CITI Technical Report 01-5

Kerberized Credential Translation: A Solution to Web Access Control

Olga Kornievskaia, Peter Honeyman, Bill Doster, Kevin Coffman
{aglo,honey,billdo,kwc}@citi.umich.edu

Abstract

Kerberos, a widely used network authentication mechanism, is integrated into numerous applications, UNIX and Windows 2000 login, AFS, Telnet, and SSH to name a few. Yet, Web applications rely on SSL to achieve authenticated and secure connections. SSL provides strong authentication by using certificates and public key challenge response authentication. The expansion of the Internet requires each system to leverage the strength of the other, which suggests the necessity of interoperability between them.
This paper describes the design, implementation and performance of a system that provides controlled access to Kerberized services through a browser. This system provides a single sign-on through Kerberos, which produces both Kerberos and public key credentials. The Web server uses a plugin that translates user's public key credentials to Kerberos credentials. The Web server's subsequent authenticated actions taken on a user's behalf are limited in time and scope. Performance measurements show how the overhead introduced by credential translation is amortized over the login session.

February 12, 2001

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Kerberized Credential Translation: A Solution to Web Access Control

Olga Kornievskaia, Peter Honeyman, Bill Doster, Kevin Coffman
{aglo,honey,billdo,kwc}@citi.umich.edu

1 Introduction

Access control for Web space has often been viewed in terms of gating access to Web pages and the job of the Web server was limited to simple file reads. With the expansion of the Internet many services are becoming accessible from the Web, increasing the Web server's importance. In addition to file access, a Web server frequently may serve information stored in backend databases. The functionality provided by Web servers has considerably grown making it into one of the most popular technologies used on the Internet. New features like generation of active server pages and CGI scripts require computing and processing power at the server side. A Web interface to backend services is considered to be more user-friendly and accessible to compared to predominant text-based interfaces.

The possibilities opened by the use of Web server to access a variety of backend services poses new questions on how to retain access control of backend services. In this paper, we provide a practical mechanism for controlling access to resources provided by Kerberized services like file servers, directory servers, and mail servers across web boundaries.

Our solution preserves authorization mechanisms already in place at the backend servers. This obviates constructing and maintaining consistent replicas of authorization policies across authentication domains.

In practice, an authorization mechanism is tied to authentication mechanism. A mismatch in authentication mechanisms prevents a Web server from using authorization mechanisms provided by backend servers. While Web servers support SSL authentication with certificates, this does not provide credentials for access to AFS file servers, LDAP directory servers, and KPOP/IMAP mail servers, which use Kerberos for client authentication.

Consider the following scenario:

Alice attends the University of Michigan, where she enjoys a variety of computing services available to her. One of the most commonly used services is AFS file service which is protected by Kerberos. Alice, being a very private person, doesn't want others to have access to her files. Through the access control mechanisms provided by AFS, she limits access to specific users. But if these users prefer to access Alice's files through the Web the flexibility of AFS access controls disappear.

Web presence for other Kerberized services also suffers. For example, Alice would like to change her umich.edu X.500 directory entry from a browser. The directory is stored in an LDAP directory that uses Kerberos for user authentication to control read and write access. Alice would also like to read mail form a browser. This too requires that the Web server authenticates as Alice to the Kerberized mail server.

A practical solution is needed that works with existing software and is easy to deploy, administer, and maintain. The process should demand minimal interaction with a user, providing transparent access to the resources. To limit misuse of user's credentials, the Web server must be constrained in its actions. Furthermore, a central, easily administered location for enforcing security policies controlling Web server's actions is required.

If an AFS client is running on the machine, a simple solution presents itself. Instead of making an HTTP request, a user can access AFS file space directly by typing `file://localhost/afs/...`. But it is fair to say that most machines do not run AFS. Also, the solution fails to provide a general mechanism for accessing services from the Web: browsers can not anticipate all possible service access types.

A frequently used solution is to send a Kerberos

identity and password through SSL, but this gives unlimited power to the Web server to impersonate users, a significant risk. It is also hazardous to expect users to know when it is safe to pass her password to a Web server.

This paper describes the design, implementation and performance of a system that provides controlled access to Kerberized services through a browser. The system provides a single sign-on through Kerberos authentication: users authenticate once and are given Kerberos and PK credentials. The latter are used for Web authentication. Our system includes a Web server plugin that translates user's PK credentials to Kerberos credentials. Our design assures that Web server actions taken on a user's behalf are limited in time and scope.

The remainder of this paper is organized as follows. Section 2 provides background material and discusses related work. Section 3 presents an architecture for access to Kerberized services through a browser. Section 4 gives implementation details. Section 5 describes performance. Section 6 summarizes, poses some unanswered questions, and presents directions for future work.

2 Background

First, we review Kerberos, a popular network authentication system based on symmetric key cryptography. Its success stories come from environments with well defined administrative boundaries. We then provide an overview of SSL, a security protocol based on public key cryptography that is widely used on the Web. The Internet spans many Kerberized realms and requires security solutions that do not have centralized management. SSL provides authenticated and secure connections between any two nodes in the Internet.

We conclude the section with an overview of related work.

2.1 Overview of Kerberos

Kerberos [11] is a network authentication system based on the Needham-Schroeder protocol [10]. Kerberos authentication is illustrated in Figure 1. Authentication is achieved when one party proves knowledge of a shared secret to another. To avoid quadratic explosion of key agreement, Kerberos relies on a trusted third party, referred to as a Key Distribution Center (*KDC*). *Alice*, a Kerberos principal, and *Bob*, a Kerberized service, each establish a shared secret with the *KDC*.

At login, *Alice* receives a ticket granting ticket, *TGT*, from the *KDC*. She uses her password to retrieve a session key encrypted in the reply. *TGT* allows *Alice* to obtain tickets from a Ticket Granting Service for other Kerberized services. To access a Kerberized service, *Alice* presents her *TGT* and receives a service ticket, $\{Alice, Bob, \mathrm{K}_{A,B}\}_{K_B}$. To authenticate to *Bob*, *Alice* constructs an authenticator, $\{\mathrm{T}\}_{K_{A,B}}$, proving to *Bob* that she knows the session key inside of the service ticket.

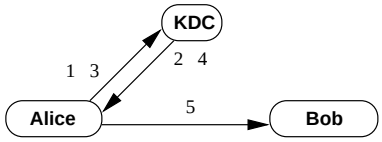
2.2 Overview of SSL/TLS

Secure Socket Layer [1] [7, 8] is a protocol that provides secure connections, addressing the need for entity authentication, confidentiality, and integrity of messages on the Internet. SSL uses public key cryptography, in particular certificates, to accomplish authentication and secret key cryptography to provide confidentiality and integrity of the communication channel. Support for SSL is nearly universal among Web browsers and servers.

SSL consists of two sub-protocols: the SSL *record* protocol and the SSL *handshake* protocol. The SSL record protocol defines the format used to transmit data. The SSL handshake protocol uses the record protocol to negotiate a security context for a session. SSL supports numerous encryption and digest mechanisms that the client and the server negotiate during the SSL handshake.

Figure 2 shows the exchange of messages in the handshake, details of which are discussed in Section 3.2. Authentication is based on a public key challenge/response protocol and X.509 [5] identity certificates. SSL supports mutual authentication. First, a user authenticates the server. The user has the responsibility to make sure it can trust the certificate it received in CERTIFICATE message from the server. That responsibility includes verifying the certificate signatures, validity times, and revo-

[1] SSL is renamed by IETF as Transport Layer Security, *TLS* [2],



LOGIN PHASE:	ONCE PER SESSION
1. *Alice* → *KDC*:	"Hi, I'm *Alice*"
2. *KDC* → *Alice*:	$TGT = \{Alice, TGS, \mathrm{K}_{A,TGS}\}_{K_{TGS}}, \{\mathrm{T}\}_{K_A}, \{\mathrm{K}_{A,KCT}\}_{K_A}$
ACCESSING SERVICES:	EVERY TIME BEFORE TALKING TO A SERVICE
3. *Alice* → *TGS*:	$Alice, Bob, TGT, \{\mathrm{T}\}_{K_{A,TGS}}$
4. *KCT* → *Alice*:	$TKT = \{Alice, Bob, \mathrm{K}_{A,B}\}_{K_B}, \{\mathrm{T}\}_{K_{A,TGS}}], \{\mathrm{K}_{A,B}\}_{K_{A,TGS}}$
5. *Alice* → *Bob*:	"Hi, I'm *Alice*", $TKT, \{\mathrm{T}\}_{K_{A,B}}$

Figure 1: **Kerberos authentication.** Two phases are shown: initial authentication and service ticket acquisition. *KDC* is the Kerberos Key Distribution Center. *TGS* is the Ticket Granting Service. Most implementations combine these services. K_{TGS} is a key shared between a *TGS* and *KDC*. K_A is a key shared between *Alice* and *KDC*, derived from *Alice's* password. $\mathrm{K}_{A,TGS}$ is a session key for *Alice* and *TGS*. $\mathrm{K}_{A,B}$ is a session key for *Alice* and *Bob*. T is a timestamp used to prevent replay attacks. Kerberos assumes synchronized clocks.

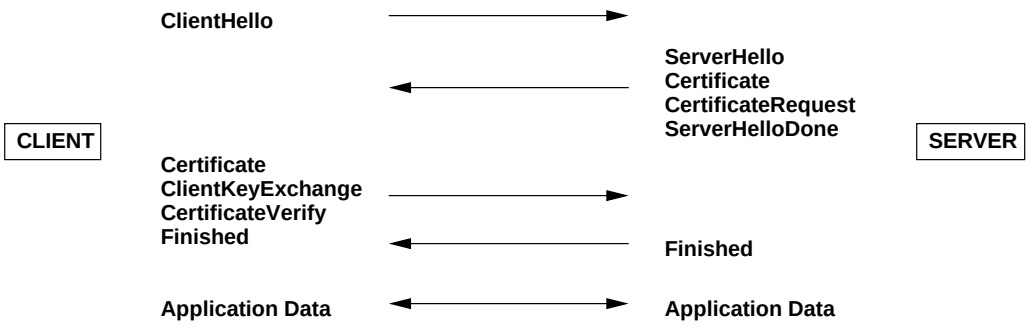


Figure 2: **SSL handshake protocol.** CLIENT HELLO carries a version, timestamp, and session id, which allows a user to resume a previous session. SERVER HELLO confirms the version and session id. Server sends its CERTIFICATE and requests user's in CERTIFICATE REQUEST. SERVER HELLO DONE specifies the end of a negotiation phase. Client sends her public key certificate in CERTIFICATE message. Client sends session information (encrypted with the server's public key) in a CLIENTKEYEXCHANGE message. Client sends a CERTIFICATE VERIFY message, which signs important information about the session using the client's private key. the server uses the public key from the client's certificate to verify the client's identity. FINISHED messages serve to end the negotiation process. Secured APPLICATION DATA messages follow. Optional SSL messages are omitted.

cation status. Then, the user sends her public key certificate. The user must also prove that she possesses the private key corresponding to the certificate's public key. For the proof, the user creates a message that contains a digitally signed cryptographic hash of information available to both the user and the server. The server then verifies the signature to be sure that the user possesses the appropriate private key.

The SSL protocol supports renegotiation of the security context after a handshake is done. A client initiates a new handshake by sending a CLIENT HELLO message. If the server wishes to initiate a handshake, it sends an empty SERVER HELLO message to the client and the client responds to it with a new CLIENT HELLO.

Establishing an SSL session requires sophisticated cryptographic calculations and a significant number of protocol messages. To minimize the overhead of these calculations and messages, SSL provides a mechanism by which two parties can reuse previously negotiated SSL parameters. With this method, the parties do not repeat the cryptographic operations, they simply resume an earlier session. The user proposes to resume a previous session by including that session's SessionID value in the CLIENT HELLO. It is up to the server to decide whether to allow the reuse of the session. To refer to this mechanism, we say that the user and server engage in a partial SSL handshake.

2.3 Related Work

Issuing and maintaining millions of user's certificates is hard. One way to accomplish client authentication on the Web without certificates is to make use of the extension to TLS cipher suites that includes Kerberos as an authentication mechanism[9]. It is also possible to delegate credentials through Kerberized TLS. However, the authors accurately point out that "there is a risk that the client may be tricked into requesting a ticket for a rogue server..." and "... the client must apply its local policy to determine whether or not to forward its credentials".

None of the browsers support Kerberized TLS. There is an implementation of Kerberized TLS [17] that relies on a local proxy. But browsers are often limited to a single proxy, complicating system management.

There is a simple alternative solution to enable the Web server to act on a user's behalf. A user can send his password (securely, of course) to the Web server. The solution has been implemented as an Apache module [6, 15, 16]. Similarly, there have been proposals to delegate a TGT to the Web server. In both cases, the Web server is given an unlimited power to impersonate users, a significant security compromise. Other mechanisms that delegate Kerberos credentials have been implemented in Minotaur [4] and SideCar [12].

PKINIT [19] allows a user to use a digital certificate in the initial Kerberos authentication. Public key distributed authentication (PKDA) [14] goes a step further and proposes for Kerberized services to support PK authentication mechanisms. For both PKINIT and PKDA, it is assumed that the user is in direct communication with the server without an interposed Web server.

Systems like Akenti [18], Keynote [1], and GAA API [13] provide ways for applications to specify and interpret security policies. These access control mechanisms are based on public key authentication mechanisms and certificates. Applications that lack an authorization mechanism of their own can greatly benefit from these mechanisms. However, our goal is to make use of already existing authorization mechanisms at the backend services.

3 Design

Our goal is to design, implement, and deploy a system that allows users to access Kerberized services through a Web server while making use of existing infrastructures and security policies.

The following considerations guide our design.

- The system must use off the shelf software whenever possible: conventional Web browsers and servers, Kerberos authentication mechanism, unmodified backend services.
- The solution must not introduce a large burden on system administrators. Administration and management of software is difficult and frequently results in security compromise of the very systems that administrators are trying to protect.

- The solution must not introduce a large burden on the user. The system must be easy to use. Added features should not require user interaction. For example, uses should not be forced to obtain additional credentials.
- The Web server is vulnerable to attacks, so it must be constrained in the actions it is allowed to take on a user's behalf.
- The system must provide a central, easily administered location for policy decisions regarding Web server's actions.

We make the following security assumptions.

- The Web server has adequate physical security.
- The Kerberized Credential Translator, described in Section 3.3, has physical security comparable to the *KDC*.
- In the simplest case, all participants trust a common certification authority. We assume the existence of a workable Public Key Infrastructure.

Our system consists of components that we describe in details in the sections below. Section 3.1 describes KX.509, a single sign-on mechanism that produces both Kerberos and PK credentials. KX.509 create a binding between user's Kerberos and PK identities. Section 3.2 discusses client authentication and the Web server's responsibilities in meeting user requests. Section 3.3 introduces our Kerberized Credential Translator, an extension to TGS that converts PK credentials to Kerberos tickets.

3.1 KX.509

In this section, we briefly describe KX.509, a Kerberized service that creates a short-lived X.509 certificate. Doster et al. describe details of the protocol [3].

The exchange of messages and other details of the protocol are shown in Figure 3. As in Kerberos, *Alice* gets a *TGT* from the *KDC*. To acquire an X.509 certificate, she first requests a service ticket for a Kerberized Certification Authority, *KCA*. At the same time, *Alice* generates a public/private key pair and prepares a message for the *KCA*. Along with the public key, she sends the KCA service ticket, $\{Alice, KCA, \mathrm{K}_{A,KCA}\}_{K_{KCA}}$, and an authenticator, $\{\mathrm{T}\}_{K_{A,KCA}}$. To ensure that the public key has not been tampered with, the HMAC of the key is sent in the same message. The session key, $K_{A,KCA}$, is used to compute the HMAC of the key.

KCA authenticates *Alice* by checking the validity of the ticket and the authenticator. It verifies that the public key has not been modified. KCA then generates an X.509 certificate and sends it back to *Alice*. The certificate is sent in the clear, but to prevent tampering, HMAC of the reply is attached. The lifetime of the certificate is set to the lifetime of the user's Kerberos credentials. The user's Kerberos identity is included inside of the certificate, creating the needed binding.

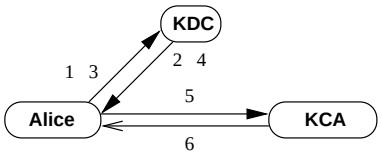
3.2 Web Server

This section describes the Web server's role in processing a request for a Kerberized service. Our goal is to provide the Web server with a means to access resources on a user's behalf. We built a Web server plugin that engages in proxy authentication by performing the following actions: (i) authenticate the user, (ii) request Kerberos credentials from a credential translator, and (iii) fulfill the user's request by accessing a Kerberized service.

In the first step, client authentication takes place in the SSL handshake. We assume *Alice* possesses a certificate verifiable by the Web server. The certificate must be issued by a certification authority trusted by the Web server. The client authentication step in SSL requires the user to sign a digest of the all the handshake message prior to this one with his private key. SSLv3 uses a keyed digest, HMAC, with the SSL session key. The signature is included in CLIENT VERIFY message of the SSL handshake.

In the second step, The Web server records a transcript of the handshake, details of which are shown in Figure 4. Then, the Web server presents the captured transcript and the SSL session key to a Kerberized Credential Translator (described in details in Section 3.3) for verification.

In the third step the Web server uses received credentials to access a Kerberized service. Revealing



1-4	Kerberos login
5. *Alice* → *KCA*:	$TKT = \{Alice, KCA, K_{A,KCA}\}_{K_{KCA}}$, $\{T\}_{K_{A,KCA}}$, Public Key, $HMAC_{K_{A,KCA}}$(Public Key)
6. *KCA* → *Alice*:	X.509 certificate, $HMAC_{K_{A,KCA}}$(X.509 certificate)

Figure 3: **KX.509 protocol.** Steps 1-4 from Kerberos are not shown. Steps 5 and 6 give the details of messages in the KX.509. *Alice* sends a service ticket, an authenticator, a public key, and its HMAC. A keyed digest is based on the session key, $K_{A,KCA}$ and prevents modifications of the data.

SSL transcript	
1. *Client* → *Server*:	CLIENT HELLO: Version = VC, Timestamp = TC, Session ID = IDC
2. *Server* → *Client*:	SERVER HELLO: Version = VS, Timestamp = TS, Session ID = IDS
3. *Server* → *Client*:	SERVER CERTIFICATE: X.509 certificate = SCert
4. *Server* → *Client*:	SERVER CERTIFICATE REQ: Cert Type = CT, CA chain = CAC
5. *Client* → *Server*:	CLIENT CERTIFICATE: X.509 certificate = CCert
6. *Client* → *Server*:	CLIENT KEY EXCHANGE: $[\text{Key material}]_{K_{WSPK}}$
6. *Client* → *Server*:	CLIENT VERIFY (SSLv3): $[HMAC_{K_{MK}}(VC, TC, IDC, VS, CT, CAC, TS, IDS, SCert, CCert)]_{K_{private}}$

Figure 4: **SSL transcript.** The messages listed constitute an SSL transcript. CLIENT HELLO carries a version, timestamp, and session id, which allows a user to resume a previous session. SERVER HELLO confirms the version and session id. Server sends its CERTIFICATE and requests user's in CERTIFICATE REQUEST. SERVER HELLO DONE specifies the end of a negotiation phase. Client sends her public key certificate in CERTIFICATE message. Client sends session information (encrypted with the server's public key, K_{WSPK}) in a CLIENTKEYEXCHANGE message. Client sends a CERTIFICATE VERIFY message, which signs important information about the session using the client's private key. The server uses the public key from the client's certificate to verify the client's identity. K_{MK} is the negotiated SSL session key. $K_{private}$ is the user's private key. A timestamp in CLIENT HELLO message is used to check freshness of the handshake.

the SSL session key in the previous step gives the credential translator the power to eavesdrop, so we force the Web server to request renegotiation to establish a new session key, one that is not known to the KCT.

User's Kerberos credentials are cached by the Web server to improve performance. The lifetime of the service ticket issued by the credential translator should be short, minimizing potential misuse of credential. At the same time, the service ticket should have a long enough lifetime so that multiple requests from a user do not incur the cost of getting a service ticket each time.

3.3 Kerberized Credential Translator

We define a *Credential Translator (CT)* as a service that converts one type of credential into another. KCA is a credential translator that translates Kerberos credentials to PK credentials. In this section, we introduce a new Kerberized credential translator service called KCT. The role of this service is to convert a user's PK credentials to Kerberos credentials.

Figure 5 shows the KCT protocol. First, the Web server authenticates to the KCT by presenting a service ticket, $\{Web\ Server, KCT, \mathrm{K}_{WS,KCT}\}_{K_{KCT}}$, and the corresponding authenticator, $\{\mathrm{T}\}_{K_{WS,KCT}}$. Along with its Kerberos credentials, the Web server sends the SSL transcript, the name of the service ticket being requested, and the SSL session key. After validating the Web server's credentials, KCT does the following:

- Verifies that the SSL transcript records a valid handshake.
- Validates user and server certificates and checks that each was signed by a trusted KCA.
- Verifies client's signature in the CLIENT VERIFY by recomputing the hash of the handshake messages up to the CLIENT VERIFY message and comparing it to the corresponding part of the SSL handshake.
- Verifies that the identity inside of the server's certificate matches the Kerberos identity. This step is needed to ensure that the Web server participated in the SSL handshake.
- Assures the freshness of the transcript by either checking the freshness of timestamps present in the hello messages or for a valid nonce. In case of latter, the Web server acquires a nonce apriori from the KCT and includes it in the SERVER HELLO message.
- Generates a service ticket for the user.
- Encrypts the session key included in the service ticket under the Web server's key, $\mathrm{K}_{WS,KCT}$.
- Sends the ticket, authenticator, and encrypted session key back to the Web server.
- Logs the transaction for auditing purposes.

We see that the KCT needs access to the database of keys maintained by the *KDC*. Consequently, the KCT requires the same physical security as the *KDC*. In practice, we run KCT on the same machine as the *KDC*, which achieves the physical security requirement and avoids the challenge of consistent replication of the Kerberos database.

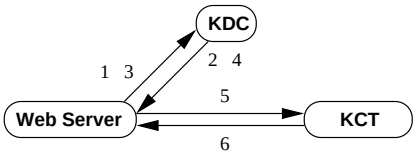
4 WebAFS Prototype

We have implemented a prototype that allows a user to submit requests to a Web server that accesses a Kerberized AFS file server on the user's behalf. An overview of the system is shown in Figure 6. In the remainder of this section, we provide details about implementation of each of the components involved in the system.

4.1 KX.509

We implemented KX.509 protocol to work for both Netscape Navigator (on UNIX, Windows, and MacOS) and Internet Explorer (on Windows). The `kx509` client and the `KCA` server are the two basic components involved in issuing users certificates. No browser modifications is required for Internet Explorer. To support the use of our certificates in Netscape Navigator we require the user to add a cryptographic module to the browser.

The Netscape browser has a special storage space for certificates, but the implementation of the certificate cache is platform dependent, undocumented,



1-4	Original Kerberos done once per lifetime a session
5. *Web Server* → *KCT*:	$TKT = \{$*Web Server*, *KCT*, $\mathrm{K}_{WS,KCT}\}_{K_{KCT}}$, $\{\mathrm{T}\}_{K_{WS,KCT}}$, SSL transcript, $\{$MK, Service$\}_{K_{WS,KCT}}$
6. *KCT* → *Web Server*:	$TKT = \{$*Alice*, *Service*, $\mathrm{K}_{WS,Service}\}_{K_{Service}}$, $\{\mathrm{K}_{WS,Service}\}_{K_{WS,KCT}}$, $\{\mathrm{T}\}_{K_{WS,KCT}}$

Figure 5: **Credential translation protocol.** Steps 1-4 are original Kerberos authentication of the Web server. It is done once per the lifetime of a service ticket for the KCT service. Steps 5 and 6 show the conversation with the KCT. *Service* is the requested backend service. Depending on the version of SSL, a master secret key, *MK* is included in the request to the *KCT*.

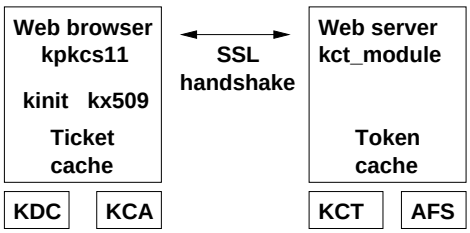


Figure 6: **WebAFS architecture.** We show details of architectural components present in the implementation of the proposed system. The new components are: kpkcs11, kx509, KCA, kct_module, and KCT. The first three components are part of credential translation from Kerberos to PK credentials. The last two are parts of translation in the other direction.

and version dependent. Thus, we chose to save certificates in user's Kerberos ticket cache.

Typically, a ticket cache stores a user's TGT and service tickets. MIT's implementation of Kerberos on UNIX allows for variable size tickets, allowing us to store any data of size up to 1250 bytes, which is sufficient to store a certificate and private key. Figure 7 shows the output of the `klist` command, which displays current contents of a ticket cache. The entry `cert.x509/umich.edu@umich.edu` is the regular service ticket for the KCA. `cert.kx509/umich.edu@umich.edu` contains the user's certificate and private key.

Netscape needs help to find our certificates. To this end, we use the browser's standard interface for adding a cryptographic module, one that we call `kpkcs11`. The browser invokes our security module before establishing an SSL connection. The `kpkcs11` looks up a certificate in the ticket cache and gives it to Netscape when client authentication is required.

Storing certificates in a ticket cache has the advantage that when a user logs out from the computer the data stored in the ticket cache is destroyed. Windows stores certificates is in the registry. There is no standard mechanism that clears out a certificate cache.

4.2 Web Server

To enable the server to act on a user's behalf, we added a module to the Apache Web server, about 1700 lines of code. The module relies on the modified version of the OpenSSL library to save the SSL transcript. The OpenSSL (version 0.9.5) modifications are minimal (less than 200 lines of code) and include a new data structure and calls to a function that saves the incoming and outgoing buffers.

Now, we look more closely at the problems that arise from differences in the SSL protocol specifications and implementations, and from harsh browser realities, which made the solution more complex and introduced delays.

In our prototype, we use timestamps present in SSL handshake to check the freshness of the handshake. Unfortunately, SSLv2 does not include timestamps in the hello messages. Worse yet, Netscape Navigator by default starts the SSL handshake with an SSLv2 CLIENT HELLO message. Only after receiving the reply from the Web server suggesting the use of SSLv3 does the browser switch to the higher version. The resulting handshake is overall a valid handshake without an SSLv3 client timestamp. To get the timestamp, we require the Web server to request renegotiation. SSL specifications allow renegotiation only after the ongoing handshake is complete. Thus, two full SSL handshakes must take place. The Opensmall SSL library was also modified to force immediate renegotiation when the SSLv2 CLIENT HELLO message is received.

One feature of SSL protocol, called a partial handshake required special attentions. When a partial SSL handshake happens, the Web server checks if AFS credentials are cached: if so, then the server proceeds with making an AFS request. Otherwise, the Web server forces an SSL renegotiation followed by a full SSL handshake. After creating a transcript, the Web server, as before, submits a request to the KCT for an AFS service ticket.

4.3 Kerberized Credential Translator

The responsibilities of the KCT are to verify the validity of a request and issue an AFS ticket on the user's behalf. To fulfill this role, KCT must have special privileges. It must be able to read the *KDC* database and get a master key of the AFS file server. Currently, tickets are issued only for AFS. In deployment, the Web server will specify the service for which it needs a ticket. In this case, KCT will need a security policy to make authorization decisions about who can ask for what.

Because Kerberos libraries are not thread safe, KCT can not be implemented as a multithreaded application. To improve performance, we spawn a process to handle an incoming request. To achieve the required physical security, we run the KCT on the same machine as the *KDC*. Implementation of KCT is about 1700 lines of code.

5 Performance

In this section we discuss the performance of the system, examining the cost of making a request to a Web server which in turn requests a service from

```
$:  klist
Ticket cache:  FILE:/tmp/krb5cc_500
Default principal:  aglo@UMICH.EDU

Valid starting      Expires             Service principal
01/19/01 13:39:56   01/19/01 23:42:15   krbtgt/UMICH.EDU@UMICH.EDU


Kerberos 4 ticket cache:  /tmp/tkt500
Principal:  aglo@UMICH.EDU

Issued              Expires             Principal
01/19/01 13:39:56   01/19/01 23:39:56   krbtgt.UMICH.EDU@UMICH.EDU
01/19/01 13:43:07   01/19/01 23:48:07   cert.x509@UMICH.EDU
01/19/01 13:43:28   01/19/01 13:43:28   cert.kx509@UMICH.EDU
```

Figure 7: **Output of klist.** KX.509 certificate and the private key are stored in the Kerberos IV ticket cache under the service names of `cert.x509` and `cert.kx509`.

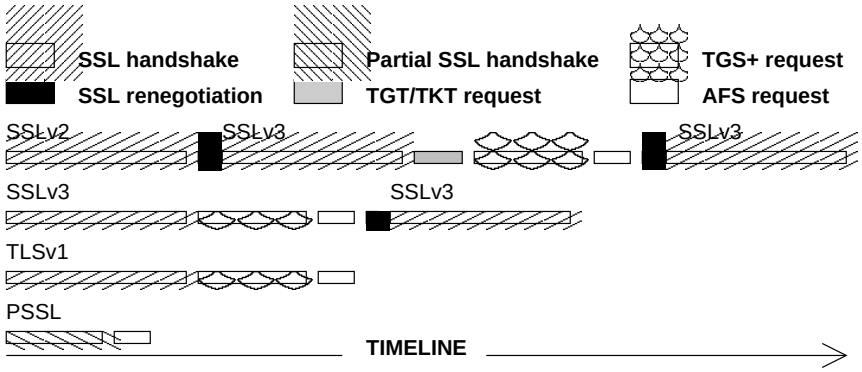


Figure 8: **Timelines of possible requests.** We show the components of a user's request in four scenarios illustrated as timelines. The legend identifies each of the components involved. We consider all different versions of an SSL protocol, v2, v3, TLSv1, and a partial handshake. In the first scenario no Kerberos credentials have been acquired by the Web server. Access to an AFS file server is used as an example.

a backend server on a user's behalf. The experiments described in this section were performed on unloaded Intel 133MHz Pentium workstation running Linux 6.2 kernel version 2.2. All the components were executed on the same machine so no network and file access delays are present. Our focus is on understanding overhead induced by the system.

The software was tested against commodity browsers, but it is hard to glean detailed measurements from commercial software, so we used OpenSSL to mimic the browser's actions. The client software is a modified version of `s_client`, an application that comes with OpenSSL. All requests were made for a 1K file. For each of the test cases 30 trials were measured and averaged results are presented. Variance was negligible.

We define several terms we use while talking about different scenarios. We define a *browser session* to be the time from launch to termination of the browser application. We define a *server session* to be the time from the first request to a Web server until the termination of the browser software. Within a browser session a user starts multiple server sessions. Requests for different files from the same Web server fall into a single server session. Requests to different Web servers are associated with different server sessions.

Figure 8 shows the breakdown of a user's request into the basic components. Four scenarios are illustrated as timelines. A typical request consists of some of the following stages:

1. SSLv2 handshake (or a partial SSL handshake)
2. Request renegotiation
3. SSLv3 handshake
4. Refresh Web server's Kerberos credentials
5. Request Kerberos credentials from KCT
6. Request renegotiation
7. SSLv3 handshake

We divided a user's request into different components, for example an SSL handshake, and measured each of the components individually. Table 1 shows the delays associated with the basic components involved in a user's request.

Components	**Delay(s)**
1 handshake	1.252
2 handshakes	2.495
TGT/KCT_TKT	0.029
KCT request	0.255
Partial SSL	0.022

Table 1: **Delays of basic components.** The first row shows SSL handshake latency. The second row shows the delays seen after two consecutive SSL handshakes with a request to renegotiate in between. The third row shows the time for the Web Server to refresh Kerberos credentials. The fourth row shows delays associated with the KCT request/reply. The last row shows the latency for a partial SSL handshake. Rows 1,2, and 5 reflect end-to-end delays seen by the user. Rows 3 and 4 measure network cost seen by the Web server while talking to the *KDC* and KCT.

End-to-End	**Time(s)**
SSLv2 hello no TGT	4.080
SSLv2 hello 1st request	4.040
SSLv2 cached creds	2.500
SSLv3 hello no TGT	2.857
SSLv3 hello request	2.801
SSLv3 cached creds	1.252

Table 2: **End-to-end delays.** Each of the scenarios represent a possible user request. We measured end-to-end latency seen by the user.

Table 2 shows the end-to-end delays seen by the user for different types of requests. We describe each of the scenarios in details and point out which ones are more common. We divide requests into two groups based on whether or not user's credentials are cached at the Web server.

- **No cached credentials.** First, we consider the cases where user's credentials are not cached. This happens when a user is making the first request to the Web server or when her credentials have been evicted from the Web server's LRU cache.
  - **Once a day:** *SSLv2 hello no TGT* and *SSLv3 hello no TGT*. In these two scenarios, the Web server has stale credentials so the user's request gets penalized by the time needed to get new Kerberos credentials. The lifetime of our Web server's TGT is 24 hours.
  - **Once per server session:** *SSLv2 hello 1st request.* When contacting a Web server for the first time, the default behavior of Netscape Navigator is to start with an SSLv2 CLIENT HELLO message. Unless the browser is restarted, all subsequent requests will start with an SSLv3 CLIENT HELLO. This scenarios measures the overhead of the three handshakes and a KCT request. The first additional handshake is to get a valid timestamp in the CLIENT HELLO message. The second additional handshake renegotiates the SSL session key which was revealed to the KCT.
  - **Most common request:** *SSLv3 hello request.* Internet Explorer starts with an SSLv3 CLIENT HELLO. Any requests from this browser either fall into this category or the partial handshake.
- **Cached credentials.** Now, we review the scenarios where user's credentials are cached at the Web server. Caching is important because it saves the overhead of getting Kerberos credentials. Furthermore, no SSL renegotiation plus handshake is needed at the end. The only overhead the system imposes is that associated with token management.
  - **Frequent:** *Partial handshake cached credentials* The lifetime of the session key negotiated in the full handshake is configurable by the web server. If more than one request is made within five minutes of a full handshake, a partial handshake takes place. Five minutes is a default value used by Apache Web servers. We can safely assume that user's credentials are already cached at that point. The time required for a partial handshake is considerably smaller than for a full handshake. The frequency of these requests depends on user's access pattern.
  - **Common:** *SSLv3/TLSv1 cached credentials.* Once the user contacts a Web server, his credentials are cached until they get evicted due to expired lifetime or lack of space. When requests to the Web server are separated by more than 5 minutes, a user experiences end-to-end delay presented in last row of Table 2.
  - **Unlikely:** *SSLv2 hello cached credentials.* The browser sends an SSLv2 CLIENT HELLO message to the Web server if it never contacted it within the current browser session. However, it is still possible for user's credentials to be cached at the Web server, if the user restarted the browser within the lifetime of the cached credentials.

To summarize, an SSL handshake costs 1.252 s. Delays associated with refreshing a TGT and making KCT requests are small, 0.022s and 0.255s respectively. The overhead is amortized over a browser session.

In the most common case, credentials are cached and SSLv3 connections are used, so the system incurs negligible overhead. Further testing in more complex environments is necessary and will be done in the future. However, these preliminary results are encouraging.

6 Discussion

In this paper we described a system that provides users with access to Kerberized services through a browser. In this section we summarize the functionality of each of the components involved in the system and discuss open questions.

While many backend services use Kerberos for authentication, Web servers authenticate with public key cryptography. We address the mismatch of authentication credentials between the Web server and Kerberized service by introducing a new service that translates PK credentials to Kerberos tickets. The Web server engages in proxy authentication. The process consists of SSL client authentication, request to a credential translation service, and finally authentication to the Kerberized service on user's behalf.

We built a single sign-on mechanism that allows users to get X.509 certificates in addition to their Kerberos credentials. Through the KX.509 protocol, we create a binding between a user's Kerberos and PK identities. The issues surrounding this binding are quite broad and must be further resolved.

A client uses his certificate to establish an authenticated and secured channel to a Web server. The Web server logs an SSL transcript and makes an authenticated request to a new service that translates user's PK credentials to Kerberos credentials.

The authorization model of the credential translator is primitive and is the focus of our future work. The current model supports generic access control lists: for each Web server there is an entry listing the Kerberized services for which it can request tickets. We are looking into integrating Akenti [18] access control mechanisms into the system.

We built a prototype, WebAFS, that allows users to access AFS restricted files through browsers. It requires minor modifications to existing software, such as a plugin module to the Netscape Navigator and modifications to the OpenSSL library. We wrote four components: kx509 and KCA take care of issuing user's certificate, an Apache module services requests, and a KCT translates between two types of credentials.

We measured the overhead introduced by our system. We showed the delays associated with the building blocks of a user's request. The results show that substantial amount of time is spent in establishing an SSL connection, but that requesting credentials for the server is amortized over a browser session.

Credential translation need not apply only to Web traffic. It is extensible to any SSL-enabled client and SSL-enabled server communication. Furthermore, credential translation need not to be limited to producing Kerberos credentials. Consider a remote login application an SSL-enabled Telnet. Assuming a user has a certificate on his local computer, we can eliminate the need to sent his password over the network. A user can use his certificate, mutually authenticate with the remove host, and empower it to act on user's behalf. We are considering these and other extensions on our future work.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The keynote trust managment system version 2, September 1999. RFC2704.

[2] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. RFC2246.

[3] W. Doster, M. Watts, and D. Hyde. The KX.509 protocol. CITI Technical Report 01-2, Februaru 2001.

[4] P. Dousti. Project minotaur: Kerberizing the web. Software at Carnegie Mellon University.

[5] ITU-T (formerly CCITT) Information technology Open Systems Interconnection. Recommendation x.509: The directory authentication framework, December 1988.

[6] Apache Software Foundation. Apache web server. http://www.apache.org.

[7] A. Freier, P. Karton, and P. Kocher. Secure socket layer 3.0, March 1996. Internet draft.

[8] A. Freier, P. Karton, and P. Kocher. The SSL protocol version 3.0, March 1996. Netscape Communications Corporation.

[9] M. Hur and A. Medvinsky. Kerberos cipher suites in transport layer security (TLS), May 2001. Internet draft.

[10] R. Needham and M. Shroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993 – 999, December 1978.

[11] C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[12] SideCar project. http://www.cit.cornell.edu/kerberos/sidecar.html.

[13] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *Proceedings of the DISCEX*, January 2000.

[14] M. Sirbu and J. Chuang. Distributed authentication in kerberos using public key cryptography. In *Symposium On Network and Distributed System Security*, 1997.

[15] Stone Cold Software. Apache kerberos module. http://stonecold.unity.ncsu.edu/software.

[16] D. Song. Kerberized WWW access. http://www.monkey.org/ ~dugsong/krb-www.

[17] V. Staats. Kerberized TLS, June 2000. Private communications.

[18] M. Thompson, W. Johnson, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate based access control for widely distributed resources. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[19] B. Tung, C. Neuman, and J. Wray. Public key cryptography for initial authentication in kerberos, April 2000. Internet draft.