EXTENSIBLE PRE-AUTHENTICATION IN KERBEROS

by

Phillip Hellewell

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Phillip Hellewell

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Kent E. Seamons, Chair
Date	Quinn O. Snell
Date	Mike Jones

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Phillip Hellewell in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Kent E. Seamons
Chair, Graduate Committee

Accepted for the Department

Parris Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

EXTENSIBLE PRE-AUTHENTICATION IN KERBEROS

Phillip Hellewell

Department of Computer Science

Master of Science

Organizations need to provide services to a wide range of people, including strangers outside their local security domain. As the number of users grows larger, it becomes increasingly tedious to maintain and provision user accounts. It remains an open problem to create a system for provisioning outsiders that is secure, flexible, efficient, scalable, and easy to manage.

Kerberos is a secure, industry-standard protocol. Currently, Kerberos operates as a closed system; all users must be specified upfront and managed on an individual basis. This paper presents EPAK (Extensible Pre-Authentication in Kerberos), a framework that enables Kerberos to operate as an open system. Implemented as a Kerberos extension, EPAK enables many authentication schemes to be loosely coupled with Kerberos, without further modification to Kerberos. EPAK provides the mutual benefits of enhancing the flexibility of Kerberos and increasing the viability of alternate authentication systems as they move to the enterprise.

ACKNOWLEDGMENTS

I would like to thank my graduate advisor, Dr. Kent E. Seamons, for his analysis and guidance, and my graduate committee, Dr. Quinn Snell and Dr. Mike Jones, for their feedback. I would also like to thank Tim van der Horst, Andrew Harding, Reed Abbott, and other reviewers for their helpful comments.

This research was supported by funding from the National Science Foundation under grant no. CCR-0325951, prime cooperative agreement no. IIS-0331707, and The Regents of the University of California.

Table of Contents

List of Tables

List of Figures

Chapter 1 — Introduction

Open systems allow the authentication of users who are outside the local security domain and do not have a pre-existing relationship with the authentication server. To provide sufficient scalability, the system can employ attribute-based access control for mapping groups of users to role(s). The RT framework [17] is an example of such a system.

Kerberos has met the security demands of many businesses, but managing Kerberos grows more difficult as outside users become involved. Open system authentication systems address this by relying on third parties to manage users, passwords, keys, and other credentials. An authentication server trusts third parties to validate users.

Adopting new authentication schemes to replace Kerberos may be prohibitive because access control systems and applications are often built up around the Kerberos infrastructure. For example, Microsoft's Active Directory is a well-established, enterprise-level authorization system built around Kerberos. Extending Kerberos provides an attractive solution that allows systems like Active Directory to remain intact.

Incorporating open system authentication into Kerberos enhances the flexibility of Kerberos while increasing the usefulness and adoption of open systems. Kerberos becomes more powerful as it leverages open systems to provide services to more people, and open systems become more practical as they merge into existing Kerberos infrastructures.

After a motivating scenario described below, Chapter 2 gives a background of Kerberos. Then Chapters 3 to 5 describe the design and implementation of EPAK

and two EPAK-based protocols that enable Kerberos to operate as an open system. Chapter 6 contains a threat analysis of EPAK and Chapters 7 and 8 give related work, conclusions, and future work.

Motivating Scenario Suppose Company A desires to create a collaborative file-sharing service accessible to the employees of Company B. It would also like to leverage its existing security infrastructure (e.g., Active Directory) to manage users. Rather than manage accounts for each employee of Company B, Company A would like to group them all into a local user *employeeB*.

At the same time, Company A wishes to grant Company C read-only access to the file sharing site to monitor the work in progress but not make any changes. Employees from Company C could be mapped to the local user *employeeC*.

What if employees from Company B and C could be authenticated to Company A's domain merely by proving ownership of their email address? Company A could grant and remove access to outsiders simply by adding and removing entries from an access control list (ACL) that maps email addresses to local users. To provide the scalability needed for an open system, the ACL could allow wildcards for grouping addresses together (e.g., `*@companyB.com`).

Chapter 2 — Kerberos

Kerberos [22] is a distributed, identity-based authentication system that provides a method for a user to gain access to an application server. Kerberos allows a user to authenticate once and then connect to servers within the realm of the Kerberos network, without authenticating again for a period of time.

Kerberos is time-tested and widely used. Version 5 was standardized over a decade ago [22], and is in use by many enterprises today. It is used in business, government, military, and educational institutions, including those that use Microsoft Windows Server as a domain controller [19].

The Kerberos server consists of an Authentication Server (AS) and a Ticket-Granting Server (TGS). The AS and TGS are responsible for creating and issuing tickets to the clients upon request. The AS and TGS usually run on the same computer, and are collectively known as the Key Distribution Center (KDC).

The Kerberos authentication process works in three phases (see Figure 2.1). In the first phase, the client sends an AS-REQ with the user name to the AS, which responds with an AS-REP that includes a ticket-granting ticket (TGT) and a session key. The session key can only be unlocked by the user's password, and is required for the second phase. In the second phase, the client sends a TGS-REQ with the TGT from phase 1 to the TGS, which responds with a service-granting ticket (SGT) in the TGS-REP. In the final phase, the SGT is presented to the application server, which then grants the service.

Users and servers have names called principals [27]. Server principals are composed of a primary name, instance, and a realm, written as name/instance@REALM. Client principals, e.g., name@REALM, do not have an instance.

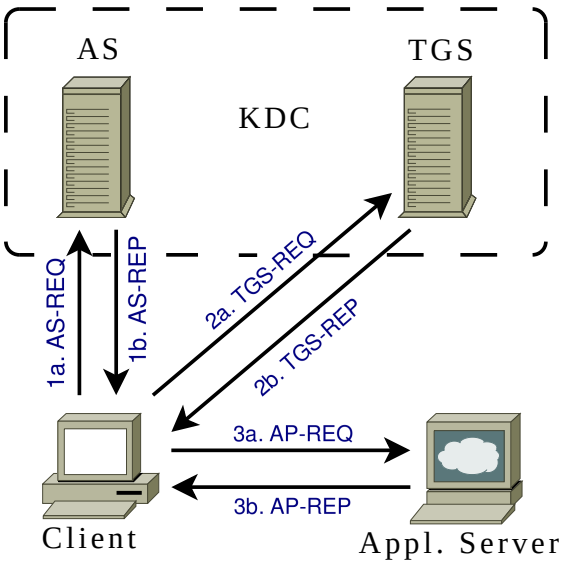

Figure 2.1: The Kerberos protocol. To access a service, the client first requests a ticket-granting ticket (TGT) from the Authentication Server (AS) in phase 1. This phase is also known as AS authentication. The client then uses that ticket to obtain a service-granting ticket from the Ticket-Granting Server (TGS) in phase 2. Finally, the client presents the service-granting ticket to the application server to access the service (phase 3).

A Kerberos server (KDC) must maintain several secret keys. A single key, K_{tgs}, is used to encrypt the TGT returned in step 1b (see Figure 2.1). Several keys, K_{c_x}, one for each client, are used to encrypt the session key, also returned in step 1b. Finally, several keys, K_{v_x}, one for each server, are used to encrypt the SGT returned in step 2b.

When the AS and TGS are combined, K_{tgs} can be stored in a private database used only by the KDC. The client keys, K_{c_x} can also be stored in the private database, because only the AS needs direct access to them (clients derive the key from their password).

However, the server keys must be shared between the Kerberos server and application servers. For example, an ftp daemon service will need access to the ftp/fqdn@REALM key so that it can decrypt the SGT sent in step 3a, encrypted by the TGS in step 2b. In Heimdal and MIT Kerberos, popular open-source implementations of Kerberos, shared keys are stored in a keytab file called `krb5.keytab`, which has strict permissions for read/write access to the admin (`root`) user only.

A credential cache on a client machine stores tickets obtained by a user, such as the TGT and SGTs. Each credential includes a client principal, server principal, encrypted ticket (opaque to the user) and a session key that matches the session key hidden inside the ticket. The credential cache must be secured to prevent impersonation. Heimdal Kerberos secures credentials by storing them in a temporary file, `/tmp/krb5cc_$UID`, which has read/write permissions only for the user who obtained the credential. Other implementations, e.g., Microsoft's, store credentials in memory for greater security.

2.1 Pre-Authentication

Kerberos version 5 introduces a pre-authentication mechanism that allows a client to prove its authenticity before being issued a TGT. A pre-authentication data (`padata`) field in the AS request is set to a value that proves the client's authenticity, such as a timestamp encrypted with the user's password-based key (a mechanism enabled by default on MIT and Windows 2000/2003 implementations). When pre-authentication is mandated by the AS, it prevents an attacker from obtaining an AS reply at will for any user and performing an offline dictionary attack against the encrypted data.

Many different pre-authentication mechanisms may be used, such as smart cards or public keys. For example, PKINIT [33] sends a timestamp encrypted with the user's private key. More `padata` types are defined in the Kerberos RFC [22].

2.2 Security Features

Kerberos has several important security features. User's passwords are never communicated over the network, and session keys are used to communicate securely between the client and the Kerberos server and between the client and the application server. These session keys are always communicated in an encrypted form. In addition, the session key between the client program and the application server may be used for secure communication after the protocol has finished.

Kerberos is also stateless [10]. Session keys are included inside messages, and do not have to be maintained by Kerberos servers. This statelessness increases scalability.

Single sign-on (SSO) is another important feature of Kerberos. With SSO, a user's password must only be entered once per session. The TGT and session key obtained in phase 1 are saved, so each time the user wants to gain access to a service,

only phases 2 and 3 are performed. This feature provides convenience, efficiency, and added security.

Some authentication systems enable SSO via automation in their implementation. For example, a user's password may be remembered and provided automatically during authentication. That technique adds convenience, but not efficiency. Such a system can still benefit from the SSO afforded by incorporation into Kerberos.

2.3 Authentication and Authorization

Although Kerberos is often described as an authentication and authorization protocol, it "does not, by itself, provide authorization" [22]. It does provide a mechanism whereby authorization information can be embedded into a Kerberos ticket in an *authorization-data* field [22], but not all implementations support this field.

In addition, since the *authorization-data* field contains data "specific to the end service" [22], a lack of interoperability may arise between Kerberos authentication servers and application servers that do not understand the same authorization data. The Windows 2000 implementation of Kerberos suffers from this incompatibility [16].

Although this paper deals primarily with authentication, the authorization mechanisms commonly built around Kerberos affect how we must design the authentication protocol so that the overall system remains usable. Our goal is to pursue a design that will not obligate changes to authorization mechanisms.

2.4 Cross-Realm Authentication

Kerberos currently provides a mechanism for cross-realm authentication that enables an authenticated user in one realm to obtain services in another realm, but

cross-realm authentication does not scale well. It requires each realm to mutually trust each other, and to share a secret key. For N realms, there must be $N(N-1)/2$ shared keys [26](p. 94).

Public key extensions to Kerberos such as PKDA [25] improve scalability by eliminating the need to establish such a large number of shared secrets [10]. Unfortunately, even with public-key-enabled Kerberos, a user in one realm must be provisioned explicitly in another realm to gain access to certain services. The authorization systems built around Kerberos usually require known principals for any level of access.

2.5 Limitations

Conventional Kerberos fails to operate as an open system because every user must be known *a priori*. A shared secret between the AS and the user (a password-derived key) must be maintained by the AS, and each user has a 1-to-1 mapping with a principal name.

Most Kerberos extensions are not designed to make Kerberos operate as an open system. Extensions such as PKINIT [33] and other public-key extensions (see Chapter 7) extend credential management to third parties (trusted CAs), but the third parties usually cooperate directly with the Kerberos administrator in creating certificates with principal names that exist in the Kerberos database.

Our goal is to extend Kerberos to be an open authentication system, but modifying Kerberos for each new authentication type is burdensome. Traditionally, new authentication types go through an approval process by the standardizing committee. Once defined, extensions are often rigid and cannot be updated without being re-approved and assigned new pre-auth type numbers. PKINIT has undergone this process.

One might wish to incorporate a proprietary extension into Kerberos without involving the standardization process, but this can be difficult or even impossible when the source code is not available (e.g., Microsoft's implementation). Even when the source code is available, continual resources must be expended to maintain a patch against the latest version of the Kerberos source code.

Chapter 3 — EPAK Design

Extensible Pre-Authentication in Kerberos (EPAK) serves as a model for extending Kerberos to support a variety of authentication schemes. If large security providers such as Microsoft were to adopt EPAK, many businesses would benefit by having the ability to plug in different authentication protocols, including those that would enable Kerberos to operate as an open system.

EPAK extends the initial authentication phase of Kerberos, just like many previous Kerberos extensions. Since only the initial authentication phase is changed, the security infrastructure built up around Kerberos can remain unchanged.

Unlike existing Kerberos extensions, EPAK enables the integration of many authentication schemes into Kerberos without further modification to Kerberos implementations.

3.1 Goals

The design goals for EPAK are to:

- Allow extensible integration of authentication systems
- Enable attribute-based authentication in Kerberos
- Preserve the existing security properties of Kerberos
- Improve efficiency and usability
- Provide scalable account provisioning for outsiders
- Maintain backwards compatibility with Kerberos

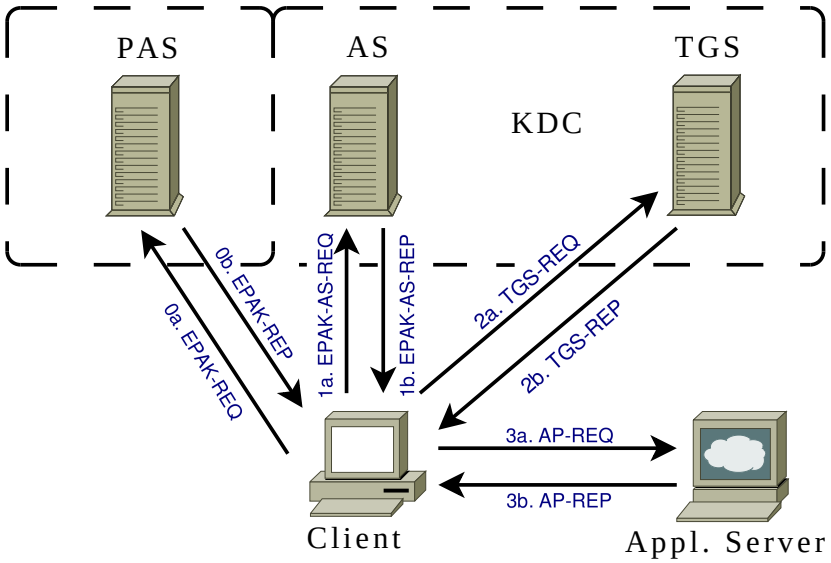

Figure 3.1: The EPAK protocol adds a preliminary phase to Kerberos, phase 0, where the client requests and obtains an authentication-granting ticket (AGT) and a session key $K_{c,as}$ from the Pre-Authentication Server (PAS). The AGT is then supplied as `padata` to the AS in step 1a. Step 1b contains a normal AS-REP with the exception that the session key $K_{c,tgs}$ is encrypted with $K_{c,as}$ instead of a password-derived key. Phases 2 and 3 are left unchanged.

3.2 Architecture

EPAK naturally extends Kerberos by adding a single phase similar to the existing phases (see Figure 3.1). The EPAK framework enables phase 1 of Kerberos to succeed after a Pre-Authentication Client (PAC) authenticates to a Pre-Authentication Server (PAS) using the desired authentication scheme. The PAS determines which users can authenticate to which principals. If authentication succeeds, the PAS returns an authentication-granting ticket (AGT) used as `padata` in the AS request, and a randomly-generated session key for decrypting the AS reply.

Since an AGT only needs to remain valid long enough to perform an AS request

to obtain a TGT, the AGT is, by default, non-renewable and short-lived.

PAS Realms K_{epak} is a randomly-generated key known only to the PAS and AS. By encrypting the AGT with K_{epak} the PAS ensures that only the AS can decrypt it. To provide load balancing and fault tolerance, the PAS may be distributed among multiple machines.

A Kerberos administrator can also outsource pre-authentication by allowing trusted parties to host their own PAS. In this setup, each PAS has its own shared key with the AS, similar to cross-realm authentication. Each party controlling a PAS is known as a *PAS realm* (see Figure 3.2).

To prevent name conflicts and maintain an arms-length trust relationship with each PAS realm, the Kerberos administrator specifies an ACL for each PAS realm, indicating all principals the PAS realm is permitted to authenticate. The AS determines in phase 1 which PAS issued the AGT and enforces the corresponding ACL.

Outsourcing the PAS offloads principal management in addition to computational work and network traffic. It also allows heterogeneous PAS's supporting different authentication mechanisms within the same Kerberos realm. These features increase scalability, but similar to cross-realm authentication, the tight relationship and shared keys limit scalability.

Principal Mapping As mentioned earlier, the PAS is responsible for mapping users to Kerberos principals. In other words, it must implement a strategy for determining which users are allowed to authenticate as which principals.

A straightforward strategy is a *1-1* mapping from users to principals. For example, if users are identified by an email address, a formula can be used to convert

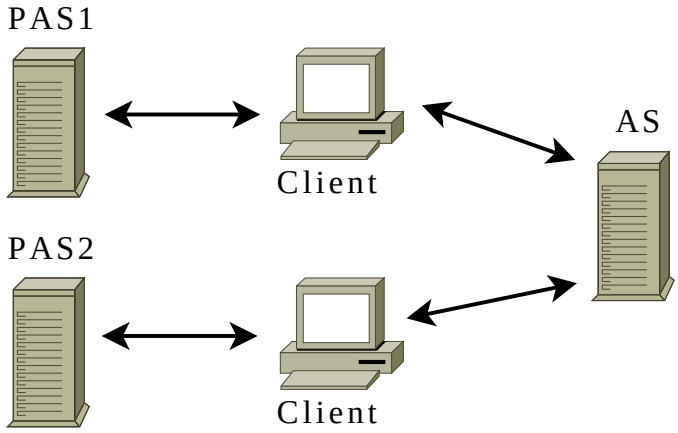

Figure 3.2: Pre-authentication can be distributed to multiple PAS's, where each PAS constitutes a *PAS realm*. Within a single Kerberos realm, a client may perform pre-authentication with any trusted PAS to obtain an AGT.

---

email addresses into corresponding principals, e.g., john@gmail.com can authenticate as john_gmail_com@REALM. The PAS may utilize an ACL for valid users, or simply rely on the AS to reject principals that do not exist.

Although this approach is more open than traditional Kerberos, which requires a shared secret (user password) to be maintained by the Kerberos server, it remains a closed system since the AS maintains a tight relationship with the PAS in provisioning a principal for each valid user. Even when the PAS is distributed among many trusted parties, this limitation still remains.

A rule-based approach like *1-1* mapping is not dynamic enough to allow Kerberos to scale to a large number of outsiders because users are still provisioned individually. A more scalable alternative that transitions Kerberos to an open system is to map a group of users to a Kerberos principal without requiring that each individual user be provisioned in the local Kerberos realm in advance. Two such strategies are described below.

The first strategy, an *m-1* mapping, provides a coarse-grained approach to map users to a single principal. For instance, all users at partner companies can be mapped to a guest principal, e.g., guest@REALM. This dynamic arrangement provides increased scalability because the local Kerberos administrator manages only a single principal and is shielded from all changes to the user population at partner companies. However, mapping users to a single principal is not flexible because all outsiders are treated uniformly.

The second strategy, an *m-n* mapping, is a fine-grained approach that provides a balance of flexibility and scalability. An attribute-based ACL, policy file, or other technique specifies which groups of users can authenticate as which principals. Principals can be defined to represent large groups (e.g., companyC@REALM, partner@REALM).

Combining this *m-n* mapping technique with multiple PAS realms produces an even finer-grained, adaptable solution for user management. Consider the scenario presented in Chapter 1. Company A avoids having to manage accounts for each employee of Company B by grouping them all together (e.g., `*@companyB.com`). Although dynamic and scalable, this configuration may be too coarse-grained for Company A's needs. To enable a more fine-grained setup, Company A could entrust Company B to run a PAS that authenticates users to specific principals, such as `developerB`, `customerB`, and `salesB`. This does not preclude Company A from continuing to use a coarse-grained approach with Company C.

3.3 Protocol

The EPAK protocol consists of four messages, defined in Table 3.1, where the AGT is referred to as the *epakticket*. The EPAK protocol is divided into two authentication phases: pre-authentication and AS authentication.

EPAK Messages

EPAK-REQ	$epakvno \parallel epakdata$
EPAK-REP	$epakvno \parallel epakdata \parallel pasrealm \parallel K_{c,as} \parallel E_{K_{epak}}[epakticket]$
EPAK-AS-REQ	AS-REQ with padata=PA-EPAK-AS-REQ
EPAK-AS-REP	AS-REP with padata=PA-EPAK-AS-REP
	and $K_{c,tgs}$ encrypted with $K_{c,as}$ instead of K_c
PA-EPAK-AS-REQ	$epakvno \parallel pasrealm \parallel E_{K_{epak}}[epakticket] \parallel E_{K_{c,as}}[epakauth]$
PA-EPAK-AS-REP	$epakvno \parallel result$

EPAK Message Elements

epakvno	EPAK version = 1
epakdata	$cname \parallel crealm \parallel starttime \parallel endtime$
epakticket	$K_{c,as} \parallel epakdata$
epakauth	$cname \parallel crealm \parallel cksum \parallel cusec \parallel ctime$
$K_{c,as}$	Random session key generated by PAS
K_{epak}	PAS's key for encrypting *epakticket*
pasrealm	PAS's realm
cname	Client name (principal name)
crealm	Client realm (principal realm)
starttime	Starting time of *epakticket*
endtime	Expiration time of *epakticket*
cksum	Checksum of AS request (excl. `padata`)
cusec, *ctime*	Timestamp [22]
result	Authentication error/success code

Table 3.1: EPAK Message Definitions

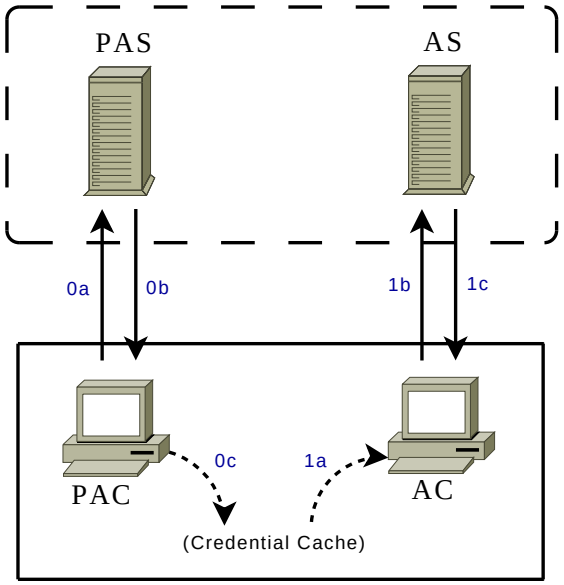

Figure 3.3: The EPAK Protocol. Pre-authentication is performed in phase 0. As the final step the AGT (*epakticket*) and session key $K_{c,as}$ are stored in the client's credential cache. In the first step of phase 1, the credential is read from the cache to be used as `padata` for AS authentication. Since the PAC and AC are separate programs that communicate through the client's credential cache, phase 0 can be customized by EPAK-based protocols without further modification to phase 1.

Pre-Authentication Phase During pre-authentication, a valid client obtains an EPAK-REP from the PAS. The pre-authentication protocol is shown in Figure 3.3, phase 0:

a) The PAC sends an EPAK-REQ to the PAS to indicate the principal requesting authentication. Additional messages may be exchanged in order for the client to complete the authentication

b) The PAS responds with an EPAK-REP

c) The *epakticket* and session key $K_{c,as}$ of the EPAK-REP are stored in the

client's credential cache under the server name `epakt/REALM@`$pasrealm$

The *epakdata* identifies the client, and specifies requested ticket start/end times. The times are then restricted by the PAS in the EPAK-REP to enforce the maximum lifetime.

The EPAK-REP must be communicated securely to protect the session key $K_{c,as}$ from eavesdroppers, and to prevent replay. TLS or another suitable mechanism may be used to transmit the EPAK-REP securely.

The PAS verifies the EPAK version number and then performs any other steps the particular authentication algorithm might require. The PAS must only return an EPAK-REP if the user proves authenticity and is allowed to authenticate to the desired principal.

The following rules for setting the ticket start/end times in the EPAK-REP given the requested start/end time in the EPAK-REQ must be enforced by the PAS:

1. $\nexists(starttime_{req}) \Rightarrow (starttime_{reply} \leftarrow now)$
2. $(starttime_{req} < now) \Rightarrow (starttime_{reply} \leftarrow now)$
3. $(endtime_{req} > now + maxlife) \Rightarrow (endtime_{reply} \leftarrow now + maxlife)$

If endtime is less than starttime, it could be treated as an error, but returning the ticket is safe because such a ticket is invalid and would be useless when presented to the AS.

AS Authentication Phase The protocol for AS authentication with EPAK pre-authentication data is shown in Figure 3.3, phase 1:

a) The encrypted *epakticket*, *pasrealm*, and session key $K_{c,as}$ are retrieved from the client's credential cache

b) The client generates *epakauth* and sends an AS request with PA-EPAK-AS-REQ as the `padata`

c) The server responds with an AS response with PA-EPAK-AS-REP as the `padata`. The session key $K_{c,tgs}$ is encrypted with the session key $K_{c,as}$

The *epakauth* included in the PA-EPAK-AS-REQ shows that the client has recent knowledge of the session key in the *epakticket*. It serves the same purpose as the Authenticator used in phase 2 and 3 [22].

If authentication fails, the PA-EPAK-AS-REP contains an error result value and the encrypted part of the AS reply is set to unusable random data. Alternatively, a Kerberos error message may be returned.

The *pasrealm* indicates which PAS issued the *epakticket*, and is used to look up the appropriate EPAK key needed to decrypt the ticket. It is also used when multiple PAS realms are involved to look up a corresponding ACL.

The following rules of verification of the PA-EPAK-AS-REQ must be enforced by the AS before returning a successful AS reply with an appropriately encrypted session key:

1. *epakvno* must be a valid version number
2. *epakauth* must be valid as in RFC 4120
3. *epakticket* realm must match the realm of the AS
4. *epakticket starttime* $\leq$ *now*
5. *epakticket endtime* $>$ *now*
6. *epakticket* principal must exist in Kerberos database

7. *epakticket* and AS request principals must match

8. *epakticket* principal must appear in ACL (if used)

Rule 6 maintains harmony with current Kerberos implementations, and its absence would necessitate dynamic creation of principals, or modifications to later phases of Kerberos to handle unknown principals. Such changes would have far-reaching effects into the systems built around Kerberos.

The lifetime of the TGT is limited so as not to extend beyond the lifetime of the *epakticket*.

3.4 EPAK Benefits

EPAK benefits both Kerberos and the authentication systems that can now be incorporated into Kerberos.

EPAK improves Kerberos by facilitating the incorporation of new authentication schemes, which can be added without further modification to the Kerberos client (e.g., kinit) or AS (e.g., kdc). EPAK also provides a clear separation between the pre-authentication and authentication phases, thus enabling loose integration of diverse systems.

Incorporating attribute-based authentication schemes into Kerberos enables it to operate as an open system and consequently allows services built on Kerberos to be manageably expanded to larger communities. Also, when Kerberos operates as an open system the need for a shared key between the AS and each client is eliminated. Moreover, the risk of compromise to the client keys central repository is removed.

Authentication systems with complex interactions, or long execution times, benefit from the SSO feature of Kerberos as authentications occur only once per session. Only having to authenticate once per session limits these potential performance bottlenecks.

EPAK lowers the barriers to integrating alternative authentication mechanisms into Kerberos, allowing newer or lesser-known schemes to enjoy a faster adoption rate.

3.5 Backward compatibility

As with other Kerberos extensions, a Kerberos server with EPAK still supports normal Kerberos password-based authentication.

A Kerberos server without EPAK support fails gracefully with a "pre-auth type not supported" error when it receives an EPAK authentication request.

3.6 Limitations

One drawback of EPAK is that it requires at least one extra round of communication. The PAC must communicate with the PAS to obtain the *epakticket*. Other Kerberos extensions, such as PKINIT [33], can provide pre-authentication data in the AS request without needing a previous phase.

Chapter 4 — Open Systems in EPAK

To demonstrate the generality and flexibility of EPAK, we have chosen two authentication systems to integrate into Kerberos: Simple Authentication for the Web (SAW) and trust negotiation. Both SAW and trust negotiation build on the EPAK framework to enable Kerberos to operate as an open system.

4.1 SAW

Simple Authentication for the Web (SAW) [28] leverages email (or other personal messages, e.g., text and instant messages) to authenticate users. SAW significantly improves upon the basic technique employed by the "Forgot your password?" link common to many web sites.

In SAW, users must demonstrate their ability to retrieve two short-lived, single-use *Authentication Tokens* (see Figure 4.1). If a user-supplied email address is authorized, a random secret, $AuthToken_{complete}$, is generated and split into two shares as follows:

$$AuthToken_{complete} \oplus AuthToken_{email} = AuthToken_{user}$$

where $AuthToken_{email}$ is another randomly generated value. $AuthToken_{user}$ is returned directly to the user over the secure link used to initiate the authentication process (e.g., HTTPS) while $AuthToken_{email}$ is emailed. If the user returns both tokens then the authentication is successful.

Since the $AuthToken_{user}$ is returned over a secure link, passively observing the $AuthToken_{email}$ is worthless.

Vulnerability to Active Impersonation By submitting a victim's email address to a site an attacker obtains a valid $AuthToken_{user}$. Consequently, by observ-

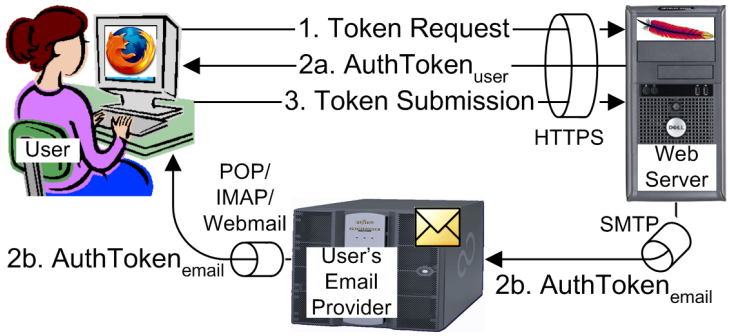

Figure 4.1: The SAW protocol. Based on the user's email address, submitted in (1), a server distributes two authentication tokens. AuthToken$_{user}$ (2a) is returned directly to the user while AuthToken$_{email}$ (2b) is emailed. Both tokens must be returned (3) to successfully authenticate. Each login attempt involves its own unique, short-lived, single-use tokens.

---

ing the victim's incoming email messages, the attacker acquires the corresponding $AuthToken_{email}$ and is able to authenticate as the victim. This is called an active impersonation attack.

SAW's threat analysis argues that SAW provides an acceptable level of risk, even in light of this attack, because sites that employ email-based password resets (EBPR) are also susceptible to a similar attack in which an attacker requests a password reset for the victim and then observes the resulting email message sent by the site. The prolific adoption of EBPR indicates that these risks are manageable.

One-Round SAW Step 3 of SAW is eliminated in *one-round* SAW by setting $AuthToken_{complete}$, normally a random value, to the item requested by the user. Since only authentic users can reconstruct $AuthToken_{complete}$, only those users will be able to obtain the item. As the token splitting used by SAW creates two shares of equal size to the secret it splits, it is advised for a large item to encrypt the

item, split the encryption key, and then deliver the encrypted item with one of the encryption key shares.

Group-Based SAW SAW is often used in closed systems, i.e., an ACL specifies all authorized email addresses. This works well for sites (e.g., forums or photo-sharing) willing to provision accounts for each user.

Unfortunately, this one-to-one specification of users to permissions is insufficient for open systems. For example, this approach requires Business A, from the scenario described in Chapter 1, to maintain an ACL containing some or all of the employee emails of its affiliate, Company B.

For more flexibility, SAW can be modified to use ACLs that contain wildcards or regular expressions. This is known as *group-based* SAW. With this enhancement, Business A can specify that anyone with a Company B email address (e.g., `*@companyB.com`) is allowed access.

4.2 SAWK Naïve Approach

A naïve approach to integrating SAW into Kerberos would be to send an email address in the AS request, inside the `padata` of type PA-SAW-AS-REQ. The AS would reply with the $AuthToken_{user}$ and the session key $K_{c,tgs}$ encrypted with the $AuthToken_{complete}$, and would email $AuthToken_{email}$ to the user, who would be able to reconstruct $AuthToken_{complete}$ and unlock the session key.

As with most Kerberos extensions, the adoption of SAW with this naïve approach would be impeded until it was approved and integrated into popular Kerberos implementations. Before integration, a patch file would have to be maintained, and Kerberos would have to be built manually to enable this functionality.

In addition, this approach provides no mechanism for securing $AuthToken_{user}$, making it susceptible to eavesdropping.

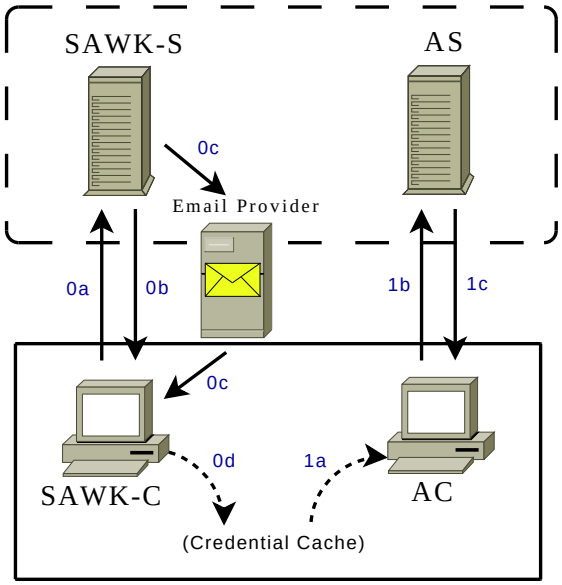

Figure 4.2: The SAWK protocol uses EPAK to enable SAW authentication in Kerberos. The SAWK Server (SAWK-S) and SAWK Client (SAWK-C) embody the PAS and PAC, respectively. In phase 0, one-round, group-based SAW authentication is used to obtain an EPAK-REP. The resulting ticket and session key are stored in the credential cache, after which phase 1 (AS authentication) is performed.

4.3 SAWK Protocol

Simple Authentication for the Web in Kerberos (SAWK) is an EPAK-based protocol that enables flexible, email-based authentication in Kerberos, and avoids the limitations of the naïve approach.

Pre-Authentication Phase The protocol for SAWK pre-authentication is shown in Figure 4.2, phase 0:

a) The SAWK-C sends an EPAK-REQ and email address to the SAWK-S

b) If the address is allowed to authenticate as the principal specified in the EPAK-REQ, the SAWK-S responds with $AuthToken_{user}$ and an EPAK-REP

encrypted with the random $AuthToken_{complete}$

c) $AuthToken_{email}$ is emailed to the specified address and is used to reproduce $AuthToken_{complete}$ and decrypt the EPAK-REP

d) The *epakticket* and session key $K_{c,as}$ of the EPAK-REP are stored in the client's credential cache

The communication between the SAWK-C and SAWK-S in steps 0a and 0b is performed over a secure channel (e.g., TLS) to thwart eavesdropping and impersonations of the SAWK-S.

SAWK uses group-based SAW for a flexible mapping of email addresses to principals. The addresses are specified as regular expressions, which provide an *m-n* mapping with a high level of scalability.

AS Authentication Phase The protocol for AS authentication following SAWK pre-authentication is shown in Figure 4.2, phase 1. This phase is identical to phase 1 of EPAK. As previously mentioned, EPAK-based authentication protocols can be integrated into Kerberos without further modification to the Kerberos client and server programs.

4.4 Trust Negotiation

Trust negotiation [29, 1] is a protocol for establishing trust between strangers with no preexisting relationship. Automated trust negotiation works by exchanging digital credentials until enough trust has been established to gain access to a service or resource. If each party has the required credentials, and their policies allow them to be shown to each other, then trust negotiation will succeed and the resource will be granted.

Digital credentials used for trust negotiation serve the same purpose as paper credentials one might carry in a wallet, such as a driver's license, insurance card, or a student ID. Unlike most authentication systems, which are identity-based, trust negotiation is attribute-based, which provides the flexibility to authenticate based on credential properties. For example, Bob can use his digital driver's license to prove that he is old enough to register for a community college.

An access control policy defines what credentials must be supplied before access to a resource is granted. Policies can also be used to protect credentials, because credentials themselves may be sensitive. For instance, a credit card credential can be protected so that it won't be disclosed unless the other party has a Better Business Bureau credential.

4.5 TNK Protocol

Trust Negotiation in Kerberos (TNK) is an EPAK-based protocol that uses trust negotiation to authenticate clients.

Pre-Authentication Phase The protocol for TNK pre-authentication is shown in Figure 4.3, phase 0:

a) The TNK-C sends an EPAK-REQ to the TNK-S

b) Trust negotiation is performed until the policy has been satisfied, or trust negotiation fails

c) If the policy is satisfied, an EPAK-REP is returned

d) The *epakticket* and session key $K_{c,as}$ of the EPAK-REP are stored in the client's credential cache

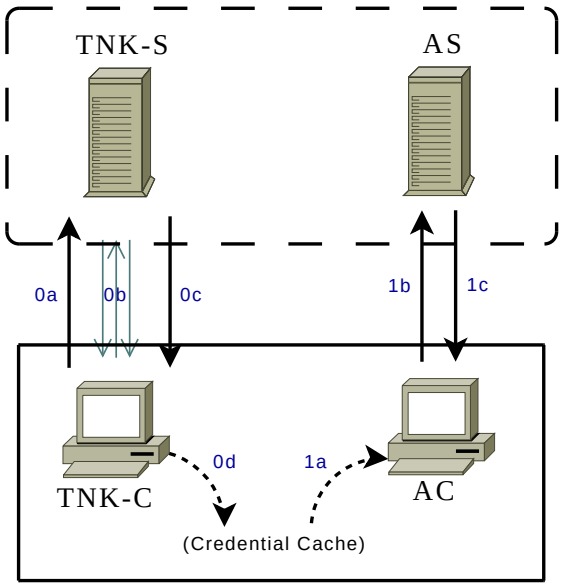

Figure 4.3: The TNK protocol uses EPAK to enable trust negotiation in Kerberos. The TNK Server (TNK-S) and TNK Client (TNK-C) embody the PAS and PAC, respectively. In phase 0, trust negotiation is performed to obtain an EPAK-REP. The resulting ticket and session key are stored in the credential cache, after which phase 1 (AS authentication) is performed.

The principal name in the EPAK-REQ serves as the role the user must satisfy before the EPAK-REP is disclosed. By its very nature, trust negotiation provides a scalable, attribute-based mapping of users to principals.

The communication between the TNK-C and TNK-S is performed over a secure TLS connection to protect potentially sensitive credentials (step 0b), provide server authentication, and to prevent an eavesdropper from viewing the session key $K_{c,as}$ in the EPAK-REP (step 0c).

AS Authentication Phase The protocol for AS authentication after TNK pre-authentication is shown in Figure 4.3, phase 1. This phase is identical to phase 1 of EPAK.

4.6 TNK vs PKINIT

PKINIT [33] is a Kerberos extension that uses public-key cryptography for initial authentication in Kerberos. Similar to TNK, only phase 1 of the Kerberos protocol changes. But unlike TNK, PKINIT authentication is handled completely in the AS request and AS reply, and does not require additional rounds.

In PKINIT, the user sends a certificate to the AS. After verifying the validity of the certificate (signed by a trusted CA and not revoked or expired), the AS responds with the TGT and session key. The session key is encrypted with the user's public key extracted from the certificate, instead of a password-derived key. PKINIT also allows a key generated through a Diffie-Hellman key exchange to be used for this encryption.

PKINIT relies on trusted CAs to issue certificates for users. The principal names are usually specified directly in the certificates, creating a one-to-one mapping between certificates and principals. This limits PKINIT's ability to operate as an open system, since the CAs must work directly with the Kerberos administrator in

managing principals.

PKINIT can function as an open system if the AS is modified to use a different binding mechanism from certificate properties to Kerberos principals. A nice approach would be to modify PKINIT to use a form of credential mapping to map large groups to principal names. For example, the subject name of the certificate maps to a principal name via regular expression mapping, similar to the SAWK ACL list. Other certificate properties could also be involved in the mapping to provide an even more flexible, attribute-based solution, similar to TNK.

Chapter 5 — EPAK Implementation

The flow of messages in EPAK is shown in Figure 5.1. EPAK is implemented as a patch to Heimdal Kerberos [12], and this section is geared towards those familiar with Kerberos implementations.

Changes to the client include modifying `kinit` to support EPAK, and adding helper programs `genpatrequest` and `savepat`. Changes to the server include modifying `kdc` to support EPAK, and adding the helper program `genpatreply`.

genpatrequest This utility program is used by the PAC to generate an EPAK-REQ. The principal name is specified (if different from the user's name), as well as the desired ticket lifetime and start time.

Usage: `genpatrequest [-l time] [-s time] file [principal]`

genpatreply This utility program is used by the PAS to generate an EPAK-REP from an EPAK-REQ. An EPAK-REP is only generated if the EPAK-REQ is valid. The rules specified in Section 3.3 are enforced. The existence of the client principal is not enforced because:

1. The PAS may not have access to the Kerberos database, especially if the PAS is running on a different machine.

2. The principal name will be verified later, by the kdc when it receives the AS request.

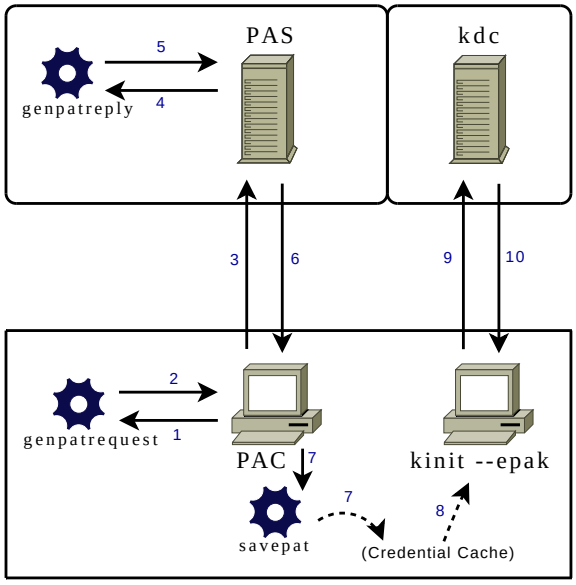

Figure 5.1: EPAK Implementation. The `kinit` and `kdc` programs support EPAK. The new helper programs `genpatrequest`, `genpatreply`, and `savepat` create and process the EPAK-REQ and EPAK-REP messages. The PAC invokes `genpatrequest` to obtain an EPAK-REQ (steps 1-2) that is sent to the PAS (step 3). Additional steps are then performed, as necessary, to authenticate the user. The PAS generates an EPAK-REP by invoking `genpatreply` (steps 4-5) and transmits it securely (e.g., TLS) to the PAC (step 6) to be stored in the client's credential cache (step 7). AS authentication is then performed (steps 8-10).

`genpatreply` must be run by a privileged user with access to the EPAK key (K_{epak}) stored in the `krb5.keytab` file. K_{epak} is used to encrypt the *epakticket* in the EPAK-REP.

Usage: `genpatreply requestfile replyfile`

savepat To save an EPAK-REP to the client credential cache, the PAC uses the `savepat` utility. The *epakticket*, session key $K_{c,as}$, and *pasrealm* from the EPAK-REP are formatted into a `krb5_creds` which is then stored into the credential cache with `krb5_cc_store_cred()`. The credential can be viewed by running the existing `klist` program.

Usage: `savepat [-c ccache] replyfile`

kinit The `kinit` program supports a new option, `--epak`. When run with this option, instead of doing password-based authentication, it performs EPAK authentication. The `epakt/REALM` service credential, which holds an *epakticket* and session key $K_{c,as}$, is read from the credential cache. If it does not exist or is expired, `kinit` aborts with an error. An *epakauth* is created and encrypted with $K_{c,as}$, and is sent along with the *epakticket* in a `padata` of type PA-EPAK-AS-REQ to the AS. If the AS reply includes a PA-EPAK-AS-REP indicating success, `kinit` uses $K_{c,as}$ to decrypt the encrypted part of the AS reply.

kdc The `kdc` program, which performs the function of the AS, supports EPAK by recognizing and responding appropriately to PA-EPAK-AS-REQ `padata`. K_{epak} is used to decrypt the *epakticket*, and the rules specified in Section 3.3 are enforced, including verification of *epakauth*, principal name, and ticket times. The AS reply includes the TGT and session key $K_{c,tgs}$ like normal, but $K_{c,tgs}$ is encrypted with

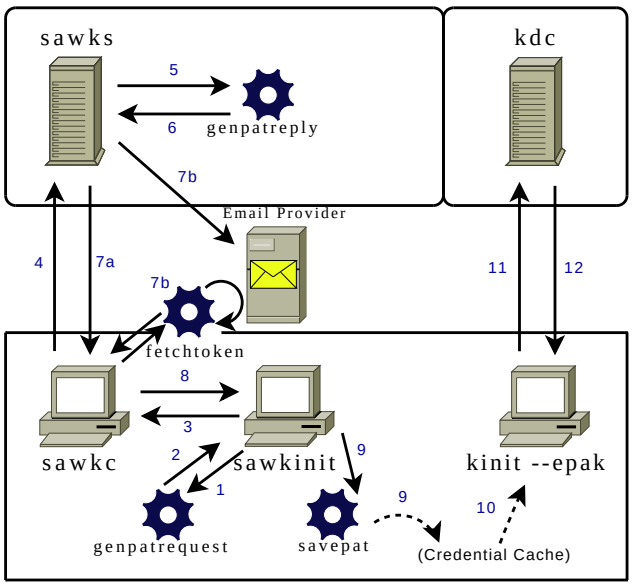

Figure 5.2: SAWK Implementation. Pre-authentication is performed (steps 1-9) by running `sawkinit`. The SAWK Client (`sawkc`) and SAWK Server (`sawks`) communicate securely over TLS. `sawkinit` invokes `genpatrequest` to create an EPAK-REQ (steps 1-2) that is passed to the `sawkc` (step 3). The EPAK-REQ is then transmitted to the `sawks` (step 4) and is used by `genpatreply` to create an EPAK-REP (step 5-6). The EPAK-REP is encrypted with $AuthToken_{complete}$ and returned to the `sawkc` along with $AuthToken_{user}$ (step 7a). $AuthToken_{email}$ is emailed to the user and retrieved by the `fetchtoken` program (step 7b), invoked by `sawkc`. $AuthToken_{email}$ is combined with $AuthToken_{user}$ to to decrypt the EPAK-REP, which is then returned to `sawkinit` (step 8) and stored in the credential cache (step 9). AS authentication is then performed (steps 10-12).

`employeeB@MYCOMPANY`	= `^.*@companyB.com$`
`employeeC@MYCOMPANY`	= `^.*@companyC.com$`
`partner@MYCOMPANY`	= `^.*@company(B|C).com$`
`guest@MYCOMPANY`	= `^.*@.*$`

Table 5.1: Example SAWK-S ACL. Regular expressions map email addresses to principals. For example, john@companyB.com can authenticate as either employeeB, partner, or guest.

$K_{c,as}$ instead of a K_c. A PA-EPAK-AS-REP is also included in the reply.

A new option, `epak_ticket_lifetime`, can be specified in the krb5.conf to indicate the maximum lifetime of an EPAK ticket. If not specified, this value defaults to eight hours.

5.1 SAWK Implementation

The SAWK implementation is shown in Figure 5.2.

sawk This small script runs `sawkinit` followed by `kinit --epak`, to perform preauthentication and AS authentication in one command.

sawkinit The `sawkinit` program is a small script that launches `genpatrequest`, `sawkc`, and `savepat`. The ticket times, if specified, are forwarded to `genpatrequest`, and the credential cache name is forwarded to `savepat`.

A configuration file, `sawkinit.conf`, specifies the location of the three programs mentioned above. It also specifies the hostname and port of the machine running the SAWK-S.

Usage: `sawkinit [-l time] [-s time] [-c ccache] [principal]`

sawkc and sawks These two programs, implemented in Java, communicate to perform one-round, group-based SAW authentication. The `sawkc` and `sawks` communicate over TLS to protect the $AuthToken_{user}$ returned in step 7a.

The email address, specified in `sawkc.properties`, is sent along with the EPAK-REQ in step 4.

In step 7a, the `sawks` responds with three items: the $AuthToken_{user}$, an EPAK-REP encrypted with $AuthToken_{complete}$, and a transaction ID that helps identify the email of step 7b. To prevent leaking valid/invalid addresses, an authentication failure is handled by returning a random value in place of the EPAK-REP.

Valid addresses and their mappings to principal names are specified in the `sawks.acl` file, which uses regular expressions to group email addresses. An example ACL is shown in Table 5.1.

The helper program `fetchtoken` in step 7b retrieves the $AuthToken_{email}$, which is XOR-ed with $AuthToken_{user}$ to produce $AuthToken_{complete}$ and decrypt the EPAK-REP.

Usage: `sawkc requestfile replyfile [host] [port]`

fetchtoken This helper program, written in C, polls the email provider to obtain the $AuthToken_{email}$. The email to retrieve is identified by a transaction ID and the SAWK-S hostname (to help prevent phishing attacks).

The email subject line contains the transaction ID, hostname, and $AuthToken_{email}$ to facilitate quick retrieval. The $AuthToken_{email}$ is saved to a specified token file to be read by `sawkc`.

Account properties for email retrieval are specified in a configuration file named `fetchtoken.conf`. These properties include username, email protocol, mail server,

and timeout. Valid email protocols include POP3 and IMAP, optionally over TLS.

Usage: `fetchtoken trans_id sawks tokfile`

5.2 TNK Implementation

The TNK implementation is shown in Figure 5.3.

tnk This script runs `tnkinit` followed by `kinit --epak`, to perform both pre-authentication and AS authentication in one command.

tnkinit The `tnkinit` program is a small script that launches `genpatrequest`, `tnkc`, and `savepat`. The ticket times, if specified, are forwarded to `genpatrequest`, and the credential cache name is forwarded to `savepat`.

A configuration file, `tnkinit.conf`, specifies the location of the three programs mentioned above. It also specifies the hostname and port of the machine running the TNK-S.

Usage: `tnkinit [-l time] [-s time] [-c ccache] [principal]`

tnkc and tnks These two Java programs perform trust negotiation to obtain the resource "Authenticated as principal X". Policy files dictate what credentials must be released to obtain this resource. The `tnkc` and `tnks` communicate over TLS to protect potentially sensitive credentials.

The EPAK-REQ is sent in step 4, trust negotiation is performed in step 7, and the EPAK-REP is returned in step 8 if trust negotiation succeeds.

Usage: `tnkc requestfile replyfile [host] [port]`

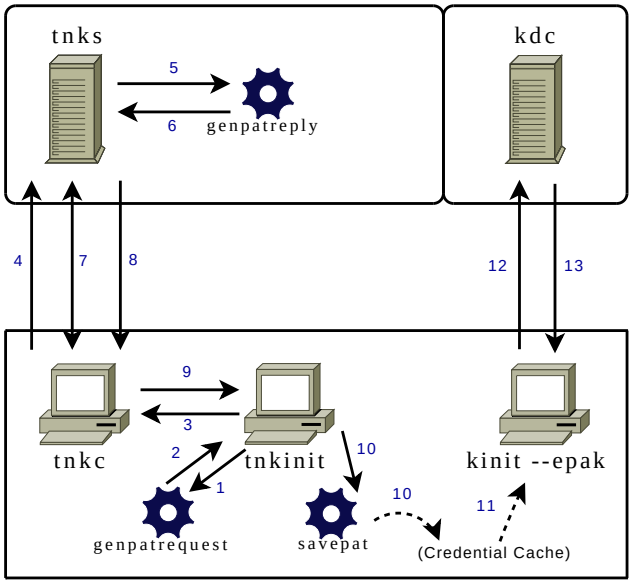

Figure 5.3: TNK Implementation. Pre-authentication is performed (steps 1-10) by running `tnkinit`. The TNK Client (`tnkc`) and TNK Server (`tnkc`) communicate securely over TLS. `tnkinit` invokes `genpatrequest` to create an EPAK-REQ (steps 1-2) that is passed to the `tnkc` (step 3). The EPAK-REQ is then transmitted to the `tnks` (step 4) and is used by `genpatreqply` to create an EPAK-REP (step 5-6). Trust negotiation is performed between the `tnkc` and `tnks` (step 7). If trust negotiation succeeds, the EPAK-REP is transmitted (step 8) to the `tnkc` and is returned to `sawkinit` (step 9) and stored in the credential cache (step 10). AS authentication is then performed (steps 11-13).

5.3 Practice and Experience

Even with documentation for the Heimdal API, the barrier to entry for modifying Heimdal is high, requiring a detailed understanding of its data structures and the functions that operate on them. A proficiency in C and in the automake build system is also required.

The stand-alone programs `genpatrequest`, `genpatreply`, and `savepat` help EPAK authentication systems work with EPAK-REQ and EPAK-REP messages without the need to link against the Heimdal krb5 library. To alleviate the difficulty of interprocess communication, these programs read and write to files, with the filenames passed as command-line parameters.

Working with ASN.1 presents a small challenge. The EPAK-REQ and EPAK-REP messages are, for the most part, handled by the utility programs mentioned above, which use the Heimdal ASN.1 implementation. However, both the SAWK-S and TNK-S need to parse the principal name out of the EPAK-REQ. This is solved by using an ASN.1 Java library, but a more favorable solution might be to create a utility program to extract the name.

While testing the performance of TNK authentication, an anomaly was discovered. The execution time for trust negotiation varied wildly from a few seconds to almost a minute. The culprit was a random-number generator that often blocked when more entropy was required. To solve this, Java was reconfigured to use `/dev/urandom` instead of `/dev/random`. Even though `/dev/random` provides more assurance of truly random numbers, `/dev/urandom` still provides a sufficient amount of unpredictability.

Chapter 6 — Threat Analysis

Kerberos has many important security properties that make it resistant to attacks (see Section 2.2). In general, it provides authenticity for clients and servers, integrity of message data, and confidentiality of secrets, e.g., session keys. The last phase of Kerberos can be used to set up a shared secret between the client and application server to provide confidentiality for subsequent communication.

Although Kerberos is not impervious to all forms of attacks, time has proven it to be effective at ensuring a high level of security.

EPAK aims to retain the confidentiality, integrity, and authenticity properties of Kerberos by extending Kerberos in a natural way. By extending phase 1 with the standard `padata` mechanism, and leaving phase 2 and 3 unchanged, we reduce the opportunity for new flaws.

Phase 0 (pre-authentication) reuses the well-established Kerberos concept of sending a request to obtain a ticket and session key, but it also introduces a new attack vector that must be examined for each particular authentication scheme. For example, SAWK and TNK inherit the security risks of SAW and trust negotiation, respectively. In addition, SAWK and TNK rely on the security of TLS. The design and use of the EPAK-REQ and EPAK-REP messages in phase 0 must also be analyzed.

The *epakticket* and *epakauth* are always communicated in encrypted form. The *epakticket*, just like all Kerberos tickets, is opaque to the client. It is encrypted with K_{epak}, a shared secret between the PAS and the AS. The *epakauth*, created by the client, is protected with the session key $K_{c,as}$ so that it can only be viewed by the AS. It is short-lived and assures that the client presenting the ticket is the one

who was issued the ticket.

The *epakticket* cannot be successfully modified by a client to authenticate as a different principal. Like all Kerberos tickets, modifying the *epakticket* results in a corrupt ticket which, if submitted, will be rejected by the server due to data integrity failure (invalid HMAC).

A client should not be allowed to obtain a ticket with an arbitrary lifetime. The PAS restricts the ticket lifetime by using the `genpatreply` program, which enforces the rules specified in Section 3.3.

The PAS authenticates valid users only. To do so, it only returns an EPAK-REP after the client has proven its authenticity, or it returns the EPAK-REP in an encrypted form such that the client must prove its authenticity to obtain the decryption key.

The PAS is responsible for enforcing the security of the EPAK-REP, to prevent replay and to prevent an eavesdropper from gaining access. Only the session key $K_{c,as}$ of the EPAK-REP needs to be encrypted, but in our implementations of SAWK and TNK, the entire EPAK-REP is encrypted within TLS.

When TLS is used to communicate between the PAC and PAS, it provides server authentication and protects against attacks like DNS spoofing. With TNK, additional server authentication may also be performed as part of the trust negotiation.

If the EPAK-REQ is not communicated securely, an eavesdropper can replay it. However, an attacker must still prove his authenticity before he can obtain an EPAK-REP from the PAS.

Access to a client's credential cache enables impersonation. Kerberos tickets can be used multiple times and therefore require persistent storage. Both system administrators and those with physical access to a client's machine can impersonate

a user during the lifetime of a valid ticket. An *epakticket* is non-renewable, so it presents less risk than other Kerberos tickets. This risk is also mitigated by Kerberos implementations that store credentials in memory instead of on disk.

The `padata` of phase 1 (PA-EPAK-AS-REQ) may be replayed by an attacker, but is ineffective for two reasons. First, the *epakauth* has a short-lived timestamp that provides a small window in which a replay may be performed. Second, even if the replay is dispatched within the limited time frame, the AS reply is useless to an attacker, who does not have the session key $K_{c,as}$ disclosed during pre-authentication. $K_{c,as}$ is needed to decrypt the session key $K_{c,tgs}$ in the AS reply, which is required for phase 2.

An *epakticket* for one principal cannot be used to authenticate to a different principal; an expired *epakticket* will also be rejected. The AS only accepts valid tickets meeting the conditions delineated in Section 3.3.

Chapter 7 — Related Work

Public key based Kerberos for Distributed Authentication (PKDA [25]) relieves the load on a Kerberos KDC server by off-loading the authentication process to the application servers. Clients do not make contact with the KDC at all in this protocol. In effect, it is meant as a replacement for SSL, but the authors themselves admit that SSL is a "formidable" solution.

PKINIT [33] is a Kerberos extension that moves Kerberos beyond password-based authentication to public-key cryptography, which provides greater scalability. EPAK builds on the ideas of PKINIT and other public-key extensions to enhance Kerberos in similar ways.

Role-based Access Control (RBAC) [24] is an approach to mapping user identities to roles within an organization. Users authenticate to known subjects, and then subjects are assigned a role(s). All access control policies are specified in terms of roles. This indirection provides scalability. As users enter and leave the system, the role assignment rules change, but all access control policies remain the same. EPAK leverages this same idea in the way it maps users to Kerberos principals. In its pure form, RBAC is a closed system. The ideas presented in this paper can be applied to RBAC systems to make them open.

GSSAPI [18] is a generic API for client/server authentication. Since most Kerberos distributions include a GSSAPI implementation, applications that support GSSAPI also support Kerberos. Extending Kerberos with EPAK allows these applications to support many other authentication systems. Alternatively, an authentication system could just implement the GSSAPI interface, but that would not afford it the benefits of Kerberos (like SSO), and it could not be used with Kerber-

ized services that do not support GSSAPI.

Chapter 8 — Conclusions and Future Work

EPAK is an attractive framework that facilitates the incorporation of diverse authentication schemes into Kerberos. EPAK clearly separates pre-authentication and AS authentication to enable heterogeneous systems to be loosely coupled with Kerberos. Two concrete examples, SAWK and TNK, demonstrate the extensibility of EPAK.

SAWK and TNK provide grouping techniques that allow Kerberos to scale to a large number of people. Services can be provided to outsiders without managing individual user accounts. Large Kerberos deployments (e.g., Microsoft Active Directory) could adopt EPAK to empower businesses with this rich environment.

Newer, lesser-known authentication systems can be integrated into Kerberos to increase their usability and performance. New protocols can be adopted without changing Kerberos-based security frameworks. Slow-running authentication schemes can leverage Kerberos' SSO capability.

A timing analysis of SAWK and TNK, and a comparison to other work on session resumption is left as future work. In addition, more authentication protocols can be incorporated into Kerberos using EPAK to identify and reaffirm its extensibility and to guide future directions.

Standardization of EPAK is a worthy goal. The types PA-EPAK-AS-REQ and PA-EPAK-AS-REP need to be assigned reserved values so they can be used without risk of conflict. The submission of an RFC for EPAK is a natural next step for it to become an IETF standard.

References

[1] Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust-X: A Peer-to-Peer Framework for Trust Establishment. *IEEE Transactions on Knowledge and Data Engineering*, 2004.

[2] F. Butler, I. Cervesato, A.D. Jaggard, and A. Scedrov. A Formal Analysis of Some Properties of Kerberos 5 using MSR. In *IEEE Computer Security Foundations Workshop*, Jun 2002.

[3] Giovanni Di Crescenzo and Olga Kornievskaia. Efficient Kerberized Multicast in a Practical Distributed Setting. *Lecture Notes in Computer Science*, 2001.

[4] Don Davis. Kerberos Plus RSA for World Wide Web Security. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, Jul 1995.

[5] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, Jan 1999.

[6] I. Downnard. Public-Key Cryptography Extensions into Kerberos. *Potentials, IEEE*, 21, Dec 2002.

[7] Armando Fox and Steven D. Gribble. Security on the Move: Indirect Authentication using Kerberos. In *Mobile Computing and Networking*, 1996.

[8] Ravi Ganesan. Yaksha: Augmenting Kerberos with Public Key Cryptography. In *Network and Distributed System Security*, 1995.

[9] Gary Ian Gaskell. Integrating Smart Cards into Kerberos. Master's thesis, Queensland University of Technology, Feb 2000.

[10] Alan Harbitter and Daniel A. Menascé. Performance of Public Key-Enabled Kerberos Authentication in Large Networks. In *IEEE Conference on Security and Privacy*, Oakland, CA, May 2001.

[11] Alan Harbitter and Daniel A. Menascé. The Performance of Public Key-Enabled Kerberos Authentication in Mobile Computing Applications. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, Philadelphia, PA, Nov 2001. ACM Press.

[12] Heimdal Kerberos 5 Implementation. http://www.pdc.kth.se/heimdal/.

[13] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2000.

[14] Adam Hess, Jared Jacobson, Hyrum Mills, Ryan Wamsley, Kent E. Seamons, and Bryan Smith. Advanced Client/Server Authentication in TLS. In *Network and Distributed System Security Symposium*, San Diego, CA, Feb 2002.

[15] Matthew Hur, Brian Tung, Tatyana Ryutov, Clifford Neuman, Ari Medvinski, Gene Tsudik, and Bill Sommerfeld. Public Key Cryptography for Cross-Realm Authentication in Kerberos, Nov 2001.

[16] Dave Kearns. Kerberos and Windows 2000. *Network World Fusion*, Mar 2000.

[17] N. Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust management framework. In *IEEE Symposium on Security and Privacy*, May 2002.

[18] J. Linn. RFC: 2743: Generic Security Service Application Program Interface Version 2, Jan 2000.

[19] Microsoft. Windows 2000 Kerberos Authentication. Microsoft Technical White Paper, Jul 1999.

[20] Microsoft. Microsoft .Net Passport. Microsoft Technical White Paper, Jan 2004.

[21] Patrick C. Moore, Wilbur R. Johnson, and Richard J. Detry. Adapting Globus and Kerberos for a Secure ASCI Grid. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Denver, CO, Nov 2001. ACM Press.

[22] Clifford Neuman, Tom Yu, Sam Hartman, and Ken Raeburn. RFC 4120: The Kerberos Network Authentication Service (V5), Jul 2005.

[23] S. Sakane and K. Kamada. Applying Kerberos to the communication environment for information appliances. In *Applications and the Internet Workshops*, 2003.

[24] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, Feb 1996.

[25] Marvin A. Sirbu and John Chung-I Chuang. Distributed Authentication in Kerberos Using Public Key Cryptography. In *Network and Distributed System Security*, Feb 1997.

[26] William Stallings. *Network Security Essentials*. Prentice Hall, 2000.

[27] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Technical Conference*, 1988.

[28] Timothy W. van der Horst and Kent E. Seamons. Simple Authentication for the Web. In *To Appear: Security and Privacy in Communications Networks*, 2007.

[29] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated Trust Negotiation. In *Information Survivability Conference and Exposition*, Jan 2000.

[30] M. Winslett, T. Yu, K.E. Seamons, A. Hess, R. Jarvis, B. Smith, and L. Yu. Negotiating Trust on the Web. *IEEE Internet Computing Special Issue on Trust Management*, 6(6), November/December 2002.

[31] Ting Yu, Xiaosong Ma, and Marianne Winslett. PRUNES: An Efficient and Complete Strategy for Automated Trust Negotiation over the Internet. In *7th ACM Conference on Computer and Communications Security*, Athens, Greece, Nov 2000.

[32] Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable Strategies in Automated Trust Negotiation. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, Philadelphia, PA, Nov 2001. ACM Press.

[33] Larry Zhu and Brian Tung. RFC: 4556: Public Key Cryptography for Initial Authentication in Kerberos (PKINIT), Jun 2006.

Appendix A — Source Code

Heimdal 0.8.1 source can be obtained from:

- `http://www.pdc.kth.se/heimdal/`

EPAK, SAWK, and TNK source can be obtained from:

- `http://isrl.cs.byu.edu/epak/`

Appendix B — EPAK ASN.1 Definitions

```
EPAK DEFINITIONS ::=
BEGIN

IMPORTS Realm, Principal, KerberosTime, EncryptionKey, EncryptedData,
    Checksum, krb5int32 FROM krb5;

epakvno INTEGER ::= 1 -- Current EPAK protocol version number.

-- EPAK Data: Main data including principal names, etc.
EPAKData ::= SEQUENCE {
    -- Client principal (name and realm).
    cprinc[0]       Principal,

    -- Client requests desired start and end time.
    -- Server responds with granted start/end time.
    -- (EPAKTicket is not renewable).
    starttime[1]    KerberosTime OPTIONAL,
    endtime[2]      KerberosTime
}

-- The EPAK Ticket is always encrypted by the EPAK key, aka K(epak).
EPAKTicket ::= SEQUENCE {
    -- Session key K(c,as).  (A random session key between client
    -- and AS, generated by pre-authentication server).
    key[0]          EncryptionKey,

    -- Main data including principal names, etc.
    epakdata[1]     EPAKData
}

-- The EPAK Authenticator helps prove that this client was recently
-- granted the EPAK Ticket.  Serves same purpose as authenticators
-- in RFC 4120.
EPAKAuth ::= SEQUENCE {
    cprinc[0]       Principal,
    cksum[2]        Checksum OPTIONAL,
    cusec[3]        krb5int32,
    ctime[4]        KerberosTime
```

```
}

-- EPAK Request: Used to obtain pre-authentication for a client
--               from a custom pre-authentication server.
EPAK-REQ ::= SEQUENCE {
    -- EPAK Version number.
    epakvno[0]      INTEGER (-2147483648..2147483647),

    -- Main data including principal names, etc.
    epakdata[1]     EPAKData
}

-- EPAK Reply: Response from pre-authentication server.
--             Contains pre-authentication data to be used in AS-REQ.
EPAK-REP ::= SEQUENCE {
    -- EPAK Version number.
    epakvno[0]      INTEGER (-2147483648..2147483647),

    -- Main data including principal names, etc.
    epakdata[1]     EPAKData,

    -- Realm of pre-authentication server (PAS)
    pasrealm        Realm,

    -- Session Key K(c,as) that will be needed to decode the AS-REP.
    -- (Random session key between client and AS).
    key[3]          EncryptionKey,

    -- Encrypted EPAK Ticket, used as the pre-auth data in AS-REQ.
    -- The ticket also contains the session key K(c,as).
    epakticket[4]   EncryptedData
}

-- EPAK pre-authentication data for AS-REQ.
PA-EPAK-AS-REQ ::= SEQUENCE {
    -- EPAK Version number.
    epakvno[0]      INTEGER (-2147483648..2147483647),

    -- Realm of pre-authentication server (PAS)
    pasrealm        Realm,

    -- Encrypted EPAK Ticket, which is the pre-auth data.
```

```
    -- The ticket also contains the session key K(c,as).
    epakticket[1]   EncryptedData,

    -- Encrypted EPAK Authenticator, to help prevent replay.
    epakauth[2]     EncryptedData
}

-- EPAK pre-authentication data for AS-REP.
PA-EPAK-AS-REP ::= SEQUENCE {
    -- EPAK Version number.
    epakvno[0]      INTEGER (-2147483648..2147483647),

    -- Server responds with 0 if pre-auth succeeded.
    result[1]       INTEGER (-2147483648..2147483647)
}
```

Appendix C — EPAK Installation Guide

Build Heimdal Kerberos w/EPAK support

1. Download Heimdal Kerberos 0.8.1 (`heimdal-0.8.1.tar.gz`)
   - MD5 = `7ff8c4850bce9702d9d3cf9eff05abaa`
   - See Appendix A
2. Download EPAK patch (`epak.patch`)
   - See Appendix A
3. Install 3rd party tools and libraries
   - yacc/bison
   - flex
   - xt library (libxt-dev)
   - Berkeley DB (libdb3-dev)
   - ncurses (libncurses5-dev)
4. Extract Heimdal Kerberos
   - `tar -zxvf heimdal-0.8.1.tar.gz`
5. Apply EPAK Patch
   - `cd heimdal-0.8.1/`
   - `patch -p1 < /path/to/epak.patch`
6. Build Heimdal Kerberos with EPAK enabled
   - `./configure --enable-epak --enable-epakdebug`
     - (`--enable-epakdebug` is optional)
   - `make`
7. Install Heimdal Kerberos (optional)
   - make install

Set up /etc/krb5.conf on server and client

A sample `krb5.conf` is presented below. The `epak_ticket_lifetime` should be set to a value similar to the other ticket lifetimes, such as eight or ten hours (36000 seconds).

```
[libdefaults]
        ticket_lifetime = 36000
        epak_ticket_lifetime = 36000
        default_realm = SSHOCK.HOME
        no-addresses = true

[realms]
        SSHOCK.HOME = {
                kdc = sshock.homeipx.net
                admin_server = sshock.homeipx.net
                default_domain = sshock.homeipx.net
        }

[domain_realm]
        sshock.homeipx.net = SSHOCK.HOME

[logging]
        kdc = FILE:/var/log/krb5kdc.log
        admin_server = FILE:/var/log/kadmin.log
        default = FILE:/var/log/krb5lib.log
```

Set up Kerberos server

Note: These commands must be run as `root` (or using `sudo`).

1. Create heimdal directory
   - `mkdir /var/heimdal`
   - `chmod 700 /var/heimdal`
2. Create master key file
   - `cd heimdal-0.8.1/`
   - `kdc/kstash --random-key`
3. Initialize database

- `kadmin/kadmin -l`
- `init REALM`
  - where `REALM` is the name of your realm
  - init will ask some questions about max ticket life

4. Add a principal for your username (optional)
   - `add username`
5. Add one or more principal that will be used with pre-authentication
   - `add --random-key princname`
6. Add `epakt/REALM` (EPAK Ticket) service principal
   - Run these commands on the PAS machine
   - `kadmin/kadmin -l`
   - `add --random-key epakt/REALM`
   - ext epakt/REALM
     - This puts the key into the keytab file `/etc/krb5.keytab`.
     - If KDC resides on a different machine, you must export the principal to the keytab of the KDC as well.
7. To test ftp or telnet, add service principals and setup daemons
   - `add --random-key host/hostname`
     - where `hostname` is the domain name of the ftp or telnet server
     - `host/localhost` may work fine for testing.
   - `ext host/myhostname`
     - This puts the key into the keytab file `/etc/krb5.keytab`.
     - If ftp or telnet server is on a different machine than the KDC, you must export the principal to the keytab on that machine as well.
   - Setup `telnetd` and `ftpd` in your `inetd.conf`.
     - Make sure your clients use kerberized `telnet` and `ftp` programs.
   - If you didn't install kerberos with `make install`, you must add a link from `/usr/heimdal/bin/login` to `heimdal-0.6.3/appl/login/login`.

Download and setup SAWK and/or TNK for testing EPAK

See Appendix A for source code.