introduction	1
1.1 Enterprise Environment	1
1.2 Security Issues	1
1.3 Literature Search	2
1.4 Problem Statement	3
1.4.1 Encryption	4
1.4.2 Authentication	5
1.4.3 Secure Transmission Protocol	8
1.4.4 Secure Storage Areas	9
1.5 Thesis Organization	9
2 Background	10
2.1 The Basic Language: HTML	11
2.2 The Server: HTTP	13
2.3 Dynamically Created HTML: CGI	15
2.4 How to Secure the Protocol	17
2.4.1 Digital Certificates	17
2.4.2 Secure Socket Layer: SSL	19
2.5 Authentication Methods: Kerberos	23
2.6 Securing the File Area: Andrew File System	25
2.7 Database Shared Resources	26
3 The Architecture of a Solution	29
3.1 Identifying the User	29
3.2 Setting up an SSL Connection	30
3.3 Uploading, Downloading, and Modifying Data	31
3.4 Building a Secure Enterprise Application	33
3.4.1 Upgrading the Web Server	34
3.4.2 Preparing the User's Browser	37
3.4.3 CGI Application Setup	37
3.4.4 Creating IP Based Authentication	39

4	Submit	40
4.1	Submit Background	40
4.2	Submit Implementation	41
5	Conclusions	48
5.1	Conclusions	48
	Bibliography	49
A	Apache Kerberos Module	53
B	access.conf	80
C	Certificate Authority Creation	83
D	Sybase Scripts	85
E	Submit Perl Library	94
F	Submit	106

List of Figures

1.1	Pros and Cons of Current Security Methods.	3
1.2	Encryption.	4
1.3	A Sample UNIX /etc/passwd Entry.	4
1.4	A Simple Perl One Way Encryption.	4
1.5	A Sample Crypt Session.	5
1.6	Basic Authentication.	6
1.7	Third Party Authentication - A wants to talk to B.	7
2.1	General Purpose Communication Model.	10
2.2	A Simple HTML Document.	12
2.3	Multipart/form-data: File Uploads in HTML Version 4.	12
2.4	Hypertext Transfer Protocol.	13
2.5	A Sample JavaScript.	16
2.6	t.pl - A Simple CGI Program.	17
2.7	Output of CGI program.	18
2.8	Digital Certificates Explained.	19
2.9	A Sample X.509 Digital Certificate.	20
2.10	SSL High Level Setup.	20
2.11	A Sample Apache MIME Types File.	21
2.12	CGI Program to Download Certificates.	21
2.13	Netscape Insecure and Secure Key Icon.	22
2.14	Tcpdump output.	23
2.15	AFS Tickets and Tokens.	26
2.16	PTS Command.	26
2.17	Perl Script Which Connects to an SQL Server.	27
3.1	New Model of the Application with Security Enhancements.	30
3.2	Initial Communication.	30
3.3	Obtaining the Public Certificate.	31
3.4	Site Certificate.	32
3.5	SSL Handshake.	32
3.6	Authentication.	33
3.7	CGI Execution on the HTTP Server.	33

3.8	Apache Secure Server.	35
3.9	Kerberos Authentication in access.conf.	35
3.10	A URL for Loading a Digital Certificate.	37
3.11	mdtest.pl, A Perl Example of MD5.	38
3.12	Output of the Perl MD5 Program.	38
4.1	Sparcstation 5 Machine Configuration.	42
4.2	Apache Secure Server.	42
4.3	Algorithm for Submit.	45
4.4	Sybase Tables.	47
4.5	A Flat-file Database.	47

Chapter 1

Introduction

1.1 Enterprise Environment

What is an enterprise environment? An enterprise environment is the collective of an organization's software, hardware, networking, and data systems. The environment can effectively place all enterprise computing power on each user's desktop. It also creates an organization-wide network, perhaps linking many smaller networks. Typically, an enterprise environment comprises workstations, servers, shared database resources, and a communications network. In an enterprise environment processing is distributed among workstations and servers where servers specialize in their function. Large databases exist which contain shared information that is available to users in the enterprise environment. Individual computers are linked together via a network backbone which allows devices to communicate with one another. However in the enterprise environment, the network is not necessarily guaranteed to be secure. In this work, we study the problems associated with securing computer applications in an enterprise environment, provide a general solution architecture for securing computer applications, and discuss a prototype implementation of the general solution architecture.

1.2 Security Issues

Before enterprise environments, users were protected from each other by the operating system running on a central processing unit (CPU). In the enterprise environment, users no longer work on a single system, but instead work in the computing collective.

Applications, which were once bound to a single system, may now use the full resources of an enterprise environment: remote databases, high capacity networks, and a multiple CPU configuration. Collaborative ventures require users to cooperate or share data with each other. Data transactions within the environment need to occur in a secure manner. Why? If transactions are not secure, one risks revealing private data to individuals who are not part of the collaborative venture.

“Secure manner” has many meanings based on the functions the application performs. At one end of the spectrum, there is a need for protection, but in which the consequences of a security breach are minimal. At the opposite end, the requirement is to protect the classified data at all costs. In a minimal consequence environment, a simple security protocol might require the user to enter a birth date, phone number, or other trivial data item which would be used to establish the user’s identity. While this “soft-security” approach works with applications that have minimal consequences of a security breach, other applications involving classified data require a maximum security approach. This maximum security approach may require complex challenge/response schemes, complex passwords, or additional security hardware for example a key to the computer console or a keyed magnetic card.

1.3 Literature Search

Security literature offers competing technologies to provide computer network security in an enterprise environment. The Web Realm Authentication Protocol [27] (WRAP) is a cookie-based approach to securing the enterprise environment which requires homogeneous machine architectures within the enterprise environment and browsers which are capable of cookie technology. Another approach, Secure MIME [14] (S/MIME), which implements security as a part of the Multipurpose Internet Mail Extensions (MIME) standard requires substantial modifications to existing software packages to make them compliant to this protocol. Another approach is IP Secure (IPSEC) [14]. IPSEC protects IP packets. IPSEC provides security using HMACS, a special form of key seeded hashes. Figure 1.1 illustrates a table listing advantages and disadvantages of these methods. After evaluating the these approaches to securing applications, a new method needs to be found which requires little, if any, modification to the computer applications, is easy to setup and use, and operates over a variety of hardware architectures and operating systems.

	Advantage	Disadvantage
WRAP	Easy for the user to implement. Centrally managed. Easy to add to a web based application.	Currently the code is in alpha. Requires Cookies to be enabled. Currently authentication enabled only on the same hardware architecture.
S/MIME	Able to provide services based on digital certificates Able to protect objects inside of the individual messages.	Currently few applications make use of this method.
IPSEC	Easy to create securely linked network entities.	Currently few, if any, applications take advantage of this method. Setup and use is complicated and difficult.

Figure 1.1: Pros and Cons of Current Security Methods.

1.4 Problem Statement

This thesis provides a model for securing network-based computer applications in an enterprise environment. Transmissions over the network need to occur in a secure manner. The data protection (armor) that we use for these connections must keep the data sent private. Another goal of this research is to find a method of providing a secure application on heterogeneous machine architectures without having to make substantial modifications to the client computer applications.

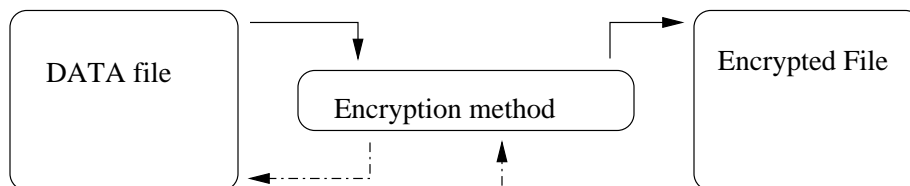
In order to accomplish these goals, we need to address problems of:

1. Encryption
2. Authentication
3. Secure Transmission Protocol
4. Secure Storage Areas

These problems are explored in further detail in the following sections.

1.4.1 Encryption

Encryption is the process of converting a data file into a coded ciphertext. Figure 1.2 shows the basic flow through an encryption operation. Many forms of encryption exist.



A data file is passed into an encryption method (which may require additional information (cipher key, etc.) Some encryption methods allow for decryption of the data while others are one way operations.

Figure 1.2: Encryption.

Some encryption methods are one way: UNIX password hashes (see Figure 1.3) or MD5 digest [24] keys.

```
tkl:XABJd9N3RfR2U:100:100:Tim Lowman:/usr/users/staff/tkl:/bin/csh
```

Figure 1.3: A Sample UNIX `/etc/passwd` Entry.

One way encryption produces an encrypted ciphertext which cannot be reconstituted back into the original. For example in Figure 1.4, the crypt function in Perl is given the text string “This is a test” and a salt value, “XA”.

```
perl -e 'print crypt("This is a test", "XA");'
XAJmKDacV6BL6
```

Figure 1.4: A Simple Perl One Way Encryption.

Perl will then use the UNIX DES encryption routine, with the two character salt value, to produce an encrypted string. Once this encryption has taken place, no algorithm exists which will convert the encrypted string back to its original form. In the case of one way encryption, the encrypted form is stored, usually in a secure location on the server.

This is the approach taken by many password systems. User passwords are encrypted and stored in the system. When a user wants access, he supplies his password, it is encrypted, and compared against the stored encrypted form. If the encrypted results match, the user is granted access. If the results do not match, the user is denied access.

Other encryption methods: UNIX crypt or PGP allow the user to apply the original cipherkey to decrypt the ciphertext thus reproducing the original plaintext data file.

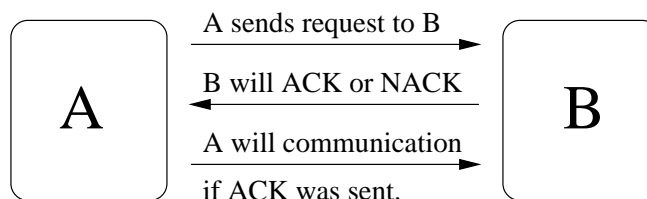
```
[418] astorath /users/staff/tkl>cat testfile
This is a test
[419] astorath /users/staff/tkl>crypt < testfile > testfile.crypt
Enter key:katmandu
[420] astorath /users/staff/tkl>less testfile.crypt
<92><92>l1*f<D4>\<8B><8D>^Y^LD<C2><E7>
[421] astorath /users/staff/tkl>crypt < testfile.crypt > testfile.orig
Enter key:katmandu
[422] astorath /users/staff/tkl>cat testfile.orig
This is a test
```

Figure 1.5: A Sample Crypt Session.

In Figure 1.5, the UNIX crypt function is used to encrypt a data file, “testfile”. The data file is read by the crypt command and a password is requested. “crypt” implements a one-rotor machine designed along the lines of the German Enigma [16], with a 256-element rotor. The password supplied is used to set the rotor. In order to decrypt the file, the same password is entered to the crypt command.

1.4.2 Authentication

Authentication is the verification of the identity of a person or process that allows access to a resource. The resource might be data or it might be a service: network, machine access, etc. At the heart of this problem is making sure that a user is who he says he is. Authentication can be accomplished by shared secrets, such as UNIX passwords, or by other methods: voice recognition, thumb prints, retinal scans, etc. [15] Most password schemes fall under the category of password challenges to verify the shared secret password between the user and the authenticating service. Several methods of authentication exist which fall under this category.



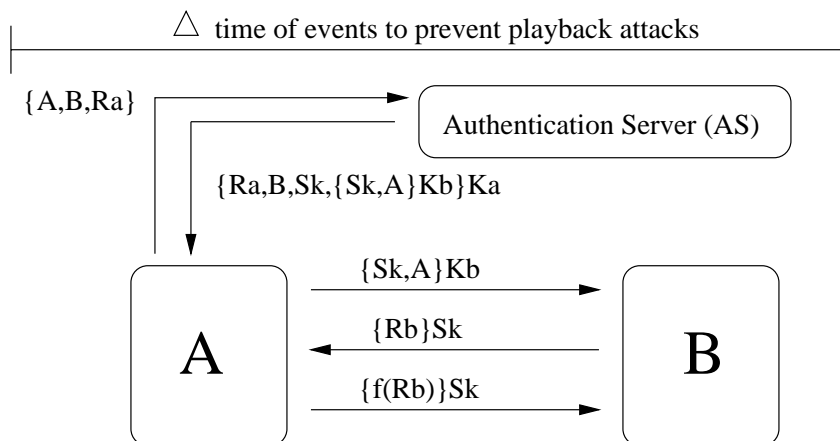
A wants to talk to B.

Figure 1.6: Basic Authentication.

The basic form of authentication asks the service for permission (see Figure 1.6). In this basic authentication, the client asks for permission to use the service and the service either grants or denies access based on a challenge to the user. Challenge Handshake Authentication Protocol (CHAP) is one approach. CHAP uses a random challenge and a secret key to provide authentication. One of the drawbacks of this approach is that the secret key is in clear text on the server system. If the server were compromised, the secret key could allow authentication as any user.

Another approach is to use standard UNIX password challenges. Passwords are typed in by the user and authenticated against the encrypted password stored on the server. This method is “soft-secure” since the password is transmitted across the network in clear text mode. Network “sniffers”, devices/programs which copy datagram packets from the network and reassemble them in order to reproduce the datagram, allow any user with a sniffer on the network to see these clear text passwords and authenticate as another user. For maximum security, we must choose an authentication method which does not store the passwords/secrets in clear text and does not pass the password information in clear text over our enterprise network. We also need a strong method of authentication which allows enterprise applications to trust the user.

In an enterprise environment, another form of authentication is common: third party authentication (see Figure 1.7). For this authentication method to operate, both A and B must have registered with an authentication server by providing their secret keys. When A wants to talk to B, A generates a random number for its identification (R_a) and passes this with its id (A), and the host it wishes to talk with (B), to the authentication server. The authentication server (AS) returns the random identification number and session information encrypted using A’s secret key (K_a). The session information contains a session



R_a = Random number generated by A
 R_b = Random number generated by B
 K_a = A's secret key (known by both A and AS)
 K_b = B's secret key (known to both B and AS)
 S_k = Session key (generated by AS and never sent in plain text over the network)
 $\{X\}Y$ = Encrypt the packet containing X with the key Y
 $f(X)$ = apply some function to X to get the next result

Figure 1.7: Third Party Authentication - A wants to talk to B.

key generated by AS, the identification number, and the same session key again encrypted with B's secret key. A then decrypts the session information using its secret key and sends the session key and requesting host (A) encrypted with B's secret key to B. B then decrypts the session key and requesting host, and B uses the session key to talk with A after verifying that it was A that was requesting the communication. Once the session key is established on both sides, secure communication can occur. For each subsequent communication a new message number is produced by applying a known function. The authentication set in this model occurs when A sends a request to the AS in order to speak with B. If A was not permitted to speak with B, no session information would be returned.

Consider the following example of third party authentication using these sample constructs. Suppose Alice wishes to talk to Bob. Our secret key algorithm will be represented by a number showing how many letters forward in the alphabet to rotate. Thus a key of "2" would mean that the letter A would be "C", the letter "B" would be "D", and "Z" would be "B". Alice knows that her secret key is "4" and Bob knows that his secret

key is “2”. Chuck, the third party authentication person, knows that both Alice’s secret key is “4” and that Bob’s secret key is “2”. Alice picks an id number and sends it and the person she wants to speak with (Bob) to Chuck. Chuck checks to see if Bob wants to communicate with Alice and if he does, he returns to Alice the id number she used (Bob’s name) a special session key, and the same session key encrypted with Bob’s secret key. In order to prevent anyone other than Alice reading the message, Chuck encrypts the entire return packet with Alice’s secret key. Alice receives the key and decrypts it using her key of “4”. She now has the session key and a special packet for Bob containing a copy of the session key encrypted with Bob’s secret key. Alice cannot read the encrypted session key since she does not know Bob’s secret key. Bob receives this information from Alice and uses his secret key to decrypt the session information and the name of the person who wished to speak with him. He can now check to see if the person who asked to speak with him is the same as the person listed with the session key. If they are the same, Bob can then create a message identification number and using the session key (which both Alice and Bob know) encrypt the data with the session key and send messages back and forth. To keep messages in order, a function can be applied to the message identification number to generate the next message number.

1.4.3 Secure Transmission Protocol

The problem with selecting a secure protocol is that the protocol must be usable on many different machine platforms. Applications which are limited to a single platform and are not portable are not truly viable in an enterprise environment. The application programmer has two choices when confronted with this problem: use an existing protocol or create a new protocol. The problem creating a new protocol is the application programmer faces code conversion difficulties. Converting socket based applications to stream, ANSI code to POSIX, and System V constructs to BSD when porting are examples of such difficulties. The overhead of porting software would certainly slow, if not halt, a multi-platform application. If the application programmer chooses an existing protocol, then the protocol must support all of the necessary features (preventing users from collecting or modifying data passed between applications) allowing the secure transfer of information in the enterprise environment.

1.4.4 Secure Storage Areas

The areas where the application shares information within an enterprise environment must be protected. These areas may be file system areas, memory areas, or other mass storage devices. Without protection, other entities may gain access to privileged data. The secured area will act as a repository for the user's stored data. The application will act on behalf of the user to manipulate this stored data and provide access to it in a secure manner. By authenticating users, we have reasonable certainty of the user's identity. We can grant access to stored data based on their identity. Several methods exist for securing file system areas. Unix File system (UFS) provides security by a series of protection bits. Each of these bits grants access for a specific user, a group of users, or any user on the system. Windows NT provides another approach: user groups. Groups exist or can be created for specific purposes. Members of these groups are granted access to specified directories in the NT file system.

1.5 Thesis Organization

In Chapter 2, background terms and ideas used in this document are explained. Chapter 3 sets forth a generic, secure network-based computer application model. Chapter 4 describes a specific secure application implementation of this network-based computer application model. Finally, Chapter 5 presents conclusions and directions for future work.

Chapter 2

Background

Figure 2.1 illustrates a typical client/server architecture of an enterprise application. Before describing the components of the model, a brief explanation of the terms and ideas used in this paper will be presented.

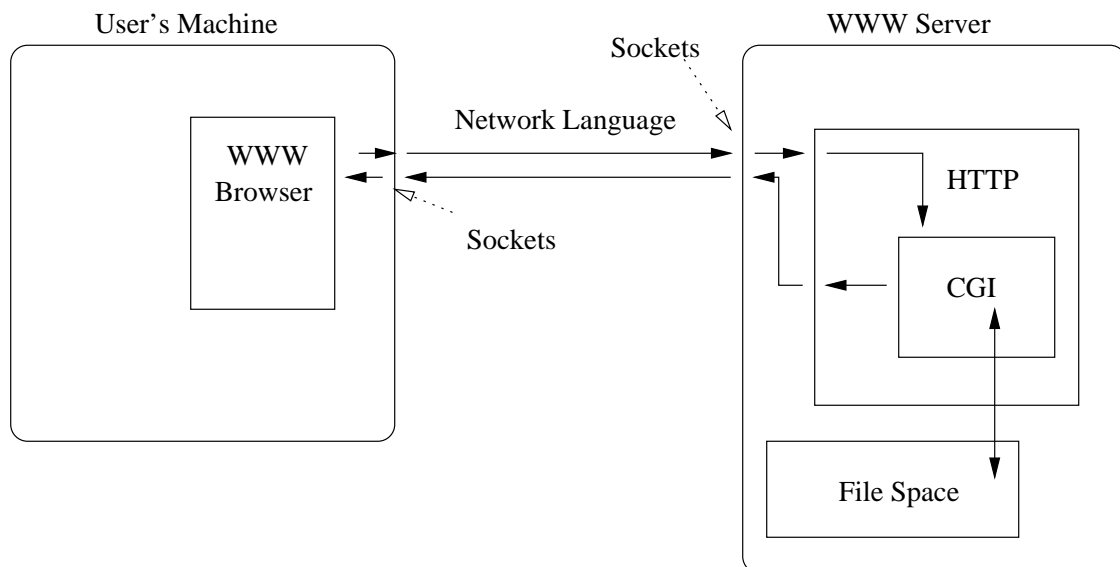


Figure 2.1: General Purpose Communication Model.

1. The basic language: Hypertext Mark-up Language (HTML)
2. Dynamically Created HTML: Common Gateway Interface (CGI)
3. The Server: Hypertext Transfer Protocol Server (HTTP)

4. Securing the Protocol: Digital Certificates and Secure Socket Layer (SSL)
5. Kerberos: An authentication method
6. A Secure File System: Andrew File System (AFS)
7. Database Shared Resources

2.1 The Basic Language: HTML

HTML aims to provide an easy to use, distributed, hypermedia system. The early roots of HTML came from Ted Nelson who in the 1960s came up with the idea of using a “hyperlinked” document retrieval system to connect many different documents on many different machines. Later in 1990, at CERN, the European Laboratory for Particle Physics, Tim Berners-Lee expanded this hypertext linking system to operate as an elaborate document retrieval system he called the “World Wide Web” [2]. Documents in this “Web” were retrievable by users of CERN utilizing a series of “notes with links (like references) between them” [2], thus allowing easy access to the many documents stored in the archives at CERN.

Since that time, HTML developed into a powerful system designed to support multimedia presentations. The HTML language is made up of many different tags. Each tag is a command which will be interpreted by the browser (a program such as Netscape, Internet Explorer, or Lynx which allows a user to browse the Web) to display the data in the browser window. Tags in HTML typically follow the form `<command>` (start command), followed by text which is subject to this command, followed by `</command>` (end command). A typical HTML version document is presented in Figure 2.2. The symbols `<!-- ... -->` are HTML comments describing each line in the HTML document.

The first `<HTML>` is the opening of the HTML document. `<HEAD>` starts the heading section of the document. In the heading section, one can define any META tags which will be used to describe the document content and the title of the document. After closing the heading section with the `</HEAD>` closing tag, the body of the HTML document starts. `<BODY>` marks the beginning of the body of the document. The body is where the contents of an HTML document are kept. `</BODY>` ends the body section of an HTML document. Finally, the closing `</HTML>` tag signals the end of the document.

```

<HTML> <!-- Begin the HTML Document -->
  <HEAD> <!-- Begin the Heading of the Document -->
    <TITLE>Title of Document</TITLE> <!-- Title of the Document -->
  </HEAD> <!-- Close the Heading of the Document -->
  <BODY> <!-- Begin the Body of the Document -->
  Document Contents ...
  </BODY> <!-- Close the Body of the Document -->
</HTML> <!-- Close the HTML Document -->

```

Figure 2.2: A Simple HTML Document.

A Quick primer for HTML can be found at <http://www.w3c.org/MarkUp/Guide> [22]. Another excellent tutorial is located at http://www.csc.ncsu.edu/csc_info/csc251/www/tutorial/index.html [18]. The version of HTML discussed in this thesis is “HTML version 4” [23] (HTML4). HTML4 has an important feature which allows our model to operate: multipart/form-data [21]. Multipart/form-data allows the browser to upload data from the user’s data space to the web server. The syntax for file uploads can be seen in Figure 2.3.

```

<FORM ENCTYPE="multipart/form-data" ACTION="_URL_" METHOD=POST>
File to process: <INPUT NAME="userfile1" TYPE="file">
<INPUT TYPE="submit" VALUE="Send File">
</FORM>

```

Figure 2.3: Multipart/form-data: File Uploads in HTML Version 4.

Once the user selects the file to upload (he may type in the name of the file or use the “browse” facility to select the file using the graphical user interface provided by the web browser), the browser will then send the form data containing the file to the web server. Since the file may contain binary or other data which would make the standard “application/x-www-form-urlencoded” [21] unsuitable, the encoding method selected will be that of the application type of the file or “application/octet-stream” [11]. All of these encoding types are a form of Multipurpose Internet Mail Extensions [10](MIME). MIME extensions allow for transmission of data via the Internet which might contain textual messages in character sets other than US-ASCII, non-textual message bodies, multi-part

message bodies, and textual header information in character sets other than US-ASCII. A complete list of MIME types can be found at

<ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types>

and are not included in this document (several thousand of these types exist).

Why choose HTML as our communications language? Since HTML browsers are established on so many different platforms, they provide the ideal medium to allow communications in an enterprise environment. HTML's rich language set allows for FORM data communication, the ability to present graphics and sound with any presentation, and the ability to upload text files using the file upload multipart/form-data added to HTML4. File integrity is maintained by having the uploaded files encoded by the chosen MIME methods. All of these features allow for seamless communication with the user regardless of data types being exchanged.

2.2 The Server: HTTP

HTTP is "an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems" [1]. HTTP is the typical means by which HTML data are transferred from a server to a client browser and vice versa. HTTP is a low bandwidth protocol which is stateless. Figure 2.4 shows that a user's browser can contact a given server using a Uniform Resource Locator (URL). A URL is much like an address for a house on a given street. Given an address, the house can be located easily.

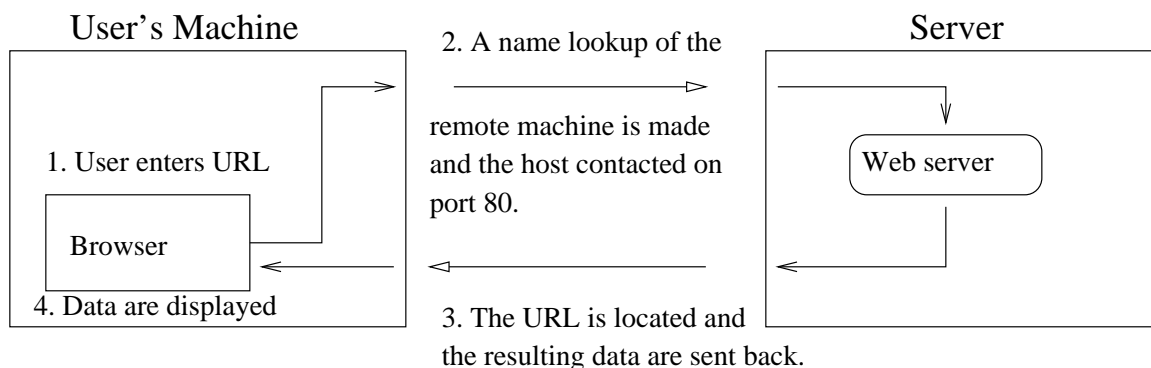


Figure 2.4: Hypertext Transfer Protocol.

As mentioned earlier this protocol is stateless. During each cycle of contacting the

HTTP server, the HTTP server loses all state information: variables, connecting machine name, IP address, etc. Since the connection is stateless, any information which needs to be saved must be encoded into the HTML document, otherwise it will be lost. A common method of encoding these data is using hidden fields. Hidden fields are not displayed in the browser but remain in the HTML data sent from the server. Another method of storing the data between cycles is by storing it on the server using an ID mechanism. The data are stored in files using the ID number for the connection to the HTTP server. When using this server-stored method, the stored data must be removed from the server when it expires to prevent filling the storage area. The problem with using this method is determining when these data expire.

CGI allows the server to execute arbitrary applications. A CGI application is activated by a Uniform Resource Locator (URL) call. Just as HTTP is stateless, so is a CGI call. When the CGI program is activated via a URL call, the server executes the CGI program and passes HTTP server data (environment variables) and the contents of standard in (STDIN) to the CGI program. The CGI program, executing with the privileges and userid of the HTTP server, reads this input, processes the data, and outputs the newly processed data back to the HTTP server. The server reads the “content-type” data header placed on the returned data by the CGI program and sends the data back to the user’s browser. Between CGI calls no state information is kept since the program starts, executes, and then completes/exits. The same method, hidden fields, can be used to continue data elements from one CGI call to another CGI call.

The “NCSA HTTPd server” [25] was one of the original HTTP servers. This HTTP server was designed to work with NCSA Mosaic, an early browser. NCSA HTTPd supports CGI programs, imagemap, security access based on URL directory, HTTP 1.0 basic authentication, MD5 digest authentication, and Kerberos versions 4 and 5 authentication. NCSA HTTPd is no longer under development as of version 1.5.2a. [25] In mid-1994 Rob McCool, who worked at NCSA, left the company and with other volunteers, started an organized effort to develop their own HTTP server with additional extensions. The server changes were distributed in the form of “patches.” Thus came, by way of a play on words, the “Apache Server” [13] (a-patchy server). The Apache server is the HTTP server used in our model and prototype. It is available for most, if not all, UNIX systems, Windows(9x/NT) platforms, and Macintosh systems.

2.3 Dynamically Created HTML: CGI

CGI provides a standard interface between a web server and a user's program. Normally an HTML page is "static, which means [the HTML page] exists in a constant state" [20]. These static pages do not change without someone editing the pages manually. CGI allows for dynamic pages. Dynamic pages are pages which are created "on-the-fly" by the web server executing an external program. The executing program operates in the common gateway defined interface. A particular coding language is not required for a CGI program. The code can be written in C, C++, Pascal, FORTRAN, or a scripting language such as Perl, C-Shell, Korn Shell, or Bourne Shell. Typically a CGI program executes on the HTTP server and produces HTML code. Since Java applets and JavaScript execute on the user's browser, these are two examples of languages which are not generally used to create CGI programs. Java applets and JavaScript are both downloaded to the user's browser which executes there. Figure 2.5 illustrates an example of JavaScript mixed with HTML which will execute on a user's browser. This external CGI program implements the specific functionality of an enterprise application.

When the CGI program executes, it reads from standard input any information that is provided by the user and is passed operating system environment variables by the web server. The data are then processed by program and new content is produced. The new content is then passed back to the web server with a "content-type" [11] describing what type of information is contained in the new data. The new data could be an image, HTML, a ASCII text document, or any other MIME type. The web server recognizes what type the new data are said to be and passes that information and the new data back to the user's browser. A simple example of a CGI program is included in Figure 2.6 and the output is in Figure 2.7.

The program shown in Figure 2.6, `t.pl`, creates a dynamic HTML page which displays all the operating system environment variables and standard input passed to the CGI program. `t.pl` is a typical program one might have in a CGI directory in order to debug or test the HTTP server or a new form. This program (`t.pl`) illustrates the basic functionality of CGI programs.

Commonly, CGI programs are used to produce HTML form data. Once generated, an HTML form is sent to the user's browser, interpreted, filled in by the user, and sent back to the same or different CGI program for additional processing. We use CGI capability to

```
<script>
<!--
if (document.images) {
  image1on = new Image();
  image1on.src = "./graphics/fiszz8.gif";
  image2on = new Image();
  image2on.src = "./graphics/fiszz8.gif";
  image3on = new Image();
  image3on.src = "./graphics/fiszz8.gif";
  image1off = new Image();
  image1off.src = "./graphics/fiszz5.gif";
  image2off = new Image();
  image2off.src = "./graphics/fiszz5.gif";
  image3off = new Image();
  image3off.src = "./graphics/fiszz5.gif";
  image4off = new Image();
  image4off.src = "./graphics/fiszz5.gif";
}
function turnOn(imageName) {
  if (document.images) {
    document[imageName].src = eval(imageName + "on.src");
  }
}
function turnOff(imageName) {
  if (document.images) {
    document[imageName].src = eval(imageName + "off.src");
  }
}
// -->
</script>
```

Figure 2.5: A Sample JavaScript.

intercept multi-part encoded files, decode the files, and place these files in secure locations.

Web servers run under a particular userid on the server machine. While many times this is considered a possible security hole [6], our model uses this feature to its advantage. By giving the HTTP server's trusted userid the authority to make modifications to protected space, the web server becomes a trusted process which can place files and manipulate sensitive data in a secure manner. The trusted web server then becomes the entity that moves files and changes data inside of our protected file space.


```
#!/usr/local/bin/perl

print "Content-type: text/html\n\n";
foreach $k (sort keys %ENV) {
    print "$k = $ENV{$k}<BR>\n";
}
print "Data from STDIN:<BR>\n";
while (<STDIN>) {
    print;
    print "<BR>\n";
}
```

Figure 2.6: t.pl - A Simple CGI Program.

2.4 How to Secure the Protocol

2.4.1 Digital Certificates

Digital certificates make up an important part of public key cryptography. A user must have a pair of cryptographic keys, a private key and a public key, to send or receive messages. The private and public keys are composed of long strings of data containing 500 to 1000 bits [7]. The user stores his private key somewhere safe (encrypted on his hard-drive, in a secured file space area) but makes his public key known to those individuals that he wishes to communicate. Suppose Alice wants to send a secure message to Bob. Secure means that the message can be verified as coming from her, Figure 2.8 shows how this secure communication occurs using digital certificates.

Alice first uses cryptographic software to generate a private key (keypriv) and a public key (keypub). She then sends the public key to a certification authority (CA) and asks for a digital certificate. The CA, through whatever means it requires, validates the authenticity of Alice's identity. After the CA is sure Alice is Alice, it issues her a digital certificate confirming that Alice's public key is really Alice's public key. Inside of the certificate is the CA's digital signature. This signature can be validated by anyone who knows the CA's public key to prove that the CA validated Alice's public key.

Alice then digitally signs her message to Bob. First she uses a hash function on the message to create a message digest. The message digest is then encrypted with Alice's

```
DOCUMENT_ROOT = /usr/local/htdocs
GATEWAY_INTERFACE = CGI/1.1
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
HTTP_CONNECTION = Keep-Alive
HTTP_HOST = www.csc.ncsu.edu
HTTP_USER_AGENT = Mozilla/3.04Gold (X11; I; SunOS 5.5.1 sun4m)
PATH = /usr/sbin:/usr/bin
QUERY_STRING =
REMOTE_ADDR = 152.1.61.13
REMOTE_HOST = astorath.csc.ncsu.edu
REMOTE_PORT = 34234
REQUEST_METHOD = GET
REQUEST_URI = /cgi-bin/t.pl
SCRIPT_FILENAME = /usr/local/httpd/cgi-bin/t.pl
SCRIPT_NAME = /cgi-bin/t.pl
SCRIPT_URI = http://www.csc.ncsu.edu/cgi-bin/t.pl
SCRIPT_URL = /cgi-bin/t.pl
SERVER_ADMIN = webmaster@csc.ncsu.edu
SERVER_NAME = www.csc.ncsu.edu
SERVER_PORT = 80
SERVER_PROTOCOL = HTTP/1.0
SERVER_SIGNATURE =
SERVER_SOFTWARE = Apache/1.3.4 (Unix)
TZ = US/Eastern
```

Figure 2.7: Output of CGI program.

private key to make her digital signature. She sends her signature, the message, and a copy of her digital certificate (remember, this digital certificate includes a copy of her public key). Bob receives the above data. Bob uses the CA's public key to verify the CA's digital certificate on Alice's certificate. Bob can not be sure that the certificate he has is indeed Alice's, since someone could be impersonating Alice by sending Bob messages using a forged certificate. Bob then uses the Alice's known public key extracted from the digital certificate to decrypt Alice's digital signature, which recreates the message digest. Bob then, using the same hash function, creates a message digest of the message sent. If Bob's message digest and the recreated message digest from Alice's digital signature match, Bob can be certain that Alice sent him the message and that it has not been tampered with by anyone along the way.

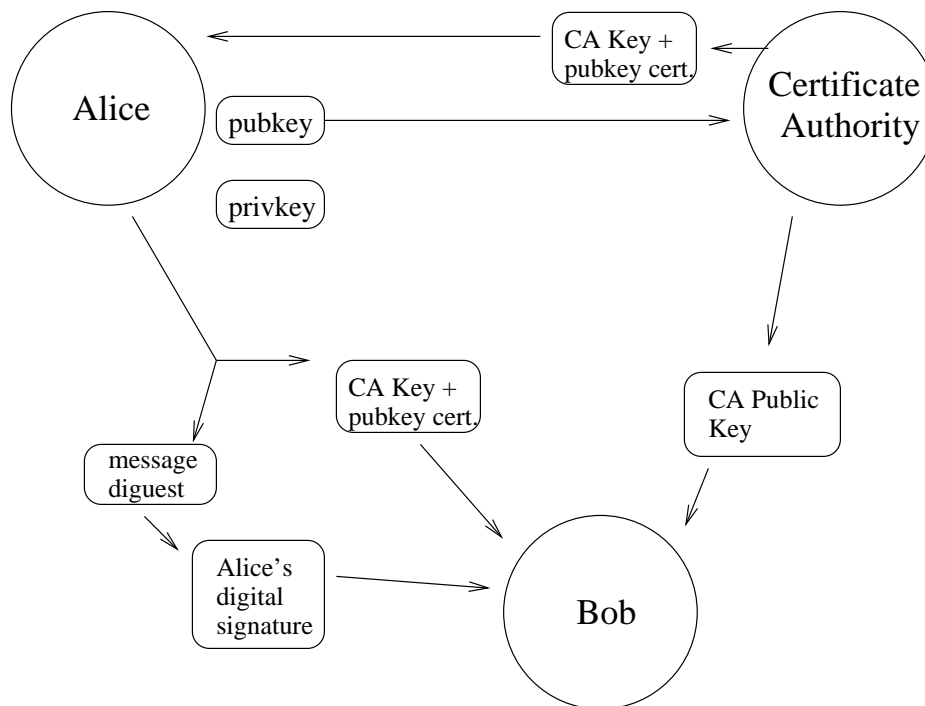


Figure 2.8: Digital Certificates Explained.

2.4.2 Secure Socket Layer: SSL

“Secure Socket Layer” [12] (SSL) was developed by Netscape Communications to provide a method of secure transactions over the Internet. SSL provides a method of client/server communications which prevents “eavesdropping, tampering, or message forgery” [12]. SSL is application protocol independent which means that different protocols (e.g. Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and even Telnet) can be transparently layered on top of SSL without having to alter the chosen protocol. The SSL protocol consists of two phases: server authentication and an optional client authentication. During server authentication, the client requests a digital certificate and cipher preferences from the server. We use a “X.509” [5] certificate. X.509 is a certificate format proposed by the International Telecommunications Union (ITU-T), formerly known as CCITT, to provide standards for digital certificate formats. X.509 is part of the ITU-T X.500 [4] directory services. An example of an X.509 certificate can be seen in Figure 2.9.

The client creates a master key, which is encrypted with the server’s public key taken from the X.509 certificate, and transmits it to the server. The server recovers the

```

-----BEGIN CERTIFICATE-----
MIICTTCCAbagAwIBAgIBADANBgkqhkiG9w0BAQQFADBMMQswCQYDVQQGEwJHQjEM
MAoGA1UEChMDVUNMMRgwFgYDVQQLEw9JQ0UtVEVMIFByb2p1Y3QxFTATBgNVBAMT
DFRydXNORmFjdG9yeTAeFw05NzAOMjIxNDM5MTRaFw05ODAOmJIxNDM5MTRaMEwx
CzAJBGNVBAYTAkdCMQwwCgYDVQQKEwNVQ0wxGDAWBGNVBAsTD01DRS1URUwgUHJv
amVjdDEVMBMGA1UEAxMMVHJ1c3RlYWN0b3J5MIGcMAoGBFUIAQECAgQAA4GNADCB
iQKBgQCEier8NcXkUW1f0G6aC6u0i8q/98JqS6RxK5YmHIGKCKuTWAUjzLfUa4dt
U9igGCjTuxaDqlzEim+t/02pmiBZT9HaX++35MjQPUWmsChcYU5WyzGERXi+rQaw
zlwS73zM8qiPj/971XYycWhgLOVaiDSPxRXEUdWoaGruom4mNQIDAQABo0IwQDAD
BgNVHQ4EFgQUHal1LZr7oVg5z6lYzrhTgZRCmcUwDgYDVROPAQH/BAQDAgH2MA8G
A1UdEwEB/wQFMAMBAf8wDQYJKoZIhvcNAQEEBQADgYEAfaggf16FZoioecjv0dq8
/DXo/u11iMzvXn08gjX/zl2b4wtPbSh0SY5FhkSm8GeySasz+/Nwb/uzfnIhokWi
lfPZHt1CWtXbIy/TN51eJyq04ceDCQDwvLC2enVg9KB+GJ34b5c5VaPRzq8MBxsA
S7ELuYGtmYgYm9NZ0Ir7yU0=
-----END CERTIFICATE-----

```

Figure 2.9: A Sample X.509 Digital Certificate.

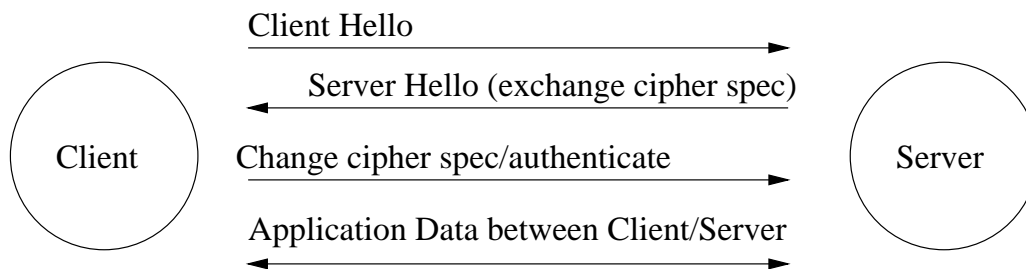


Figure 2.10: SSL High Level Setup.

master key by decrypting it with the server's private key and authenticates itself with the client by returning a message authenticated by the master key. Any data sent after this point are encrypted and authenticated with keys derived from the original master key. In the optional second phase, the server can challenge the client. The client authenticates itself to the server by returning the client's digital signature on the challenge, as well as the server's public-key. Figure 2.10 illustrates a typical SSL setup.

Once the SSL authentication challenge has been passed, the server and browser communicate via an encrypted data-stream. To pass the SSL handshake, the user must have a copy of the X.509 certificate from the server on a local machine in order to encrypt the master key with the server's public key.

The server certificate can be downloaded from the server using the “application/x-x509-ca-cert” [11] MIME type. MIME types are usually of the form *CLASSIFICATION/TYPE*. For instance, take our application/x-x509-ca-cert type. The classification is application and its type is x-x509-ca-cert. By associating helper applications and utilizing built in MIME extensions, applications such as browsers are able to perform the required actions to use the MIME data. Figure 2.11 shows a section of a MIME types file which describes the feature of the application/x-x509-ca-cert type.

```

...
application/news-transmission
application/octet-stream      bin
application/pdf              pdf
application/x-x509-ca-cert    der
video/x-msvideo              avi
video/x-sgi-movie            movie
...

```

Figure 2.11: A Sample Apache MIME Types File.

```

#!/bin/sh
if [ -n "$PATH_INFO" -a -r "$PATH_INFO" ]; then
    echo "Content-Type: application/x-x509-ca-cert"; echo
    cat $PATH_INFO
else
    cat << END
Content type: text/html
<HTML>
<HEAD><TITLE>Certificate not found!</TITLE></HEAD>
<BODY>
<P>
Sorry, I can't find the certificate that I planned to send back.
</BODY>
</HTML>
END
fi
exit 0

```

Figure 2.12: CGI Program to Download Certificates.

Downloading the certificate can be accomplished by adding the application/x-x509-ca-cert type to an HTTP server's known MIME types or by using a CGI script such as is included in Figure 2.12. The certificate is typically verified by a Certificate Authority (CA). In this case, we self-sign the site certificate we plan to use, thus setting us as our own CA. We chose to implement this method due to the fees associated with using a known CA such as Verisign, Thawte, or CyberTrust. The package used to do this is the SSLey package which is found at <http://www.psy.uq.oz.au/~ftp/Crypto/>. SSLey is a free implementation of Netscape's Secure Socket Layer. SSLey implements both the SSL version 2 [8] and the SSL version 3 [12] protocols. For our model, the SSLey package is used to add SSL capability to the HTTP server. The SSLey package is used to generate and sign certificates.

Once we have generated our certificate and the user has downloaded it into his browser, the HTTP server then compares the user's certificate with its copy of the CA during the SSL handshake. Once the authentication occurs, the web server communicates with the browser using encrypted connections. The user can differentiate between secure connections and insecure connections by looking at the secure icon in the browser.



Figure 2.13: Netscape Insecure and Secure Key Icon.

Netscape's Navigator uses a key symbol (see Figure 2.13) for this icon while Microsoft's Internet Explorer uses a padlock symbol. Suppose we use Navigator as our example. When the key icon is lit and not broken, a secure connection has been established. When the key is broken and unlit, the connection is not secure.

Why use SSL? SSL provides a means of encrypting data sent from the user's browser to the HTTP server and vice versa. In today's society, the technology is readily available to "sniff" networks. Sniffers take packets from the network and collect them in large data files. Tcpcdump is a simple sniffer that exists on most machines. Figure 2.14 shows output from the tcpcdump program. The data contained within the packets can be seen. Saving large number of packets would allow unsecured sources a view of the data transmitted between the user and the web server. Since the data travelling between the user and web server might contain sensitive information such as passwords, private data, etc. SSL provides an armor for the TCP/IP connection which prevents external sources

from viewing the data. Thus when passing sensitive information in an enterprise model, armored connections prevent unauthorized access to these data or the passwords which would grant access to these data.

```

12:59:55.354 8:0:9:90:14:c5 ff:ff:ff:ff:ff:ff 0064 100:
                e0e0 03ff ff00 6000 0098 013a 00ff ffff
                ffff ff04 5298 013a 0008 0009 9014 c504
                5200 0203 0c30 3830 3030 3939 3031 3443
                3538 3038 484e
12:59:55.354 8:0:9:90:14:c5 ff:ff:ff:ff:ff:ff 8137 96:
                ffff 0060 0000 9801 3a00 ffff ffff ffff
                0452 9801 3a00 0800 0990 14c5 0452 0002
                030c 3038 3030 3039 3930 3134 4335 3832
                3848 4e50 4939
12:59:56.018 arp who-has uni00bf.unity.ncsu.edu tell pa-ece03.ece.ncsu.edu
12:59:56.143 0:60:b0:2e:34:76 ff:ff:ff:ff:ff:ff 0063 99:
                e0e0 03ff ff00 6000 0098 013a 00ff ffff
                ffff ff04 5298 013a 0000 60b0 2e34 7604
                5200 0203 0c30 3036 3042 3032 4533 3437
                3638 3043 544e
12:59:56.143 0:60:b0:2e:34:76 ff:ff:ff:ff:ff:ff 8137 96:
                ffff 0060 0000 9801 3a00 ffff ffff ffff
                030c 3030 3630 4230 3245 3334 3736 3832
                4354 4e50 4932

```

Figure 2.14: Tcpdump output.

2.5 Authentication Methods: Kerberos

Kerberos [19] is an authentication protocol originally developed at MIT. Kerberos provides integrity and authentication in our model. It operates by a method known as shared secrets [19].

A Kerberos authentication server (KAS) and one or more ticket granting servers (TGS) are the primary components of this approach. First the user requests a ticket, containing the user's name and the name of the TGS, from the KAS. The KAS looks up the user in its database and generates a session key which will be used between the user and the various TGSs. The KAS encrypts this session key using the user's secret key (a one

way hashing algorithm of the user's password) and creates a ticket granting ticket (TGT) which is encrypted with the TGS's secret key, known only to the KAS and the TGS, for the user to present to the TGS. The KAS sends the TGT back to the user. The user decrypts the message and recovers the session key. Next, the user creates an authenticator consisting of his name and a time stamp, all encrypted with the session key. The user then sends a request to the TGS for a ticket to a particular target server. The request contains the name of the server, the TGT received from the authentication server, and the encrypted authenticator. The TGS decrypts the TGT with its secret key and then uses the session key included in the TGT to decrypt the authenticator. The TGS compares the information in the authenticator with the information in the ticket, the user's network address with the address the request was sent from, and the time stamp with the current time. If everything matches, it allows the request to proceed. The TGS creates a new session key for the user and target server and incorporates this key into a valid ticket for the user to present to the requested server. The ticket also contains the user's name, network address, a time stamp, an expiration time for the ticket, and the name of the server. The TGS also encrypts the new target-user session key using the session key shared by the user and the TGS. The TGS then sends both messages to the user. The user receives and decrypts these messages, extracting the target-user session key from the ticket. The user is now ready to authenticate with the target server. The user creates a new authenticator with the target-user session key and sends this encrypted authenticator and the original TGT to the target server. The target server decrypts the ticket and the authenticator, and it compares the session keys, target-user session keys, time stamp, and user's address. If everything checks out, the user is allowed access to that server. Since the time stamps are compared, it prevents a captured Kerberos session from being played back at a later time. Play back attacks are commonly used to thwart authentication methods which do not have time stamps. Sessions between a client and server are recorded and then played back to the server. Since each challenge would be correct, the server would willingly perform the set of operations again. Suppose the play back was adding one cent to an account balance. You can see how this play back method could be used again and again to add thousands of dollars to the account. Time stamps, such as those used in Kerberos, prevent play back attacks by voiding the requested operations if the time stamps are too far from the current time.

A user has a password, a shared secret, which is encrypted and compared with an encrypted copy stored on the server. If these two encrypted copies match, the Kerberos

system provides one with a ticket granting ticket which can be used later to authenticate other transactions. A good paper describing the operation of Kerberos on a non-technical level is the “Dialogue in Four Scenes” [3] by Bill Bryant. In this paper, the logical progression of the idea of Kerberos’s shared secret and ticket granting ticket scheme is explained.

Why use Kerberos? In an enterprise environment, any authentication method could be employed. The main reason for choosing Kerberos is the ability to incorporate Kerberos authentication algorithms into the Apache HTTP server. Once a password challenge has been made, the Kerberos authentication provides reasonable certainty that the user is who he claims to be. Since Kerberos passwords are never passed in plain text on the network, this method provides a good solution to problems of sniffers picking up the user’s password from the network.

2.6 Securing the File Area: Andrew File System

The Andrew File System (AFS) was originally developed at Carnegie Mellon University. AFS provides a distributed file system to multiple platforms and a method of authentication known as access control lists (ACLs). ACL authentication comes from the Kerberos ticket granting ticket, which in turn grants a file system ticket, **afs.eos.ncsu.edu@EOS.NCSU.EDU**, shown in Figure 2.15.

The file system ticket is then used to grant an AFS token. Figure 2.15 shows an example of the tickets and tokens used to gain AFS permissions. One can view tickets and tokens using the **klist** and **token** commands, respectively. The AFS token is used, much like the Kerberos ticket, to authenticate the user and allow him access to the file system. You can view the contents of an ACL by using the **pts** [26] command. **pts** allows the user to modify or list AFS ids in an ACL. Figure 2.16 shows an example of the **pts** command.

The AFS token is used to access the file system. Another method of authentication is IP authentication. IP authentication allows AFS system administrators to create AFS users whose AFS id matches their IP address. The AFS id is then placed into an ACL which will allow that IP address the same access as a normal user, but the authentication does not require any additional challenges except that the IP address of the machine originating the request be the same as the AFS id. By using this method of authentication, no password challenges or requests are required. All processes running on the machine have the ability to be authentic and can read/write data to the secure area using the AFS IP-based id.

```

[36] astorath /users/staff/tkl> klist
Ticket file:      /tmp/tkt269
Principal:       tkl@EOS.NCSU.EDU

   Issued             Expires             Principal
Nov 17 07:11:52   Nov 17 17:11:52   krbtgt.EOS.NCSU.EDU@EOS.NCSU.EDU
Nov 17 07:11:53   Nov 17 17:11:53   afs.eos.ncsu.edu@EOS.NCSU.EDU
Nov 17 07:11:54   Nov 17 17:11:54   afs.bp.ncsu.edu@EOS.NCSU.EDU
Nov 17 07:11:54   Nov 17 17:11:54   afs.unity.ncsu.edu@EOS.NCSU.EDU
Nov 17 07:24:02   Nov 17 17:14:02   afs.tx.ncsu.edu@EOS.NCSU.EDU

[37] astorath /users/staff/tkl> tokens
Tokens held by the Cache Manager:

User's (AFS ID 269) tokens for afs@unity.ncsu.edu [Expires Nov 17 17:11]
User's (AFS ID 269) tokens for afs@eos.ncsu.edu [Expires Nov 17 17:11]
User's (AFS ID 269) tokens for afs@bp.ncsu.edu [Expires Nov 17 17:11]
--End of list--

```

Figure 2.15: AFS Tickets and Tokens.

```

[39] astorath /users/staff/tkl> pts membership course-admin
Members of course-admin (id: -1965614004) are:
  submit
  tkl
  152.1.9.104

```

Figure 2.16: PTS Command.

AFS was chosen because it provides a distributed file system over many different platforms. Network file system, NFS, would also make a good choice, but due to the lack of NFS in our environment, AFS was chosen for the file system.

2.7 Database Shared Resources

A common problem in enterprise applications is that data need to be shared data. Shared data provides information to the computer applications. The information could be

course lockers, mailing lists, user ids, social security numbers, etc. A Structured Query Language (SQL) database server provides an effective means of managing the data.

```
#!/usr/local/bin/perl
#
# Example program using the Sybase::DBlib library
# Written by Dr. Charles J. Brabec
# Used with Permission
#
### include Sybase::DBlib extensions
use Sybase::DBlib;
### make sure we know where sybase lives
$ENV{'SYBASE'} = "/afs/eos.ncsu.edu/project/csc_sybase";
### Login information for the Sybase server
$username = "perl";
$password = "larrywall";
$server = "CSCDEV";
$Programe = "DBlib Demo";
### Open a connection to the server
$dbh = new Sybase::DBlib $username,
        $password, $server, $Programe;
### Set our SQL command
$stat= $dbh->dbcmd("select * from info\n");
print STDERR "DBcmd Stat = $stat\n";
### Execute our SQL command
$stat = $dbh->dbsqlxexec;
print STDERR "DBsqlxexec Stat = $stat\n";
### Retrieve the results
$stat = $dbh->dbresults;
print STDERR "DBresults Stat = $stat\n";
### Parse the returned rows, one at a time
while (@data = $dbh->dbnextrow) {
    print "$data[0]\n";
}
### Close the server connection
$dbh->dbcclose;
exit 0;
```

Figure 2.17: Perl Script Which Connects to an SQL Server.

In our prototype application we are creating, Sybase was chosen to fill the need of an SQL server. Any SQL server could have been chosen to fill this need. Other SQL servers

are Informix, MySQL, mSQL, Postgress, and Microsoft Back Office SQL server. The data can be stored in an SQL server and queried to provide information to the application. The data are extracted from the SQL server using Perl SQL modules. Figure 2.17 shows an example of connecting to an SQL server using Perl and Sybase SQL server.

Chapter 3

The Architecture of a Solution

What type of solution to the enterprise network security problem does this paper present? The solution presented here includes a description of appropriate network structures, software algorithms, cipher-ware encryption/authentication, and instructions on how these elements must interact to:

1. Provide a means of identifying the user is who he claims to be.
2. Create a secure environment for passing data, including the userid and password, from the user to the server and vice versa.
3. Upload, download, and modify protected data.

A pictorial design of the solution model can be seen in Figure 3.1. This new application model includes security enhancements to authenticate users (Kerberos), communicate privately (SSL network connections), and store protected data (secure AFS filespace). The Apache HTTP server allows the secure interoperation of these security features. The following sections will describe how the secure application model operates inside of the enterprise environment.

3.1 Identifying the User

The method used to identify the user is Kerberos (described in Section 2.5). By having the user enter a Kerberos userid and password and having the Kerberos server authenticate the user, the application can have reasonable certainty that the user is who

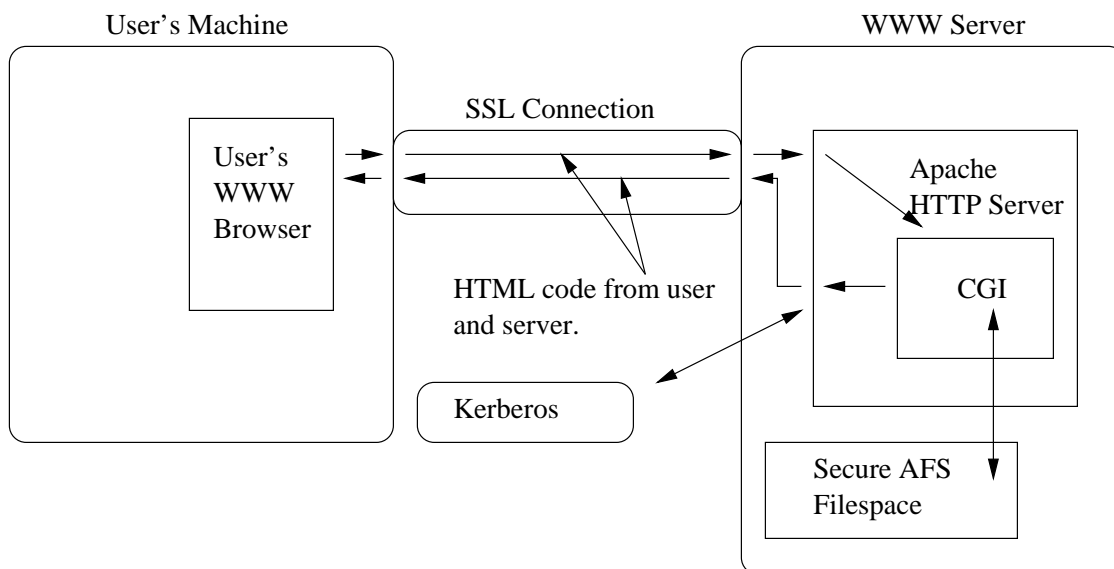


Figure 3.1: New Model of the Application with Security Enhancements.

he claims to be. “Reasonable certainty” is defined as a positive identification based on the authentication method. The computer application can now use this userid with the certainty that it is authentic.

3.2 Setting up an SSL Connection

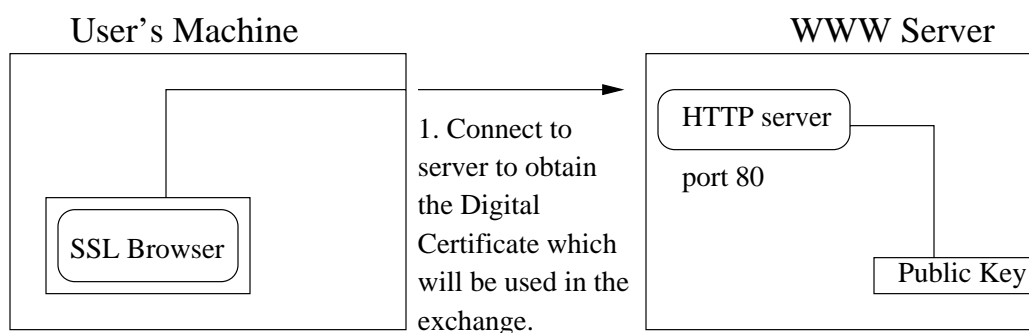


Figure 3.2: Initial Communication.

We assume the user has an SSL¹ capable browser on his machine. The user's browser, as in Figure 3.2, connects to the insecure HTTP server on the remote server.

¹Netscape version 3.0+ and IE version 3.0+ are SSL capable.

During this initial connection, the browser requests the public digital certificate of the server from the insecure HTTP server if that digital certificate does not already exist in the browser's certificate list. The HTTP server locates the public certificate and returns the certificate using the application/x-x509-ca-cert MIME type. The public key certificate is located in unsecured space on the web server. Figure 3.3 illustrates an example of this transaction. Since the public key cannot be used to gain access to secure data, the key is free to travel in clear text over the network connection.

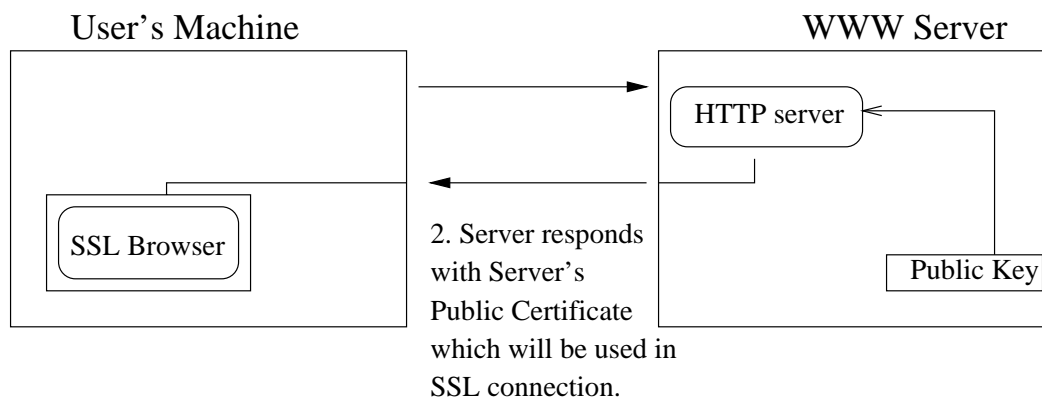


Figure 3.3: Obtaining the Public Certificate.

Once the user's browser has the digital certificate loaded into the browser (see Figure 3.4), the user's browser then begins the SSL handshake (described in Section 2.4.2) with the secure server (see Figure 3.5).

After the initial SSL handshake, some form of authentication must take place. It is this authentication which proves to the secure HTTP server that the user is who he claims to be. Since the authentication happens after the initial SSL handshake, the data the user enters (passwords, userids, etc.) will be encrypted so as not to be passed in clear text over the network. Figure 3.6 shows the steps of the authentication process.

3.3 Uploading, Downloading, and Modifying Data

Once the authentication has been completed, the user can execute CGI programs on the secure HTTP server. Since the HTTP server is a trusted user, the scripts have the privilege to change the data stored in the secure data areas on the server. Figure 3.7 shows the process of executing a CGI program on the HTTP server.

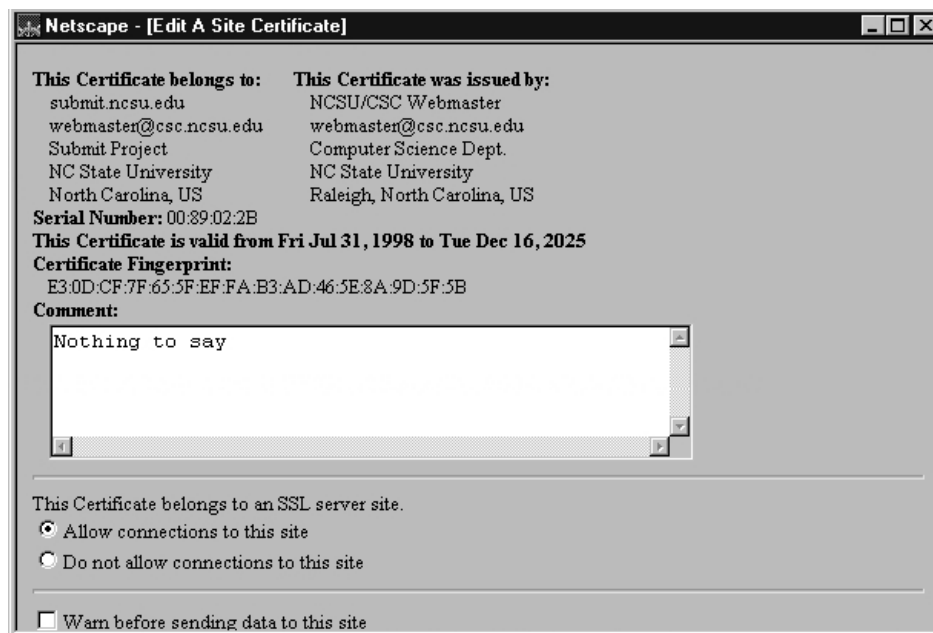
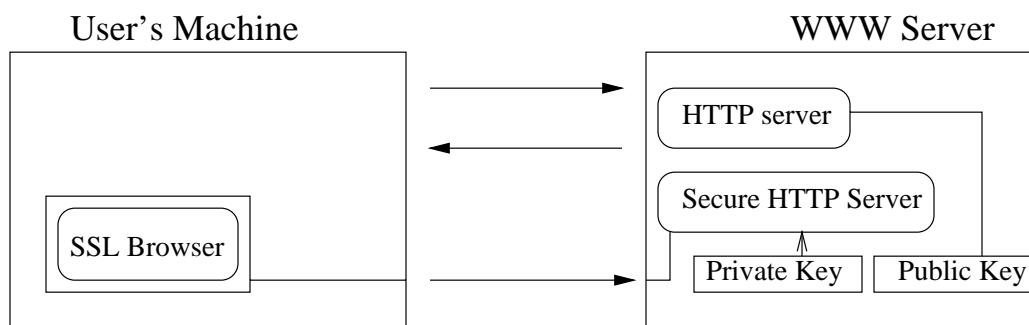


Figure 3.4: Site Certificate.



3. Begin the SSL handshake protocol and forward to port 443 (SSL port).

Figure 3.5: SSL Handshake.

By using the multipart form file upload facility in HTML4, the user can upload or download data from the HTTP server via the CGI scripts. Since this transaction occurs inside of the SSL connection, the data uploaded or download are protected from sniffers.

This model satisfies all of the security requirements by using SSL to encrypt the data, including the userid and password, between the user's browser and the secure HTTP server. The secure HTTP server is able to authenticate the user to the level of certainty

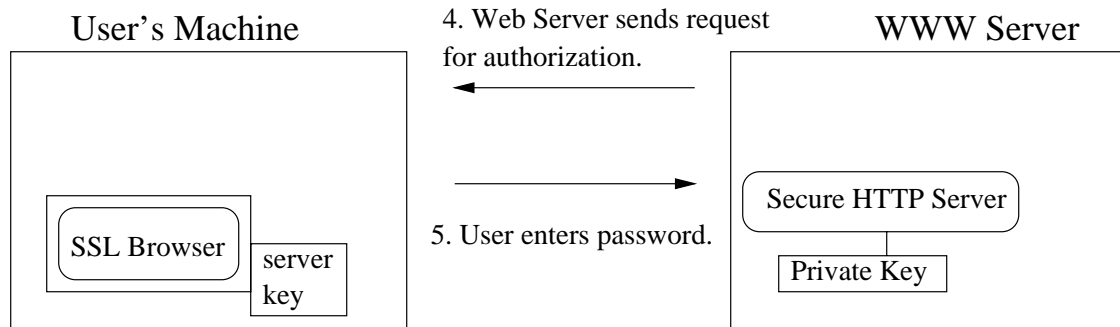


Figure 3.6: Authentication.

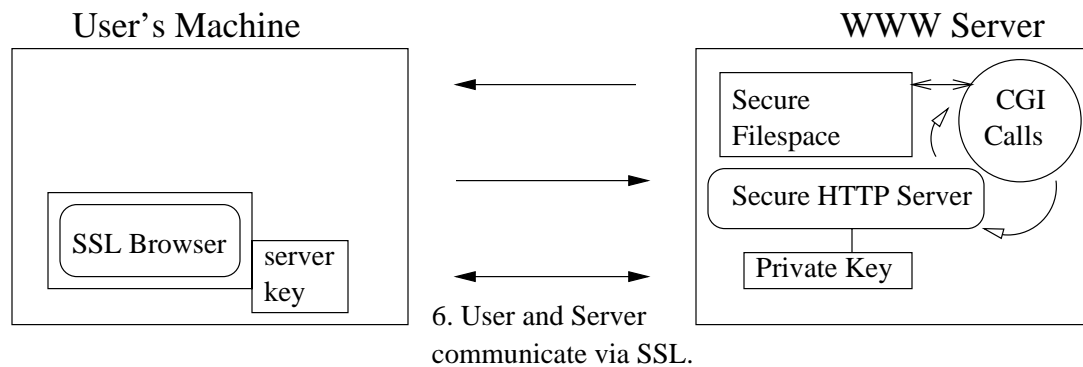


Figure 3.7: CGI Execution on the HTTP Server.

required for most enterprise applications. The HTML4 multipart form data constructs and the CGI programs running on the secure HTTP server allow uploading, downloading, and modification of the data stored on the HTTP server. This is the general software architecture of the prototype system.

3.4 Building a Secure Enterprise Application

In this section we present the detailed steps for creating a secure enterprise application. Our approach requires the following activities:

1. Upgrading the web server
 - (a) Apache web server
 - (b) Kerberos authentication mechanism

- (c) SSLeay
 - (d) Digital certificates
2. Preparing the user's browser
 3. Setting up the CGI application
 - (a) Importing the userid from the web server
 - (b) Adding digests to important form data
 4. Creating IP based authentication

3.4.1 Upgrading the Web Server

Apache Web Server

The web server used in our solution is Apache [13] version 1.3.1. The Apache web server was chosen for its many exceptional features: loadable modules, enhanced logging, ease of setup, and exceptional user support base. Our solution's Apache server is built using `gcc` [28], the GNU compiler, and with Apache's additional modules to provide authentication for Kerberos and SSL support.

Kerberos Authentication Mechanism

Since Eos (the enterprise environment used at NC State) relies on Kerberos as its method of authenticating users, Kerberos is an excellent choice for our solution. In order to add Kerberos authentication to the Apache web server, a module and patch is applied to the Apache source code. The module is `mod_auth_kerb.c` in the Appendix A. `mod_auth_kerb.c` was written by James E. Robinson, III. The Kerberos module allows for the authentication of userids based on a user's Kerberos password. The Kerberos method of authentication is illustrated in Figure 3.8.

The URL request comes in on port 443, the secure web server port. The Uniform Resource Locator (URL) is parsed by the URL processing routines in the Apache server. The base portion of the URL is located in a file called `access.conf` (part of the Apache configuration files). An example of this file is shown in Figure 3.9.

If the URL was `http://submit.ncsu.edu/eos-bin/submit`, then `http://submit.ncsu.edu/eos-bin` would be the basename. The basename path is resolved, using the

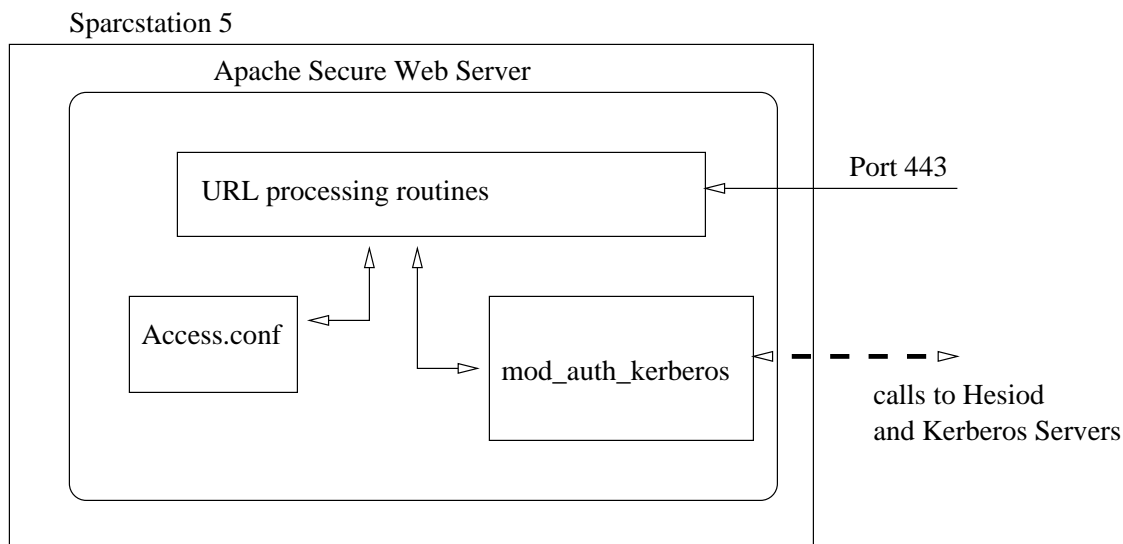


Figure 3.8: Apache Secure Server.

```

<Directory /local/etc/https/eos-bin>
AuthName NCSU Realm
AuthType KerberosV4
<Limit GET PUT POST>
require valid-user
order allow,deny
allow from all
</Limit>
</Directory>

```

Figure 3.9: Kerberos Authentication in access.conf.

Apache URL processing routines, to a local directory on the web server. In this example, the basename resolves to the directory `/local/etc/https/eos-bin`. The directory is searched for in the `access.conf` file. If the directory is located in the `access.conf`, then authentication (in this case KerberosV4) must occur before the URL is resolved. Once the authentication occurs the URL can be processed. In KerberosV4, the user types in his userid and password and is authenticated.

SSLeay

The module used to provide the SSL portion is the “SSLeay” [29] 0.9.0b package. SSLeay is a free implementation of the Netscape SSL. SSLeay implements SSL, versions 2 and 3, TLS version 1, DES, RSA, RC4, IDEA, and Blowfish². Building this package provides the necessary cipher-ware libraries: **libcrypto.a**, **libssl.a**, and **libRSAGlue.a** such that the Apache SSL patch can be used. Ben Laurie provides a patch [17] which can be applied to the standard Apache 1.3.1 version to allow one to build a secure web server. Once the SSLeay package is built and the SSL patch applied to the standard Apache code, the server’s digital certificate must be obtained.

Adding the Digital Certificate

In order for the Apache secure server (created as described in the previous two sections) to operate, it must have a digital certificate. Two choices are available for obtaining this certificate: buying a certificate or creating a certificate. In order to buy a certificate from Verisign, Thawte, or one of the many Certificate Authority (CA) vendors, the administrator operating the Apache server must apply for the certificate, be processed by the vendor (supplying the server id, contact information, and other validation information) and pay for the certificate. The alternative is to create a CA digital certificate, and then create and sign the certificate. SSLeay can be used to first create a CA signing certificate. This low cost approach can be used to generate the same type of digital certificate that CA vendors generate. The steps for generating a digital certificate are given in Appendix C.

After the certificate is created, the certificate is placed in a read-only directory accessible only by the web server. The Apache configuration lines in the **httpd.conf** file are modified to reflect the certificate’s location:

```
SSLCACertificatePath /local/etc/https/SSLconf/conf
SSLCACertificateFile /local/etc/https/SSLconf/conf/httpsd.pem
```

These configuration variables provide the path and location of the digital certificate used by the Apache secure web server.

²DES, RSA, RC4, IDEA, and Blowfish are common ciphers.

3.4.2 Preparing the User's Browser

In order for the user's SSL capable browser to operate, it must have a copy of the digital certificate from the server. If the certificate was purchased from a CA vendor, the certificate is automatically loaded in the user's browser when he first contacts the secure server. This action happens automatically because the user's browser was distributed with a copy of the CA vendors certificate. The SSL capable browser understands that the digital certificate it is receiving from the secure web server is signed by an authority which is already trusts (the CA vendor certificate which is already loaded in the browser). However if the secure server's digital certificate was created using the SSLeay method, the certificate has to be manually loaded to prevent the user from being questioned by the browser each session about the validity of the digital certificate being loaded. One can setup a URL link which allows this certificate download to take place. Figure 3.10 shows an example of one such URL.

```
<A HREF="http://www5.csc.ncsu.edu/CA-cert.der">Certificate</A>
```

Figure 3.10: A URL for Loading a Digital Certificate.

3.4.3 CGI Application Setup

A few modifications must be made to existing CGI programs in order to use out solution: all application programs import the userid from the web server and add digests to important form data.

Importing the Userid

The Apache secure web server provides many different types of information accessible to CGI programs. The userid of the authorized user is one type of information provided. The userid can be imported into the CGI program by means of the environment variable `REMOTE_USER` and is set by the secure web server after the authorization phase. Since a CGI program does not execute until after the authorization phase, the CGI is guaranteed not to be supplied a userid unless the authorization was successful (the program would not be executed if authorization failed).

Adding Digests to Important Form Data

CGI programs often produce forms to gather data from the user. Since CGI is stateless, the form data sent to the user is not retained by the server and exists only within the user's browser. If a user were to modify the data before returning the form to the secure web server, a security breach would occur. In order to prevent data modification, an MD5 digest key is created of each of the form items before being sent to the user's browser.

```
#!/usr/local/bin/perl
use MD5;
$md5 = new MD5;
$md5->add('There are six white wolves',
         'In the dead of the night');
$digest = $md5->digest();
print "Digest is " . unpack("H*", $digest) . "\n";
$md5->reset();
$md5->add('There are six white wolves',
         'In the dead of the Night');
$digest = $md5->digest();
print "Digest is " . unpack("H*", $digest) . "\n";
```

Figure 3.11: mdtest.pl, A Perl Example of MD5.

The MD5 digest algorithm creates a one way digest hash. An example of an MD5 implementation in Perl can be seen in Figure 3.11. In the first part of the Perl program (mdtest.pl), a digest is created using the strings “There are six white wolves” and “In the dead of the night”. The digest produced is “042e73ea45e003caf7a1e77fd7a3ed37”. If a small change is made to the strings (the “n” in night is changed to “N”), the digest produces a completely different digest: “09810ea2e3430502e42036fca023a73e”. One can see how small changes cause big differences in the digest produced.

```
[197] astorath /users/staff/tkl>./mdtest.pl
Digest is 042e73ea45e003caf7a1e77fd7a3ed37
Digest is 09810ea2e3430502e42036fca023a73e
```

Figure 3.12: Output of the Perl MD5 Program.

The digest value can be passed to the user's browser using a hidden text field. A hidden text field is not displayed on the user's browser, but the hidden field is returned with the form data. By applying the MD5 digest algorithm to the returning form data and comparing it with the hidden digest field, the CGI program would be aware of any data falsification by the user.

3.4.4 Creating IP Based Authentication

AFS provides the secure file space in our solution architecture. IP authentication is the key to how the secure file space is accessed. AFS provides a mechanism for allowing a machine to become the equivalent of a user. This mechanism is called IP authentication. A system administrator who has administrative privileges for the AFS realm first creates a user who has the same userid as the IP address of the web server. The web server user created can then be added to ACL groups for a directory and given permissions to read, write, or modify data to that directory.

Chapter 4

Submit

In this section we describe our prototype application, submitV2, which supports online homework submission in our enterprise environment (Eos).

4.1 Submit Background

Submit is a homework submission program originally written by Dan Sharpe and Tim Lowman in early 1992. It was later rewritten and revised by Tim Lowman in 1997, 1998, and 1999. The goal of the work was to write a secure, command-line, homework submission program for the Eos environment. Submit operated by means of AFS ACL protection groups and a collection of “lockers” (dynamically mountable directories). Each locker contains a master ACL group, `tkl:CSCXXX/admin/sup`, which is granted full access to the locker. The locker contains a directory called “LEC”, which contains numbered directories for each section of the course. A “SUBMITTED” directory is located inside of each of these numbered directories. Each SUBMITTED directory contains a directory named for each student in that particular section of the course. The SUBMITTED directory is accessible only by the course administrators, those who are members of `tkl:CSCXXX/admin/sup`, the course TAs, and the students of that particular section. Access by the TAs and students is controlled for each section by means of the `tkl:CSCXXX/YYY/tas` and `tkl:CSCXXX/YYY/students` ACL groups, respectively. Using a command line interface, students specify the files they wish to submit for a particular homework assignment. If the student’s group is given read/write access to the directory which matches their userid, then the submit program copies the specified files from the user’s directory to the SUBMITTED area. AFS provides the sole

mechanism for authorizing users to submit homework assignments.

Since AFS is used as the method of authentication, many AFS groups must be generated each semester and updated as students add or drop classes. The number of AFS protection groups usually grows to such a large number that creating and maintaining these groups requires over one thousand groups. AFS ACL servers [Protection Database Servers (PTS)] soon become swamped by the simple act of destroying a semester's groups. Thus a new method, utilizing fewer ACL groups, was necessary.

Submit operates on machines where the C source code can be compiled. While most UNIX machines fall into this category, Windows NT machines and Macintosh computers do not. This limitation prevents user homework submissions from non-UNIX machines. The increasing number of students using non-UNIX machines mandates the quest for a solution to multi-platform submissions.

NC State teaches many Video Based Education (VBE) courses. The courses are taught by way of taped classroom instruction and other multimedia tools: video whiteboards, online conferencing, and WWW based instruction. Submit, being an application which is local to the Eos system, is not available for VBE students. The homework submission solution must allow students at remote sites to submit, using many different machine architectures. Our prototype submitV2 (version 2.0 of Submit) is a tool which solves these problems.

4.2 Submit Implementation

Our prototype web server host machine is a Sparcstation 5. The Sparcstation's operating system, Solaris, provides the rich environment needed. The configuration of the Sparcstation host is shown in Figure 4.1.

The prototype system's web server is a secure Apache web server built according to the methods outlined in Chapter 3. As mentioned in Section 3.4.1, `mod_auth_kerb.c` is the module which provides authentication for the SSL connection. Figure 4.2 shows a basic diagram of the authentication process via this module.

```

SunOS Release 5.5.1 Version Generic [UNIX(R) System V Release 4.0]
Copyright (c) 1983-1996, Sun Microsystems, Inc.
vac: enabled in write through mode
cpu0: FMI,MB86904 (mid 0 impl 0x0 ver 0x4 clock 70 MHz)
mem = 65536K (0x4000000)
avail mem = 61558784
Ethernet address = 8:0:20:71:8f:a0
root nexus = SUNW,SPARCstation-5

```

Figure 4.1: Sparcstation 5 Machine Configuration.

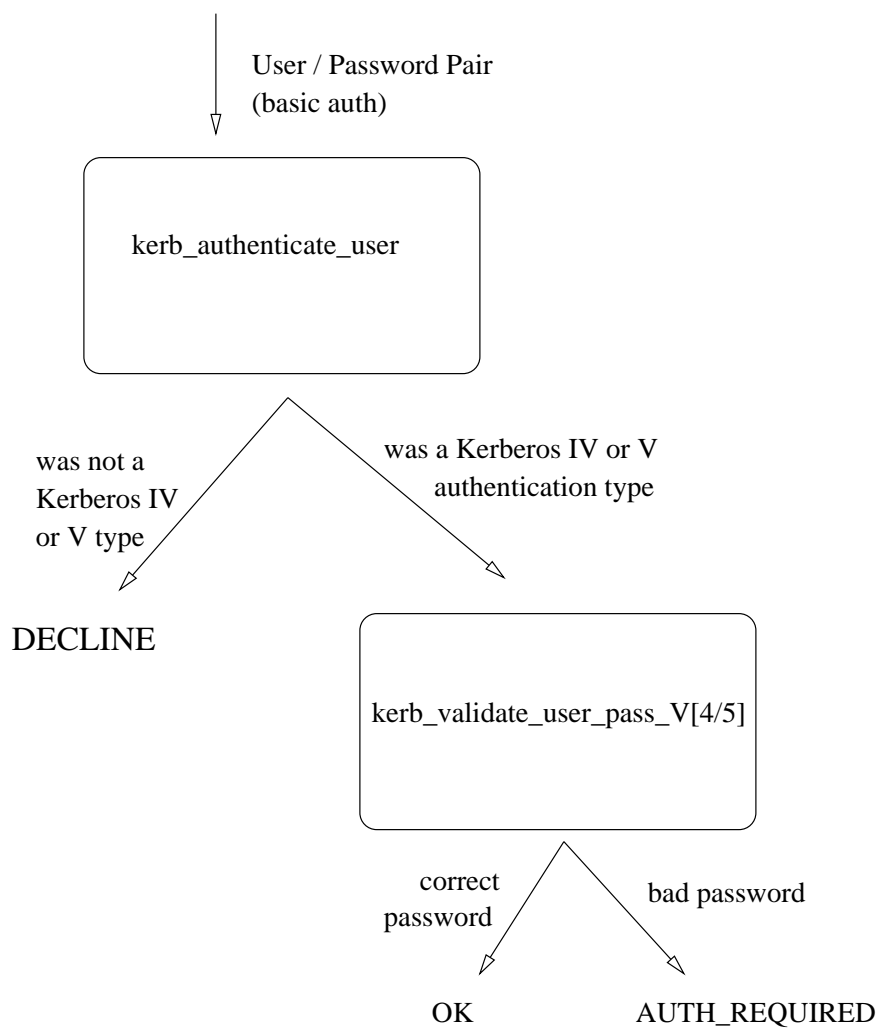


Figure 4.2: Apache Secure Server.

When the SSL handshake initiates, the userid/password pair enters the `kerb_authenticate_user` function. This function retrieves grplist data from the Hesiod¹ server (in case additional grplist authentication is required). The authentication type is checked to see if it is Kerberos related. If the authentication type is not Kerberos, `kerb_authenticate_user` returns `DECLINED`. Once the userid/password pair are processed into Kerberos records, they are passed to `kerb_validate_user_pass_V[4,5]` (4 being the Kerberos IV version and 5 being the Kerberos V version). The password is verified by attempting to obtain a Kerberos ticket granting ticket in the routine `kerb_validate_user_pass_V4`. If the routine returns `OK`, the password was able to gain a ticket. The routine returns `AUTH_REQUIRED` if the ticket granting ticket could not be obtained. If no ticket was obtained (the Kerberos routine returned `DECLINED`), a web server code of 401 is returned with the `AUTH_REQUIRED` code showing that authentication is required to run/view this URL.

The prototype application (`submitV2`) is written in Perl. Perl was chosen because of the ease of coding CGI applications, its rich support of string processing, and its ability to add additional support modules to the core distribution. The additional modules needed for `submitV2`, which is listed in Appendix F, are [these modules are available from the Comprehensive Perl Archive Network (CPAN)] [9]:

1. AFS - a custom written AFS module adapted from Roland Schemers's AFS code
2. CGI.pm-2.42 - CGI generation modules
3. Crypt-DES-1.0 - Crypt-DES encryption algorithms
4. Data-Dumper-2.09 - Data Dumper
5. Getopt-Tabular-0.2 - Tabular Getopt
6. GetoptLong-2.17 - Getopt long
7. HTML-Parser-2.20 - HTML Parser
8. HTML-TableLayout-1.001006 - HTML Table Layout
9. Hesiod-1.0 - Hesiod Interface

¹Hesiod is a modified BIND server which provides user information such as password, mail drops, or group lists in addition to BIND data.

10. IO-1.20 - Input Output addition
11. IO-File-Multi-1.01 - Multi-file I/O addition
12. IO-stringy-1.203 - String processing I/O addition
13. LockFile-Simple-0.1 - Lock file support
14. Log-Logger-1.01 - System logging support
15. Logfile-0.202 - Log file processing support
16. MD5-1.7 - MD5 digest support
17. MIME-Base64-2.06 - MIME support for BASE64 encoding
18. MIME-tools-4.121 - MIME tool support
19. MailTools-1.11 - Mailer packages
20. PGP-Sign-0.09 - PGP support
21. Schedule-At-1.02 - Scheduler support
22. SyslogScan-0.32 - Syslog processing support
23. Time-HiRes-01.18 - Timing support
24. Time-modules-98.060902 - Time function modules
25. libnet-1.0605 - Network/Internet support
26. libwww-perl-5.35 - WWW support (LWP)
27. sybperl-2.09 - Sybase Support

The basic algorithm of the submitV2 program is shown in Figure 4.3. Since CGI programs are stateless, submitV2 uses a series of hidden variables to store information needed between CGI executions. It also creates an MD5 digest which is used to prevent form data from being modified by the user and re-submitted to the server. The basic algorithm is as follows. The initial submitV2 form is displayed when the user connects

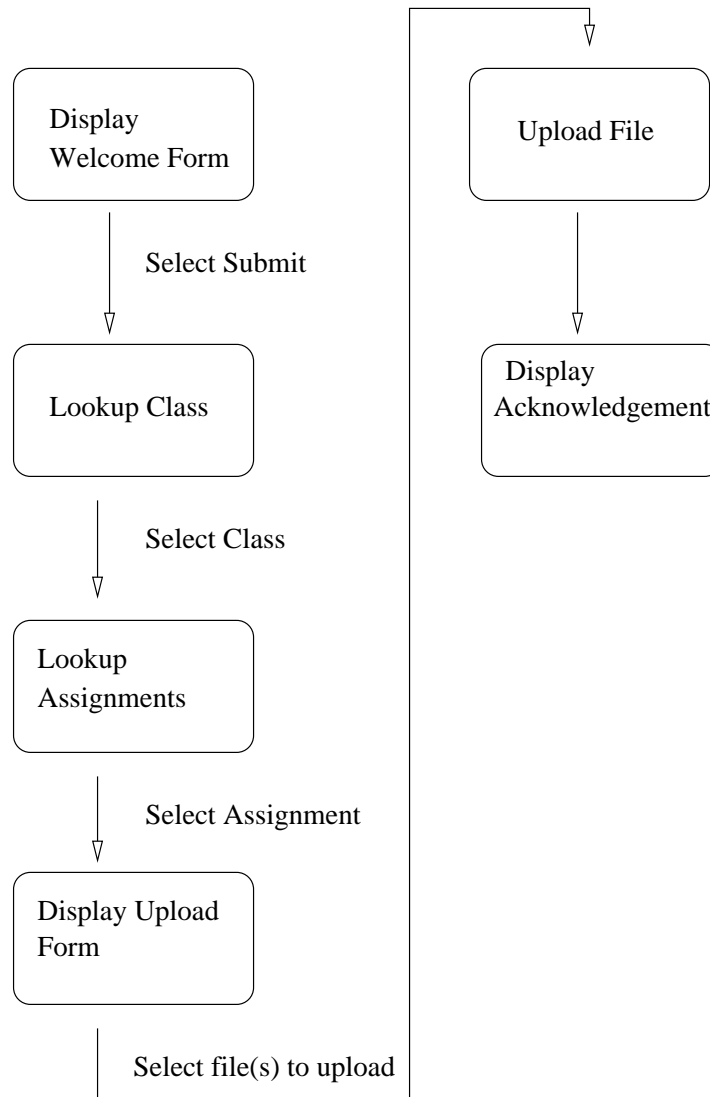


Figure 4.3: Algorithm for Submit.

to the secure server. The user selects the “submit” facility triggering a Sybase² query to obtain a class list for the student. The returned classes are formatted using HTML4 code to display a pull-down menu, and the dynamically created HTML4 document is sent to the user.

The user then selects the class for which he wishes to submit his homework assignment and clicks on the “submit” button. This button triggers another submitV2 CGI

²The CSC department maintains a Sybase database of student information including courses, student data, and email addresses. This information is available to the submitV2 CGI program via Syperl calls.

call and the form data are passed to submitV2. SubmitV2 processes the data from the form, the class the user selected. SubmitV2 retrieves all the possible assignments available for that class and returns another dynamically created HTML4 form. This time the form has two hidden variables “key” which contains an MD5 digest of the course selected and “course” which contains the actual course selected. The form data also have radio-group buttons listing the possible homework assignments.

The user can click the radio-group button beside the assigned homework and then on the “submit” button. This button triggers another call to submitV2. SubmitV2 receives the form data (with the hidden fields) and the selected homework assignment. To verify the course, passed back in the hidden field, was not altered, submitV2 computes a MD5 digest based on that value. This computed value is compared with the MD5 digest passed as a hidden field. If the values differ, submitV2 knows the form data were modified and can take appropriate action. SubmitV2 checks to see if the homework assignment selected is active. If the assignment is not active, a message that the homework assignment is not valid is returned to the user. If the assignment is valid, a new form is dynamically generated. The new form contains the hidden variables “courses” (the course), “assignments” (the assignment), and “key” (the new MD5 digest key comprised of the “courses” string and the “assignments” string). The form also contains a multi-part/form-data file selection dialogue box. The file selection box will be used to select the file the user wishes to upload.

Once the user has selected the file to upload, the “submit” button is pressed. The click triggers another submitV2 call. SubmitV2 creates a new MD5 digest of the course and assignment names and compares it against the MD5 digest key received with the form data. Again if these digests fail to match, the user is displayed a message that the form data has been modified. If the digests match, the file is uploaded and placed in the homework submission area for that course.

The database packages used for this prototype is Sybase(tm) SQL server. Information is loaded into the database by a series of Sybperl scripts. The database is accessed via the Sybperl module mentioned in earlier. SQL queries occur via a common Sybase account. Multiple queries can happen simultaneously. The Sybase server regenerates its data each morning from the most recent information provided by NC State’s Academic Computing Services. Information held in this database includes the student’s classes, preferred email addresses, current course standing: Freshman, Sophomore, Junior, etc, and the course data on which courses are being offered. Several indexed tables exist which allow faster searches

of the 245,000 entries. Figure 4.4 shows a listing of table names and types created for this prototype. Additional table information is include in Appendix D.

Name	Owner	Object_type
addr	dbo	user table
enroll	dbo	user table
people	dbo	user table
stu_addr	dbo	user table
stu_crs	dbo	user table
user_crs	dbo	user table
userinfo	dbo	user table

Figure 4.4: Sybase Tables.

Another type of database used by submitV2 is flat-file text files holding information about each assignment. These files hold a list of active homework assignments, start and end assignment times, and additional information for late homework submissions. Figure 4.5 shows an example flat-file database.

```
hw1:0:909991439:001
hw1:0:999999999:tk1
hw2:0:988888888:001
```

Figure 4.5: A Flat-file Database.

SubmitV2 modifies these flat files in order to create the information that the program will later use to determine if an assignment is open for submission, is available for late submission, needs to be turned on, or needs to be turned off.

The submitV2 prototype resolves the problems of securely submitting homework assignments from remote sites, from different machine architectures, and requires fewer AFS ACL groups. A submitV2 production system is currently under discussion by several NC State departments.

Chapter 5

Conclusions

5.1 Conclusions

By utilizing the secure Apache web server, SSL encoding, Kerberos authentication, AFS's IP based file protection, and Perl as the CGI language, a secure application prototype can be shown to operate in a manner which protects the user's information in a large enterprise environment such as Eos. Users are protected from each other. User information passed over a common network is protected from sniffing. By using the HTML transmission method, we have enabled anyone with a SSL capable browser to use this prototype. The AFS portion used for file security could be substituted with NFS file security and a setuid program. Likewise, Kerberos as an authentication mechanism could be substituted by another authentication protocol which may be available at that site. The cost of this setup was quite small: our CA key was self generated and all the software used is in the public domain with the exception of Sybase. Another public domain database such as MySQL could be substituted into this scenario with very little modification. Our secure model is appropriate for many other application: online book trading, online advising, or a multitude of others which require user authentication and security in a large enterprise environment.

Bibliography

- [1] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, November 1996. <http://www.cis.ohio-state.edu/rfc/rfc1945.txt> (28 October 1998).
- [2] Tim Berners-Lee. The original proposal of the WWW, May 1990. <http://www.w3c.org/History/1989/proposal.html> (28 October 1998).
- [3] Bill Bryant. Designing an authentication system: A dialogue in four scenes. Technical report, MIT: Project Athena, February 1988.
- [4] CCITT. Recommendation X.500: *The Directory Overview of Concepts, Models, and Services*. Technical report, CCITT, 1988.
- [5] CCITT. Recommendation X.509: *The Directory Authentication Framework*. Technical report, CCITT, 1988.
- [6] CERT. Security for a public web site, September 1997. <http://www.cert.org/security-improvement/modules/m02.html> (8 November 1998).
- [7] William Cheswick, Warwick Ford, and James Gosling. How computer security works. *Scientific American*, 279(4):108, October 1998.
- [8] Netscape Corporation. Netscape SSL 2.0 certificate format. Technical report, Netscape Corporation, 1997.
- [9] CPAN. Comprehensive Perl archive network (CPAN), March 1999. <ftp://ftp.csc.ncsu.edu/pub/CPAN/SITES.html> (28 October 1998).

- [10] N. Freed and N. Borenstein. Multipurpose Internet mail extensions (MIME) part one: Format of Internet message bodies, November 1996. <http://www.cis.ohio-state.edu/rfc/rfc2045.txt> (28 October 1998).
- [11] N. Freed and N. Borenstein. Multipurpose Internet mail extensions (MIME) part two: Media types, November 1996. <http://www.cis.ohio-state.edu/rfc/rfc2046.txt> (28 October 1998).
- [12] A. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. Technical report, Netscape Communications, November 1996.
- [13] Apache Group. Apache project, October 1998. <http://www.apache.org/> (28 October 1998).
- [14] John Hughes. Brief comparison of SSL, S/MIME, and IPSEC security technologies, February 1998. http://www.entegrity.com/papers/WHITE_PAPER_4_SSL-SMIME.htm (3 March 1999).
- [15] I/O Software Inc. I/O Software – biometrics explained, September 1998. <http://www.iosoftware.com/about/biometrics.htm> (3 March 1999).
- [16] David Kahn. *Seizing the Enigma: The race to break the German U-boat codes, 1939-1943*. Houghton Mifflin Co., Boston, 1991.
- [17] Ben Laurie. Apache-SSL, March 1998. <https://www.apache-ssl.org> (3 March 1999).
- [18] Tim Lowman. HTML tutorial, May 1996. http://www.csc.ncsu.edu/csc_info/csc251/www/tutorial/ (28 October 1998).
- [19] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT: Project Athena Technical Plan, October 1988.
- [20] NCSA. Common gateway interface, October 1996. <http://hoohoo.ncsa.uiuc.edu/cgi/> (28 October 1998).
- [21] E. Nebel and L. Masinter. Form-based file upload in HTML, November 1995. <http://www.cis.ohio-state.edu/rfc/rfc1867.txt> (28 October 1998).

- [22] Dave Raggett. Raggett's 10 minute guide to HTML, July 1998. <http://www.w3c.org/MarkUp/Guide/> (28 October 1998).
- [23] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.0 specification, April 1998. <http://www.w3c.org/TR/1998/REC-html40/> (28 October 1998).
- [24] R. Riest. The MD5 message-digest algorithm, April 1992. <http://www.cis.ohio-state.edu/rfc/rfc1321.txt> (28 October 1998).
- [25] NCSA HTTPd Development Team. The NCSA HTTPd home page, January 1998. <http://hoochoo.ncsa.uiuc.edu/> (29 October 1998).
- [26] Transarc Corporation. AFS installation guide. Technical report, Transarc Corporation, February 1991.
- [27] North Carolina State University. Security for a public web site, February 1999. <http://www.ncsu.edu/wrap> (8 November 1998).
- [28] The GNU Webmasters. GNU's not Unix! - the GNU project and the Free Software Foundation (FSF), September 1998. <http://www.gnu.org/> (8 November 1998).
- [29] Eric Young. SSLeay: SSL and supporting libraries, June 1998. <ftp://ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL/> (28 October 1998).

Appendices

Appendix A

Apache Kerberos Module

`mod_auth_kerb.c` is an Apache module which performs Kerberos authentication.

```

/*
 * James E. Robinson, III <jamesncstate.net>
 *
 * Source and Documentation can be found at:
 * http://www.ncstate.net/nts/research/software/mod_auth_kerb/
 *
 * $Id: mod_auth_kerb.c,v 3.6 1998/09/01 00:34:07 jero bins Exp $
 *
 * --
 * James E. Robinson, III | jamesncstate.net | Lead Systems Programmer
 * NC State University | NCState.Net | http://www.ncstate.net/
 * Information Technology | PGP key at http://www.ncstate.net/james/pgp/
 *
 */

/*
 * Further modifications to this code done at the National Center
 * for Supercomputing Applications by Von Welch <vwelchncsa.edu>
 *
 * And also Modifications for sending back WWW-Authenticate headers on
 * each 401 by Mark Mentovai <markmozienet.com>
 */

```

```

/* =====
* Copyright (c) 1996-1998 The Apache Group. All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
* notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in
* the documentation and/or other materials provided with the
* distribution.
*
* 3. All advertising materials mentioning features or use of this
* software must display the following acknowledgment:
* "This product includes software developed by the Apache Group
* for use in the Apache HTTP server project (http://www.apache.org/)."http://www.apache.org/)."

```

```

* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Group and was originally based
* on public domain software written at the National Center for
* Supercomputing Applications, University of Illinois, Urbana-Champaign.
* For more information on the Apache Group and the Apache HTTP server
* project, please see <http://www.apache.org/>.
*
*/

/*
* Portions developed at the National Center for Supercomputing
* Applications at the University of Illinois at Urbana-Champaign.
*/

/* Ouch, conflicting header file. Make sure ssl des.h is ignored. */
#define HEADER_DES_H

#ifndef APACHE_SSL
#undef APACHE_SSL
#endif /* APACHE_SSL */

#include "httpd.h"
#include "http_config.h"
#include "http_core.h"
#include "http_log.h"
#include "http_protocol.h

/*
* Prevent warning about closesocket redefinition (Apache's
* ap_config.h and Kerberos' cc-unix.h both define it as close)
*/
#ifdef closesocket
#undef closesocket
#endif /* closesocket */

#ifdef HESIOD
#include <hesiod.h>

```

```

#endif /* HESIOD */

#if (defined(DUAL_AUTH)) && (! defined(KRB5))
#define KRB5
#endif

#if (defined(DUAL_AUTH)) && (! defined(KRB4))
#define KRB4
#endif

#ifdef KRB5
#include "krb5.h"
#endif

#ifdef KRB4
#include "krb.h"
#include "des.h"
#endif

#ifdef KRB5
#define KRB5_DEFAULT_OPTIONS 0

#ifndef KRB5_SAVE_CREDENTIALS
#define KRB5_DEFAULT_LIFE    60 * 5 /* 5 minutes */
#else
#define KRB5_DEFAULT_LIFE    60 * 30 /* 30 minutes */
#endif

krb5_data tgtname = {
    0,
    KRB5_TGS_NAME_SIZE,
    KRB5_TGS_NAME
};

char k5_srvtab[MAX_STRING_LEN] = "";
#endif

#ifdef KRB5_SAVE_CREDENTIALS
/* Are we saving our credentials? */
static int krb5_save_credentials;

```



```

#endif

#ifndef MAX_KDATA_LEN
#define MAX_KDATA_LEN 2048 /* Biggest Kerberos passwd string */
#endif

/* make this the default */
#ifdef KRB_DEF_REALM
#ifdef KRB_REALM
#undef KRB_REALM
#endif /* KRB_REALM */
#define KRB_REALM KRB_DEF_REALM
#endif /* KRB_DEF_REALM */

/* default location for srvtab file */
#ifndef KRB_V4_SRVTAB
#define KRB_V4_SRVTAB KEYFILE
#endif

#ifdef DEFAULT_TKT_LIFE
#undef DEFAULT_TKT_LIFE
#define DEFAULT_TKT_LIFE 1 /* default tkt life to 5 mins */
#endif

#ifdef KRB4
/* our httpd server kerberos realm */
static char realm[REALM_SZ];
#endif

/* forward declaration, see defination at end of file */
module kerb_auth_module;

/* configuration record defination*/
typedef struct {
    char *auth_def_krb_realm;
} kerb_auth_config_rec;

/* map the valid commands to the config record */
/* we allow the user to set AuthRealm in the dir config */
command_rec kerb_auth_cmds[] = {

```

```

    { "KrbAuthRealm", ap_set_string_slot,
      (void*)XtOffsetOf(kerb_auth_config_rec, auth_def_krb_realm),
      OR_AUTHCFG, TAKE1, "Kerberos realm to associate with users." },
    { NULL }
};

/* Tables for converting binary values to and from hexadecimal */

static char hex[] = "0123456789abcdef";

static char dec[256] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 0 - 15 */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 16 - 37 */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* ' ' - '/' */
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0, /* '0' - '9' */
    0,10,11,12,13,14,15, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* '*' */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 'P' - '_' */
    0,10,11,12,13,14,15, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* "' - 'o' */
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 'p' - DEL */
};

/*****
 * kdata_to_str - convert 8-bit char array to ascii string
 *
 * Accepts:  input array and length
 * Returns:  a pointer to the result, or null pointer on malloc failure
 *          The caller is responsible for freeing the returned value.
 *****/

static char *kdata_to_str(char *in_data, int length)
{
    char *result, *p;
    int i;

    p = result = malloc(length*2+1);

    if (!result) {
        return (char *) NULL;
    }
}

```

```

    for (i=0; i < length; i++) {
        *p++ = hex[(in_data[i]>>4)&0xf];
        *p++ = hex[(in_data[i]&0xf)];
    }

    *p++ = '\0';

    return result;
}

/*****
 * str_to_kdata - Converts ascii string to a (binary) char array
 *
 * Accepts: string to convert
 *          pointer to output array
 * Returns: length of output array, NULL on failure
 *
 *****/
static int str_to_kdata(const char *in_str, char *out_str)
{
    int inlen, outlen;

    inlen = strlen(in_str);
    if (inlen & 1) {
        return NULL; /* must be even number, in this scheme */
    }

    inlen /= 2;

    if (inlen > MAX_KDATA_LEN) {
        return NULL;
    }

    for (outlen=0; *in_str; outlen++, in_str += 2) {
        out_str[outlen] = (dec[in_str[0]]<<4) + dec[in_str[1]];
    }

    return outlen;
}

```

```

/*****
 * kerb_get_server_realm - retrieve the realm information for the server
 *
 * Accepts: request record
 * Returns: sets the global variable 'realm' to the server's realm
 *
 *****/
#ifdef KRB4
static void kerb_get_server_realm( request_rec *r)
{
    /* get the config information */
    kerb_auth_config_rec *sec =
        (kerb_auth_config_rec *)ap_get_module_config(r->per_dir_config,
            &kerb_auth_module);

    /*
     * did the user specify a different realm than compiled in default
     * for the server?
     */
    if(sec->auth_def_krb_realm) {
        strncpy(realm, sec->auth_def_krb_realm, REALM_SZ);
    } else {
        /* if not, try and get the one from the system */
        if (kerb_get_lrealm(realm, 1) != KSUCCESS) {
            /* else, result to the compiled in default */
            strncpy(realm, KRB_REALM, REALM_SZ);
        }
    }

    return;
}
#endif

/*****
 * kerb_validate_user_pass - validate user with principal and passwd
 *
 * Accepts: request record and sent password
 * Returns: OK, DECLINED, AUTH_REQUIRED or SERVER_ERROR
 *
 *****/

```

```

*****/
#ifdef KRB4
static int kerb_validate_user_pass_V4( request_rec *r, const char *sent_pw )
{
    char errstr[MAX_STRING_LEN];
    int krbval;
    int kpass_ok = 0; /* not allowed until we say so */
    char krb_tkt_file[MAXPATHLEN - 1];

    /* need to be sure ticket file is set to something reasonable
     * since multiple processes could be authenticating at once
     * better make sure files don't conflict
     */
    sprintf(krb_tkt_file, "/tmp/apache_tkt_%ld", (long)getpid());
    krb_set_tkt_string(krb_tkt_file);

    /* see if they are who they say they are */
    krbval = krb_get_pw_in_tkt(r->connection->user, "", realm,
        "krbtgt", realm, DEFAULT_TKT_LIFE, (char *)sent_pw);

    /* destroy all the evidence */
    /* well, it is still in memory on the browser. . */
    dest_tkt();

    /* see what kerberos told us */
    switch (krbval) {
        case INTK_OK:
        case INTK_W_NOTALL:
            kpass_ok = 1;
            break;

        default:
            sprintf(errstr, "Kerberos error: %s", krb_err_txt[krbval]);

            /* first, the error log */
            ap_log_reason (errstr, r->uri, r);
            break;
    }

    /* are they ok? */

```

```

if ( kpass_ok ) {
    return OK;
}

/* if not, let somebody else take care of it */
return AUTH_REQUIRED;
}
#endif
#ifdef KRB5
static int kerb_validate_user_pass_V5( request_rec *r, const char *sent_pw )
{
    krb5_context    kcontext;
    krb5_ccache     ccache = NULL;
    krb5_principal  me;
    krb5_principal  server;
    krb5_creds      my_creds;
    krb5_timestamp  now;
    long            lifetime = KRB5_DEFAULT_LIFE;
    int             options = KRB5_DEFAULT_OPTIONS;
    krb5_preauthtype *preauth = NULL;
    krb5_error_code code;

    char            *client_name;
    char            errstr[MAX_STRING_LEN];

    krb5_init_context(&kcontext);
    krb5_init_ets(kcontext);

    if (code = krb5_parse_name(kcontext, r->connection->user, &me)) {
        sprintf(errstr, "krb5_parse_name(): Parsing name %s - %s (%d)",
            r->connection->user, error_message(code), code);
        ap_log_reason (errstr, r->uri, r);
        return SERVER_ERROR;
    }

    if (code = krb5_unparse_name(kcontext, me, &client_name)) {
        sprintf(errstr, "krb5_unparse_name(): Unparsing name - %s (%d)",
            error_message(code), code);
        ap_log_reason (errstr, r->uri, r);
        return SERVER_ERROR;
    }
}

```

```

}

#ifdef KRB5_SAVE_CREDENTIALS
/*
 * krb5_save_credentials is set in kerb_validate_user_pass_V5
 */
if (krb5_save_credentials) {
    /*
     * We want to save the credentials somewhere a CGI script can find them.
     * I've choosen to put them in /tmp/krb5cc_<username>. Obviously this
     * holds some problems if <username> contains slashes (/), but I'm
     * not sure what a better solution is at this point.
     */

    char        cache_name[MAX_STRING_LEN];

    /*
     * We want to check the username to make sure there is no funny bussiness
     * going on.
     */
    if (strchr(r->connection->user, '.') ||
        strchr(r->connection->user, '/')) {

        sprintf(errstr, "Checking krb5 password: Bad character in username \"%s\".",
            r->connection->user);
        ap_log_reason (errstr, r->uri, r);
        return SERVER_ERROR;
    }

    sprintf(cache_name, "/tmp/krb5cc_%s", r->connection->user);

    if (code = krb5_cc_resolve(kcontext, cache_name, &ccache)) {
        sprintf(errstr,
            "krb5_cc_resolve(%s): Getting credential cache - %s (%d)",
            cache_name, error_message(code), code);
        ap_log_reason (errstr, r->uri, r);
        return SERVER_ERROR;
    }

    if (code = krb5_cc_initialize(kcontext, ccache, me)) {

```

```

    sprintf(errstr,
            "krb5_cc_initialize(): Initializing credential cache - %s (%d)",
            error_message(code), code);
    ap_log_reason (errstr, r->uri, r);
    return SERVER_ERROR;
}

/*
 * XXX This should really be an option somewhere.
 */
options |= KDC_OPT_FORWARDABLE;
}

/*
 * If we're not saving the credentials, then we want to leave ccache == NULL
 */

#endif /* KRB5_SAVE_CREDENTIALS */

memset((char *)&my_creds, 0, sizeof(my_creds));

my_creds.client = me;

if (code = krb5_build_principal_ext(kcontext, &server,
    krb5_princ_realm(kcontext, me)->length,
    krb5_princ_realm(kcontext, me)->data,
    tgtname.length, tgtname.data,
    krb5_princ_realm(kcontext, me)->length,
    krb5_princ_realm(kcontext, me)->data,
    0)) {
    sprintf(errstr, "krb5_build_principal_ext(): Building server name - %s (%d)",
            error_message(code), code);
    ap_log_reason (errstr, r->uri, r);
    return SERVER_ERROR;
}

my_creds.server = server;

if (code = krb5_timeofday(kcontext, &now)) {
    sprintf(errstr, "krb5_timeofday(): Getting time of day - error %s (%d)",

```



```

        error_message(code), code);
    ap_log_reason (errstr, r->uri, r);
    return SERVER_ERROR;
}

my_creds.times.starttime = 0;
my_creds.times.endtime = now + lifetime;
my_creds.times.renew_till = 0;

code = krb5_get_in_tkt_with_password(kcontext,
    options, 0, NULL, preauth, sent_pw, ccache, &my_creds, 0);

/* This frees the principals as well */
krb5_free_cred_contents(kcontext, &my_creds);

#ifdef KRB5_SAVE_CREDENTIALS
    if (krb5_save_credentials) {
        if (code == 0) {
            krb5_cc_close(kcontext, ccache);
        } else {
            krb5_cc_destroy(kcontext, ccache);
        }
    }
#endif /* KRB5_SAVE_CREDENTIALS */

    if (code) {
        switch(code) {
            case KRB5KRB_AP_ERR_BAD_INTEGRITY:
                sprintf(errstr,
                    "krb5_get_in_tkt_with_password(): User \"%s\" - Password incorrect",
                    r->connection->user);
                break;

            case KRB5KDC_ERR_C_PRINCIPAL_UNKNOWN:
                sprintf(errstr,
                    "krb5_get_in_tkt_with_password(): Unknown user \"%s\"",
                    r->connection->user);
                break;

            default:

```

```

        sprintf(errstr,
                "krb5_get_in_tkt_with_password(): User \"%s\" - %s (%d)",
                r->connection->user, error_message(code), code);
        break;
    }
    ap_log_reason (errstr, r->uri, r);

    return AUTH_REQUIRED;
}

return OK;
}
#endif

/*****
 * kerb_validate_ticket - validate the tkt sent by the browser
 *
 * Accepts: request record
 * Returns: OK, DECLINED, AUTH_REQUIRED or SERVER_ERROR
 *
 *****/
#ifdef KRB4
static int kerb_validate_ticket_V4( request_rec *r )
{
    int krbval;
    char errstr[MAX_STRING_LEN];
    KTEXT_ST authent;
    char instance[INST_SZ];
    static AUTH_DAT kdata;
    char k4_srvtab[MAX_STRING_LEN];
    des_cblock session;          /* Our session key */
    des_key_schedule schedule;   /* Schedule for our session key */
    char *type, *p;
    const char *sent_pw, *auth_line;

    /* get the auth line the user sent us */
    auth_line = ap_table_get (r->headers_in, "Authorization");

    /* did they send us the information required? */
    if ( auth_line == NULL ) {

```

```

    /* tell the client what to expect */
    ap_table_set (r->err_headers_out, "WWW-Authenticate", "KerberosV4");
    return AUTH_REQUIRED;
}

/* get the auth type sent to us */
type = ap_getword(r->pool, &auth_line, ' ');

/*
 * These two fields in the connection record are used elsewhere
 * in the Apache modules. (check_access)
 * They really appreciate them being set to useful values. :)
 */
r->connection->user = ap_getword(r->pool, &auth_line, ' ');
r->connection->ap_auth_type = "KerberosV4";

sent_pw = ap_getword(r->pool, &auth_line, '\0');

if ((authent.length = str_to_kdata(sent_pw, (char *)authent.dat)) == NULL) {
    sprintf(errstr, "Invalid Kerberos authenticator");
    /* log it to the error file */
    ap_log_reason (errstr, r->uri, r);
    /* let somebody else take care of it */
    return AUTH_REQUIRED;
}

/* configure the srvtab file location */
strncpy(k4_srvtab, KRB_V4_SRVTAB, MAX_STRING_LEN);

/* configure the default instance */
strcpy(instance, "*");

/* Verify authenticator */
if (k4_srvtab[0]) {
    krbval = krb_rd_req(&authent, "khttp", instance, 0L, &kdata, k4_srvtab);
} else {
    /* not likely to work, but we'll try without the srvtab */
    krbval = krb_rd_req(&authent, "khttp", instance, 0L, &kdata, NULL);
}

```

```

/* see what the master says */
/* this code should be similar to the other case statement */
if (krbval) {
    /* No eh? log it so the admin can see what happened */
    sprintf(errstr, krb_err_txt[krbval]);
    ap_log_reason (errstr, r->uri, r);
    /* let somebody else take care of it */
    return AUTH_REQUIRED;
}

/* Check Kerberos principal versus the username they sent us */
if (strncmp(kdata.pname, r->connection->user, ANAME_SZ)) {
    sprintf(errstr, "Permission denied; name/username mismatch.");
    ap_log_reason (errstr, r->uri, r);
    /* let somebody else take care of it */
    return AUTH_REQUIRED;
}

/* Save the session key */
memcpy(session, kdata.session, sizeof(des_cblock));
key_sched(session, schedule);

/* Construct the response for mutual authentication */
authent.length = sizeof(des_cblock);
memset(authent.dat, 0x00, sizeof(des_cblock));
*((long *)authent.dat) = htonl(kdata.checksum + 1);
des_ecb_encrypt((des_cblock *)authent.dat, (des_cblock *)authent.dat,
    schedule, 1);

/* Convert response to string and place in buffer */
p = kdata_to_str((char *)authent.dat, authent.length);

if (p) {
    sprintf(errstr, "[%s] User %s authenticated", p, r->connection->user);
    free(p);
    ap_table_set (r->headers_out, "WWW-Authenticate", errstr);
} else {
    /* Out of memory */
    sprintf(errstr, "Not enough memory to create reply, eek!");
    ap_log_reason (errstr, r->uri, r);
}

```

```

    /* let somebody else take care of it */
    return AUTH_REQUIRED;
}

return OK;
}
#endif
#ifdef KRB5
static int kerb_validate_ticket_V5( request_rec *r )
{
    krb5_data      k5authent;
    krb5_error_code code;
    krb5_context   k5context;
    krb5_keytab    k5keytabid = NULL;
    krb5_auth_context *k5auth_context = NULL;
    krb5_principal serverp;
    krb5_principal clientp;
    krb5_ticket    *k5ticket = NULL;
    krb5_data      k5ap_rep_data;
    char *type, *p;
    char errstr[MAX_STRING_LEN], tmpstr[MAX_KDATA_LEN];
    const char *sent_pw, *auth_line;

    auth_line = ap_table_get (r->headers_in, "Authorization");

    /* did they send us the information required? */
    if ( auth_line == NULL ) {
        /* tell the client what to expect */
        ap_table_set (r->err_headers_out, "WWW-Authenticate", "KerberosV5");
        return AUTH_REQUIRED;
    }

    /* get the auth type sent to us */
    type = ap_getword(r->pool, &auth_line, ' ');

    /*
     * These two fields in the connection record are used elsewhere
     * in the Apache modules. (check_access)
     * They really appreciate them being set to useful values. :)
     */

```

```

r->connection->user = ap_getword(r->pool, &auth_line, ' ');
r->connection->ap_auth_type = "KerberosV5";

sent_pw = ap_getword(r->pool, &auth_line, '\0');

if ((k5authent.length = str_to_kdata(sent_pw, tmpstr)) == NULL) {
    sprintf(errstr, "Invalid Kerberos authenticator");
    /* log it to the error file */
    ap_log_reason (errstr, r->uri, r);
    /* let somebody else take care of it */
    return AUTH_REQUIRED;
}

k5authent.data = tmpstr;

code = krb5_init_context(&k5context);

if (code) {
    sprintf(errstr, "krb5_init_context(): Error - %s", error_message(code));
    ap_log_reason (errstr, r->uri, r);
    return AUTH_REQUIRED;
}

krb5_init_ets(k5context);

/* find server principal name; NULL means krb libs determine my hostname */
code = krb5_sname_to_principal(k5context,
    NULL, "khttp", KRB5_NT_SRV_HST, &serverp);

if (code) {
    sprintf(errstr,
        "krb5_sname_to_principal(): Error finding server principal name: %s",
        error_message(code));
    ap_log_reason (errstr, r->uri, r);
    return AUTH_REQUIRED;
}

/* Check for user-specified keytab */
if (k5_srvtab[0]) {
    code = krb5_kt_resolve(k5context, k5_srvtab, &k5keytabid);

```

```

if (code) {
    sprintf(errstr, "krb5_kt_resolve(): Error resolving keytab file: %s",
            error_message(code));
    ap_log_reason (errstr, r->uri, r);
    return AUTH_REQUIRED;
}
}

/* and most importantly, check the client's authenticator */
code = krb5_rd_req(k5context,
                  &k5auth_context, &k5authent, serverp, k5keytabid, NULL, &k5ticket);

if (code) {
    sprintf(errstr, "krb5_rd_req(): Error - %s", error_message(code));
    ap_log_reason (errstr, r->uri, r);
    return AUTH_REQUIRED;
}

clientp = k5ticket->enc_part2->client;

/* send an AP_REP message to complete mutual authentication */
code = krb5_mk_rep(k5context, k5auth_context, &k5ap_rep_data);

if (code) {
    sprintf(errstr, "krb5_mk_rep(): Error - %s", error_message(code));
    ap_log_reason (errstr, r->uri, r);
    return AUTH_REQUIRED;
}

/* Convert response to string and place in buffer */
p = kdata_to_str(k5ap_rep_data.data, k5ap_rep_data.length);

if (p) {
    sprintf(errstr, "[%s] User %s authenticated", p, r->connection->user);
    free(p);
    ap_table_set (r->headers_out, "WWW-Authenticate", errstr);
} else {
    /* Out of memory */
    sprintf(errstr, "Not enough memory to create reply, eek!");
    ap_log_reason (errstr, r->uri, r);
}

```

```

    return AUTH_REQUIRED;
}

return OK;
}
#endif

#ifdef HESIOD
/*****
 * check_user_auth - entry point for kerberos with hesiod group authentication
 *
 * Accepts: request record
 * Returns: OK, DECLINED, AUTH_REQUIRED or SERVER_ERROR
 *
 *****/
int check_user_auth (request_rec *r) {
    char *user = r->connection->user;
    int m = r->method_number;
    int method_restricted = 0;
    register int x;
    const char *t, *w;
    array_header *reqs_arr = requires (r);
    require_line *reqs;
    table *grpstatus;
    char **user_groups, *grpset;
    char errstr[50];
    int i;

    if (!reqs_arr) {
        return kerb_authenticate_user(r);
    }

    reqs = (require_line *)reqs_arr->elts;

    for (x=0; x < reqs_arr->nelts; x++) {

        if (!(reqs[x].method_mask & (1 << m))) {
            continue;
        }
    }
}

```



```

method_restricted = 1;

t = reqs[x].requirement; /* t = "require valid-user/user/group u g" */
w = ap_getword(r->pool, &t, ' '); /* w = "valid-user/user/group" */

if (!strcmp(w,"valid-user")) {
    return kerb_authenticate_user(r);
} else if (!strcmp(w,"user")) {
    return kerb_authenticate_user(r);
} else if (!strcmp(w,"group")) {

    if (kerb_authenticate_user(r) != OK) {
        return AUTH_REQUIRED; /* not in our realm */
    }

    user_groups = hes_resolve(r->connection->user, "grplist");

    if (!user_groups) {
        return AUTH_REQUIRED; /* no groups found? */
    }

    grpstatus = user_groups[0];

    while (t[0]) {
        w = ap_getword(r->pool, &t, ' '); /* w = "u", then "g" */
        grpset = strstr(grpstatus, w);

        if (grpset) {
            i = strlen(w);
            if ((grpset[i]==':' || (grpset[i]==NULL)) {
                return OK;
            }
        }
    }
}

return AUTH_REQUIRED;
}
#endif /* HESIOD */

```

```

/*****
 * kerb_authenticate_user - entry point for kerberos authentication
 *
 * Accepts: request record
 * Returns: OK, DECLINED, AUTH_REQUIRED or SERVER_ERROR
 *
 *****/
int kerb_authenticate_user (request_rec *r)
{
    int ticket = 1;    /* ticket or no, assume yes */
    int res;          /* fcn result code */
    const char *type, *name; /* auth information */
    const char *auth_line, *sent_pw;
    int KerberosV4 = 0; /* authtype v4? */
    int KerberosV5 = 0; /* authtype v5? */
    char *ptr;        /* string place holder */

    /* get the type specified in .htaccess */
    type = ap_auth_type(r);

    /* get the user realm specified in .htaccess */
    name = ap_auth_name(r);

    /* if AuthType is not Kerberos, then don't touch it */
#ifdef KRB4
    if( (type != NULL) && (strncasecmp(type, "KerberosV4", 10) == 0) ) {
        KerberosV4 = 1;
    }
#endif

#ifdef KRB5
    if( (type != NULL) && (strncasecmp(type, "KerberosV5", 10) == 0) ) {
        KerberosV5 = 1;
    }
#endif

    if ( !(KerberosV4 || KerberosV5) ) {
        /* we decline the offer... */
        return DECLINED;
    }
}

```

```

}

#ifdef KRB5_SAVE_CREDENTIALS
/*
 * Big Hack: Check for a SaveCredentials on end of authtype.
 */
if ((type!=NULL) && strstr(type, "SaveCredentials")) {
    krb5_save_credentials = 1;
} else {
    krb5_save_credentials = 0;
}
#endif /* KRB5_SAVE_CREDENTIALS */

/* get what the user sent us in the HTTP header */
auth_line = ap_table_get (r->headers_in, "Authorization");

/* did we get an auth line? */
if(!auth_line) {
    /* nope, well, tell the client what to send */

#ifdef (defined(KRB4) || (defined(KRB5)))
    /* check for those browsers that we know need basic anyways */
    /* */
    if (ap_table_get(r->subprocess_env, "use_basic_auth")) {

        ap_table_set (r->err_headers_out, "WWW-Authenticate",
            "Basic realm=\"Kerberos\"");

    } else if (ap_table_get(r->subprocess_env, "use_kerberos_auth")) {
#endif
#ifdef KRB4
        if ( KerberosV4 ) {
            ap_table_set (r->err_headers_out, "WWW-Authenticate",
                ap_pstrcat(r->pool, "KerberosV4 realm=\"", name, "\", NULL));
        }
#endif
#ifdef KRB5
        if ( KerberosV5 ) {
            ap_table_set (r->err_headers_out, "WWW-Authenticate",
                ap_pstrcat(r->pool, "KerberosV5 realm=\"", name, "\", NULL));
        }
#endif

```

```

    }
#endif

    } else {
        /* this is the new "more correct" default behavior */
        ap_table_set (r->err_headers_out, "WWW-Authenticate",
            ap_pstrcat(r->pool, "Basic realm=\"", name, "\"", NULL));
    }
#endif

    return AUTH_REQUIRED;
}

/*
 * Did they send us a Basic auth?
 */
if (strncasecmp(ap_getword (r->pool, &auth_line, ' '), "Basic", 5) == 0) {
    /*
     * No problem!  Let's get the userid/passwd pair they sent and
     * see if we can buy them a ticket.
     */
    sent_pw = ap_uudecode (r->pool, auth_line);

    /*
     * These two fields in the connection record are used elsewhere
     * in the Apache modules. (check_access)
     * They really appreciate them being set to useful values. :)
     */
    r->connection->user = ap_getword (r->pool, &sent_pw, ':');
    r->connection->ap_auth_type = "Basic";

    /* flag this as a ticket-less connection for processing */
    ticket = 0;
}

#ifdef KRB4
    /* do this for each connection since it can be set per directory */
    kerb_get_server_realm(r);
#endif

```

```

/* based on what we have so far, let's see if we can help them out */
if ( ticket ) {
    /* what? a ticket? must have a hacked browser */
#ifdef KRB4
    if ( KerberosV4 ) {
        res = kerb_validate_ticket_V4(r);
    }
#endif

#ifdef KRB5
    if ( KerberosV5 ) {
        res = kerb_validate_ticket_V5(r);
    }
#endif
} else {

    /* do not allow user to override realm setting of server */
    if ( (ptr = strchr(r->connection->user,'@')) != NULL ) {
        /* just null terminate the string sooner */
        *ptr = '\0';
    }

    /* do they know the magic word? */
#ifdef KRB4
    if ( KerberosV4 ) {
        res = kerb_validate_user_pass_V4(r, sent_pw);
#ifdef (defined(DUAL_AUTH)) && (defined(KRB5))
        if ( res != OK ) {
            res = kerb_validate_user_pass_V5(r, sent_pw);
        }
#endif
#endif /* DUAL_AUTH && KRB5 */
    }
#endif /* KRB4 */

#ifdef KRB5
    if ( KerberosV5 ) {
        res = kerb_validate_user_pass_V5(r, sent_pw);
#ifdef (defined(DUAL_AUTH)) && (defined(KRB4))
        if ( res != OK ) {
            res = kerb_validate_user_pass_V4(r, sent_pw);

```

```

    }
#endif /* DUAL_AUTH && KRB4 */
    }
#endif /* KRB5 */

    memset((char *)sent_pw, 0x00, strlen(sent_pw));
}

/*
 * Send a WWW-Authenticate: header back to the client even if they
 * supplied a bad password - as per HTTP standards.
 *
 * The WWW-Authenticate: header must be sent along with every 401
 * response, which returning AUTH_REQUIRED will generate.
 */

if ( res == AUTH_REQUIRED ) {
#if (defined(KRB4) || (defined(KRB5))
    /* check for those browsers that we know need basic anyways */
    if (ap_table_get(r->subprocess_env, "use_basic_auth")) {
        ap_table_set (r->err_headers_out, "WWW-Authenticate",
            "Basic realm=\"Kerberos\"");
    } else if (ap_table_get(r->subprocess_env, "use_kerberos_auth")) {
#ifdef KRB4
        if ( KerberosV4 ) {
            ap_table_set (r->err_headers_out, "WWW-Authenticate",
                ap_pstrcat(r->pool, "KerberosV4 realm=\"", name, "\", NULL));
        }
#endif
#ifdef KRB5
        if ( KerberosV5 ) {
            ap_table_set (r->err_headers_out, "WWW-Authenticate",
                ap_pstrcat(r->pool, "KerberosV5 realm=\"", name, "\", NULL));
        }
#endif
    } else {
        /* this is the new "more correct" default behavior */
        ap_table_set (r->err_headers_out, "WWW-Authenticate",
            ap_pstrcat(r->pool, "Basic realm=\"", name, "\", NULL));
    }
}

```

```

#endif /* KRB4 || KRB5 */
    }

    return res;
}

/*****
 * create_kerb_auth_dir_config - does the configuration
 *
 * Accepts: memory pool and dummy char
 * Returns: pointer to allocated config record
 *
 *****/
void *create_kerb_auth_dir_config (pool *p, char *d)
{
    return ap_palloc (p, sizeof(kerb_auth_config_rec));
}

/* register the functions in the correct places */
module kerb_auth_module = {
    STANDARD_MODULE_STUFF,
    NULL, /* initializer */
    create_kerb_auth_dir_config, /* dir config creator */
    NULL, /* dir merger — default is to override */
    NULL, /* server config */
    NULL, /* merge server config */
    kerb_auth_cmds, /* command table */
    NULL, /* handlers */
    NULL, /* filename translation */
    kerb_authenticate_user, /* check_user_id */
#ifdef HESIOD
    check_user_auth, /* check auth */
#else
    NULL, /* check auth */
#endif /* HESIOD */
    NULL, /* check access */
    NULL, /* type_checker */
    NULL, /* fixups */
    NULL /* logger */
};

```

Appendix B

access.conf

Access.conf is an Apache configuration file which is used to control access to URL paths.

```
# access.conf: Global access configuration
# Online docs at http://www.apache.org/

# This file defines server settings which affect which types of services
# are allowed, and in what circumstances.

# Each directory to which Apache has access, can be configured with respect
# to which services and features are allowed and/or disabled in that
# directory (and its subdirectories).

# Originally by Rob McCool

# This should be changed to whatever you set DocumentRoot to.

<Directory /local/etc/https/htdocs>

# This may also be "None", "All", or any combination of "Indexes",
# "Includes", "FollowSymLinks", "ExecCGI", or "MultiViews".

# Note that "MultiViews" must be named *explicitly* --- "Options All"
# doesn't give it to you (or at least, not yet).

Options Indexes FollowSymLinks
```



```
# This controls which options the .htaccess files in directories can
# override. Can also be "All", or any combination of "Options", "FileInfo",
# "AuthConfig", and "Limit"

AllowOverride AuthConfig Limit

# Controls who can get stuff from this server.

order allow,deny
allow from all

</Directory>

# /usr/local/etc/httpd/cgi-bin should be changed to whatever your ScriptAliased
# CGI directory exists, if you have that configured.

<Directory /local/etc/https/cgi-bin>
AllowOverride AuthConfig
Options None
</Directory>

# You may place any other directories or locations you wish to have
# access information for after this one.

# require Realm authentication on these spaces
#

<Directory /local/etc/https/eos-bin>
AuthName NCSU Realm
AuthType KerberosV4
<Limit GET PUT POST>
require valid-user
order allow,deny
allow from all
</Limit>
</Directory>

<Directory /local/etc/https/ssldocs>
AuthName NCSU Realm
```

```
AuthType KerberosV4
<Limit GET PUT POST>
require valid-user
order allow,deny
allow from all
</Limit>
</Directory>
```

Appendix C

Certificate Authority Creation

The following steps were performed to generate generating our Certificate Authority (CA) key.

```
req -config CA.conf -new -nodes -out request.pem -keyout request.pem
US
North Carolina
Raleigh
NC State University
Computer Science Dept.
submit.ncsu.edu
webmaster@csc.ncsu.edu
"" (no passwd)
"" (no company name)

ca -config CA.conf -in request.pem -out newcert.pem
<passwd>
sign? Y
commit? Y

vi newcert.pem
Remove all but certificate
Add RSA key from request.pem

mv newcert.pem submit.pem
mv request.pem submit.req.pem
```

```
x509 -noout -hash < submit.pem
```

Save HASH string

Copy certificate to httpsd.pem on Apache server.

```
ln -s httpsd.pem HASH.0
```

Restart Apache Server.

Appendix D

Sybase Scripts

The following Sybase scripts create and build the student and course database tables.

```
-- ADD_UNITYACS:
-- creates basic tables and login
--
-- USAGE: isql -Usa -iadd_unityacs -e
--
-- DATABASE:
create database acsdb
    on default = 140
go
sp_dboption acsdb,
    "select into/bulkcopy", true
go
sp_dboption acsdb,
    "trunc log on chkpt", true
go
-- LOGINS:
sp_addlogin XXXXXXXX,"XXXXXXXXXX", acsdb
go
use acsdb
go
sp_changedbowner XXXXXXXX
go
-- TABLES:
```

```

create table people (
  ssn          varchar(9)   NOT NULL,
  username     varchar(12)  NOT NULL,
  firstname    varchar(25)  NULL,
  middlename   varchar(25)  NULL,
  lastname     varchar(25)  NULL,
)
go
create table userinfo (
  ssn          varchar(9)   NOT NULL,
  username     varchar(12)  NOT NULL,
  uid          int          NOT NULL,
  fullname     varchar(75)  NOT NULL,
  add_date     datetime     NOT NULL,
  active_date  datetime     NULL,
  disable_date datetime     NULL,
  status_code  char(2)      NULL,
  last_access_time datetime NULL,
)
go
create table stu_crs (
  sem_yy          char(2)   NULL,
  sem_code        char(1)   NULL,
  student_id     char(9)   NULL,
  index_name     char(30)  NULL,
  sex            char(1)   NULL,
  ethnic_code    char(1)   NULL,
  reg_status     char(1)   NULL,
  sch_code       char(2)   NULL,
  curr_code      char(5)   NULL,
  class_code     char(2)   NULL,
  enrol_stat     char(2)   NULL,
  tui_res_cd     char(3)   NULL,
  career_lvl     char(1)   NULL,
  sem_hrs        real      NULL,
  crs_prefix     char(5)   NULL,
  crs_num        char(3)   NULL,
  crs_suffix     char(1)   NULL,
  sect_num       char(3)   NULL,
  crs_cr_hrs     real      NULL,

```

```

grade_opt          char(1)    NULL,
crs_status         char(1)    NULL,
crs_st_chg        char(6)    NULL,
tui_rate          char(1)    NULL,
date_first_enroll char(3)    NULL,
extract_date       smalldatetime NULL,
)
go
create clustered index course_index
on stu_crs (crs_prefix, crs_num, sect_num)
go
create index id_index on stu_crs (student_id)
go
create table stu_addr (
student_id  char(9) NOT NULL,
index_name  varchar(30) NOT NULL,
label_name  varchar(30) NOT NULL,
sex         char(1) NOT NULL,
marital     char(1) NOT NULL,
ethnic      varchar(30) NOT NULL,
dob         smalldatetime NOT NULL,
sem_code    char(1) NOT NULL,
sem_name    char(7) NOT NULL,
sem_yy      char(2) NOT NULL,
reg_status  char(1) NOT NULL,
sch_code    char(2) NOT NULL,
sch_name    varchar(45) NOT NULL,
curr_code   varchar(5) NOT NULL,
curr_name   varchar(40) NOT NULL,
sch_code2   varchar(2) NOT NULL,
sch_name2   varchar(45) NOT NULL,
curr_code2  varchar(5) NOT NULL,
curr_name2  varchar(40) NOT NULL,
class_code  char(2) NOT NULL,
class_name  varchar(30) NOT NULL,
privacy     varchar(20) NOT NULL,
p_street1   varchar(23) NOT NULL,
p_street2   varchar(23) NOT NULL,
p_city      varchar(25) NOT NULL,
p_state     char(2) NOT NULL,

```

```
p_zip          varchar(10) NOT NULL,
p_s_c_name     varchar(30) NOT NULL,
p_phone       varchar(12) NOT NULL,
c_street1     varchar(23) NOT NULL,
c_street2     varchar(23) NOT NULL,
c_city        varchar(25) NOT NULL,
c_state       char(2) NOT NULL,
c_zip         varchar(10) NOT NULL,
l_street1     varchar(23) NOT NULL,
l_street2     varchar(23) NOT NULL,
l_city        varchar(25) NOT NULL,
l_state       char(2) NOT NULL,
l_zip         varchar(10) NOT NULL,
l_phone       varchar(12) NOT NULL,
l_housing_type char(1) NOT NULL,
father_nam    varchar(30) NOT NULL,
father_liv    char(1) NOT NULL,
mother_nam    varchar(30) NOT NULL,
mother_liv    char(1) NOT NULL,
guard_name    varchar(30) NOT NULL,
date_first_enro char(3) NOT NULL,
email_addr    varchar(60) NOT NULL,
email_addr_override char(1) NOT NULL,
extract_date  smalldatetime NOT NULL,
)
go

-- USER PERMISSIONS
-- ALL DONE
checkpoint
go
quit
```



```
-- USAGE: isql -ibuild_addr -e
--
--
-- BUILD TABLE:
select distinct
  p.student_id, p.index_name, p.sem_code, p.sem_yy,
  p.sem_name, p.privacy, p.email_addr,
  p.email_addr_override, s.username
into addr
from stu_addr p, people s
where p.student_id = s.ssn
go
-- BUILD INDEX:
create index a_index on addr (student_id)
go
-- ALL DONE
checkpoint
go
quit
```

```
-- USAGE: isql -ibuild_enroll -e
--
--
-- REGEN INDEXES FOR STU_CRIS:
update statistics stu_crs
go
-- BUILD TABLE:
select distinct
  p.*, s.curr_code, s.class_code
into enroll
from people p, stu_crs s
where p.ssn = s.student_id
go
-- ALL DONE
checkpoint
go
quit
```

```

#!/usr/local/bin/perl

$homepath = "/usr/sybase/XXXXXXXX";
$infile = "$homepath/userinfo";
$outfile = "$homepath/people";
%suffix = (
    "JR", 1,
    "SR", 1,
    "II", 1,
    "III", 1,
    "IV", 1,
    "V", 1,
    "VI", 1,
    "VII", 1,
    "VIII", 1,
);

print STDERR "Building people data from userinfo file...";

open(FIN,$infile) || die "Failed to open input file: $infile\n";
open(FOUT,">$outfile") || die "Failed to write output file: $outfile\n";
while ($lin = <FIN>) {
    $ssn = substr($lin, 0, 9);
    $username = substr($lin, 9, 12);
    $uid = substr($lin, 21, 6);
    $fullname = substr($lin, 27, 75);
    $firstname=""; $middlename=""; $lastname="";
    $ssn =~ s/\s//g;
    $username =~ s/\s//g;
    $fullname =~ tr/a-z/A-Z/;
    $fullname =~ s/[^A-Z\-\s]//g;
    $fullname =~ s/\s+/ /g;
    $fullname =~ s/^ +//;
    $fullname =~ s/ +$//;
    @names = split(/ /,$fullname);
    $lastname = pop(@names);
    if ($suffix{$lastname}) {
        $lastname = pop(@names)." ".$lastname;
    }
    $firstname = shift(@names);

```

```
    $middlename = join(" ",@names);  
    print FOUT join("\t",$ssn, $username, $firstname, $middlename, $lastname),"\n";  
}  
close(FOUT);  
close(FIN);  
  
print STDERR "Done!\n";  
exit;
```

```
-- USAGE: isql -ibuild_user_crs -e
--
--
-- BUILD TABLE:
select distinct
  p.ssn, p.username, s.crs_prefix, s.crs_num, s.crs_suffix, s.sect_num,
  s.sem_yy, s.sem_code, s.curr_code, s.index_name, s.class_code
into user_crs
from people p, stu_crs s
where p.ssn = s.student_id
go
-- BUILD INDEX:
create index u_index on user_crs (username)
go
-- ALL DONE
checkpoint
go
quit
```

Appendix E

Submit Perl Library

The following Perl support library contains all functions used by submitV2.

```
use Sybase::DBlib;
use CGI qw(:standard :html3);
use Hesiod;
use AFS;

# uncomment to see the AFS rx errors
#AFS::raise_exception(1);

# Where is sybase and the interface file?
$ENV{'SYBASE'} = '/local/sybase';
$USER = $ENV{'REMOTE_USER'};

# Where is the current courses file?
$CURRENT_COURSES = "/afs/eos.ncsu.edu/info/csc_info/CURRENT-COURSES";

# Where is sybase and the interface file?
$ENV{'SYBASE'} = '/local/sybase';

# What is the sybase user, passowrd, and server?
$SYBUSER = 'XXXXXXXX';
$SYBSERVER = 'XXXXXXXX';
$SYBPASSWD = 'XXXXXXXXXX';
```

```

# Graphics
$BASEURL = 'https://submit.ncsu.edu';
$ADMINURL = 'https://submit.ncsu.edu/eos-bin/submit-admin';
$GRAPHIX = "$BASEURL/graphics";
$TSTART = "$GRAPHIX/fancy-tstart.gif";
$TBAR   = "$GRAPHIX/fancy-tbar.gif";
$LOGO   = "$GRAPHIX/submit-logo.gif";
$TEND   = "$GRAPHIX/fancy-tend.gif";
$SIDE   = "$GRAPHIX/fancy-side.gif";
$BSTART = "$GRAPHIX/fancy-bstart.gif";
$BBAR   = "$GRAPHIX/fancy-bbar.gif";
$BEND   = "$GRAPHIX/fancy-bend.gif";

%console = (
  'end' => {'graphic'=>"$GRAPHIX/console-bar-end.gif",
            'action'=>""},
  'bar' => {'graphic'=>"$GRAPHIX/console-bar.gif",
            'action'=>""},
  'submit-admin' => {'graphic'=>"$GRAPHIX/console-submit-admin.gif",
                    'action'=>"$ADMINURL"},
  'side' => {'graphic'=>"$GRAPHIX/console-side.gif",
            'action'=>""},
#
  'update-student' => {'graphic'=>"$GRAPHIX/console-update-student.gif",
                      'action'=>"$ADMINURL?tool=\"update-student\""},
  'delete-student' => {'graphic'=>"$GRAPHIX/console-delete-student.gif",
                      'action'=>"$ADMINURL?tool=\"delete-student\""},
  'add-student' => {'graphic'=>"$GRAPHIX/console-add-student.gif",
                   'action'=>"$ADMINURL?tool=\"add-student\""},
#
  'add-support' => {'graphic'=>"$GRAPHIX/console-add-support.gif",
                  'action'=>"$ADMINURL?tool=\"add-support\""},
  'delete-support' => {'graphic'=>"$GRAPHIX/console-delete-support.gif",
                     'action'=>"$ADMINURL?tool=\"delete-support\""},
#
  'add-assignment' => {'graphic'=>"$GRAPHIX/console-add-assignment.gif",
                    'action'=>"$ADMINURL?tool=\"add-assignment\""},
  'delete-assignment' => {'graphic'=>"$GRAPHIX/console-delete-assignment.gif",
                        'action'=>"$ADMINURL?tool=\"delete-assignment\""},
);

```

```

# Colors
$BGCOLOR = '#000000';
$TEXT = '#FFB00';
$VLINK = '#COCOCO';
$LINK = '#FFFF00';
$ALINK = '#COFFCO';

#----- Error Handlers -----
##
##
sub cgi_die {
##
##
    my (@err) = @_;
    print header;
    print start_html(-title=>"Error"),
        "An error has occurred:",
        br,
        join(' ', @err),
        end_html;
}

#----- Sybase Handlers -----
##
##
sub syb_lookup_user {
##
##
    my ($user) = @_;

    my $dbh = new Sybase::DBlib $SYBUSER, $SYBPASSWD, $SYBSERVER, "Submit";
    $dbh->dbcmd("select distinct username, crs_prefix, crs_num, sect_num\n");
    $dbh->dbcmd("from user_crs\n");
    $dbh->dbcmd("where username = \'$user\'\n");
    my $sybstat = $dbh->dbsqlexec;
    $sybstat = $dbh->dbresults;

    my @courses = ();
    my @sybdata = ();

```



```

while (@sybdata = $dbh->dbnextrow) {
    push(@courses, $sybdata[1] . $sybdata[2] . '-' . $sybdata[3]);
}
$dbh->dbclose;

return(\@courses);
}

#----- Page Layout -----
##
##
sub begin_submit {
##
##
    print header(-expires=>' +30s');
    print "\n";
    print start_html(-title=>'NC State Online Submit',
        -BGCOLOR=>"$BGCOLOR",
        -TEXT=> "$TEXT",
        -LINK=> "$LINK",
        -VLINK=>"$VLINK",
        -ALINK=>"$ALINK");

    print "\n";
}

##
##
sub end_submit {
##
##
    print end_html;
}

##
sub begin_table {
##
    print "<TABLE CELLSPACING=\0\0" BORDER=\0\0" CELLPADDING=\0\0" \n";
    print "WIDTH=\640\>\n";
    print "<TR>\n";
    print "    <TD><IMG SRC=\$TSTART\0" WIDTH=\100\0" HEIGHT=\39\0"></TD>\n";

```

```

print " <TD><IMG SRC=\"$TBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n";
print " <TD><IMG SRC=\"$LOGO\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n";
print " <TD><IMG SRC=\"$TBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n";
print " <TD><IMG SRC=\"$TEND\" WIDTH=\"20\" HEIGHT=\"39\"></TD>\n";
print "</TR>\n";
}

##
##
sub print_data {
    my ($data) = @_;
    # you need a better calculation for this.
    my $height = int(length($data) / 2);
    print "<TR>\n";
    print " <TD><IMG SRC=\"$SIDE\" WIDTH=\"100\" HEIGHT=\"750\"></TD>\n";
    print " <TD COLSPAN=\"4\"> $data </TD>\n";
    print "</TR>\n";
}

##
##
sub console_line {
    my (@line) = @_;
    my $x;
    for ($i = 0; $i <= 2; $i++) {
        if ($line[$i] eq "") {
            $line[$i] = "bar";
        }
    }
    print "<TR>\n";
    print " <TD><IMG SRC=\"$console{'side'}->{'graphic'}\"\",
    \"WIDTH=\"100\" HEIGHT=\"39\"></TD>\n";
    if ($console{$line[0]}->{'action'}) {
        print " <TD><A HREF=\"$console{$line[0]}->{'action'}\">\",
        \"<IMG SRC=\"$console{$line[0]}->{'graphic'}\" BORDER=\"0\"\",
        \"WIDTH=\"180\" HEIGHT=\"39\"></A></TD>\n";
    } else {
        print " <TD><IMG SRC=\"$console{$line[0]}->{'graphic'}\"\",
        \"WIDTH=\"180\" HEIGHT=\"39\"></TD>\n";
    }
}

```

```

if ($console{$line[1]}->{'action'}) {
    print "    <TD><A HREF=\"\$console{$line[1]}->{'action'}\">\",
          \"<IMG SRC=\"\$console{$line[1]}->{'graphic'}\" \" \",
          \"BORDER=\"0\" WIDTH=\"180\" HEIGHT=\"39\"></A></TD>\n\";
} else {
    print "    <TD><IMG SRC=\"\$console{$line[1]}->{'graphic'}\"\",
          \"WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
}
}
if ($console{$line[2]}->{'action'}) {
    print "    <TD><A HREF=\"\$console{$line[2]}->{'action'}\">\",
          \"<IMG SRC=\"\$console{$line[2]}->{'graphic'}\" \" \",
          BORDER=\"0\" WIDTH=\"180\" HEIGHT=\"39\"></A></TD>\n\";
} else {
    print "    <TD><IMG SRC=\"\$console{$line[2]}->{'graphic'}\"\",
          \"WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
}
}
print "    <TD WIDTH=\"20\"><IMG SRC=\"\$console{'end'}->{'graphic'}\"\",
      \"WIDTH=\"20\" HEIGHT=\"39\"></TD>\n\";
print \"</TR>\n\";
}

##
##
sub end_table {
    print \"<TR>\n\";
    print "    <TD><IMG SRC=\"\$BSTART\" WIDTH=\"100\" HEIGHT=\"39\"></TD>\n\";
    print "    <TD><IMG SRC=\"\$BBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
    print "    <TD><IMG SRC=\"\$BBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
    print "    <TD><IMG SRC=\"\$BBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
    print "    <TD><IMG SRC=\"\$BBAR\" WIDTH=\"180\" HEIGHT=\"39\"></TD>\n\";
    print "    <TD><IMG SRC=\"\$BEND\" WIDTH=\"20\" HEIGHT=\"39\"></TD>\n\";
    print \"<TR>\n\";
    print \"</TABLE>\n\";
}

#----- Course Locker Tools -----
##
##
sub get_locker_location {
    my ($locker_name) = @_;
    $locker_name = lc($locker_name);

```

```

# makes assumption that all lockers are XXXYYY_info
if ($locker_name !~ /_info$/) {
    $locker_name .= '_info';
}
my ($location, $path, $write, $shortcut) = split(' ',
    resolve($locker_name, 'filsys'), 4);
return ($path);
}

##
##
sub get_submission_path {
    my ($course) = @_;
    my ($class, $section) = split('-', $course, 2);
    my $path = &get_locker_location($class);
    $path = $path . "/SUBMITTED/$section/";
    return ($path);
}

##
##
sub create_path {
    my ($path) = @_;
    my @parts = split('/', $path);
    foreach $x (@parts) {
        $pathname .= "/"$x";
        next if ($x eq "");
        if (! -d $pathname) {
            mkdir ("$pathname", 0755);
        }
    }
}

##
##
sub get_user_info {
    my ($user, $path) = @_;
# looks up the user and returns the information about him.
    $path = $path . "/ADMIN/ROLL";
    open (DATA, "$path") ||
        &cgi_die("Could not open the $path/ADMIN/ROLL file\n");
}

```

```

my @roll = ();
my $line = "";
while ($line = <DATA>) {
    @roll = split(':', $line, 5);
    if ($roll[0] =~ /^user$/) {
        last;
    }
}
close(DATA);
return(\@roll);
}

##
##
sub get_homework_assignments {
    my ($user, $course) = @_;
# looks up the section and then what assignments are available.

    my ($class, $sec) = split('-', $course, 2);
    $class = lc($class);

    my ($path) = &get_locker_location($class);

    open(ASSIGNMENTS, "$path/ADMIN/ASSIGNMENTS") ||
        &cgi_die ("Could not open the $path/ADMIN/ASSIGNMENTS file\n");

# initialization
my $line = "";
my $current_time = time();
my %assignments = ();

while ($line = <ASSIGNMENTS>) {
    chomp $line;
    my ($hw_name, $start, $end, $type) = split(':', $line, 4);
    if (($start < $current_time) && ($end > $current_time) &&
        ($sec eq $type)) {
        $assignments{$hw_name} = $line;
    } elsif (($start < $current_time) && ($end > $current_time) &&
        ($user eq $type)) {
        $assignments{$hw_name} = $line;
    }
}

```

```

    }
  }
  return (\%assignments);
}

#
# get_courses
#
sub get_courses {
  my (@courses) = ();
  open (CURRENT_COURSES, "$CURRENT_COURSES");
  while (<CURRENT_COURSES>) {
    if (/^(~%+)\%([~:]+)([~:]+)$/) {
      push @courses, $3;
    }
  }
  close(CURRENT_COUSES);
  return (\@courses);
}

#
# pts_2_support
#
sub pts_2_support {
  my ($course) = @_;
  my ($pts) = newpts;
  my ($over) = 0;
  my (@members) = $pts->members("tk1:$course/admin/sup",1,$over);
  my $x = 0;
  my $path = &get_locker_location($course);

  open(SUP, "> $path/ADMIN/SUP");
  close(SUP);
}

#
# add support
#
sub add_support {

```

```

my($user, $course) = @_;
&pts_add($user, "tkl:$course/admin/sup");
}

#----- AFS Tools -----
##
##
sub pts_membership {
    my ($group) = @_;

    my $pts = newpts;
    my @members = ();
    my $over = 0;

    @members = $pts->members($group, 1, $over);

    return (\@members);
}

##
##
sub pts_add {
    my ($user, $group) = @_;

    my $pts = newpts;
    my $success = $pts->adduser($user, $group);
    return ($success); # 1 = true 0 = error
}

##
##
sub pts_del {
    my ($user, $group) = @_;

    my $pts = newpts;
    my $success = $pts->removeuser($user, $group);

    return ($success); # 1 = true 0 = error
}

```

```

##
##
sub pts_create {
    my ($group, $owner) = @_;
    my $pts = newpts;
    my $id = $pts ->creategroup($name,$owner, $id);
    return ($id);
}

##
##
sub pts_destroy {
    my ($group) = @_;
    my $pts = newpts;
    my $success = $pts ->delete($name);
    return ($success);
}

##
##
sub pts_chown {
    my ($group, $newowner) = @_;
    my $pts = newpts;
    my $success = $pts->chown($group, $newowner);
    return($success);
}

##
## ----- Authentication -----
##
sub check_auth {
    my ($user) = @_;
    # not complete
    return (1);
}

sub check_auth_sub {
    my ($user, $course) = @_;
    my ($pts) = newpts;
    my ($over) = 0;

```



```
my (@members) = ();

@members = $pts->members("tkl:$course/admin/sup", 1, $over);
return (1) if (scalar(grep(/^$user$/, @members)));
return (0);
}
```

Appendix F

Submit

The following Perl code implements SubmitV2, our prototype application.

```
#!/local/bin/perl
#
#
require "../submit-lib/submit-lib.pl";
use MD5;
use CGI qw(:standard :html3);
$CGI::POST_MAX = 1024 * 8000; # MAX UPLOAD is 8mb.
@phase = param('phase');
@key = param('key');

if (!param()) {
# first pass into the page (select class and (check or submit))
    *courses = &syb_lookup_user($USER);
    $md5 = new MD5;
    $md5->add ($USER);
    $md5hash = unpack("H*", $md5->digest());
    $layout = em('Username: '). $USER. p.
        start_form.
        em('Select Course: ').
        popup_menu(-name=>'course',
            -values=>[@courses]).
        p."\n".
        em('Choose One: ').
```

```

        radio_group(-name=>'type',
                    -values=>['submit', 'check'],
                    -default=>'submit').

    p.
    hidden(-name=>'phase', -default=>['phase1']).
    hidden(-name=>'key', -default=>[$md5hash]).
    submit.
    end_form;

    &begin_submit;
    &begin_table;
    &print_data($layout);
    &end_table;
    &end_submit;
} elsif ($phase[0] eq 'phase1') {
    # submitting a homework
    # display form.
    # request file.
    *assignments = &get_homework_assignments($USER,param('course'));
    param('phase', 'upload');
    $md5 = new MD5;
    $md5->add ($USER);
    $md5hash = unpack("H*", $md5->digest());

    if ($md5hash ne $key[0]) {
        &cgi_die ("MD5 key does not match.\n");
    }

    $md5->reset;
    $md5->add($USER, param('course'));
    $md5hash = unpack("H*", $md5->digest());

    $layout .= em('Username: '). $USER. p.
                em('Submitting for class: '). param('course'). p.
                start_multipart_form.
                hidden('course', param('course')).
                hidden(-name=>'phase', -default=>['upload']).
                hidden(-name=>'key', -default=>[$md5hash]).
                "Choose the assignment: ".
                radio_group(-name=>'assignment', -values=>[keys(%assignments)]). p.

```

```

        filefield(-name=>'upload').
        p.
        submit.
        end_form;
    &begin_submit;
    &begin_table;
    &print_data($layout);
    &end_table;
    &end_submit;
} elsif ($phase[0] eq 'upload') {

    $md5 = new MD5;
    $md5->add ($USER, param('course'));
    $md5hash = unpack("H*", $md5->digest());

    if ($md5hash ne $key[0]) {
        &cgi_die ("MD5 key does not match.\n");
    }

    $dataname = join(' ', param('upload'));
    $dataname =~ s#^.*(?:/)*$#$1#; # take care of /usr/foo/bar/dog as dog
    $dataname =~ s#^.*\\(?:\\)*$#$1#; # take care of c:\usr\foo\bar\dog as dog

    $basename = &get_submission_path(param('course'));
    $basename = $basename . param('assignment') . "/" . $USER;
    $dataname = $basename . "/" . $dataname;

    &create_path($basename);

    $filename = param('upload');
    open (OUTFILE, "> $dataname");
    while ($bytesread = read($filename, $buffer, 1024)) {
        print OUTFILE $buffer;
    }
    close(OUTFILE);

    # do the graphics
    &begin_submit;
    &begin_table;
    &print_data($layout . $dataname);

```

```
&end_table;
&end_submit;
} elsif (param('type') eq 'check') {
    # checking on an assignment.
    # display files.
    &cgi_die('You can not get there from here. ');
} else {
    # we are here because there was an error.
    &cgi_die('You can not get there from here. ');
}
```