Kerberized Credential Translation: A Solution to Web Access Control

Olga Kornievskaia
Peter Honeyman
Bill Doster
Kevin Coffman

Center for Information Technology Integration
University of Michigan
Ann Arbor

{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 establish 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 importance 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 that produces both Kerberos and public key credentials. The Web server uses a plugin that translates 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.

1 Introduction

Access control for Web space is often viewed in terms of gating access to Web pages where the job of the Web server is limited to simple file reads. The functionality provided by Web servers has grown considerably making it the most popular technology on the Internet. With the expansion of the Internet, many new kinds of services are accessible from the Web, increasing Web servers' importance and scope. For example, a Web server may serve information stored in backend databases. A Web interface to backend services is considered to be more user-friendly and accessible compared to predominant text-based interfaces.

The possibilities opened by the use of a Web server to access a variety of backend services pose challenging questions on how to retain access control of backend services. A Web server could potentially become another access control decision point, increasing the burden on the server and its administrators. It would have to comply with the same security requirements as all of the backend services it fronts, increasing its potential as a place for system compromise.

A solution that provides end-to-end authorization would allow the end service to retain control over the authorization decisions. Furthermore, it would obviate constructing and maintaining consistent replicas of authorization policies.

In practice, authorization mechanisms are tied to authentication mechanism: end-to-end authorization requires end-to-end authentication. 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. To provide end-to-end authorization, we address the problem of end-to-end authentication.

We motivate the end-to-end authentication problem by considering the following scenario:

Alice attends the University of Michigan, where she enjoys access to a variety of computing services. 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, then the flexibility of AFS access controls disappear.

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

If an AFS client is running on Alice's workstation, a simple solution presents itself. Instead of making an HTTP request, a user can access AFS file space directly with `file://localhost/afs/`$\cdots$. 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.

In this scenario, end-to-end authentication presents the question of how to convey Kerberos credentials to the Web server. One solution is for the client to acquire the needed credentials and delegate them to the Web server. 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 a user to know when it is safe to give her password to a Web server.

Kerberos supports a mechanism for delegation of rights. However, browsers do not support any form of delegation. 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 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 the Web server's actions is required.

This paper describes the design, implementation and performance of a system that provides controlled access to Kerberized services through conventional browsers. 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 users' 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 and presents directions for future work.

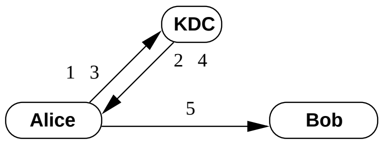
2 Background

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 universally supported on the Web. The Internet spans many Kerberos 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 [19] is a network authentication system based on the Needham-Schroeder protocol [17]. Kerberos authentication is illustrated in Figure 1. Authentication is achieved when one party proves knowledge of a shared secret to another. To avoid



Login phase:	once per session
1. *Alice* $\rightarrow$ *KDC*:	"Hi, I'm *Alice*"
2. *KDC* $\rightarrow$ *Alice*:	TGT = $\{Alice, \text{TGS}, \text{K}_{A,TGS}\}_{K_{TGS}}$, $\{\text{K}_{A,TGS}, \text{T}\}_{K_A}$
Accessing services:	every time before talking to a service
3. *Alice* $\rightarrow$ *TGS*:	*Alice*, *Bob*, TGT, $\{\text{T}\}_{K_{A,TGS}}$
4. *TGS* $\rightarrow$ *Alice*:	TKT = $\{Alice, Bob, \text{K}_{A,B}\}_{K_B}$, $\{\text{K}_{A,B}, \text{T}\}_{K_{A,TGS}}$
5. *Alice* $\rightarrow$ *Bob*:	"Hi, I'm *Alice*", TKT, $\{\text{T}\}_{K_{A,B}}$

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 the TGS and KDC. K_A is a key shared between *Alice* and the KDC, derived from *Alice's* password. $\text{K}_{A,TGS}$ is a session key for *Alice* and TGS. $\text{K}_{A,B}$ is a session key for *Alice* and *Bob*. T is a timestamp used to prevent replay attacks.

quadratic explosion of key agreement requirements, 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. The 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, $\{Alice, Bob, \text{K}_{A,B}\}_{K_B}$. To authenticate to *Bob*, *Alice* constructs a timestamp based authenticator, $\{\text{T}\}_{K_{A,B}}$, proving to *Bob* that she knows the session key inside of the service ticket.

2.2 Overview of SSL/TLS

Secure Socket Layer [1] [12, 13] 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 universal among Web browsers and servers.

[1]SSL is renamed by IETF as Transport Layer Security, TLS [6]

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 [7, 22] and X.509 [11] identity certificates.

SSL supports mutual authentication. First, a user authenticates the server. The user has the responsibility to assure that it can trust the certificate received in the Certificate message from the server. That responsibility includes verifying the certificate signatures, validity times, and revocation status. The user then 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.

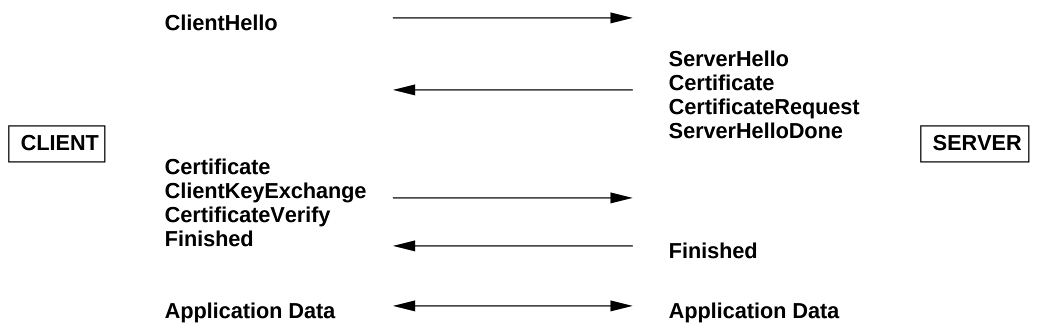


Figure 2: **SSL handshake protocol.** ClientHello carries a version, random value (part of which is a timestamp), and session id (which allows a user to resume a previous session). A timestamp is used to guarantee the uniqueness of the random value. ServerHello confirms the version and session id. Server sends its Certificate and requests the user's in CertificateRequest. ServerHelloDone specifies the end of a negotiation phase. Client sends her public key certificate in Certificate. Client sends session information (encrypted with the server's public key) in ClientKeyExchange. Client sends CertificateVerify which contains a signed digest of messages exchanged upto this pointed. 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.

To reduce the risk of key compromise, the SSL protocol supports renegotiation of the security context. A client initiates a new handshake by sending a ClientHello message. If the server wishes to initiate a handshake, it sends an empty ServerHello message to the client and the client responds with a new ClientHello.

Establishing an SSL session requires sophisticated cryptographic calculations and numerous 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 ClientHello. It is up to the server to decide whether to allow the reuse of the session. We call this a partial SSL handshake.

2.3 Related Work

This section describes related work on distributed authorization and interoperability among authentication mechanisms. Many efforts have focused on creating formal systems that allow reasoning about delegated (restricted) rights and express (general) authorization statements. Many researchers have focused on creating powerful and expressive languages for making and verifying security assertions efficiently. Among them are SPKI/SDSI [5, 10, 21], PolicyMaker, and its successor KeyNote [2, 3, 4], GAA API [23], Akenti [28], and Neuman's proxied authorization [18]. Applications that lack an authorization mechanism of their own greatly benefit from these mechanisms. However, our goal is to make use of already existing authorization mechanisms at the backend services.

There has been work on interoperability of Kerberos and PKI. PKINIT [30] allows a user to use a digital certificate in the initial Kerberos authentication. Public key distributed authentication (PKDA) [24] 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.

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 [1, 27, 25]. In this case, the Web server is given an unlimited power to impersonate users, a significant security risk.

There are several projects that propose to use Kerberos for Web authentication without sending user passwords. Minotaur [9] depends on a client side plugin in to acquire a service ticket for a Web server. However, it has been shown that, in its current design, Minotaur's handling of HTTP POST is insecure. Another system, called SideCar [20], achieves Kerberos authentication by talking to a dedicated process on a client's machine. Failure to start the daemon process prevents the client from being able to do Web authentication. Yet another solution makes use of the extension to TLS cipher suites that includes Kerberos as an authentication mechanism [14].

Kerberos authentication to a Web server is not enough for end-to-end authorization. There must be support for delegating Kerberos credentials after the client authenticates to the Web server, which is addressed by Jackson et al. in their proposal on how to delegate credentials (currently, Kerberos and X509 certificates) in TLS [15]. There are a few problems with considering this approach as a solution. First, no browsers currently support Kerberized TLS. There is an implementation of Kerberized TLS [26] that relies on a local proxy, but browsers are often limited to a single proxy, complicating system management. Furthermore, the description of the exact content of the protocol is vague, making it hard to validate the security of the protocol.

Tuecke et al. [29] propose a specific delegation mechanism that allows a user to delegate an identity certificate to a third party. The receiver must engage in a special verification process that validates these certificates to identify the real sender. Authentication to a commodity server with these certificates cannot be considered secure, as each entity in the delegated path serves as a certification authority and can create a certificate under whatever identity it pleases.

The problem with delegation is that the client may be tricked into requesting a ticket by a rogue server. It has been repeatedly demonstrated that we can not always trust a valid server's certificate, most recently by the Microsoft/VeriSign debacle [16]. Delegation places a large administrative burden on the client. First, a client must be able to understand and apply security policies to determine whether or not to forward his credentials. To avoid the hassle, users frequently allow for unlimited and unchecked delegation. It is not reasonable to assume that for each compromised Web server each user will update her security policy to address the problem. Lastly, browser support for restricted delegation always leaves us wishing for more.

3 Design

Our goal is to design, implement, and deploy a system that allows users access to 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.
- We depend on minimal PKI functionality. We are not trying to solve PKI problems such as reliable and efficient key revocation. This leads to the following additional assumptions.
- We assume the ability to instantiate a root certification authority, be it a self-signed CA certificate or one signed by an acknowledged root CA, such as Versign.

- We assume the CA certificate can be distributed efficiently and securely. All the client machines need to have such a certificate installed in their Web browser CA certificate list (unless the certificate is signed by one of the well acknowledged root CAs). All other servers in the system need to possess the CA certificate.

- We assume the root certificate can be revoked.[2] A mechanism is needed that notifies all clients and servers.

- We assume the (long-lived) certificates issued to the services can be revoked.

Our system consists of components that we describe in detail in the sections below. Section 3.1 describes KX.509, a single sign-on mechanism that produces both Kerberos and PK credentials and creates a binding between them. 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 [8].

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.

The KCA authenticates *Alice* by checking the validity of the ticket and the authenticator. It verifies that the public key has not been modified. The KCA then generates an X.509 certificate and sends it back to *Alice*. The certificate is sent in the clear; to prevent tampering, the 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 the certificate, creating the necessary binding.

[2]We know it usually can't.

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.

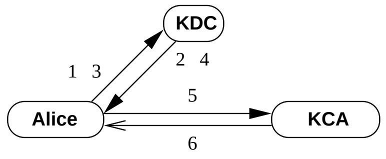
In the first step, client authentication takes place in the SSL handshake. We assume *Alice* possesses a certificate verifiable by the Web server, i.e., the certificate must be issued by a certification authority trusted by the Web server. The client authentication step in SSL requires the user to sign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

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 Section 3.3) for verification.

In the third step, the Web server uses received credentials to access a Kerberized service. Revealing the SSL session key in the previous step gives the credential translator the power to eavesdrop, so we require the Web server to request renegotiation to establish a new session key, one that is not known to the KCT. This is a trade-off between security and performance. [3]

The user's Kerberos credentials are cached by the Web server to improve performance. The lifetime

[3]One could argue that because the KCT is as powerful as the KDC and can impersonate any user, then the KCT itself can place a request to a Kerberized service, and, thus, the KCT can be trusted with the knowledge of the SSL session key.



1-4	Kerberos login
5. *Alice* $\rightarrow$ *KCA*:	TKT, Auth, Public Key, $\text{HMAC}_{K_{A,KCA}}$(Public Key)
6. *KCA* $\rightarrow$ *Alice*:	X.509 certificate, $\text{HMAC}_{K_{A,KCA}}$(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 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 modification of the data.

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

Figure 4: **SSL transcript.** The messages listed constitute an SSL transcript. CLIENTHELLO carries a version, random number (first four bytes occupied by a timestamp), and session id, which allows the user to resume a previous session. SERVERHELLO confirms the version and session id. A server sends its CERTIFICATE and requests the user's in CERTIFICATEREQUEST. SERVERHELLODONE specifies the end of the negotiation phase. A client sends her public key certificate in CERTIFICATE. A client sends the session information (encrypted with the server's public key, K_{WSPK}) in CLIENTKEYEXCHANGE. Key material included in this message depends on the key exchange protocol. For example, in the case of RSA, a client generates a premaster secret that both parties use to generate key (encryption and digest) material, including K_{MK}. A client sends CERTIFICATEVERIFY, which includes a key-based digest of all the messages prior to this one signed with 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 key generated from the key material sent by the client in CLIENTKEYEXCHANGE. We call it the *SSL session key.* $K_{private}$ is the user's private key. A timestamp in CLIENTHELLO is used to verify freshness of the handshake.

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 lifetime long enough that multiple requests from the user do not incur the cost of getting a service ticket each time. A compromise of the Web server enables the intruder to use the currently cached credentials and to acquire credentials on the user's behalf for any of the requests to this compromised Web server.

3.3 Kerberized Credential Translator

We define a *Credential Translator (CT)* as a service that converts one type of credential into another. In this section, we introduce a Kerberized credential translator (KCT) that converts 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, $K_{WS,KCT}\}_{K_{KCT}}$, and the corresponding authenticator, $\{T\}_{K_{WS,KCT}}$. 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, the KCT performs the following steps:

- Validates user and server certificates and checks that each was signed by a trusted CA.
- Verifies client's signature in CertificateVerify by recomputing the hash of the handshake messages up to CertificateVerify 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 checking the freshness of timestamps or nonces present in the hello messages. In the latter case, the Web server acquires a nonce from the KCT and includes it in ServerHello.
- Generates a service ticket for the user.
- Encrypts the session key included in the service ticket under the Web server's session key, $K_{WS,KCT}$.
- Returns the ticket, authenticator, and encrypted session key to the Web server.
- Logs the transaction for postmortem auditing.

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 the KCT on the same hardware as the KDC, which achieves the physical security requirement and sidesteps the challenge of consistent replication of the Kerberos database.

4 WebAFS Prototype

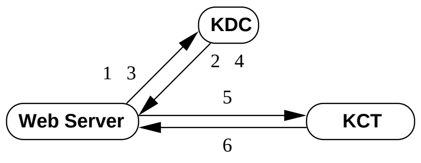
We have implemented a prototype that allows a user to submit requests to a Web server that accesses a Kerberized AFS file server on the user's behalf. An overview of the system is shown in Figure 6. In the remainder of this section, we provide details about the implementation of each of the components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

4.1 KX.509

We implemented the 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. Additional details about the implementation can be found in a related technical report [8].

Navigator maintains a private cache of certificates, but the implementation is platform dependent, undocumented, and version dependent. Thus, we elect to save certificates in user's Kerberos ticket cache, which requires the user to add a cryptographic module to the browser. No such modification is required for Explorer.

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 a private key. Figure



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

Figure 5: **Credential translation protocol.** Steps 1-4, not shown, indicate Kerberos authentication of the Web server. They are performed 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, an SSL secret key, MK is included in the request to the KCT.

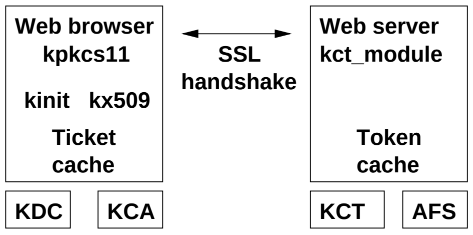


Figure 6: **WebAFS architecture.** We show details of architectural components present in the implementation of the proposed system. The new components are: kpkcs11, kx509, KCA, kct_module, and KCT. The first three components are for credential translation from Kerberos to PK credentials. The last two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

7 shows the output of the `klist` command, which displays the current contents of a ticket cache. The entry `cert.x509/umich.edu@umich.edu` is the service ticket for the KCA. `cert.kx509/umich.edu@umich.edu` contains the user's certificate and private key.

As we mentioned, Navigator needs help to find our certificates. To this end, we use the browser's standard interface to add a cryptographic module that we call `kpkcs11`. When client authentication is required, `kpkcs11` looks up a certificate in the ticket cache and gives it to Navigator.

In our implementation, the user identity information that KCA includes in the certificate is retrieved from a naming service (an X.500 directory). Given a Kerberos principal, KCA looks up the user's first and last name. Additionally, at the end of the distinguished name we attach an email field with the principal name in the local part and the realm in the remote part, for example, `aglo@UMICH.EDU`.

4.2 Web Server

To enable the server to act on a user's behalf, we added a module to the Apache Web server, under 2000 lines of code. The module relies on a version of the `openssl` (0.9.5) library modified to save the SSL transcript. Modifications to the library are minimal (under 200 lines of code) and include a new data structure and calls to a function that saves the incoming and outgoing handshake messages.

We now look more closely at the problems that arise from differences in the SSL protocol specifications and implementations, and from harsh browser realities, which make the solution more complex and introduce 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, Navigator by default starts the SSL handshake with an SSLv2 ClientHello 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, but lacks 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, so two full SSL handshakes must take place.

One feature of the SSL protocol, called a partial handshake, requires special attention. When a partial SSL handshake happens, the Web server checks if AFS credentials are cached; if so, then the server proceeds with the 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 the request and issue an AFS ticket on the user's behalf. To fulfill this role the KCT must have special privileges: it must be able to read the KDC database and use the key of the AFS Kerberos principal. Currently, tickets are issued only for AFS. In deployment, the Web server will specify the service for which it needs a ticket, at which point the KCT will need a security policy to make authorization decisions about who can ask for what.

As of this writing, the MIT Kerberos libraries are not thread-safe, so the KCT cannot be implemented as a multithreaded application. To improve performance, we spawn a process to handle incoming requests. To achieve the required physical security, we run the KCT on the same hardware as the KDC. Implementation of the KCT is under 2000 lines of code.

5 Performance

In this section we discuss the performance of the system by examining the cost of making a request to a Web server, which, in turn, requests a service from a backend server on a user's behalf. The experiments described in this section were performed on an unloaded Intel 133MHz Pentium workstation running RedHat Linux 6.2 (kernel version 2.2). Our focus is on understanding overhead induced by the system, so all the components were executed on the same hardware to avoid network and file access delays.

The software was tested against commodity

```
$: klist
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 13:40:07   01/19/01 23:39:56   cert.x509@UMICH.EDU
01/19/01 13:40:07   01/19/01 23:39:56   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.kx509`. `cert.x509` is the service ticket for the KCA. the other entry is the service ticket for the TGS.

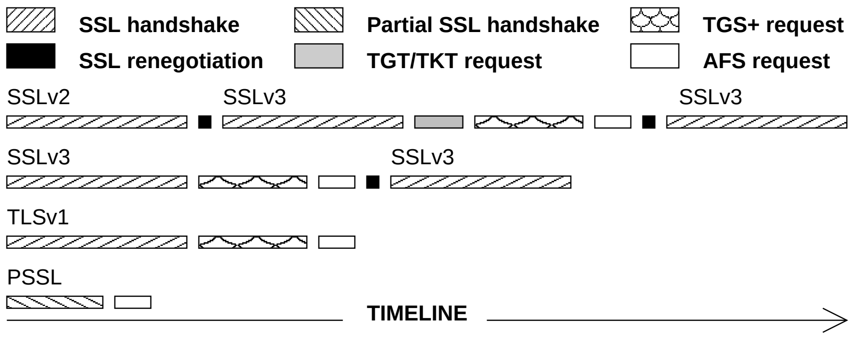


Figure 8: **Timelines for WebAFS requests.** We show the components of a user request in four scenarios illustrated as timelines. The legend identifies each of the components involved. We consider all the different versions of an SSL protocol, v2, v3, TLSv1, and a partial handshake. Access to an AFS file server is used as an example.

browsers, but it is hard to glean detailed measurements from commercial software, so we used `openssl` tools to mimic the browser's actions. We used `openssl`'s generic SSL/TLS client, called `s_client`. All requests were made for a 1K file. For each of the test cases 30 trials were measured and averaged.

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 application. 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

Components	**Delay(s)**
1 handshake	1.25
2 handshakes	2.50
TGT/KCT_TKT	0.03
KCT request	0.26
Partial SSL	0.02

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 latencies seen by the Web server while talking to the KDC and KCT.

We divide 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.

End-to-End	**Time(s)**
SSLv2 hello no TGT	4.08
SSLv2 hello 1st request	4.04
SSLv2 cached creds	2.50
SSLv3 hello no TGT	2.86
SSLv3 hello request	2.80
SSLv3 cached creds	1.25

Table 2: **End-to-end delays.** Each of the scenarios represents 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 detail and point out which ones are more common. We divide requests into two groups, depending on whether 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 by the Web server 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 Navigator is to start with an SSLv2 CLIENTHELLO message. Until the browser is restarted, all subsequent requests will start with an SSLv3 CLIENTHELLO. This scenario measures the overhead of the three handshakes and a KCT request. The first additional handshake produces a valid timestamp in the CLIENTHELLO message. The second additional handshake renegotiates the SSL session key, which was revealed to the KCT.
    - **Most common request:** *SSLv3 hello request*. Explorer starts with an SSLv3 CLIENTHELLO. Any requests from this browser fall either into this category or the partial handshake.
- **Cached credentials.** We now review the scenarios where the 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 the user's access pattern.

- **Common:** *SSLv3/TLSv1 cached credentials.* Once the user contacts a Web server, her 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 five 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 ClientHello message to the Web server if it never contacted it within the current browser session. However, it is still possible for the 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 1.25 seconds. Delays associated with refreshing a TGT and making KCT requests are small: 0.02 and 0.26 seconds, respectively.

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 point out the issues that require further research.

While many backend services use Kerberos for authentication, Web servers use SSL to 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, a request to a credential translation service, and finally authentication to the Kerberized service on a user's behalf.

We built a single sign-on mechanism that allows users to obtain 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 require further study.

A client uses her certificate to establish an authenticated and secured channel to a Web server. The Web server logs the SSL transcript and makes an authenticated request to a new service that translates the 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 [28] access control mechanisms into the system.

We built a prototype, WebAFS, that allows users to access restricted AFS files through browsers. It requires minor modifications to existing software, such as a plugin module to 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 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 a substantial amount of time is spent in establishing an SSL connection, but that requesting credentials for the server is nicely 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 be limited to producing Kerberos credentials. Consider a remote login application such as an SSL-enabled Telnet. Assuming a user has a certificate on his local computer, we can obviate the need to send his password over the network. A user can use his certificate, mutually authenticate with the remote host, and empower it to act on his behalf. We are considering these and other extensions in our future work.

7 Acknowledgments

We thank the anonymous reviewers for their helpful comments. We also thank CITI staff for their participation in the project and their valuable comments. Our special thanks go to Dr. Naomaru Itoi for many insights and constant encouragement.

References

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

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust managment system version 2, September 1999. RFC 2704.

[3] M. Blaze, J. Feigenbaum, and A. Keromytis. Keynote: Trust management for public-key infrastructure. In *Proceedings Cambridge 1998 Security Protocols International Workshop*, 1998.

[4] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *Proceedings of Second International Conference on Financial Cryptography*, 1998.

[5] D. Clarke, J. Elien, C. Ellison, F. Morcos, and R. Rivest. Certificate chain discovery in SPKI/SDSI. Draft Paper, November 1999.

[6] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. RFC 2246.

[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transaction on Information Theory*, 22:644–654, November 1976.

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

[9] P. Dousti. Project Minotaur: Kerberizing the Web, software at Carnegie Mellon University. `http://andrew2.andrew.cmu.edu/minotaur`.

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, September 1999. RFC 2693.

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

[12] A. Freier, P. Karton, and P. Kocher. Secure Socket Layer 3.0, March 1996. Internet draft.

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

[14] M. Hur and A. Medvinsky. Kerberos cipher suites in Transport Layer Security (TLS), May 2001. Internet draft.

[15] K. Jackson, S. Tuecke, and D. Engert. TLS delegation protocol, February 2001. Internet draft.

[16] Microsoft Security Bulletin MS01-017. Erroneous VeriSign-issued digital certificates pose spoofing hazard, March 2001.

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

[18] C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributing Computing Systems*, May 1993.

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

[20] SideCar project. Software at Cornell University. `http://www.cit.cornell.edu/kerberos/sidecar.html`.

[21] R. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rump session, 1996.

[22] R. Rivest, A. Shamir, and L. Adleman. A method of obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

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

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

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

[26] V. Staats. Kerberized TLS, June 2000. Private communication.

[27] Stone Cold Software. Apache Kerberos Module. `http://stonecold.unity.ncsu.edu`.

[28] 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.

[29] S. Tueke, D. Engert, and M. Thompson. Internet X.509 public key infrastructure impersonation certificate profile, February 2001. Internet draft.

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