Best Practices for Integrating Kerberos into Your Application

This paper describes best practices for application developers who wish to add support for the Kerberos Network Authentication System to their applications. Best practices include a discussion of approaches for integrating Kerberos, recommendations for when these approaches should be used, and examples of code using the approaches.

This paper focuses on traditional client-server or peer-to-peer applications. In particular this paper focuses on applications that:

- Exchange data over the network
- Run code on each network endpoint

Why Integrate Kerberos?

Kerberos is a key component of providing access control in today's enterprises. At the beginning of the day, users log into Kerberos, obtaining *credentials* once and then using applications throughout the day. Kerberos provides the appropriate security exchanges and guarantees so that these applications need not ask for a username or password. The user gets the convenience of not being constantly asked to log into each application. Each application gets a consistent way of naming users. The organization gets a single point at which to enforce security policy. Other components of an organization's access control strategy typically include a directory to store authorization information and attributes about users as well as management infrastructure. However, Kerberos provides the link allowing applications to participate in the enterprise's access control strategy. While this model is common in large enterprises, it is expanding to workgroups and peer-to-peer settings as Kerberos becomes more integrated into operating system platforms.

As discussed in "The Role of Kerberos in Modern Information Systems" paper (forthcoming), there are many elements to an enterprise access control strategy. Kerberos allows applications to use the access control elements of the organization or platform rather than requiring each application to provide a complete access control strategy. Even if applications did provide complete access control capabilities, integration with other enterprise authentication and authorization services would be desirable to avoid the cost of managing multiple access control elements. As an example of this integration, if an account is disabled then that account will be unable to use any applications that are integrated with the account management and authentication services. Applications can also take advantage of any authorization information

available in a directory. It may be possible to automatically provision users to use applications when the user's account is created in the organization.

Single Sign On

The most visible benefit to Kerberos for end users is *single sign-on*. The user need not sign onto each application but instead can sign onto their computer once. Kerberos accomplishes single sign-on by storing credentials that typically last approximately one work day. When the user signs onto the computer, the local Kerberos implementation contacts the *Key Distribution Center (KDC)* to authenticate the user to the KDC. When authentication succeeds, the KDC issues a *ticket*. The ticket is a time-limited message from the KDC to itself attesting to successful authentication. This ticket along with a session key known only to the local computer and the KDC forms a *credential* that can be used to sign onto applications. When Kerberos wishes to contact an application, it presents the ticket along with proof that the session key is known by the client to the KDC and receives a new service ticket for the application that is being contacted. This ticket along with an application-specific session key is used to securely contact the application.

The KDC serves as a central point to enforce the organization's authentication policy and to enforce general policies about account management.

Security Services

An effective access control policy relies on a number of security services. This section discusses the services that Kerberos provides and how these fit into application access control. Providing these services is an important goal of integrating Kerberos into an application. As such, a plan for integrating Kerberos into an application should carefully consider how these services are provided. Validating that these services are provided is essential to validating the integration of Kerberos into an application.

Authentication

Authentication is the process of verifying to a sufficient degree of confidence claims about a party or message. Typically a network application needs to know some attributes, such as the name, about the party sending it messages.

Kerberos separates authentication into two phases. Initial authentication takes place between the Kerberos client and the KDC. The mechanisms used will be set by site policy; typical examples include pre-shared secrets (passwords) or smart cards. Later, though, the client authenticates to the application. As a side effect of this exchange, the client and application share a session key that may be used in subsequent, cryptographically-protected communications.

Today's network applications require that both sides of a connection be authenticated in order to prevent phishing and other malicious attacks. It is just as important that a server authenticate to its clients so their access control can be maintained as it is for the clients to authenticate to the server. Fortunately, Kerberos makes mutual authentication easy. Kerberos is symmetric; any two parties that can authenticate in one way can also authenticate in the other direction.

Confidentiality and Integrity

Authentication of the parties to a network exchange is not sufficient to enforce reasonable access control objectives. The contents of the message themselves need to be protected. If the messages are not protected then an attacker could modify the messages, defeating the access control policy. Kerberos provides facilities to make sure that messages are not changed as they travel across the network. The messages can optionally be encrypted so only the parties that know the session key can examine their contents.

Some applications already have a protocol layer responsible for message security. For example, applications may use Transport Layer Security (TLS, the successor to SSL) or IPsec to provide message security. A technique called *channel binding* can be used to tie the Kerberos authentication to the message security layer so that the application has confidence that the messages are transported between the same parties that are authenticated by Kerberos.

Authorization

Once Kerberos has successfully authenticated the client to the service and once appropriate security services are put in place, the client and service still need to make authorization decisions in order to meet access control objectives. What objects should the client be allowed to access? What permissions should the client have? Is the service the one that the client expected to talk to? To what extent is the service trusted by the client? All these questions are part of the authorization that the client and service need to perform.

Kerberos authenticates the client and service. In some environments, Kerberos also provides information on group membership of the client. The rest of authorization is a matter for the platform and application. The "Role of Kerberos" paper describes how Kerberos fits into the platform's authorization function. This paper discusses some specific issues where authorization touches on how Kerberos is integrated into the application. The section on Federation and Client Names discusses issues surrounding matching Kerberos names to application names. The section on SIDs and the PAC discusses how Microsoft's Active Directory conveys group membership to a service.

Protocol Facilities for Integrating Kerberos

There are two important axes that describe the approaches for integrating Kerberos. The first is the protocol facility or network messages that are used. The second is the Kerberos implementation that is used. Not all combinations are possible: not all implementations provide access to all protocol facilities. This section describes the first axis and the next section describes the implementation axis.

Kerberos provides applications the flexibility to determine what network-level messages to adopt. There is very broad consistency in how applications interact with the KDC and in the initial messages a client uses to authenticate to a service. However even the message that a client uses to authenticate to a service needs to be transported in some application-specific framing. In addition, applications can adopt multiple

different approaches for integrity and confidentiality of user data. This section describes the three most common approaches.

Raw Kerberos messages are described to establish context. They should not generally be used in new applications. The GSS-API Kerberos mechanism is the preferred way to integrate Kerberos into applications. SASL is also described because it provides a simple way to integrate GSS-API mechanisms into an application, assuming that the application fits within the constraints imposed by SASL.

Raw Kerberos

The base Kerberos specification (RFC 4120) defines the underlying Kerberos message exchanges. All Kerberos applications rely on the KDC message exchanges defined in RFC 4120. Furthermore, the application authentication messages specified in this RFC are used in some form by almost all Kerberos applications.

RFC 4120 also defines two messages for integrity and confidentiality of user data. The *KRB_SAFE* message provides an integrity-protected wrapper around user data. The *KRB_PRIV* message encrypts and integrity protects user data. However, outside of some special applications these messages are rarely used—best practice is to use alternatives based on GSS-API rather than the KRB_PRIV or KRB_SAFE messages.

The KRB_SAFE message is particularly problematic. A bug in early versions of MIT Kerberos produced an incorrectly formatted KRB_SAFE message. Some implementations expect the version of the message produced by MIT Kerberos; others expect the version in the standard. As a result, KRB_SAFE interoperability is limited. Since the message is so rarely used, it has not been widely tested.

In addition, there are no standard APIs for accessing these messages. As of Windows Vista, Microsoft does not provide a mechanism for user applications to produce KRB_PRIV or KRB_SAFE messages. Solaris prior to version 10 update 3 did not provide access to the raw Kerberos API, which is required to use these messages.

The raw Kerberos messages do provide some flexibility that is not present in other interfaces. Applications that need to directly access Kerberos keys, or add fields not found in other interfaces may need to use the raw messages.

GSS-API Mechanisms: the GSS-API and SSPI

The GSS-API (RFC 2743) is a mechanism-independent facility for allowing applications to request security services such as authentication, integrity and confidentiality. The GSS-API specifications focus on an API that applications can use.

However, there is also a series of network protocols associated with GSS-API. Each mechanism has an associated network protocol. The Kerberos GSS-API mechanism (RFC 4121) describes messages that realize the GSS-API security services with Kerberos infrastructure.

The GSS-API mechanism uses the RFC 4120 application authentication exchange with some additional framing to indicate it is a Kerberos message and to provide for some GSS-API specific options. The Kerberos GSS-API mechanism provides services similar

to KRB_SAFE and KRB_PRIV, but the message formats over the network are completely different.

The GSS-API is available on most platforms today. However, even if the GSS-API is not available, GSS-API mechanisms can still be used. The message formats from the Kerberos mechanism could be built into a small, embedded system without the complexity and code space of the complete GSS-API.

There are several advantages to using the GSS-API Kerberos mechanism:

- Good cross-platform availability, including Windows, Mac OS, Java, and most Unix variants
- Good interoperability between implementations
- Support for future extensibility within Kerberos and for other security technologies

GSS-API supports security mechanisms that take an arbitrary number of round trips for authentication of a client to a service. This allows support for negotiation of which security service to use as well as future extensibility as options are added to mechanisms. As a result, GSS-API applications exhibit a characteristic loop for the application authentication. The client generates the first message by calling GSS_Init_sec_context. The client sends this message to the server, which calls GSS_Accept_sec_context. Both GSS_Init_sec_context and GSS_Accept_sec_context provide a return value in the form of a major status. This major status instructs the application what to do next. If the function returns GSS_S_CONTINUE_NEEDED, then the other side of the connection is expected to return another message and the loop should continue. Otherwise, the loop ends; depending on whether the status is successful or not—i.e., the authentication succeeds or fails. Even if an authentication function does not return, "continue needed," it may still output a message to be sent to the other party.

Some application protocols depend on knowing the number of round-trips authentication and security setup will take. GSS-API should still be used in these applications, but the application should restrict the set of mechanisms that are supported. The Kerberos mechanism always takes one round trip with mutual authentication enabled. If future extension to the Kerberos mechanism increase the number of round trips, then either the mechanism *object identifier* (OID) will change (creating a new mechanism), or extra round trips will only be used when a new optional feature is enabled.

Once initial authentication succeeds, GSS-API applications can use GSS_Wrap to request integrity or confidentiality for user data. The GSS_GetMIC function requests a message integrity code that can be transported to provide integrity for data sent in application framing.

As mentioned, GSS-API mechanisms can be used without the GSS-API. The Windows SSPI is a widely deployed example of this. Windows SSPI provides a Microsoft-specific API for security services. However, the Kerberos SSPI mechanism is the same as the Kerberos GSS-API mechanism. Applications that will use both the SSPI and GSS-API

should avoid depending on GSS-API's gap or duplicate token errors unless a skipped or duplicate message will be considered a fatal error.

See http://docs.sun.com/app/docs/doc/816-4863 for Sun's documentation on the GSS-API. This documentation is applicable to other platforms besides Solaris.

SPNEGO and GSS-API Negotiation

Most GSS-API implementations provide the "Simple and Protected Negotiation" (SPNEGO) mechanism (RFC 4178) in order to allow applications to select what security mechanism they wish to use. In order to use the SPNEGO mechanism, an application passes the object identifier naming the SPNEGO mechanism into GSS_Init_sec_context instead of the object identifier for Kerberos or the default mechanism. The GSS-API implementation will include a list of available mechanisms in the token the application sends to the service. This first token also includes a message from the mechanism that the client would prefer to use. If that mechanism is also selected by the service, the negotiation does not introduce any round trips. If the service does not select the client's preferred mechanism, then the client and service exchange a message integrity code after authentication succeeds. This protects both parties against an attacker who tries to force them to select a different security mechanism that is easier to attack.

SPNEGO is available on the Windows platform via the SSPI. Callers should use the "negotiate" security package in order to use SPNEGO.

SASL

SASL (RFC 4422) provides facilities for negotiating security mechanisms, authentication, and optional integrity and confidentiality. SASL works for stream-oriented applications that send only one stream of bytes in each direction. For example, SASL works well for applications that send a stream of bytes over one TCP connection. It does not work well for applications that need to deal with out-of-order delivery or lost data. Many internet applications use SASL.

SASL and GSS-API are complimentary technologies that work together: SASL's Kerberos support uses the GSS-API. There are actually multiple SASL mechanism families for supporting GSS-API. The oldest and most commonly used (RFC 4752) is widely implemented. A newer family (GS2) supports channel binding (see Kerberos and Other Authentication Mechanisms) but is not widely implemented. Applications that need channel binding in order to provide adequate mutual authentication should plan for an upgrade to GS2 when it becomes more readily available.

Comparing SASL and GSS-API Mechanisms

Because SASL uses GSS-API to provide higher-level Kerberos support, SASL is preferable to GSS-API mechanisms as a network protocol, but only when it meets the needs of an application. The primary advantage of SASL is that it provides more structure than GSS-API and leaves fewer details to the application. For example, SASL presents a stream abstraction rather than a message abstraction and the SASL specifications provide more detail on what an application needs to do to properly use SASL. However, SASL APIs are not as well standardized as the GSS-API, and some

application developers have encountered problems with generic SASL framework implementations. A good rule of thumb is, if using the GSS-API makes more sense than using SASL for implementation reasons, then use the GSS-API. The following table provides a summary list of factors to consider when deciding which API mechanism provides the most appropriate protocol facility for a given application.

Factor	SASL	GSS-API Mechanisms
Network Profile	Applications with a single stream of data—e.g., one TCP connection	Provides per-message services allowing applications to have multiple streams or to use datagram protocols
Abstract interface	Application reads and writes data similar to TCP or TLS	Application requests message protection and handles the details of sending data over the network
Non-Kerberos mechanism support	Simple passwords, challenge/response, one-time-password	Limited availability of non-Kerberos mechanisms in widely-deployed platforms
API Standardization	Cross-platform APIs are available and used for some applications, though standardized APIs are still evolving	Widely-available standardized API

Table 1: Factors to consider in choosing between SASL and GSS-API

SASL and TLS

SASL is often used along side TLS (or SSLv3). Many SASL mechanisms, such as simple passwords (the "plain" mechanism), do not provide adequate security alone. Other SASL mechanisms provide authentication but do not provide confidentiality or integrity services.

In this usage pattern, the application makes both SASL and TLS available at the beginning of the connection. Only SASL mechanisms that provide appropriate security to meet access control policies are made available outside of TLS. If TLS is selected, then SASL is also available as an option after TLS completes session establishment. Typically, a larger set of mechanisms is available at this point.

GSS-API mechanisms such as Kerberos that provide confidentiality and integrity should be offered *before* TLS session initiation. Applications should support using both a SASL security layer (integrity or confidentiality provided by SASL) at the same time as TLS.

Application developers have found it challenging to support both TLS and SASL security layers in the same application. How data is written to the network depends on what security services are used. Sometimes, the data goes through TLS, sometimes through SASL and sometimes both. Several applications do not support SASL security layers as a result. When used with Kerberos and other strong mechanisms, these applications do not provide mutual authentication. For this reason, best practice today is to support SASL security layers even though there is extra complexity. As channel binding becomes more widely available, best practice will likely evolve to use channel binding and not to use SASL security layers in applications that also support TLS.

Protocols that Already Support Kerberos

The goal of networking is to be able to communicate. This communication is facilitated by similar applications making the same decision about matters such as how to integrate security. The following table describes several common internet protocols and how they can be integrated with Kerberos. It also gives pointers to the specification that describes how security is integrated into the protocol; in many cases, the specification is not Kerberos specific, but instead points to a description of how the GSS-API or SASL is used to secure the protocol.

Protocol	Integration Strategy	Specification
SMTP	SASL+TLS	RFC 4754
POP3	SASL+TLS	RFC 5034 RFC 2595
SSH (Secure Shell)	GSS-API channel binding to SSH transport	RFC 4462
NFS	GSS-API	RFC 2203
DNS	GSS-API for transactional signatures	RFC 3645
LDAP	SASL+TLS	RFC 4513

Table 2: Approaches to integrating common Internet protocols with Kerberos

Many protocols are described as having the "SASL+TLS" integration strategy. This means that the protocol supports both SASL and TLS. As discussed above, the best strategy for using such an application with Kerberos is to use the GSS-API SASL mechanism and to use that mechanism's security layers. Doing so achieves the mutual authentication benefits of Kerberos. TLS, while available, is not needed in this situation. TLS is useful with other SASL mechanisms besides GSS-API. Also, some applications do not support SASL security layers; see the example section on SASL in Thunderbird for an example. Using TLS for integrity and GSS-API for authentication with such applications is better than no integrity protection at all. Once channel binding is available, this option may be even more attractive.

Survey of Kerberos Implementations

There are a number of Kerberos implementations available to developers. This section briefly surveys the implementations and describes areas that developers should consider when choosing an implementation or moving between implementations.

MIT Kerberos

The MIT Kerberos Consortium makes available a reference implementation of Kerberos. This implementation exposes a raw Kerberos API and also includes the GSS-API.

MIT Kerberos forms the basis of the Kerberos implementation in a number of platforms including Apple's Mac OS X, Sun Solaris and Redhat Linux. MIT Kerberos has also been ported to platforms such as Vxworks for embedded applications. MIT makes available a version of its product called Kerberos for Windows that runs on various Windows Platforms.

Solaris versions prior to Solaris 10 update 3 only supported the GSS-API. However, since update 3, the raw Kerberos API is also exposed.

For more information on MIT Kerberos or the Kerberos Consortium, see http://www.kerberos.org/

Heimdal

Heimdal is an implementation of Kerberos maintained by Stockholm University. It contains support for a raw Kerberos API and for the GSS-API. Heimdal and MIT Kerberos have a different set of features, although both implementations regularly add new features, so the set of features one has that the other does not is constantly changing. Heimdal has very good over-the-wire compatibility with MIT Kerberos. Heimdal is used in several platforms, including most of the BSD Unix variants, aside from Apple's Mac OS X.

Both MIT Kerberos and Heimdal have similar raw Kerberos APIs. However there are important differences in basic structures including how keys and principals are represented. As a result, it generally requires significant work to write a program that works with both implementations. This work can generally be constrained to a number of portability functions. Applications built on the GSS-API without using raw Kerberos should work against either implementation with only build system changes required.

For more information on Heimdal, see http://www.h5l.org/

Microsoft Windows

With the exception of Windows CE, all current versions of Microsoft Windows include Kerberos support. Windows supports the Kerberos GSS-API mechanism, but does *not* use the GSS-API as an application interface. Instead, Windows applications can access Kerberos via the Security Support Provider Interface (SSPI). The SSPI provides similar functionality to the GSS-API, although the specific API calls that an application needs to make are different. In Microsoft's terminology, Kerberos is one of several Security

Service Providers (SSPs) that can be used via the SSPI interface. There are a few minor differences between Microsoft's mechanism and other GSS-API Kerberos mechanisms:

- Windows does not support the duplicate token or gap token functionality of the GSS-API. If sequencing or replay detection is enabled on Windows, then no plaintext is made available if out-of-sequence or replay messages are unwrapped.
- Windows supports a DCE style mode that changes token formats slightly.
- Windows supports the ability to pass in multiple buffers when generating a token; other GSS-API implementations do not.

Microsoft's Kerberos provides single-sign-on within a Windows domain. The domain credential is made available and can be used to connect to network services. Windows also allows applications to obtain Kerberos credentials based on a password that is passed into SSPI. However, Windows does not provide a user interface to obtain default credentials for an identity different than the user's domain identity which means there is no way for a user to establish alternative default credentials that can be used when contacting network services. Windows does provide a password vault-like facility (Credential Manager) allowing users to store a password to be used when contacting specific services; if the service supports Kerberos then Windows will try to use this password to obtain Kerberos tickets.

For more information on SSPI, see http://msdn.microsoft.com/en-us/library/aa380493.aspx

SIDs and the PAC

Active Directory uses a field in the Kerberos ticket called "authorization data" to store the Microsoft PAC (Privilege Attribute Certificate). Among other things, the PAC includes a set of Security Identifiers (SIDs) for the client. These identifiers uniquely identify the account and any groups to which the account belongs within the directory. Services can trust the PAC because it is integrity protected by the KDC. Applications can make authorization decisions based on the group membership of the user. Without a PAC, services would need to query the directory to find out what groups an account belongs to.

SIDs solve the problem of account renaming. With other versions of Kerberos, the principal name is used as an identifier. If the principal name changes, then applications will no longer be able to track authorization information for that account. However, SIDs do not change when the user account name changes.

Comparing KFW and Windows Kerberos

As discussed above, Kerberos for Windows (a version of MIT Kerberos for the Windows platform abbreviated as KFW) and Windows Kerberos (the native Kerberos support that is part of the Windows operating system) are both available for the Windows platform. If the native Windows Kerberos is sufficient for a given application, using it is simplest. To use KFW, KFW needs to be installed and configured. However, KFW has important advantages in some environments:

- KFW allows machines that are not part of a domain to easily use Kerberos; this works well for individually owned machines that still need to access Kerberos protected network resources.
- KFW supports the raw Kerberos API and the GSS-API. This may allow developers to build applications that can be easily ported to both Windows and other platforms.
- KFW supports AES and other strong encryption on all versions of Windows. Windows Vista is the first version of Windows to support AES. In the future, KFW may allow new Kerberos features to be used on existing versions of Windows.
- Starting with Windows Vista and KFW 3.2, KFW can set the default identity that a Windows client will use when contacting network services. This can significantly improve the Kerberos experience when accessing services outside of a Windows domain context.

Integrating KFW into an application requires care to make sure that KFW can be installed either as part of the application or as a separate install. In addition, the installers need to be designed so that multiple applications on the same system can integrate KFW. Parties interested in integrating KFW should contact the Kerberos Consortium to coordinate their integration efforts.

Java, JGSS, and JAAS

The J2SE platform includes support for the GSS-API as well as the Kerberos GSS-API mechanism. Two components make this possible.

The Java Authentication and Authorization Service (JAAS) provides mechanisms to obtain credentials and perform initial authentication. JGSS is a binding of the GSS-API to the Java language. A client application running on a desktop could first use JAAS to obtain credentials and then use JGSS to contact a Kerberos service.

Often credentials are already available on the native platform. For example, a Windows desktop that is part of a domain will obtain credentials at login. Similarly, a Mac attached to a corporate directory will obtain credentials at login. On these platforms, JAAS can import the existing native credentials into Java applications with no user interaction. JAAS also supports the file based credential caches that MIT Kerberos and Heimdal use on UNIX platforms. If native credentials are unavailable or not used, the JAAS krb5 login module can obtain credentials and perform initial authentication directly.

The JGSS interfaces provide object-oriented access to the same conceptual API as the more traditional C interface. The same services, including context establishment, message wrapping and MAC generation are available. The Java implementation of the Kerberos GSS-API mechanism is compatible with the Kerberos mechanism available for other platforms.

Starting with JDK 6, Java can utilize a native GSS-API implementation when it is available. That means that Java will translate JGSS calls into calls to the platform's native GSS-API implementation. The native support allows Java applications to take

advantage of enhancements and security updates to the platform's native Kerberos implementation. For example, if a Solaris update adds support for better encryption technology, then Java applications running on Solaris can take advantage of this new technology.

J2SE includes a complete implementation of the Kerberos GSS-API mechanism. This implementation is used prior to JDK 6 or when the native support is not available. The ability to use JGSS even when the native platform does not have a Kerberos implementation makes depending on Java Kerberos easy. However there have been some problems when the Java implementation was used on the same system as a native implementation. Native implementations tend to add new features including new encryption types faster than the Java platform. So, it is possible for the native implementation to obtain credentials that use one of these new encryption types and for JAAS to be unable to understand the credentials. Also, while the Java Kerberos implementation does try to understand common native configuration mechanisms, there have been problems where Java had difficulty finding the KDCs for a realm or determining what realm to contact for a given host-based service. (See Service Names for detail on service realms.) Support for using native implementations when available should alleviate these problems.

See http://java.sun.com/j2se/1.5.0/docs/guide/security/jgss/tutorials/index.html for more information on JAAS and JGSS.

Planning to Integrate Kerberos

Application authors face a number of decisions when they integrate Kerberos into their applications: how will Kerberos security interact with the rest of the application, how will the application know when to use Kerberos, and which approach should be used to integrate Kerberos? Furthermore, developers need to understand what assumptions Kerberos makes about their applications and make appropriate provisions if these assumptions are not correct. This section discusses these concerns, outlines issues to consider and provides recommendations for the common cases.

A successful integration of Kerberos into an application should meet the following goals.

- The application should provide authentication and appropriate session security services with the same level of protection as provided by the Kerberos environment in which the application is deployed.
- The Kerberos security infrastructure should be sufficient to provide these authentication and session security services. While other security infrastructures, such as a PKI, may be used when available, Kerberos should not require dependencies on such infrastructures.
- The application must not undermine the security of the Kerberos environment.
- Under normal operation, the Kerberos experience should not require user interaction. User interaction when the application is first configured may be desirable, but when a user who already has Kerberos credentials attempts to use the application, they need not interact with the application in order to authenticate

and service. The most obvious use of this session key is to protect traffic between just the two parties that are authenticated. However, another approach is to use this session key to distribute a group key that is known to a larger number of parties. For example a service that is distributing the same content to a number of clients might not want to separately integrity protect or encrypt the content for each client. Instead, the service could authenticate each client and then distribute a group key to each client using the session key shared between that client and the service. Then, the service only integrity protects or encrypts the content once for all clients by using the shared group key. The service still needs to engage in authentication exchanges with each client, but can avoid the cost of performing cryptographic operations on subsequent content with each client. As with all group-key-based security schemes utilizing symmetric encryption, any party that knows the group key can impersonate the other parties if no other authentication information is used.

Kerberos is easiest to deploy when the following conditions hold:

- A client is authenticating to, and exchanging data with, a single service.
- The client knows the name of the service before it authenticates.
- The authentication exchange meets the timing requirements for Kerberos.

The rest of this section discusses common cases where one of these conditions is not met, and provides advice on how best to use Kerberos in these situations.

Proxies

Some applications use proxies—i.e., network intermediaries that examine and possibly modify application messages between other parties.

If the proxy does not need to modify messages, then it may not pose a problem for Kerberos at all. The authentication can be from the client to the service, irrespective of whether a proxy is in the middle. The messages pass through the proxy, but integrity protection is end-to-end just as it would be if no proxy were involved. The application needs to be careful that parts of the messages that proxies need to examine are not encrypted.

With any communications between a client and service that traverses one or more proxies, hop-by-hop authentication is possible. The client authenticates to the first proxy as the service. This proxy authenticates to the next proxy; eventually the final proxy authenticates to the service. The advantage of this approach is that proxies can manipulate the messages. The disadvantage is that both the client and service have significantly weaker security guarantees; many access control objectives cannot be met with hop-by-hop authentication. For example, the service has no direct evidence of the client's authenticity; it must trust the final proxy to authenticate the client. Sometimes the service and final proxy are in the same trust domain and a reasonable access control policy would allow the service to trust the final proxy to authenticate the client. However, as the number of proxies increases, it becomes less likely that such trust can be chained all the way from the final proxy to the first proxy. Support for this type of proxy needs to be built into an application as it will be the proxy that authenticates to the service—not the end user.

One seemingly attractive approach is to delegate users' credentials to a proxy so that the proxy can authenticate as the user. Best practice is to avoid application designs that depend on delegation of a user's credentials for a single application to work. There are similar situations where delegation may be acceptable, such as when delegating credentials to a trusted service so that it can access file stores, mail stores and the like. However, delegation is only appropriate where policies allow trust to be placed in certain third parties that act as delegates on behalf of *primary* or *first* parties.

A more effective approach is to combine end-to-end authentication with hop-by-hop authentication. In this hybrid approach, applications can protect some information end-to-end, but other information that proxies need to modify is protected hop-by-hop. Information that proxies do not need to examine can even be encrypted end-to-end.

Discovering the Service Name

The requirement that a client know the specific name of the service to which it is authenticating can be problematic in some contexts. Services may be known by a number of names. A user interface may present descriptions of services, but a domain name may be the actual registered name of the service.

One seemingly attractive approach is to ask the service what its name is, and then authenticate to this name. Applications must *not* adopt this approach, as doing so defeats the mutual authentication provided by Kerberos. Such an application would be vulnerable to phishing or man-in-the-middle attacks where a malicious service replaces the service that the client or user intends to use.

One approach for getting the specific name of a service is to look it up in a trusted directory. For example, an organization could operate a directory service that maps human descriptions of services to Kerberos principals and host names for network connections. The data retrieved from the directory needs to be integrity protected; this directory must be a trusted resource in the access control system. Unfortunately, while DNS is a ubiquitous, simple directory service, it is not trustworthy because it is typically deployed without integrity protection or source authentication of data.

Another similar approach is to get a referral from one service to another. The name obtained from such a referral can only be trusted as much as the referring service can be trusted.

In some situations, the service can propose a name to authenticate to, and the user can confirm this name. This only works when users have enough knowledge about appropriate names to make intelligent access control decisions about whether the name represents the service they wish to connect to.

When none of these approaches is available, a weaker approach called "leap-of-faith" is a common solution. Despite the risky-sounding name, leap-of-faith still provides significant security in a number of situations. The first time a client connects to a service, the service provides its name. The client caches this name, and uses the same name for future connections. If the service tries to change its name, then the client warns the user and prevents access until the user agrees to accept the new name. Leap-of-faith requires that the client have some way of knowing when it is connecting to a service

again; for example a list of favorite services in a user interface. In addition, leap-of-faith only works well when service names rarely, if ever, change.

Kerberos and Other Authentication Mechanisms

As discussed in the section on Protocol Facilities for Integrating Kerberos, there are various mechanisms for integrating Kerberos into an application. If Kerberos is the only authentication mechanism that an application supports then these mechanisms can be used directly. A mechanism like GSS-API can be used to give the application fine-grained control over what parts of the messages are integrity-protected and what parts of the messages are encrypted, or a mechanism like SASL can be used to provide simple protection for TCP-based applications.

However many applications support Kerberos *in addition to* other authentication mechanisms. One approach is to support only GSS-API mechanisms; this is roughly the approach that SAP R3 adopts for advanced security. Applications supporting multiple GSS-API mechanisms need to decide how to choose which mechanism is used. One easy solution is to use the "Simple and Protected Negotiation Mechanism" (see the section on SPNEGO and GSS-API Negotiation). The GSS-API only approach has the significant disadvantage of not supporting common authentication mechanisms like simple passwords; so many application developers choose a different approach. These application developers need to decide how to integrate both Kerberos and other mechanisms so that important access control services including mutual authentication and integrity are provided.

One approach is to incorporate within an application separate facilities for supporting Kerberos alongside other security mechanisms, using each mechanism's native facilities for providing integrity, confidentiality and mutual authentication. This approach works and is commonly used. It allows the application's use of security services to best match the underlying mechanism.

Other application developers choose to adopt one mechanism for providing integrity and confidentiality and use that regardless of how authentication is handled. For example, the SSH key exchange and transport facilities can be used regardless of what user authentication facilities are used. Similarly, many applications use TLS and run SASL mechanisms within a TLS tunnel. Typically in this approach, the service is authenticated to the client by the layer providing integrity and confidentiality. Both the SSH key exchange mechanism and TLS provide this authentication. For authentication mechanisms, such as simple passwords, that cannot provide mutual authentication, this is the best security approach possible.

However, taking the approach that Kerberos is only responsible for authenticating the client to the service and not for providing mutual authentication undermines one of the key advantages of Kerberos. The mutual authentication provided by Kerberos tends to be stronger than that provided by other mechanisms because it does not need to present warnings about expired or incorrect certificates and because whenever there is a mutual authentication failure, there is strong evidence of a security attack. An approach that uses the message security layer to authenticate a service to a client but does not leverage Kerberos mutual authentication fails to achieve the goal of independence from other

security infrastructure: the access control depends both on the security of Kerberos and of the mechanism providing authentication of the service. So best practice when using a lower layer to provide authentication of the service is to use channel binding.

Channel Binding (defined in RFC 5056) provides a facility to tie an authentication exchange to security services provided at a lower layer. In addition to attesting to the names of the two parties, the authentication exchange attests to a cryptographic name of the lower layer session called channel binding data. This name has the property that both parties can verify that it uniquely names the lower layer session. If the lower layer provides certain security properties outlined in the channel binding specification, both parties have confidence that mutual authentication provided by the authentication mechanism also implies mutual authentication of the parties to the message security layer. As an example, the TLS exchange contains a message called a *finish message* that includes a cryptographic hash of the entire TLS authentication exchange. The TLS finish message provides the necessary security properties outlined in the channel binding specification to be a cryptographic name for the TLS session. If GSS-API message protection facilities are used to sign this finish message then the parties to the GSS-API authentication can trust the integrity of data exchanged over the TLS channel. These parties also know the source of the exchanged data. Even if self-signed certificates are used at the TLS layer, the authentication is secure provided that the GSS-API authentication is secure. When performing channel binding, it is important that both sides check the channel binding data. The following steps will accomplish this requirement.

- The client sends the channel binding data to the service using GSS_Wrap after authentication succeeds.
- The service does not send sensitive data over the lower layer until it receives and verifies the channel binding data. The service sends an indication that the channel binding data has been verified using GSS_Wrap once it has received and verified the channel binding data.
- The client does not consider the lower layer authenticated until it has received the indication that the channel binding succeeded.
- Channel binding is not an optional facility; the client and server always send channel binding messages.

Of course, SSPI or raw Kerberos calls can be substituted for GSS-API calls. Other secure constructions are possible, including constructions where channel binding is an optional facility. However, reviewing the security of such a construction is difficult. If the above steps are not sufficient for a particular application, an expert familiar with channel binding should be consulted to review the security of the application.

Channel binding is a relatively new facility and may not be available in all environments. If it is not available, then applications should either directly use Kerberos facilities for message protection or at least be designed to take advantage of channel binding when it becomes available. Such designs need to pay attention to upgrade issues, so that an attacker cannot cause a new version of the application to appear as an old version of the application without channel binding support. To do this, the application should provide a set of authentication-related capabilities sent from the

client to the service that are protected using Kerberos message protection facilities. Services must require that these capabilities be received.

Long-term keys for Services

All principals in the Kerberos system, including users and services, have long-term keys. Typically, the user's computer (client) only has access to the user's long-term key during initial authentication to the KDC. The credentials obtained are used to avoid needing access to the long-term key throughout the lifetime of the credentials. However, the story is different for services. In the most common mode of usage, the service uses its long-term key every time a client authenticates to it.

As a consequence, Kerberos application services must be designed to have long-term keys that are accessed every time a user authenticates to the application. Also, since enrolling a new long-term key typically requires administrator involvement, Kerberos works best when new services are not created frequently. If a particular application needs to create new Kerberos services on a frequent basis, it should provide a mechanism to enroll these services into Kerberos in a sufficiently secure, automated manner. However, such mechanisms depend on the implementation of the Kerberos KDC used in an organization.

There is an alternative mode of Kerberos usage called user-to-user. This mode is intended to allow one Kerberos user (e.g., Alice) to contact a service running as a part of another user's (e.g., Bob's) session; for example, one user (Bob) might want to share files with another user (Alice). Neither user is expected to have access to a long-term service key during the time when the client (Alice) application authenticates to the service (Bob). Instead, the service (as part of Bob's session) obtains a ticket-granting ticket through the initial authentication exchange. The service (Bob) presents its ticket-granting ticket to the client (Alice), which interacts with the KDC to establish a shared session key between the principal identified by the client's (Alice's) ticket and the service's (Bob's) ticket. User-to-user Kerberos allows a service to have access to a ticket-granting ticket instead of access to a long-term key. Unfortunately, implementation support for user-to-user Kerberos may not be adequate, depending on the environment and platforms used.

Microsoft Windows supports user-to-user Kerberos via a special GSS-API mechanism. This makes user-to-user Kerberos relatively attractive for two Windows applications within the same forest of domains. These applications should be written using the SSPI in order to access this mechanism. MIT Kerberos supports user-to-user mode at the raw Kerberos level, but does not support the same GSS-API mechanism that Microsoft supports. Microsoft does not expose the raw Kerberos messages necessary to work with the support found in MIT Kerberos.

In its Mac OS X 10.5 (Leopard), Apple adopted a different approach to using Kerberos in small work-groups. Each Mac runs its own KDC and Kerberos realm. Mechanisms are provided to make it easy to register services on the local machine. So when clients authenticate to applications, the typical Kerberos model is employed where applications have long-term keys. The advantage of this approach is that Kerberos can be used with very little infrastructure—even for peer-to-peer environments. The disadvantage is that,

since each machine is its own realm and no keys are shared to join these realms to a federation, users must perform initial authentication against each computer. This limits the single-sign-on benefits. Apple mitigates this disadvantage for mac.com (a.k.a., MobileMe) customers by leveraging a public-key infrastructure.

In summary, Kerberos is easiest to use when network clients initiate connections to stable network servers with long-term keys. While it is possible to deal with services that do not have long-term keys available, doing so adds significant interoperability challenges and implementation complexity.

Initial Authentication

As a general rule, initial authentication and obtaining Kerberos credentials are handled by the platform, not the application. Often, credentials are obtained during the login process to the local computer. If MIT Kerberos is installed on a Mac OS or Windows platform, it will also obtain credentials automatically if they are not already available.

Applications generally should not attempt to obtain their own credentials. Doing so provides an unexpected user experience and may frustrate users by providing Kerberos related prompting when none is desired. Instead, it is recommended that applications use platform facilities to obtain credentials when needed, and let the underlying platform handle the case where initial authentication may be required.

Some specific use cases may require applications to obtain credentials directly; this recommendation is only a general guideline.

DNS Dependence

Kerberos depends on DNS for several functions:

- DNS SRV records are used to find the KDCs that serve a given realm.
- When service-based names are used, some implementations attempt to resolve DNS aliases (CNAME records) or reverse resolve the hostname.
- MIT Kerberos and Heimdal support a mechanism to look up the realm that should be used for a particular hostname in DNS using TXT records. This mechanism poses security problems and is disabled by default, but may be enabled within some organizations.

Given these dependencies, performing Kerberos calls can generate DNS traffic. In current implementations, these calls will block until DNS responds or times out. It is frustrating when a user intends to use Kerberos and an application takes a long time to indicate that Kerberos will fail. However, DNS-related delays have caused much worse user acceptance problems when Kerberos-related DNS delays cause an application to hang trying to contact an endpoint that ultimately does not even support Kerberos. Application designers need to carefully consider how to avoid blocking an application for long periods of time contacting an endpoint that does not support Kerberos or when DNS is failing. Mitigating strategies include caching known failures or caching whether Kerberos works with a particular endpoint. On platforms where attempting to use Kerberos will not automatically obtain credentials, then Kerberos authentication should not be used if credentials are not present.

Canonicalization of Host Names

Several situations where users connect to a service can legitimately result in the host name of the service being different from the name originally entered by the user. For instance, the user may have entered an alias corresponding to a CNAME DNS record, or the service may be part of a DNS load-balancing cluster where connections are spread across a number of systems. In these situations Kerberos deployments need to allow users to authenticate to any of the names for a given service. Microsoft Windows handles this by storing a set of aliases for a principal in its directory. A client can authenticate to any of these aliases.

Most other Kerberos implementations have only limited support for storing aliases in the KDC database. These implementations often compensate by attempting to canonicalize the name entered by the user. This canonicalization starts by resolving the name and if a CNAME record is returned, then the name pointed to by the CNAME record is used rather than the name obtained from the user. Implementations often take the additional step of reverse resolving the address obtained from DNS, and if the reverse resolution is successful, then the name in the resulting PTR record is used. This canonicalization introduces significant security concerns, because DNS has no integrity protection in typical deployments. However, without adequate aliasing support in KDC databases, it provides better usability.

Given these security concerns, recent implementations have taken steps to reduce or eliminate this canonicalization behavior. As a result, applications are unlikely to know what canonicalization behavior is used as it may change from one version to the next of a Kerberos implementation, and it may depend on local configuration. With the exception of load-balanced services , applications should ignore the details of canonicalization behavior. The application will encounter the same canonicalization behavior as other applications in the local environment; site administrators can balance the tradeoffs between security and usability in how they configure the Kerberos infrastructure. In addition, the application will not need to be updated as better mechanisms for handling aliases become available and as implementations move away from canonicalization.

Ignoring canonicalization runs into significant problems for applications that contact services behind DNS load balancers. Consider a situation where service.example.net will return a.example.net and b.example.net using the typical DNS round-robbin behavior. The first request will get “a,” the second request “b,” and so on. The application passes service.example.net into Kerberos, which canonicalizes it to a.example.net. Then, the application makes a connection to service.example.net. The application calls getaddrinfo in order to find the IP address of service.example.net. Internally, getaddrinfo canonicalizes service.example.net and connects to b.example.net. Then the application tries to authenticate, but fails because the Kerberos implementation thinks it is contacting a.example.net while the network connection is to b.example.net.

Regrettably, there is no best practice for this situation because there is no way for an application to handle the situation without knowing what the specific Kerberos implementation’s canonicalization behavior is. The application could canonicalize the

name before passing it into Kerberos; this is usable but introduces a security weakness especially in environments with KDC aliasing. Alternatively, the application could bypass Kerberos's handling of host-specific services and generate the principal name itself.

KDC Discovery

Designing an application to use Kerberos in an environment where DNS cannot be used to find KDCs or lookup host names is challenging. Kerberos is designed to be part of an overall distributed infrastructure that supports multiple applications. So, applications that expect Kerberos to work without DNS need to be an integral part of the larger infrastructure and replace the uses of DNS with some other technology. MIT Kerberos provides a mechanism where system-level plugins can be installed to specify the location of KDCs. If name types like GSS_KRB5_NT_PRINCIPAL_NAME are used, there is no dependence on DNS for host names of services.

Windows has provided a registry-based mechanism for locating KDCs within a particular realm. However the Windows Kerberos implementation is heavily integrated into the Active Directory platform; hence, changing assumptions, such as the strategy used to find KDCs, is likely to introduce significant complexity.

Federation and Client Names

The "Role of Kerberos" paper describes how the Kerberos name space is broken into separate administrative realms. All principals, whether client or service principals, belong to an administrative realm.

Two realms can share keys allowing principals in one realm to authenticate to principals in another realm. This relationship can be asymmetric; just because principles in BOSTON.EXAMPLE.COM can authenticate to principals in HQ.EXAMPLE.COM does not mean that principals in HQ.EXAMPLE.COM can authenticate to principals in BOSTON.EXAMPLE.COM. There is no general mechanism to know whether a principal in one realm can authenticate with another without trying the authentication. The authentication may fail because there is no path of shared keys to get from one realm to another. Or the authentication may fail because the service application does not consider the path of shared keys used to be sufficiently trustworthy. Services are responsible for enforcing authentication policies such as evaluating the trust in cross-realm paths. However, KDCs make a recommendation if KDC policy indicates the path should be trusted. Typically, services accept a path if the KDC recommends doing so, and reject the path in other cases; Kerberos implementations adopt this approach by default. Therefore, application authors need to be aware that service policy may cause an authentication to be rejected, but at least in the case of cross-realm paths, applications should adopt the default policy.

As a result, just because a client and service both support Kerberos, they may not be able to use Kerberos to secure their session. Applications that support both Kerberos and other authentication mechanisms need to allow other mechanisms to be used when Kerberos authentication fails. Some applications will attempt to use a second authentication mechanism in the same session. Others cache the fact that Kerberos has

failed, and will start a new session defaulting to a different mechanism. Still other applications store the set of access control mechanisms to use for a particular connection along with the preferences for that connection.

Kerberos realms form a naming hierarchy. Principals in one realm cannot be assumed to name the same entity as principals in another realm. For example, an application must not assume that alice@EXAMPLE.COM names the same user as alice@SALES.EXAMPLE.COM unless it has information external to Kerberos allowing it to make this assumption (e.g., via some authoritative directory service). Applications need to adopt a way of expressing access control and authorization information so that access can be granted to users in multiple realms, and so that granting access to a user from one realm does not imply granting access to similarly named users from other realms. One simple way to accomplish this is to use Kerberos names directly as access control identifiers. A more complicated strategy is needed for applications that use a flat namespace of names. For example, Unix systems use a flat namespace of password file entries to describe users. A feasible approach is to provide a facility to map Kerberos identifiers into the application's namespace. For example, UNIX often uses .k5login files in user home directories to list a set of Kerberos principal names that can be used to represent that user.

Service Names

Another consideration is the names of services. The "Role of Kerberos" paper has a detailed discussion of Kerberos service naming. To summarize, service names often look like *service/hostname@realm* where *service* is some string such as `ldap' describing what type of application is being named, and *hostname* is the domain name of the server on which the service runs. With this format, there can only be one separate instance of a service on a given host. If more than one instance of a service is run, they will be in the same access control separation domain as far as Kerberos is concerned. It is relatively easy to adopt a different naming scheme for services if the default is inappropriate for an application. However there are tools designed to make dealing with DNS names easier that cannot be used if a different naming scheme is adopted. As an example, with GSS-API, the GSS_NT_SRV_HST name type supports names in the default service form. The GSS_KRB5_NT_PRINCIPAL_NAME name type supports any Kerberos principal name.

Clients must determine what realm a service principal is expected to be located in. At least for host-based service names, such as names of type GSS_NT_SRV_HST, Kerberos implementations will determine the realm of the host for the application. Applications that use an alternative strategy for naming services will need to adopt some mechanism for finding the realm of the service. Merely assuming that a service is located in the same realm as the client is inadequate, because it prevents federated access to the service.

Time Synchronization

Kerberos requires loose time synchronization (within five minutes typically) between the KDC and the service. Older versions of Kerberos also required time synchronization

between the client and service. However, enhancements to Kerberos allow current implementations to synchronize against the KDC's declaration of time.

The most important reason for services to have time synchronization is so they can determine whether a ticket is still valid. Beyond that, services need to synchronize time in order to limit the information that needs to be stored to prevent replays.

Some clients will ask the service for its idea of time and then issue Kerberos authenticators as if the service's idea of time is correct. Applications should *not* do this, as attackers could use this behavior to replay authentication messages that cause applications to assume a different time, thereby allowing new sessions to be established using an old (i.e., replayed) authentication message.

Delegation

Kerberos has a delegation facility allowing credentials to be forwarded by a client to the service. The service can then act on behalf of the user as a delegate. Applications supporting remote login or remote desktop applications should support delegation, and should have mechanisms to enforce access control policies when delegation is employed. The Kerberos specification has a flag that can be included by the KDC in a successful response indicating that the service is OK as a target of delegation. One common access control policy is to allow delegates to act on behalf of principals only when the KDC indicates that delegation is permitted. This policy is a good fit when the KDC and client are owned by the same organization or otherwise use the same security policies. However there are many situations where clients have either more or less strict access control policies than KDCs. In such situations, clients should have a way of forcing or preventing delegation, even when the KDC provides information on delegation targets.

Best practices for when delegation is appropriate for other classes of applications are still evolving.

Ticket Expiration

Some Kerberos implementations, particularly MIT Kerberos when using GSS-API, will not permit a security context to be used after the expiration of the ticket used to establish it. This means that if Kerberos facilities are being used to provide integrity and confidentiality, these facilities will be unavailable after the ticket expires. Consequently, applications depending on long-lived sessions should establish new security contexts before existing security contexts expire.

Examples

This section provides pointers to practical implementations of the concepts and best practices presented in this document. Examples include code designed to clearly illustrate concepts as well as authentication support for real products, and to demonstrate how Kerberos is integrated into shipping code.

MIT GSS Example

The Kerberos Consortium provides a simple GSS-API example that illustrates how the GSS-API can be used in an application. The client makes a connection, authenticates to the server and receives an encrypted token that provides an error message or success indication. The server sets up a listening socket, authenticates the client, determines whether the client is authorized and writes an encrypted token back to the client giving a success indication or authorization error.

The server side of the GSS authentication loop is found in the Authenticate function in server.c. The client authentication loop is found in the Authenticate function is client.c.

This example code is Kerberos-specific in a couple of ways. First, the server passes in GSS_C_NO_CREDENTIAL into GSS_Accept_sec_context. The advantage of doing this is that the server does not need to know what name the client used to address it. This works well for servers that are multi-homed machines, are part of a DNS load balancing pool, or are registered in multiple realms. However, it introduces a Kerberos dependency because the service needs to confirm that the name does apply to it. For example, if both an LDAP directory and an SMTP server ran on the same machine and used this approach, the LDAP directory would need to make sure that the client was not trying to authenticate to the SMTP server on the same machine, and vice versa. A second Kerberos dependency is introduced in the server code due to client authorization—i.e., the service needs to understand Kerberos names in order to authorize the client.

An application that wanted to be independent of any particular GSS-API mechanism would need to handle the problem of server credential selection and client authorization differently. To handle server credential selection in a mechanism-independent manner, an application could either have only one name by which the service is known, or could acquire credentials for a number of names and try GSS_Accept_sec_context on each of these names in turn. The GSS_Export_name function provides a mechanism-independent solution for client authorization. This function provides a binary representation of a name suitable for storing on ACLs that can be compared using binary comparison. While these approaches do work well for mechanism-independent applications, the approaches taken in the example tend to be more usable for Kerberos-specific applications.

MIT's GSS Example can be downloaded from:
http://www.kerberos.org/software/samples/GSSExample.tgz

The Gsstest Examples for Cross-Platform GSS-API

Since the initial release of MIT Kerberos, MIT has made available the gss-sample in src/appl/gss-sample in the MIT Kerberos source tree. Originally, this code was intended to be a sample GSS-API application demonstrating how GSS-API should be used. However, the sample has grown and evolved. Today, the primary purpose of the code is to test features of GSS-API as part of automated regression tests and interoperability events. As such, the code is somewhat more complicated than desirable

for a first example of a GSS-API application. In addition, the code is fairly old; it does not use ANSI C prototypes.

Because the code has been used in test scenarios, it has been ported to several different platforms, and is thus a useful example to study when writing a cross-platform GSS-API application. Despite the evolution of the code, MIT retains the name gss-sample for its distribution. This name is very close to GSSExample, the simple example application also distributed by MIT that is discussed in the previous section.

With the release of MIT Kerberos 1.3 in July of 2003, MIT changed the messages that this sample sends. As a result, some very old ports of this application do not work with the current code.

Windows Port

There is a port of the gss-sample client to KFW located in src/windows/gss in the MIT Kerberos source tree. This port is functionally identical to the original version, but uses the KFW DLLs and provides a user interface for manipulating options.

SSPI Port

The Kerberos Consortium distributes a port of gss-sample to the SSPI. This port uses the same network protocol as the GSS-API version, but uses Microsoft's SSPI to access the native Windows Kerberos GSS-API mechanism. Comparing this port to the original demonstrates how a similar code base can be used with both the SSPI and GSS-API.

The port can be found at
http://www.kerberos.org/software/samples/ms_samples_security_sspi_gss.zip

Kerberos 5 GSS-API Shim

Martin Rex distributes a GSS-API to SSPI shim layer for the Kerberos mechanism. This shim is a Windows DLL that translates GSS-API calls into SSPI calls. With this shim, an application can be written to use the GSS-API while taking advantage of the native Kerberos support found in Windows.

The shim has not been updated since the year 2000. It definitely still serves as a helpful example for those evaluating the differences between the SSPI and GSS-API. Application developers who are considering using the code directly in a product need to evaluate whether changes since 2000 affect its stability or suitability.

The code can be found on the sap.com FTP server at
http://www.kerberos.org/software/samples/gsskrb5/

SASL for Jabber in Pidgin

The Pidgin instant messaging client (http://www.pidgin.im/) includes support for Kerberos in its Jabber protocol plugin. Pidgin uses the Cyrus SASL library (http://asg.web.cmu.edu/sasl/sasl_library.html) to support the GSS-API SASL mechanism and thus Kerberos. The SASL integration can be found in libpurple/protocol/jabber/auth.c in the Pidgin sources.

Pidgin is an example of an application that supports both TLS and SASL security layers. The Pidgin SASL integration tries to fall back to other authentication mechanisms if preferred mechanisms like GSS-API fail. By default, Cyrus SASL will fail to authenticate if its preferred mechanism fails. As discussed in the section on Federation and Client Names, even if both a client and server support Kerberos, authentication between them may not work. In these situations, Cyrus SASL's default behavior can break authentication. Pidgin will remove mechanisms that fail and try again with the remaining mechanisms to obtain a more permissive behavior than the Cyrus default behavior. One consequence of the Pidgin behavior is that it is easier to force a less-secure mechanism to be chosen by causing the more secure mechanisms to appear to fail. In this instance, Pidgin has chosen to improve usability at the expense of stronger security. Applications that implement this sort of fallback strategy need to be careful not to enable mechanisms that are sufficiently weak that they do not meet access control policies.

SASL in Thunderbird

The Thunderbird Mail Reader (http://www.mozilla.org/) has support for GSS-API SASL authentication for mail retrieval and submission. This support can be found in extensions/auth in the Mozilla source tree.

The Thunderbird integration takes a much different approach than the Pidgin integration. Thunderbird focuses on providing a consistent experience on all platforms. As such there is a desire to minimize external dependencies and thus Cyrus SASL was not an appropriate solution. Instead, Thunderbird creates its own SASL wrapper around GSS-API calls.

Thunderbird developers also wanted to avoid a link-time dependency against a specific Kerberos implementation, since there may be a different implementation available on the target system than on the build system. So, Thunderbird dynamically loads the GSS-API library and accesses symbols from it. On Windows, Thunderbird supports both KFW's GSS-API library and the SSPI.

Since some operating systems shipped with broken GSS-API libraries that would crash an application if they were used without proper configuration, Thunderbird introduces additional complexity to avoid using these libraries.

Unfortunately, the Thunderbird architecture made supporting SASL security layers sufficiently difficult that they are not supported. As such, the Thunderbird Kerberos integration does not meet the goal of providing mutual authentication.

The Thunderbird Kerberos integration provides a comprehensive illustration of how truly cross-platform applications can be built and deployed in practice.