# Yaksha[1]: Augmenting Kerberos with Public Key Cryptography

Ravi Ganesan

Center of Excellence for Electronic Commerce
Bell Atlantic
Silver Spring, MD 20904
Ravi.Ganesan@Bell-Atl.Com

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
ganesan@blaze.cs.jhu.edu

## Abstract

*The Kerberos authentication system is based on the trusted 3rd party Needham-Schroeder authentication protocol. The system is one of the few industry standards for authentication systems and its use is becoming fairly widespread. The system has some limitations, including the fact that compromise of the on-line trusted 3rd party is catastrophic and that the system is vulnerable to dictionary attacks. Further, while the system provides for authentication and key-exchange, it does not provide non-repudiation (i.e. digital signature) services, as a result of which an organization using Kerberos would have to maintain a separate security infrastructure for the latter function.*

*Many of these limitations are traceable to the decision of the Kerberos designers to solely use, freely available, symmetric key cryptosystems. Using asymmetric, or public key, cryptosystems in an authentication protocol would prevent some of the shortcomings addressed here. Several such protocols have been proposed and some have been implemented. However, all these designs are either completely different from the Kerberos system, or require major changes to the basic system. Our goal in designing Yaksha[1] is different. Having observed the fairly tortuous and time consuming process the Kerberos community has wound through to finally arrive at what is a mature, and from all appearances a fairly secure, standard, we are of the firm conviction that any attempts to improve Kerberos would do so with only minimal impact to the protocol and the source tree. In this work we describe Yaksha, a new approach to achieving these goals.*

*Yaksha uses as its building block an RSA variant independently invented by Boyd [Boyd89] and by Ganesan and Yacobi [Gane94], in which the RSA private key is split into two portions. One portion becomes a user's Yaksha password, and the other the Yaksha server's password for that user. Using this simple but useful primitive we show how we can blend the Kerberos system with a public key infrastructure to create Yaksha, a more secure version of Kerberos with minimal changes to the protocol.*

**KEYWORDS**: Authentication, Authentication Protocols, Dictionary Attacks, Digital Signatures, Kerberos, Key Exchange, Non-Repudiation, Passwords, Yaksha

## 1.0 Motivation

The Kerberos authentication system [KOHL93] based on the classic Needham-Schroeder authentication protocols [NEED78] with extensions by Denning-Sacco [DENN78], uses a trusted 3rd party model to perform authentication and key exchange between entities in a networked environment. Kerberos uses symmetric key cryptosystems as a primitive, and initial implementations use the Data Encryption Standard (DES) as an interoperability standard, though any other symmetric encryption system can be used. After close to a decade of effort, the Kerberos authentication system is now a fairly mature standard whose security properties have held up fairly well to intense scrutiny. Further, it is finally the case that vendors are delivering Kerberos as a supported product. It has also been adopted as the basis for the security service by the Open Software Foundation's (OSF) Distributed Computing Environment (DCE). Consequently, we expect

[1] In Greek mythology, Kerberos is the three headed dog that guards the gates of Hades, "the land of the dead, underworld". To guard something more valuable, for instance, the gates of heaven, we need a Yaksha. In Hindu mythology, Yakshas (Yakshini is the feminine), are 'good' demi-gods who, among other things, guard the gates of heaven. Yakshas are also extremely flexible and can transform themselves into any other form e.g. birds, cows, and presumably, three headed dogs. :-)



0-8186-7027-4/95 $4.00 

Kerberos to be among the most widespread security standards used in distributed systems over the next several years.

Kerberos does have limitations, and among the more serious ones are:

- Compromise of the central trusted on-line Kerberos server is catastrophic, since it retains long term user secrets. Kerberos is vulnerable to password guessing dictionary attacks.
- Kerberos does not provide non-repudiation services (i.e. digital signatures)

The first limitation is intrinsic to the Needham Schroeder protocol when used with symmetric cryptosystems like DES. The second problem is also a major issue since experience suggests that password guessing attacks tend to be far more common than most other forms of attacks - they are simple and effective. Finally, Kerberos was designed to provide authentication and key-exchange, and hence it may be unfair to characterize its not providing digital signatures as a "limitation". However, most organizations using Kerberos will also want to implement digital signatures, and will have to maintain separate security infrastructures for Kerberos and for digital signatures - a significant cost.

A major reason for these limitations is that Kerberos does not use asymmetric, or public key, cryptosystems. It is a fairly straightforward exercise to create a paper design of an authentication protocol that uses public-key cryptography and avoids some of these limitations. And with significantly more effort, one can design a full fledged system with a public key infrastructure which achieves the same goals as Kerberos without its associated limitations. DEC's SPX [TARD91] system is one such example. Our motivation for this work begins from a different set of constraints. Namely, we believe that the effort required to get a multi-vendor supported standard authentication system whose security properties have been widely examined is probably the hardest part of implementing a new system. For the most part, this effort has already been exerted on behalf of Kerberos, and consequently we believe any addition of public-key cryptography to Kerberos must meet the following two constraints:

- It should require minimal changes to the protocol as defined in [KOHL93]. Specifically, analogous to generational increments in the instruction set of a microprocessor, the changes to the Kerberos protocol should be incremental to increase the likelihood of backward compatibility.
- It should require minimal changes to the Kerberos source tree, and again the changes should be primarily in the form of additions.

These two constraints are driven by practical considerations, but are difficult to meet. For instance, Kohl [KOHL91] (as quoted in [SCHN94]) suggests that: "Taking advantage of public-key cryptography would require a complete reworking of the protocol". We do not believe this is necessary and this work describes Yaksha, a new method of adding public-key cryptography to Kerberos, that to a large extent meets the first constraint. Further, we strongly suspect that our approach meets the second constraint of minimizing changes to the source tree, but this can only be proven when the system is built.

## 2.0 Relevant Prior Work

We first describe the relevant prior work that we use as building blocks in our design and then also comment briefly on other approaches to the same problem. Readers familiar with the Kerberos messaging structure, public key cryptography and RSA are urged to skip directly to Section 2.5

### 2.1 Kerberos: A Protocol Overview

For the sake of clarity, in this paper we will use the "simplified" version of the Kerberos protocol described by Neuman and Ts'o in [NEUM94]. The extension of our ideas to the complete protocol, as described in [KOHL93], is straightforward. Further, the Kerberos overview in this section is based on [NEUM94], and for the sake of consistency uses almost the same notation. The fundamental message exchanges are shown below in Figure-1.

We now describe the messages in further detail. *Message-1*, known as *as_req* (request to authentication service) consists of:

$$as\_req: c, tgs, time\text{-}exp, n$$

where $c$ is the name of the client (user), *tgs* the name of the ticket granting service for which the client is requesting a ticket granting ticket $T_{c,tgs}$. *time-exp* is the requested expiry time of the ticket (typically eight hours) and $n$ is a fresh random number. This message is sent in the clear, and all parts of it are visible to an eavesdropper. The authentication server (*as*) responds with *Message 2*,

$$as\_req: \{K_{c,tgs}, time\text{-}exp, n, ...\}K_c, \{T_{c,tgs}\}K_{tgs}$$

where $K_{c,tgs}$ is the session key to be shared between the ticket granting server (*tgs*) and the user for the lifetime of this ticket. Note that we are using the notation $\{M\}K$, to denote the encryption of message $M$ using a symmetric encryption system, e.g. DES, using key $K$. $K_{c,tgs}$ and the

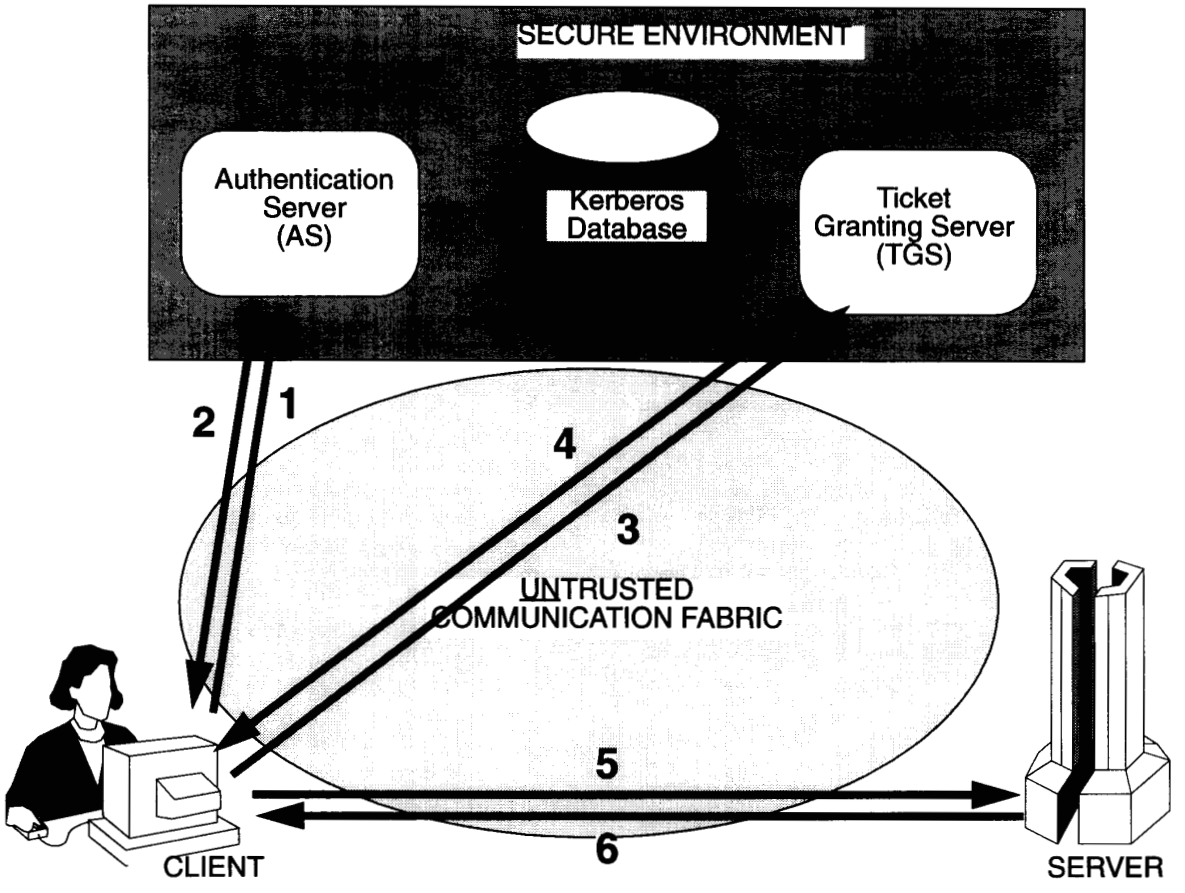


Figure 1: Kerberos message exchange overview (adapted from [NEUM94]). In message 1 the user requests a ticket granting ticket (TGT). the server creates such a ticket, looks up the user's password from the Kerberos database, encrypts the TGT with the password and sends it to the user in message 2. The user decrypts the TGT with her password, and stores the TGT on her computer. Then, when she wants to access a service, she sends message 3, which contains the TGT to the server, who verifies the TGT and sends her back, in message 4, a ticket to access the server and a session key. In message 5 she presents the ticket to the server, which verifies it and also recovers the same session key from it. If mutual authentication is required, the server, in message 6, sends back a message encrypted with the session key.

other information is encrypted with $K_c$ which is the user's password (the long term secret which is shared with the Kerberos server). Only a user who knows $K_c$ will be able to decrypt this message to obtain $K_{c,tgs}$. This key $K_{c,tgs}$ is also embedded in the ticket $T_{c,tgs}$, which in the *as_rep* is encrypted using $K_{tgs}$, a long term key known only to the *as* and the *tgs*. After decrypting the first part of the message, the user now stores the data received in the *as_rep* on the local computer. The main purpose behind this is to avoid storing the long term key $K_c$ on the computer where it may be compromised. Rather, the key $K_{c,tgs}$ is used in lieu of $K_c$. Since $K_{c,tgs}$ is relatively short lived, the damage an attacker can cause by learning this key is less.

It is worth observing that the *as* does not verify the identity of the user before responding to a user's *as_req* with an *as_rep*. Rather *as* relies on the fact that to be able to make any use of the *as_rep*, the recipient must know $K_c$. So an attacker can actually get an *as_rep* from the *as* by sending a fraudulent *as_req*. The attacker can then take the portion of the *as_rep* encrypted with $K_c$, and attempt to decrypt by taking guesses at $K_c$. Since $K_c$ is typically a user selected password, $K_c$ may well be a poor password, which the attacker can guess. Even if this "vulnerability" is closed (there is an option to do so in Version 5 of Kerberos), an attacker can always eavesdrop on the network to obtain information using which a password-guessing attack can be mounted.

When the client wishes to obtain a ticket to access a server, it sends to the *tgs*, *Message 3*,

$$tgs\_req:\ s,\ time\text{-}exp,\ n,\ \{T_{c,tgs}\}K_{tgs},\ \{ts...\}K_{c,tgs}$$

This message consists of the name of the server, *s*, the expiry time, *time-exp*, requested and the random number, *n*, in cleartext. It also contains the encrypted ticket granting ticket $\{T_{c,tgs}\}K_{tgs}$ which was received by the client in the *as_rep* message. Upon receipt of the message the *tgs*, which knows $K_{tgs}$, can decrypt and recover $T_{c,tgs}$, which is a valid ticket. In order to prevent a replay attack in which an attacker might gain some benefit by re-sending a valid $\{T_{c,tgs}\}K_{tgs}$ at a later time, the *tgs_req* message also contains an authenticator, which is a timestamp, *ts*, a check sum and other data, all encrypted with the session key $K_{c,tgs}$. Since this session key is embedded in the ticket $T_{c,tgs}$, which the *tgs* has recovered, the *tgs*, can decrypt the authenticator and verify the time stamp and check sum. By maintaining a cache of recently received authenticators, the *tgs* can detect replays.

Having verified the authenticity of the *tgs_req*, the *tgs* responds with *Message 4*,

$$tgs\_rep:\ \{K_{c,s},\ time\text{-}exp,\ n,\ s,...\}K_{c,tgs},\ \{T_{c,s}\}K_s$$

This message is very similar in structure and purpose to the *as_rep*, message. The first part consists of a session key, expiry time, etc., encrypted with $K_{c,tgs}$. The client can decrypt this to recover the session key and other information. The second portion is a ticket to access the server, encrypted with the long term key shared by the server and the *tgs*. The client now constructs *Message 5* and sends it to the server,

$$ap\_req:\ \{ts,ck,\ ...\}K_{c,s}\ \{T_{c,s}\}K_s$$

This message is similar to the *tgs_req*, in that it contains an encrypted ticket $\{T_{c,s}\}K_s$ which the server can use to recover $T_{c,s}$, which authenticates the client to the server and, among other information, contains the session key $K_{c,s}$. The server then uses $K_{c,s}$ to decrypt the first part of the message, the authenticator, which has a time-stamp, *ts*, a check-sum, *ck*, etc.

Having verified the authenticity of the client, the client and server are ready for communication. However, in some cases the client may request mutual authentication, in which case the server must first respond with *Message 6*,

$$ap\_rep:\{ts\}K_{c,s}$$

which is basically proof that the server successfully recovered $K_{c,s}$ from the ticket $T_{c,s}$, which means the server knew $K_s$, which in turn is proof of authenticity of the server.

The actual protocol has a number of options and is more complex, but the basic structure is defined by these six messages. The interested reader is referred to [KOHL93] for more details.

### 2.2 Public-Key (Asymmetric) Cryptosystems

This subsection provides a quick overview of public-key cryptography. In public-key systems each entity, *i*, has a private key, $P_i$, which is known only to the entity, and a public key, $U_i$, which is assumed to be publicly known. The system has the special property that once a message is encrypted with a user's public-key, it can only be decrypted using that user's private-key, and conversely, if a message is encrypted with a user's private-key, it can only be decrypted using that user's public-key (in some systems only operations in one direction are permissible). So, if the sender wishes to send a message to receiver, *i*, then the sender "looks-up" *i*'s public key, $U_i$, and computes $C=E(M,U_i)$ and sends $C$ to *i*. *i* can recover $M$ using its private-key, $P_i$, by computing $M=D(C,P_i)$. An adversary who makes a copy of $C$, but does not have $P_i$, cannot recover $M$. Public-key cryptosystems are not however very

efficient (e.g. RSA is roughly 1000 times slower than DES when both are implemented in hardware, and 100 times slower when both are implemented in software [SCHN94]), and typically cannot be used for large messages.

Public-key cryptosystems can also be used for digital signatures. The signer, *i*, computes $S=E(M,P_i)$ and sends $(M,S)$ to the recipient. The recipient "looks-up" *i*'s public-key, $U_i$, and then checks to see if $D(S,U_i)$ is equal to $M$. If so then the recipient is convinced that *i* signed the message, since computing an $S$, such that $M=D(S,U_i)$, requires knowledge of , *i*'s private key which only *i* knows.

### 2.3 Review of the RSA Cryptosystem

RSA[RSA78] is a public-key based cryptosystem that is believed to be very difficult to break. In the RSA system the pair $(e_i,n_i)$, is user *i*'s public-key and $d_i$ is the user's private key. Here $n_i = p \times q$, where $p$ and $q$ are large primes, and $e_i \times d_i \equiv 1 \bmod \phi(n_i)$, where $\phi(n_i) = (p-1)(q-1)$ is the Euler Toitient function which returns the number of positive numbers less than $n_i$, that are relatively prime to $n_i$. To encrypt a message being sent to user *i*, user *j* will compute $C \equiv M^{e_i} \bmod n_i$ and send $C$ to *i*. *i* can then perform $M \equiv C^{d_i} \bmod n_i$ to recover $M$. The RSA based signature of user *i* on a message, $M$, is $S \equiv M^{d_i} \bmod n_i$. The recipient of the message *j*, can perform $M \equiv S^{e_i} \bmod n_i$, to verify the signature of *i* on $M$. Note that in RSA encryption and signatures can be combined.

### 2.4 Review of Public Key Certificates

Since a recipient of a message must know the sender's public-key, a method must be provided to securely provide this information to the recipient. One common method [KENT93] is the concept of certificates. A certificate is basically a binding between an entity and its public key, as vouched for by some authority. So a certificate in a RSA based infrastructure could contain $Cert = \{i,e_i,n_i\}$. The certificate is signed by some trusted third party called a Certificate Authority (*CA*). So when *i* sends *j* a signed message $S \equiv M^{d_i} \bmod n_i$, it is accompanied by $(Cert)^{d_{CA}} \bmod n_{CA}$. *j* can recover *i*'s public key from the certificate using $(e_{CA},n_{CA})$, the Certificate Authority's public-key which is assumed to be universally available. In an informal sense, the degree of trust in the off-line Certificate Authority is (arguably) much less than the trust placed in an on-line Kerberos server.

### 2.5 Review of Boyd's RSA Variation

Boyd [Boyd89] introduced an interesting RSA variation for "digital multisignatures". In his scheme the RSA private key $d$ is split into multiple portions $d_1,d_2.....d_k$, where $d_1 \times d_2....d_k \equiv d \bmod \phi(n)$. The *i*th portion $d_i$ is given to the *i*th user. The users can then jointly sign a message . For example if there are two users $(k=2)$, then the first user computes $S_1 \equiv M^{d_1} \bmod n$, and the second user completes the signature by computing $S \equiv S_1^{d_2} \bmod n$. The resulting signature is identical to one signed by the regular RSA private key (i.e. $S_2 = M^d \bmod n$) and can hence be verified, in one operation, using the regular public-key.

***For simplicity of notation we are dropping the "mod n" from our explanation for the rest of this paper, but all exponentiations are modulo the appropriate modulus. For instance, $M^{d_i}$, refers to $M^{d_i} \bmod n_i$.***

### 2.6 Review of Ganesan-Yacobi RSA Variant

Ganesan and Yacobi [GANE94] reinvented Boyd's system, and made four significant additional contributions. All their results apply to the two party case, but are believed to be generalizable. The four results are (using the same notation as in our description of Boyd's scheme):

1. They prove mathematically that breaking the joint signature system is equivalent to breaking RSA. The attacker can be an active/passive eavesdropper or one of the participants. They assume that key generation is conducted by a trusted 3rd party, like a tamper proof chip, and the factorization of the RSA modulus and $\Phi(n)$ are discarded after key generation and not known to any of the participants.
2. They describe the following key exchange protocol User 1 sends $x^{d_1}$ to User 2. User 2 recovers $x \equiv ((x^{d_1})^{d_2})^e$. Similarly User 2 transmits $y^{d_2}$ to User 1, who recovers $y$, The users can use as the session key some function of $x$ and $y$ (e.g. $x \otimes y$). Ganesan and Yacobi proved mathematically that breaking this key exchange protocol is equivalent to

breaking RSA. The attacker can be an active/passive eavesdropper or one of the participants.

3. Next, they introduce the concept where one of the two users is actually a central server which maintains one portion of every user's private key. In order to sign a message the user must interact with this server (which they prove, cannot impersonate the user). Having to interact with such a central server to sign runs somewhat counter to prevailing conventional wisdom. However, it turns out to have several important practical advantages including instant revocation (without difficult to maintain Certificate Revocation Lists), a central point for audit and as discussed below, a method of providing digital signatures in an era where smart cards are not yet ubiquitous.
4. Ganesan and Yacobi proved mathematically that even if one of the two portions, $d_1$ and $d_2$, of the private key, is short, say 80 bits, then for an active or passive eavesdropper to break the system is still as difficult as breaking RSA. As a consequence, a digital signature infrastructure can be built where users who remember short (say ten characters) passwords, can interact with the central server to create RSA signatures. The signatures created are indistinguishable from those created using a full size RSA key stored on a smart-card. They conjectured, but do not prove, that the system is still secure from a malicious central server even when the user keys are short. Michael Weiner [WEIN94] illustrated an attack where a malicious server can mount an attack that runs in roughly $O(\sqrt{2^l})$ steps, where $l$ is the number of bits in the short user password. So for instance if a security factor of $2^{40}$ is required, than the user password should be 80 bits long. Again, this is an attack mounted by a malicious server, not an eavesdropper.

In 1. and 2. above we observe that although the goals are signatures and key-exchange respectively, authentication is a natural by-product. Ganesan and Yacobi suggest that this system is a simpler alternative to Bellovin and Merrit's Encrypted Key Exchange (EKE) [BELL92](the RSA version) and unlike EKE, does not require the two parties to share a common secret key.

Yaksha uses results 1., 3. and 4. to envisage an authentication system, in which the server, instead of sharing a common symmetric key with each user, retains a portion of each user's private RSA key. Several authentication protocols can be created using this basic idea. In this paper we restrict ourselves to illustrating how we can modify Kerberos using these results to arrive at a Kerberos-like protocol.

### 2.7 Other Potential Approaches to a Better Kerberos

The SPX system [TARD91] is a full-fledged public-key based authentication system which does not require a trusted on-line server. It's protocol is sufficiently different from Kerberos to make integration of these systems require a complete reworking of the Kerberos protocol. Bellovin and Merrit's Encrypted Key Exchange [BELL92] can potentially be integrated with Kerberos to prevent dictionary attacks. However, their multi-pass protocol would require very significant changes to the Kerberos system. A nice feature of EKE is that the authors show how it can be implemented using the RSA, Diffie-Hellman or El Gamal public key cryptosystems. Our system however, to our knowledge, works only with RSA. Also, unlike our system, EKE assumes that the participants share a common long term secret. Finally, both these systems, like Yaksha, generate public-private key pairs on the fly, but all three use these dynamic keys in totally different ways.

The Sesame project [MCMO94] also integrates public-key cryptography with Kerberos, but the focus there has been on adding public key cryptography to the inter-realm portions of Kerberos, to make those aspects more secure. Our approach can be used to meet their objectives.

## 3.0 The Yaksha Design Goals

We now discuss our design goals in more detail.

### 3.1 Removing Vulnerability to Catastrophic Failure

The Kerberos system shares a permanent secret with every user and service. Compromise of this database is catastrophic. Our most important design goal is to alleviate this problem. Practically speaking, compromise of the server in any server-centric design will result in some damage. We believe that any such compromise will be short lived (for example, if the database is surreptitiously copied, then fraudulent use of services will at some point be detected), and hence our goal is to minimize the damage that can be caused in the interval. Specifically:

*Compromise of the server should not allow the attacker to impersonate a client to the server or vice versa.*

Yaksha meets this goal, with the caveat that (in the version of our system where the user has a short private key) an attacker who compromises the server (unlike an eavesdropper) can mount an expensive dictionary attack against the user (see Section 3.2).

### 3.2 Removing Vulnerability to Dictionary Attacks

Dictionary attacks are a common form of attack, and it is well known that many systems (e.g. UNIX [MORR79] or Kerberos) are vulnerable [KARN89] to this attack. However, all dictionary attacks are not alike, and it is worth considering a taxonomy of such attacks. There are four parameters to a dictionary attack:

1. The known plaintext, *S*, which can take two forms:
   - a string *S1* which is known in advance to the attacker. An example of *S1* is a string of zeroes.
   - a string *S2* which is not known to the attacker in advance, but "he'll know it when he sees it". An example of *S2* would be any string with some form of predictable redundancy, for instance a time stamp. Another example would be if *S2* were a number with particular, easily tested, mathematical properties, for instance a prime, or a non-prime with no small factors.
2. The ciphertext *C*, typically of the form $C=F(S,k)$ where *k* is the password being sought.
3. The password space *P* being guessed consists of *N* passwords. The attacker will take guesses $p_1, p_2,...,p_N$, till he finds a $p_i$ which is equal to *k*.
4. The function *F* and its inverse (assuming one exists), which are typically public information. It is important to draw a distinction between the cases when *F* is a symmetric encryption system like DES, and when *F* is an RSA function of modular exponentiation.

These four parameters yield at least two distinct forms of dictionary attacks:

- S1 type attacks. Here the attacker typically computes $F(S1,p_i)$ (or perhaps $F^{-1}(S1,p_i)$) for every $p_i$ in *P* until he discovers a $p_i$ where, $F(S1,p_i) = C$ (or $F^{-1}(S1,p_i)=C$). This is the most dangerous form of attack since the attacker can (a) Precompute the $F(S1,p_i)$ for all or many $p_i$ and (b) The attacker can amortize his attack against several users. UNIX is vulnerable to such attacks.
- S2 type attacks. Here the attacker is typically computing $(C,p_i)$ and is hoping to find an *S2* which he can recognize. Here the attacker cannot start computations before he captures *C*. Further, since *C* will be different for each instance, no amortizations of attack are possible. The Kerberos system is vulnerable to this form of attack.

*Our design goal is a system that is not vulnerable to either form of attack from an eavesdropper.*

As mentioned earlier, in the event of server compromise, AND, the use of short keys by the user, our system is vulnerable to the second form of dictionary attack. However, since the attacker would have to use modular exponentiation as the function *F*, the resulting attack will be far slower than a dictionary attack against DES, and further, the attack cannot be amortized.

### 3.3 Minimize Protocol Changes

We have an extremely minimalistic approach to any protocol modifications. Specifically:

- We do not want additional "rounds" to any protocol exchange. We constrain ourselves to the basic six messages described earlier.
- We do not want to change any important structures, e.g. the structure of the tickets.
- We will permit additional structures to be added to the messages, but restrict these to the barest minimum to meet our security goals.

For the most part Yaksha achieves these objectives. The most significant changes we are willing to make are:

- assuming the existence of an off-line public-key Certificate Authority.
- we add certificates as additional strings to some of the messages.
- most of our changes are in the way encryption is performed. For instance, instead of a DES encryption with a user's DES key, we may encrypt using the user's RSA private key. Observe that such changes are NOT protocol changes, since the protocol does NOT specify the kind of cryptosystem to be used.

We believe that these minimal protocol changes will result in changes to the source tree being correspondingly small.

### 3.4 Upward Compatible With Smart Cards

We expect Yaksha to be deployed in environments that today do not have smart-cards, but which within five years, will have significant smart card deployments. Consequently, the design should be seamlessly upward compatible, and be able to take advantage of, smart cards. As we mentioned earlier, we see Yaksha being used with short user private keys (passwords) in the near term, and migrating to full length RSA private keys as smart cards become ubiquitous.

### 3.5 Reuse Authentication Infrastructure for Digital Signatures

Before describing this design goal, we wish to point out that the entire Yaksha design can be viewed independent of

this design requirement, and the way we meet it. The design goal is that the authentication and key-exchange infrastructure for Yaksha be reused for digital signatures. While theoretically this may not seem important, in practice it would be expensive for an organization to have to maintain two parallel sets of security infrastructures. By "reusing infrastructure" we refer specifically to three components:

- The user private secret used for authentication should also be used for digital signatures.
- Any certificate scheme used for authentication should also be used for digital signatures.
- The secure database be common

Observe that the last requirement refers to the use of digital signatures in an environment where interaction with a central server is essential. Yaksha meets these requirements, and satisfies the digital signature requirements by the addition of another two messages to the basic protocol.

We note, in passing, that one reason for our being enamored with central servers has to do with a key escrow system we have developed [GANE94b], again using the Ganesan-Yacobi scheme as a building block. In this system a central server, which does not know user private secrets, performs key exchange between two parties and upon authenticated request from authorities, reveals the session key for a particular session. Yaksha, like Kerberos, can be easily modified to perform the same function, since both generate session-keys. Unlike Kerberos, Yaksha does not have the ability to compromise a user's long term private key - a desirable property. More discussion on key-escrow is beyond the scope of this paper, but we observe that our system allows us to reuse the same infrastructure for authentication, digital signatures and key-escrow, a significant saving.

## 4.0 Yaksha

We are now ready to describe the basic Yaksha protocol. For each step of the protocol, we also reproduce the equivalent Kerberos step so that the differences are obvious. We shall explain the notation as we describe the protocol, but note now that, like in our Kerberos overview, $\{Message\}K_c$ means the Message is encrypted using a symmetric cryptosystem like DES using key $K_c$. When we say, $[Message]\hat{}X$ we mean the RSA modular exponentiation operator with the corresponding modulus $N$, i.e. $[Message]^X \bmod N$. Further, $[Cert\text{-}c]\hat{}D_{ca}$, is the public key certificate for user $c$, signed by the certificate authority, $ca$.

### 4.1 Initial key Creation

How initial keys are created in any public-key cryptosystem, is largely a question of policy. e.g. should the certifying authority also create keys? For the purposes of this paper, it suffices to present one practical scheme, and assume that the specific method is independent of the Yaksha protocol and hence can be performed in any desired fashion.

We envisage that an organization's Security Department, perhaps the same organization that issues Photo-IDs, has a terminal connected to a secure computer (e.g. a tamper proof chip). A user walks up to this terminal, enters her name, etc. This information is certified by a security officer. The computer creates an RSA public-private key pair $(E,N,D)$, prompts the user for a password, which becomes $D_c$ the user $c$'s portion of the RSA private key $D$. The computer computes $D_{cy}$ which is the Yaksha server's portion of that user's RSA key. If the computer is also the certifying authority, it computes $[c,E_c,N_c]\hat{}D_{ca}$ as that user's certificate (we are of-course greatly simplifying the complex structure [KENT93] an actual certificate would take, since those details are orthogonal to the discussion here). The computer can then transfer $D_{cy}$ and the certificate to the Yaksha server using a secure channel (for instance by encrypting with the Yaksha server's public key $(E_y,N_y)$). Finally, once smart cards are ubiquitous, the user-password may become irrelevant and the computer can download the user's (long) private key directly to his smart card.

We reiterate that no particular method of key generation is critical to the functioning of Yaksha, and the above is only meant to be one possible scenario. We do note that we expect that in Yaksha the user private key will be more long-lived than in Kerberos. Since Yaksha is not vulnerable to some of the attacks Kerberos is vulnerable to, we do not see this as a problem, and from a user perspective see it as beneficial.

At the end of this process for every user/service, there exists:

- a private portion $D_c$ known only to the user/service/entity.
- the Yaksha server has a private portion $D_{cy}$ for every user/service/entity.
- certificates exist on the Yaksha server and possibly on other services, like a name service, of the form $[c, E_c, N_c]\hat{}D_{ca}$.
- everybody knows $(E_{ca},N_{ca})$ the certifying authority's public key.

All other intermediate key generation information has been destroyed (hopefully within the safe confines of the key generation tamper proof chip).

### 4.2 Message Structure Overview

In Yaksha we desire that to authenticate itself to the Yaksha server, the client reveal knowledge of $D_c$ (which is of-course completely different from revealing $D_c$ itself) to the Yaksha server, and that the Yaksha server reveal knowledge of $D_{cy}$ to the client. When a service receives a ticket from a client it requires proof that the Yaksha server has vouched for the ticket (like in Kerberos), and further, (unlike in Kerberos) requires proof that the client has requested the ticket. i.e. it trusts neither the client nor the server individually, but trusts the message if both vouch for it. Similarly, the mutual authentication response to the client should require a message effectively vouched for by both the service and the Yaksha service.

Like in Kerberos, we do not want to store the user's private-key $D_c$ on the computer for more than the barest minimum time. Hence we use a temporary RSA private-public key pair, which is generated on the fly, and then have the client and the server collaborate to sign the public portion of this temporary key to create a temporary certificate that is valid for, say, eight hours. Note, that the authenticity of this temporary pair is verified using the long term public-key. Further, the Yaksha server never sees the private-key portion of the temporary pair.

To summarize:

- An entity's secrets are known only to the entity and no one else.
- No one party (the client or Yaksha) can spoof the server, without collaborating with the other.
- Similarly neither the server nor Yaksha can spoof the client individually.
- A user's long term private secret is not stored on any computer for any lengthy period. Instead this long term private key is used in conjunction with the corresponding Yaksha key for that user to *sign* a certificate consisting of a temporary public key.

Somewhat surprisingly, all of the above can be achieved with very minimal changes to the protocol. We do assume, that clients and servers have an easy method of retrieving certificates, perhaps from a name service. Alternately, the appropriate certificates could be attached to messages from the Yaksha server.

### 4.3 The Yaksha as_req and as_rep messages

In Kerberos the initial *as_req* message is:

Kerberos:- *as_req: c, tgs, time-exp, n*

The corresponding Yaksha message is:

Yaksha:- *as_req: c, tgs, time-exp,*
*[[TEMP-CERT]^Dc, n]^Dc*

Here *TEMP-CERT* contains *(c,* $E_{c,temp}$, $N_{c,temp}$, *expiry-time, etc.)* where $(E_{c,temp}, N_{c,temp})$ is the public portion of the temporary RSA private-public key pair which the client *c* generated, and *expiry-time*, is the interval for which it is valid. The *TEMP-CERT* is "signed" with the user's portion of his long term private key, $D_c$. This structure is then concatenated with a random string *n*, and again "encrypted" with $D_c$. This prevents an attacker who later sees *TEMP-CERT*, from seeing *[TEMP-CERT]^*$D_c$ and mounting a dictionary attack by taking guesses at $D_c$ and checking if *[TEMP-CERT]^guess = [TEMP-CERT]^*$D_c$.

Let us denote *[[TEMP-CERT]^*$D_c$, *n]^*$D_c$ by *Y*. The Yaksha server performs *[[Y]^*$D_{cy}$*]^*$E_{cy}$*]* to recover *Z=[TEMP-CERT]^*$D_c$ and *n*. The server recovers the temporary certificate by performing *[[Z]^*$D_{cy}$*]^*$E_c$*]=TEMP-CERT*. Observe that in successfully recovering a valid *TEMP-CERT*, the Yaksha server has authenticated the client. The server then completes the signature on the temporary certificate by performing *[Z]^*$D_{cy}$.

At this point in Kerberos the reply to the client is:

Kerberos:- *as_rep:*$\{K_{c,tgs}$, *time-exp, n,...*$\}K_c\{T_{c,tgs}\}K_{tgs}$

The Yaksha message is:

Yaksha:- *as_rep:*$[K_{c,tgs}$, *time-exp, n,...]^*$E_{c,temp}$,
*[T_c,tgs]^*$D_{tgsy}$, *[ [TEMP-CERT]^*$D_c$*]^*$D_{cy}$

Observe that the first two components of the Kerberos and Yaksha messages are identical, expect that the encryptions are performed using modular exponentiation and using different keys. The third component of the Yaksha *as_rep* is basically a certificate signed by the client and the Yaksha server verifying the authenticity of the temporary public-private key pair. The client can "decrypt" the first part of the message using $D_{c,temp}$ which only it knows, to recover the usual Kerberos information. The second portion is the ticket granting ticket encrypted with the Yaksha server's portion of the ticket granting service's RSA private key. Notice that the user private key $D_c$ is not needed after the *as_req*, and is never used again. Nor is the Yaksha server's

portion of this key, namely, $D_{cy}$, ever used again, thus effectively preventing any dictionary attacks against $D_c$. Yet, $D_c$ and $D_{cy}$ make their "presence felt" since they have been used to sign the temporary public key, and now the client can "sign/authenticate" messages with the corresponding temporary private key (which is a regular full size RSA key invulnerable to password guessing attacks), without danger of revealing $D_c$. Further, a message can be sent securely to the client *c* encrypted under $E_{c,temp}$, by any entity that sees the temporary certificate.

### 4.4 The Yaksha tgs_req and tgs rep Messages

In Kerberos the request to the ticket granting server takes the form:

Kerberos:- *tgs_req: s, time-exp, n,* $\{T_{c,tgs}\}K_{tgs}$, $\{ts...\}K_{c,tgs}$

The only Yaksha modifications to this Kerberos message are to (a) attach the temporary certificate to the message, and (b) to take the encrypted TGT from the Yaksha *as_rep* message, $[T_{c,tgs}]^{\wedge}D_{tgsy}$, and to sign it using the temporary private key, $D_{c,temp}$. Part (a) allows the Yaksha *tgs* server to retrieve ($E_{c,temp}$, $N_{c,temp}$) and (b) guarantees that a compromised authentication server cannot generate a valid TGT for a "fake" client. The resulting message is:

Yaksha:- *tgs_req: s, time-exp, n,*$[ [T_{c,tgs}]^{\wedge}D_{tgsy}]^{\wedge}D_{c,temp}$, $\{ts...\}K_{c,tgs}$, $[ [TEMP\text{-}CERT]^{\wedge}D_c]^{\wedge}D_{cy}$

The ticket granting server first retrieves the user's permanent certificate $[Cert\text{-}c]^{\wedge}D_{ca}$, recovers $(E_c,N_c)$, uses this to recover the *TEMP-CERT*, uses the temporary public-key in the temporary certificate to retrieve $[T_{c,tgs}]^{\wedge}D_{tgsy}$, and then uses its private key $D_{tgs}$ and public key $(E_{tgs},N_{tgs})$ to recover the ticket $T_{c,tgs}=[[[ T_{c,tgs}]^{\wedge}D_{tgsy}]^{\wedge}D_{tgs}]^{\wedge}E_{tgs}$.

At this point the ticket granting service has authenticated the user, and it is tempting to use the Kerberos *tgs_rep*:

Kerberos:- *tgs_rep:* $\{K_{c,s}$, *time-exp, n, s,...*$\}K_{c,tgs}$, $\{T_{c,s}\}K_s$

almost unchanged, for instance, simply by replacing $\{T_{c,s}\}K_s$ with $[T_{c,s}]^{\wedge}D_{sy}$. But the problem is that mutual authentication is not achieved and a compromised *as* could spoof the client into believing it is talking to the *tgs* when it is not. So we make the return message contain proof of the *tgs's* authenticity. To avoid this we simply make the *tgs* complete the signature on $[T_{c,tgs}]^{\wedge}D_{tgsy}$ message using $D_{tgs}$ and return the result to the client. so the *tgs_rep* message is:

Yaksha:- *tgs_rep:* $\{K_{c,s}$, *time-exp, n, s,...*$\}K_{c,tgs}$, $[T_{c,s}]^{\wedge}D_{sy}$, $[[T_{c,tgs}]^{\wedge}D_{tgsy}]^{\wedge}D_{tgs}$

The client retrieves (unless we choose to include it in the *tgs_rep*) the *tgs* long term certificate, uses $(E_{ca},N_{ca})$ to recover $(E_{tgs},N_{tgs})$ and verifies that $[[T_{c,tgs}]^{\wedge}D_{tgsy}]^{\wedge}D_{tgs}$ is the signature on a valid $T_{c,tgs}$. The ticket contains $K_{c,tgs}$ which the client knows (and further it has structure), so it can verify its validity. A compromised *as* cannot generate a valid $T_{c,tgs}$, signed by the *tgs*, since it does not know $D_{tgs}$.

### 4.5 The Yaksha ap_req and ap_rep Messages

The Kerberos *tgs_req* and *ap_req* messages are fundamentally identical (the former being a special type of request to a server). Similarly the Yaksha *ap_req* message is identical to the *tgs_req* message, and without further explanation we state the two messages:

Kerberos:- *ap_req:* $\{ts,ck, ...\}K_{c,s}$ $\{T_{c,s}\}K_s$

Yaksha:- $\{ts, ck,...\}K_{c,s}$, $[ [T_{c,s}]^{\wedge}D_{sy}]^{\wedge}D_{s,temp}$, $[[TEMP\text{-}CERT]^{\wedge}D_c]^{\wedge}D_{cy}$

In Yaksha we do mandate mutual authentication, and want the server to prove its knowledge of its long term private key *Ds*. As in the *tgs_rep* message this is achieved by the server replying with the service ticket $T_{c,s}$, with the signature completed, such that it can be verified using its long term public-key. Consequently the *ap_rep* messages are:

Kerberos: *ap_rep:*$\{ts\}K_{c,s}$

Yaksha: *ap_rep:* $[[T_{c,s}]^{\wedge}D_{sy}]^{\wedge}D_s$

We have changed the *tgs_rep* and *ap_rep* messages more than we may prefer, but the resultant mutual authentication, without having to trust the *as* or *tgs* respectively, justifies the change.

### 4.6 The Yaksha sign_req and sign_rep Messages

Kerberos does not perform signatures, so these two messages do not have a Kerberos counterpart. Rather these messages are essentially the signature protocol described in [GANE94] with a slight modification to remove a potential dictionary attack. The messages are:

Yaksha: *sign_req: c,* $[[H,ts]^{\wedge}D_c$ *, n*$]^{\wedge}D_{c,temp}$, $[[TEMP\text{-}CERT]^{\wedge}D_c$, *n*$]^{\wedge}D_c$

Yaksha: *sign_rep:* $[[[[H, ts]^{\wedge}D_c]^{\wedge}D_{cy}],n]^{\wedge}E_{c,temp}$

We use the same temporary public-private key pair to perform mutual authentication and encryption between the

signer and the server. If this temporary key pair is only used for this exchange, then it may be safe to assume that the temporary public-key (and the temporary certificate) is never made available to anyone but the two parties. Under this assumption, the message exchange will be simpler. However, we have chosen to be cautious, and hence our exchange is more complex. The user $c$ signs a hash, $H$, concatenated with a timestamp, $ts$, to add redundancy (in practice, a signature would have some well defined format and this would not be necessary) to the message. The client then takes this string, $[H,ts]^D_c$, and concatenates it with a random number $n$ to prevent dictionary attacks of the form $[H,ts]^guess$, and then signs again with $D_{c,temp}$. This results in $[[H,ts]^D_c,n]^D_{c,temp}$. The client sends this and the usual partially signed temporary certificate, $[[TEMP\text{-}CERT]^D_c, n]^D_c$ to the Yaksha server.

On receipt of the *sign_req*, The Yaksha server first unlocks the *TEMP-CERT* (just as in the *as_req* message), and recovers the temporary public key. The successful recovery authenticates the client to the Yaksha server. It uses this public key to recover $[H,ts]^D_c$ and $n$. The server can then recover $[H,ts]$, by performing $[[[H,ts]^D_c]^D_{cy}]^E_c$. The presence of a structured timestamp authenticates that this part of the message came from the client (we have to worry about attacks where pieces of messages may be valid, with other portions "pasted" in).

The server then computes $[[H, ts]^D_c]^D_{cy}$ which is the regular RSA signature on the hash and time stamp. It then concatenates the signature with $n$ and encrypts using the temporary public key. The client can recover the signature using the temporary private key, and then verify the authenticity of the signature using its long term public-key.

## 5.0 Conclusions

We wish to make several observations:

1. Except for the *ap_rep* message, almost all changes are restricted to either using modular exponentiation instead of DES, or requiring an additional message to be inserted. Importantly, the number of message rounds is kept identical.
2. Given that we have the power of the RSA variant at hand, several of our choices may seem curiously sub-optimal. This results from our strong desire to retain several Kerberos structures and ideas, even if they are now somewhat redundant and sub-optimal.
3. Extending Yaksha to encompass the full range of Kerberos functions as envisaged in [KOHL93], including cross-realm operations, is beyond the scope of this paper, but are a natural, if tedious, extension of the ideas contained here.
4. Given the basic ideas here, countless variations are possible. It is not claimed that the Yaksha design described here is the "best", and we suspect that as the protocol (and software) design progresses the eventual Yaksha will look slightly different.
5. There is little question that symmetric encryption is more efficient than any scheme using public-key cryptography. However, we believe that there is nothing particularly prohibitive about our designs and sensible engineering choices can alleviate any performance bottlenecks. For instance, generation of temporary public-private key pairs can be precomputed and placed in a queue-cache that is partially flushed at regular intervals. The Yaksha server itself will probably require RSA hardware, which is easily available, and the "capacity" of the server may not be high (requiring more servers). However, this may be a small price to pay when weighed against the potential havoc a catastrophic failure of a Kerberos database.

## 6.0 Acknowledgments

I would like to thank the anonymous referees for providing useful feedback. I am extremely grateful to Raymond Pyle and Rick Austin for their careful and insightful reviews. The name Yaksha is due to Meenakshi and Natesan Ganesan, with Satish Krishnamurthy pointing out that Yakshas have the power to mutate into any other creatures including three headed dogs. Finally, my special thanks to Karuna Ganesan for spending many hours improving the 'readability' of a notation heavy paper, which for my sake, she hopes someone somewhere can comprehend!

# 7.0 References

[BELL92] Bellovin, S. M. and M. Merritt, "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks", *Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy*, 1992.

[Boyd89] Boyd, C., "Digital Multisignatures", *Cryptography and Coding*, Clarendon Press, Oxford 1989, H.J. Beker and F.C. Piper, Ed.

[GANE94] Ganesan R., and Y. Yacobi, "A Secure Joint Signature and Key Exchange System", *Bellcore Technical Memorandum, TM-ARH-*, 1994.

[GANE94b] Ganesan, R. "A New Key Escrow System", *In publication process.*

[KARN89] Karn, P.R. and D.C. Feldmeier, "UNIX password security - Ten years later", *Advance in Cryptology - CRYPTO 89. G. Brassard (Ed.) Lecture Notes in Computer Science, Springer-Verlag*. 1990.

[KENT93] Kent, S., "Privacy Enhancement for Internet Electronic Mail: Part II: certificate Based Key Management", *INTERNET RFC 1422*, Feb. 1993.

[KOHL91] Kohl, J. T., "The Evolution of the Kerberos Authentication Service", *EurOpen Conference Proceedings*, May 1991.

[KOHL93] Kohl, J. T. and B.C. Neuman, "The Kerberos Network Authentication Service", *INTERNET RFC 1510*, September 1993.

[MCMO94] McMohan, P., "SESAME V2 Public Key and Authorization Extensions to Kerberos", *Proceedings of the INTERNET Society Symposium on Network and Distributed System Security*, 1994

[MORR79] Morris, R. and K. Thompson. "Password Security: A Case History", *Communications of the ACM, 22(11).* November 1979.

[NEED78] Needham, R. M., and M. D. Schroeder, "Using Encryption for Authentication in Large networks of Computers", *Communications of the ACM, v. 21,n 12,* Dec. 1978.

[NEUM94] Neuman, B. C., and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks", *IEEE Communications,* September 1994.

[RSA78] Rivest, R., A. Shamir and L. Adelman, "On Digital Signatures and Public-Key Cryptography", *Communications of the ACM, v. 27, n. 7,* July 1978.

[SCHN94] Schneier, B., *Applied Cryptography: Protocols, Algorithms and Source Code in C*, John Wiley and Sons, New York, 1994.

[TARD91] Tardo, J., and K. Alagappan, "SPX: Global Authentication Using Public-Key Certificates", *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, 1991.

[WEIN94] Weiner, M. *Personal Communication*, 1994.