OM

Yaksha: Towards Reusable Security Infrastructures

by

Ravi Ganesan

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
1996

This dissertation is dedicated to my parents Natesan and Meenakshi Ganesan, for making everything possible.

ABSTRACT

In this work we first motivate and introduce the concept of a reusable security infrastructure. Such an infrastructure will be built using a small set of proven security technology primitives and will have a single set of administrative processes, policies, databases and user keys. This single infrastructure, once implemented, will provide multiple security functions such as authentication, digital signatures, key exchange and key escrow by protocol variations. We believe that such reusable security infrastructures are the only cost effective way of implementing security on large public networks like the Internet, or within large organizations.

Next we describe the Yaksha security system which is an example of such an infrastructure. Built using an RSA variant as a building block, the system can be used for digital signatures, key exchange and key escrow. It can also be used for authentication, and several authentication protocols are feasible within the infrastructure. We choose to describe an authentication protocol which is an extension of Kerberos. Significantly, it appears that breaking the Yaksha system is equivalent to breaking RSA.

The Yaksha system achieves more than just reuse, it provides significant improvements over the state of the art. Its method of achieving digital signatures allows for short user private keys, and provides real time revocation of compromised keys. The extension of Kerberos implemented using the infrastructure removes the vulnerability to catastrophic failure and dictionary attacks inherent in the original Kerberos specification. The method of key escrow Yaksha provides does not require an authority to ever learn a user's long term private secrets and can be used for applications ranging from telephony to e-mail to file storage. Passwords are an important part of any security infrastructure, and we overview and point to some of our results on how to build strong password systems. Finally, we note that the fundamental primitives in the Yaksha infrastructure are powerful, and consequently a Yaksha infrastructure can be extended and reused in a myriad of ways.

ACKNOWLEDGEMENTS

I am very grateful to my Advisor Baruch Awerbuch for his counsel and encouragement. Thanks also to the members and alternates of my Ph.D. committee, Yair Amir, Lenore Cowen, James Fill, Leslie Hall, Samuel Lomonaco and Gerald Masson, for their participation and advice. I am also grateful and appreciative of the guidance and instruction provided to me at Johns Hopkins by Michael Goodrich, Rao Kosaraju and Gerald Masson. Thanks also to Yacov Yacobi in collaboration with whom part of this work was performed and to Dorothy Denning for her continued interest in this work.

This work was performed during my employment at Bell Atlantic and I am grateful for the sustained support. A very special thank you to my friend and mentor Raymond Pyle.

Thanks to my wife Karuna for her patience, and dedication to this project. Finally, to the rest of my family, Meenakshi Ganesan, Natesan Ganesan, Scamp Ganesan, Usha Satish, Akhila Satish and Satish Krishnamurthy for their support, and *im*patience with my progress!

CONTENTS

1. Introduction and Outline

In this thesis we first motivate and introduce the concept of a *reusable security infrastructure*. Such an infrastructure will be built using a small set of proven security technology primitives and will have a single set of administrative processes, policies, databases and user keys. This single infrastructure, once implemented, will provide multiple security functions such as authentication, digital signatures, key exchange and key escrow by protocol variations. We believe that such reusable security infrastructures are the only cost effective way of implementing security on large public networks like the Internet, or within large organizations.

Next this thesis describes the Yaksha security system which is an example of such an infrastructure. Built using an RSA [RIVE78] variant as a building block, the system can be used for digital signatures, key exchange and key escrow. It can also be used for authentication, and several authentication protocols are feasible within the infrastructure. We choose to describe an authentication protocol which is an extension of Kerberos [NEUM94]. Significantly, it appears that breaking the Yaksha system is equivalent to breaking RSA.

The Yaksha system achieves more than just reuse, it provides significant improvements over the state of the art. Its method of achieving digital signatures allows for short user private keys, and provides real time revocation of compromised keys. The extension of Kerberos implemented using the infrastructure removes the vulnerability to catastrophic failure and dictionary attacks inherent in the original Kerberos specification. The method of key escrow Yaksha provides does not require an authority to ever learn a user's long term private secrets and can be used for applications ranging from telephony to e-mail to file storage. Passwords are an important part of any security infrastructure, and we overview and point to some of our results [DAVI93, GANE94a] on how to build strong password systems. Finally, the fundamental primitives in the Yaksha infrastructure are powerful and consequently a Yaksha infrastructure can be extended and reused in a myriad of ways.

Chapter 2 of this thesis motivates the need for building reusable security infrastructures. Chapter 3 describes the basic structure of the Yaksha system. Chapters 4, 5, 6 and 7 describe how the Yaksha system can be used for digital signatures, key exchange, key escrow and authentication. Chapter 8 discusses the importance of good passwords and provides an overview of some of our results in this area. We conclude in Chapter 9.

2. Why Reusable Security Infrastructures?

In this Chapter we describe reusable security infrastructures and motivate their use.

2.1 What is a Security Infrastructure?

Curiously, the somewhat obvious concept of a *security infrastructure* is not always readily comprehended. Our intent here is not to describe every aspect of a security infrastructure, rather our goal is to give the reader an appreciation of all the piece parts that have to come together (the infrastructure) to achieve a simple security function. We choose to do this by considering a simple example of a fairly straightforward security function: authentication.

Consider a computer, e.g. a Departmental UNIX server, with multiple users, with a simple authentication system [KARN89] that ensures that only legitimate users gain access to the system. Let us take a closer look at the authentication security infrastructure. It consists of:

- *Security Algorithms:* These are the underlying security techniques themselves. For instance, in UNIX, passwords are stored after being encrypted using a one way function. When the user logs in, the system applies the same one way function to the password the user types in, and compares it to the previously stored result. We are greatly simplifying the actual details, and the actual algorithm is more complex. Developing even such a conceptually simple scheme requires careful thought and attention.
- *Security Evaluation:* The specifics of the algorithm itself have to be carefully evaluated in order to study the underlying security properties. In most security systems a mathematical proof of security is unlikely, and consequently the validation of a security algorithm is a tedious and inexact process, which often reduces to an argument of the form: "lots of bright people have tried to break the system and failed". Using a scheme which has not been carefully examined, and stood the test of time, is unwise. The UNIX authentication system for instance has been fairly exhaustively analyzed.
- *Hardware/Software:* The actual implementation of the algorithms, which in our example will likely be a piece of software. Quite apart from all the other travails of any engineering project, implementing security has the additional problem that, even if the system works exactly per normal input/output specifications, and passes all regression tests, it may still be totally flawed if it has an inadvertent 'security hole'! Very few other 'systems' have to contend with *malicious* adversaries trying to break the system. Security hardware and software have to be carefully examined for "defects" which such adversaries can use.
- *Security Administration:* Even in our simple example, there has to exist an administrator who will set up new accounts, delete old accounts, reset passwords when they are forgotten, and so on. Most large organizations probably spend most of their security budget on administration, not on the technology or systems themselves.
- *Security Policies and Processes:* In our example someone has to develop and implement various policies (e.g. how does the user authenticate herself to the

administrator to reset her password?) and processes (e.g. who is the appropriate authority to authorize a new account?, what are the rules for how tape back ups of the password files are stored?).

- *Security Audit:* Once security policies are established, we need auditors to ensure the rules are followed. In addition to human auditors, a well run system will have audit software that will perform routine checks on the system on a regular basis. For instance, on a UNIX system, a conscientious administrator will use tools like COPS and SATAN on a regular basis. Further, the authentication sub-system will generate log files which have to be audited on a regular basis.
- *Key Generation Process:* In our example, how did the user get their password initially? In a small set up the administrator could walk up to the user and tell them their initial password. What if the system had millions of users scattered across the globe? Generating and distributing initial keys in a secure fashion is a difficult process.
- *Password (key) Management:* How do we ensure that users pick 'good' passwords? How often should the passwords be "aged". Most attacks on the authentication system arise from poorly chosen user passwords; significant energy must be expended to ensure that this does not happen.
- *Users:* And at the end of it all is a user, who probably resents having to remember yet another password, who sees the security system and its processes as getting between him and his work, and who has low tolerance for the hindrance the authentication system imposes!

It is hopefully clear to the reader that even a simple security function requires a complex and costly *infrastructure*. When we consider more complex systems or more complex security functions the infrastructures can quickly get very complex and expensive.

2.2 Two Observations on Security Infrastructures

We have presented an intuitive argument that a security infrastructure can be extremely complex. Let us make two related observations:

1. *Security infrastructures are costly!* Just the cost of administering (ignoring acquisition and implementation costs) an authentication infrastructure for a large organization can run into several millions of dollars a year. Or consider a small company that wants to set up a cyber storefront on the Internet; the costs of establishing and maintaining Logon-Ids and Passwords for each of its customers can be a significant overhead.
2. *Infrastructure problems are a leading cause of security incidents; not weaknesses in the underlying technology*. We in the security research community spend much time and effort developing complex and esoteric attacks in the fundamental security technologies, and pay great attention to the next advance in cryptanalysis. Yet, it is well established that a large percentage of attacks are decidedly low-tech. More attacks result from sloppy administration and poorly chosen user passwords than any other cause. The notion of a high tech attacker digging a hole in a field in Kansas, pulling out a piece of fiber, eavesdropping using a protocol analyzer and then using

supercomputers to mount an esoteric cryptanalytic attack on an encrypted message stream is entertaining, but not (yet) relevant to the real[1] world. It is far more likely that the attacker will call up a user, pretend to be a system/security administrator and *ask* the user for a password! *i.e. the security infrastructure process is more likely to break before the technology does.*

The cost and security characteristics of infrastructures are closely related. In a typical cycle, an organization will rush computing and communications functionality to its staff, dollars for building the security infrastructure will be scarce, the resulting security infrastructure will be inadequate and low-tech security problems will result. As a community this is exactly what is happening with the Internet, where the security infrastructures are struggling to catch up with the functionality. If the choice for a group of users is between a higher speed connection to the Internet and a better firewall, guess which is likely to get the funds? If nuclear power plants were built with the safety attitude with which we are building our information infrastructure, then there would be a Chernobyl every week!

2.3 Where are Security Infrastructures Heading?

For many reasons, including a lack of awareness of the practical concerns discussed above, as a community we are proposing that we build not one, not two, but several security infrastructures. A few for authentication, a completely separate one for digital signatures (typically built around RSA in the commercial world), and perhaps a few more for key exchange and key escrow. Further, for the most part the focus remains on the *security technology*, and its feature functionality, with little or no attention paid to the *security infrastructure* that has to surround the technology.

There are some positive signs. The emergence of Kerberos as a widespread standard (soon every major operating system from MVS on IBM mainframes to NT on PCs will support Kerberos, not to mention UNIX on which Kerberos was initially developed) spells tremendous relief to organizations which currently have to deal with several distinct security infrastructures for authentication alone. The growing realization that security is on the critical path for the success of electronic commerce on the Internet has meant that serious attention (and dollars), beyond the obligatory lip service, is being devoted to the issue.

However, even if the number of authentication infrastructures is reduced to one Kerberos infrastructure, there still remains the problem of having to build completely distinct infrastructures for signatures and escrow.

2.4 Why Reusable Security Infrastructures?

If we cannot afford one high quality security infrastructure, how do we expect to find the resources to build several for different security functions? If users dislike interfacing with the few infrastructures they have to deal with today, do we seriously expect them to learn

[1] We believe this extends to even information warfare waged by national governments. It is true that national governments do spend a lot of money on 'high tech' attacks, but their vulnerability to low-tech attacks caused by low-tech problems remains high.

to use several different infrastructures? How many smart cards will we carry in our wallets?

We believe that multiple infrastructures are utterly impractical. Rather, we see the emergence of what we call *reusable security infrastructures*, where the same infrastructure provides multiple security functions. We believe such infrastructures will have the following characteristics:

- They will be built around a core set of security technology building blocks whose security has withstood the test of time, and whose specifications are completely public. For instance, DES [FIPS77], Kerberos [NEUM94] and RSA [RIVE78].
- From the user's perspective, she will have to remember a single password, or carry a single smart card with a single set of keys.
- There will be one administrative unit, one set of administrative processes and one process for key generation.
- Different security functions such as authentication, digital signatures, key exchange and key escrow, will be provided by simple protocol variations within the same infrastructure.

In the rest of this thesis we will describe the Yaksha security system, a reusable security infrastructure which has the above characteristics.

2.5 How Will Reusable Security Infrastructures Evolve?

We believe that within organizations, dollars will be spent on security vendors who can promise reuse. The savings in administration costs alone are too seductive. More interesting perhaps is our *speculation* on what will happen on the Internet. Imagine the plight of a small (or even large) business that wants to set up shop on the Internet. They will have to spend many of their dollars worrying about how to authenticate users, ensure non-repudiation of transactions and, eventually, have to worry about how to implement a key-escrow mechanism to stay within the law. All of this of course is a complete nuisance to the business which would like to stay focused on its core competency related to selling whatever it sells. Consumers also will quickly tire of having to remember different passwords for accessing different storefronts.

Enter the *full service security provider*. This provider will set up his own storefront on the Internet, but he will have only one product: security! He will build a reusable infrastructure and will register users and businesses as customers. This security provider will then act as an intermediary in any transaction, user to business, business to business, or user to user. Neither the business, nor the user, will have to worry about security; that is the provider's business! This business model exists today, very few people or businesses build their own *physical* security infrastructure, rather they rely on companies whose business is to supply the physical security infrastructure from alarm systems to security personnel. There is little reason why the same model will not eventually be established on the Internet.

Finally, we do not believe that reusable security infrastructures will be specified by standards bodies and then developed. We would rather bet on Darwin. We hope to see the emergence of a few competing security providers, each rallied around a different reusable security infrastructure. Eventually, a few, perhaps two, will survive!

The Yaksha security system, which the rest of this thesis describes, is the first proposal we know of for such a reusable security infrastructure.

3. The Yaksha Fundamentals

This chapter describes relevant background work, and then describes the Yaksha infrastructure.

3.1 Background

In this section we very briefly introduce the concept of secret and public key cryptography, describe the RSA system, the concept of certificates and Boyd's variant of RSA.

3.1.1 Secret-Key (Symmetric) Cryptosystems

In conventional cryptography, the participants, say Alice and Bob, share a common secret key, k which is known only to them. To encrypt a message Alice would typically use some encryption function E and a session key k to compute a ciphertext $C = E(M, k)$. She would then send C to Bob on a public channel. Bob can decrypt the message using a decryption function D, i.e. $M = D(C, k)$. Typically, the details of E and D are public, but an attacker Eve, who only sees C, cannot recover M since she does not know k.

How do Alice and Bob agree on the shared secret k? This is known as *the key distribution problem*. Later in this thesis we will describe trusted third party systems which solve the key distribution problem using symmetric cryptosystems only, but first we introduce the notion of public key cryptography.

3.1.2 Public-Key (Asymmetric) Cryptosystems

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. Let us denote the public key encryption function by $\overline{E}$ and the decryption function by $\overline{D}$. 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 $C = \overline{E}(M, U_i)$ and sends C to i. i can recover M using his private-key, P_i, by computing $M = \overline{D}(C, P_i)$. 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. Consequently, public and secret key cryptosystems are usually used in conjunction. Alice would typically use some symmetric encryption function E and a session key k to compute a ciphertext $C = E(M, k)$. To send the message to Bob, she would further encrypt the session key k using Bob's public key and the public key encryption function, $K = \overline{E}(k, U_B)$. She would then send Bob (C, K). Bob would first recover k using his private key, i.e. $k = \overline{D}(K, P_B)$, and can then decrypt the message using the symmetric decryption function D, i.e. $M = D(C, k)$. This solves the key distribution problem, without sacrificing the efficiencies of symmetric cryptosystems for large messages.

Public-key cryptosystems can also be used for achieving non-repudiation, i.e. a method of 'signing' a message such that the signer cannot later repudiate the message. The signer, i, computes $S = \overline{E}(M, P_i)$ and sends (M, S) to the recipient. The recipient "looks-up" *i*'s public-key, U_i, and then checks to see if $\overline{D}(S, U_i)$ is equal to *M.* If so then the recipient is convinced that *i* signed the message, since computing an S, such that $\overline{D}(S, U_i) = M$, requires knowledge of *i*'s private key which only i knows. The recipient can retain the signature as proof to prevent i from later repudiating the message. Unless the message is small, typically a one way hash function H is used to hash the message, and it is typically the hash, $h = H(M)$ that is signed, not the message itself. Further, by combining the encryption and signature functions, both privacy and non-repudiation can be achieved.

We now review an example of a public-key cryptosystem.

3.1.3 Review of the RSA Cryptosystem

RSA [RIVE78] is a public-key based cryptosystem that is believed to be very difficult to break. In the RSA system the pair of numbers (e_i, n_i), is user *i*'s public-key and the number d_i is the user's private key. Here $n_i = p \times q$, where p and q are large primes, and $e_i \times d_i = 1 \bmod \phi(n_i)$, 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. After key generation, p, q and $\phi(n_i)$ are destroyed. To encrypt a message being sent to user *i*, user j will compute $C = M^{e_i} \bmod n_i$ and send C to user i. User i can then perform $M = C^{d_i} \bmod n_i$ to recover M.

The RSA based signature of user i on a message, M, is $S = M^{d_i} \bmod n_i$. The recipient of the message, can perform $M = S^{e_i} \bmod n_i$, to verify the signature of *i* on M. Note that in RSA encryption and signatures can be combined.

RSA is not mathematically proven to be secure, but, "lots of bright people have tried to break the system and failed". The most efficient known attacks on RSA rely on factoring n_i. Since typically n_i (and d_i) are chosen to be very large numbers (hundred to two hundred decimal digits long), and there are no known efficient (polynomial time[2]) algorithms for factoring, RSA is widely believed to be secure.

[2] Polynomial in the number of bits used to represent n_i.

Observe that since d_i is a very large number, users cannot be expected to remember their own private key. Once smart cards, and smart card readers, become ubiquitous, this problem is minimized, until then however the requirement of long private keys is an issue. As we shall see later, the Yaksha system solves this problem.

3.1.4 Review of Public Key Certificates

In the previous sections, we have repeatedly referred to how a user can "look up" another user's public-key. An obvious solution would be to put everyone's public-key in some sort of universally accessible database. This raises the problem of how to secure the connection between the user and the database. The concept of certificates [KENT93] are an elegant way of solving the problem.

A certificate is basically a binding between an entity and its public key, as vouched for by some authority. So a certificate in an RSA based infrastructure could contain $Cert = \{i, e_i, n_i\}$. The certificate is signed by a trusted third party called a Certificate Authority (*CA*). So when *i* sends *j* a signed message $S = M^{d_i} \bmod n_i$, it is accompanied by $(Cert_i)^{d_{CA}} \bmod n_{CA}$. *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 any infrastructure, some keys will be compromised from time to time, and it then becomes necessary to *revoke* a certificate that has been issued. One suggestion for solving this is to use the concept of a Certificate Revocation List (CRL). Roughly speaking, ever so often (say once a day), a CA will broadcast a list of certificates that have been revoked, and every user will maintain a CRL containing all such revoked certificates. We find this concept utterly impractical. First, if a user's private key is compromised, the first few hours after compromise are likely to be the most significant. A CRL that is updated once a day is too slow, updating CRLs more often is too costly. Second, the idea of every user maintaining a list of every certificate that has been revoked simply does not scale to any practical implementation.

As we shall see later, the Yaksha system solves this problem by providing instant revocation without requiring any additional work on the end-user's part.

3.1.5 Review of Boyd's RSA Variation

Colin 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 $d_1 \times d_2 .... d_k = d \bmod \phi(n)$. 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 $S_1 = M^{d_1} \bmod n$, and the second user completes the signature by computing $S = S_1^{d_2} \bmod n$. The resulting signature is identical to one signed by the regular RSA private key (i.e. $S = M^d \bmod n$) and can hence be verified, in one operation, using the regular public-key.

3.1.6 Review of Ganesan-Yacobi Results

In joint work with Yacov Yacobi [GANE94b] we reinvented Boyd's system, and made four significant additional contributions. All our 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. We proved 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. We discuss these results in greater detail later in the chapter.
2. We described the following key exchange protocol. All arithmetic is modular the appropriate modulus; we omit showing this for brevity of notation. User 1 sends x^{d_1} to User 2. User 2 recovers $x = ((x^{d_1})^{d_2})^e$. 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. $x \otimes y$). We proved mathematically that breaking this key exchange protocol is equivalent to breaking RSA. The attacker can be an active or passive eavesdropper. For reasons that will become clearer later, we will *not* use this key exchange system as the primary key exchange mechanism in the Yaksha system.
3. Next, we introduced 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 we proved, cannot impersonate the user). This concept is central to the Yaksha infrastructure.
4. We also 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, breaking 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. We conjectured, but did 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, then the user password should be 80 bits long. Again, this is an attack mounted by a malicious server, not an eavesdropper. This result means that in the Yaksha system, if the user does not have smart cards, they can be given short memorizable private keys.

In 1. and 2. above we observed that although the goals are signatures and key-exchange respectively, authentication is a natural by-product. We suggested that this system is a simpler alternative to Bellovin and Merrit's [BELL92] Encrypted Key Exchange (the RSA version) and unlike EKE, does not require the two parties to share a common secret key. In this work however we describe a different authentication system, one based on Kerberos.

3.2 The Yaksha Infrastructure

This section overviews the details of the infrastructure which will be reused to provide different security functions. In all subsequent discussion we assume that the Yaksha infrastructure is deployed as described here.

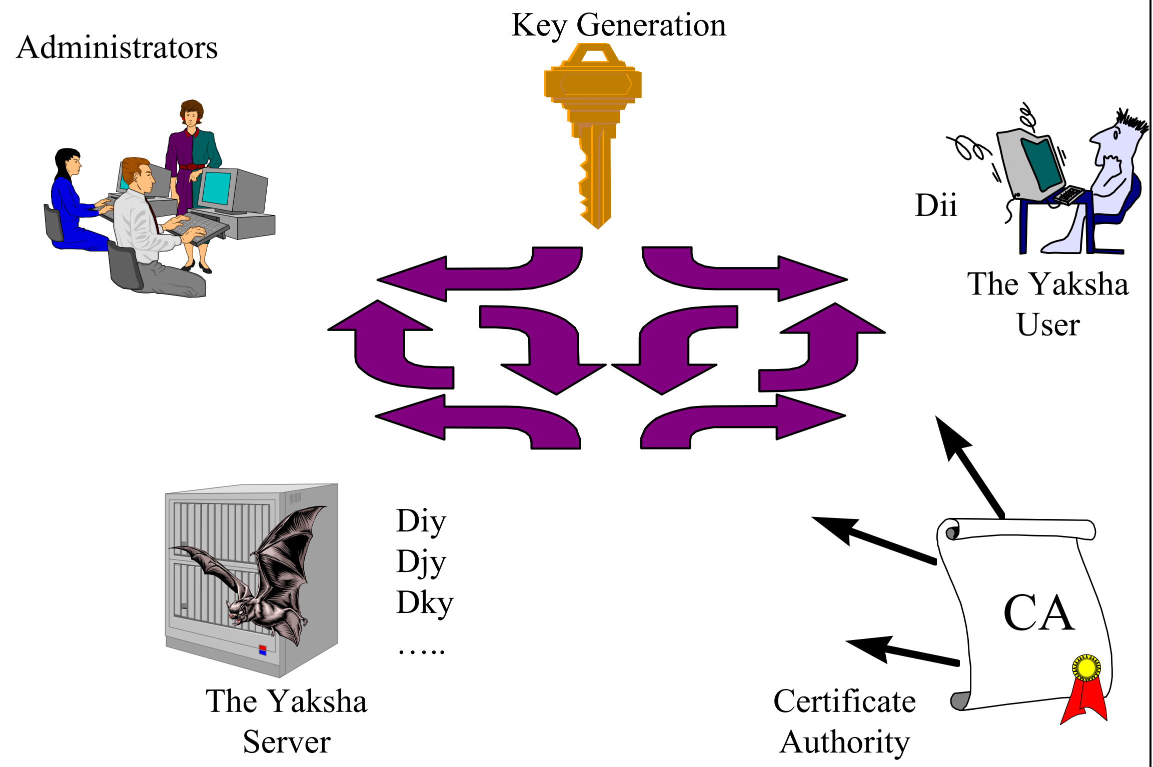
3.2.1 Overview

The Yaksha infrastructure consists of the following five components:

- *Key Generation Service:* A single key generation service which generates and distributes keys, exists in the system.
- *Yaksha Server:* The heart of the system is a Yaksha server which maintains one portion of the private key, d_{iy}, for every entity i in the system. This key is never revealed outside the server.
- *Certificate Authority:* There is a corresponding public-key for every user or entity, which is publicly available. A certificate binding the user's identity to the public key is publicly available. This CA is identical to those proposed for use with regular RSA.
- *Administrative Unit:* There is a single administrative unit which does whatever administration is required.
- *Yaksha User:* Each user or entity in the system has his own private key, d_{ii}. This key is never revealed to any other entity. Each user has software (or hardware in a smart card) to carry out their portion of the Yaksha functionality.

These five components are all there is to the Yaksha infrastructure. As we shall see, authentication, signatures, key exchange and key escrow, can all be performed without any additional components. We now discuss each of the components in greater detail.

Figure 3.1: The Yaksha Security Infrastructure. These five components comprising the infrastructure need to be implemented once, and can then be reused repeatedly to achieve different security functions.



3.2.2 Key Generation System

Algorithmically key generation is straightforward. First, the regular RSA parameters, $p_i, q_i, n_i, \phi(n_i), e_i, d_i$ are generated. Now it is necessary to generate two values d_{ii} and d_{iy} such that:

$$d_{ii} \times d_{iy} = d_i \bmod \phi(n_i) \qquad (3.1)$$

d_{iy} which will be stored on the Yaksha server is always a regular length RSA private key, say 768 bits. d_{ii} can also be long, say 768 bits, if the user has a smart card. In this case the system can generate both parameters. If the user has to memorize the password, and if we want to let the user choose this password, then we can let the user pick, say, a 10 or 11 character password (about 80 bits), which becomes d_{ii}. Another option is to generate a short key (password) for the user. In all cases, after picking one of d_{ii} or d_{iy} we solve for the other. Of-course, any equation of the form:

$$ax = b \bmod n \qquad (3.2)$$

is solvable for x only if $\gcd(a,n)$ divides b, and a few retries may be necessary (in the case where a user picks their password, it may be more user-friendly to re-generate the RSA keys itself, instead of having the user retry a different password).

After key generation, d_{ii} is given to the user i, d_{iy} to the Yaksha server, (e_i, n_i) to the Certificate Authority who generates the associated certificate, and makes it available on the system.

While algorithmically key generation in the Yaksha system is fairly similar to regular RSA key generation, there is a difference in the process. In regular RSA, users can generate keys themselves, and then register the public key with the Certificate Authority. In Yaksha, we want the user to only know d_{ii}, and the user should never see d_{iy}. Similarly the Yaksha server should only be aware of d_{iy}. Consequently we assume the existence of a separate key generation service. In a large organization, this could be a computer located at a physically secure site to which a user must go, prove identity by other means, and then get his d_{ii}. Similar to how the user would get a photo ID (or for that matter a driver's license). The computer can transmit d_{iy} to the Yaksha server encrypted under the Yaksha server's public-key, and signed using the key generation server's private key. We do not believe that this additional requirement in Yaksha is particularly onerous; every system has this 'bootstrap' problem. For instance, in regular RSA, even if the user picks their own private key, the process by which they register their public-key with the CA (at which point they have to somehow prove their identity to the CA) has the exact same issues.

3.2.3 The Yaksha Server

The Yaksha server is the heart of the system. To sign, obtain credentials, perform key exchange or key escrow, users will constantly be interacting with this server. Further, this server has d_{iy} for every entity i in the system. Consequently, the security and performance of the Yaksha server are important. Let us address these two issues:

- Security: There is little question that the Yaksha server(s) must be maintained in a highly secure fashion. However, failure of this server is not necessarily catastrophic as in the case of a Kerberos server. In Kerberos, the server knows the secret key of every user; in Yaksha, the server does not know the user's keys. Further, we would expect that all the keys the Yaksha server retains would in turn be encrypted with some sort of master key, which is maintained only inside a tamper resistant chip. If all the security functions requiring knowledge of a d_{iy} are also performed solely inside this chip, then the system's security is greatly increased[3]. How feasible is this use of tamper resistant chips? As an indication of the practicality of tamper resistant chips, consider that the recent Clipper proposal by the Federal Government calls for

[3] Note that a similar strategy will not protect a Kerberos server. In that case an attacker who has compromised the server, may be able to forge credentials. In Yaksha any action requires the user and the Yaksha server to cooperate. Neither can forge credentials by themselves.

similar tamper resistant chips in *every* telephone handset. All we require is that the Yaksha servers have such chips.

- Performance: The presence of an on-line server automatically raises questions of communications costs, and server performance. Consider communications first. It used to be the case that communications were very expensive, and that shaving a few messages off a protocol was significant. That picture has changed completely. In today's world, communications are exceedingly cheap, and in networks being built to handle video traffic, a few bits of security data are hardly likely to pose bandwidth problems. As practical proof consider how credit card networks work. Financial companies find it perfectly economical to verify credit information for purchases of even a few dollars, and even if the communications required are trans-continental in nature (or for that matter require communications halfway across the globe). Consider the second issue of scalability of the back-end computing. Clearly the Yaksha servers have to be carefully engineered with load balancing and redundancy in mind. However, there is already a precedent for the large scale use of computers attached to telecommunications networks. Most of the "intelligent features" built into today's telecommunications networks have their logic implemented on computers attached to the network (as opposed to the switches themselves). In the very near future, every phone number will become 'portable', and consumers will be able to retain a number even if they move. This requires a mapping between a logical and physical telephone network, and a look up every time a number is dialed. This information will also be maintained on computers. The point we are trying to make here is that, unlike in the past, on-line servers which perform very heavy duty transaction processing, are very practical, and while they require careful engineering, they should not automatically suggest insurmountable scalability issues.

The Yaksha server as we shall see in the course of this thesis, will interact with the user to perform numerous security functions. In all cases the Yaksha server is only required to know d_{iy}, for each entity i, and the different functions are achieved by simple protocol variations.

3.2.4 The Yaksha Certificate Authority

Perhaps the single most important point of note about the CA we use in Yaksha it is that is identical to what has been proposed (and implemented) in the Internet community. We have chosen to reuse all this good work [KENT93, CCIT88], and also ensure that the Yaksha system in general fits into and complements the existing infrastructure. This is by design; a recurrent theme in our work is to reuse existing standards and ensure interoperability. For example, we could have done away with some of the CA functionality given the semantics of the Yaksha system or designed a more efficient authentication system than one based on Kerberos. In both cases we intentionally chose the route which resulted in a somewhat redundant system, but which ensures interoperability with existing standards.

A lot can, and has been said about how CAs should be structured and how their policies should work (see the references in [SCHN94] for more pointers) . A discussion of this is beyond the scope of this work, and we simply reiterate that while the Yaksha system requires a CA, the details of the CA are orthogonal to our system.

3.2.5 The Yaksha Administrative Unit

The system as a whole has to be administered. We already discussed key generation, but the Yaksha servers themselves will require maintenance, hardware and software will have to be upgraded, policies will have to be set and maintained, and audits will have to be conducted. In all these respects the Yaksha system is much like any other security system. The main differentiation lies not in how the administration works, but in that there will be only one administrative unit for the one infrastructure, yet multiple security functions will be achieved. This should be contrasted to the alternative of having a set of administrative units for each security function.

3.2.6 The Yaksha User

In the Yaksha system every user (or entity) retains their Yaksha private key d_{ii}. In the version of the system where users do not have smart cards, d_{ii} may be a relatively short memorizable password of say 80-100 bits (10-15 characters). In this case the user remembers this password. Alternately, if smart cards and smart card readers are ubiquitous (as we believe they eventually will be), then each user has a smart card with their Yaksha key embedded in a (presumably) tamper resistant chip. Further, in some cases the use of a short key may make sense if the user's device has very strict power consumption requirements, and we wish to make computation faster and more efficient.

If the user memorizes a short passwords and enters it on a keyboard, the user portions of the Yaksha algorithms will be implemented in software and run locally at the client's computing device. The user will be prompted to enter their password at appropriate points. If smart cards are being used then most of the functionality is likely to be implemented within the smart card in hardware. In all these respects using the Yaksha system is not very different from using other security systems.

As we shall see, users also have an additional advantage using Yaksha; due to the instant revocation facility, they are more secure against the eventuality of their private key being stolen from them. No algorithm can prevent a password or smart card being forcibly stolen. In Yaksha however, the instant a theft is reported, all further usage is blocked. Simply stated, this is because to perform any function in Yaksha requires the cooperation of the user and the Yaksha server. If the Yaksha server is informed that a key is no longer valid, then all further functionality using that key is immediately shut down. This is similar to how credit cards work today.

Finally, the use of short keys by the user if smart cards are not available is more than just a minor advantage. Today very few large organizations are in a position to equip their users with smart cards and all their computing devices with smart card readers. Eventually this will change, but in the interim, a system that can work with user memorizable passwords, and eventually smoothly upgrade to smart cards, is very attractive. A similar situation exists for the average Internet user; it is very unlikely that a security system that requires the user to spend a few hundred dollars on smart cards and smart card readers will prove as attractive as a system that requires the user to occasionally enter a password.

3.3 How Secure is the Yaksha System?

This section discusses what it means for Yaksha to be secure, and then illustrates why breaking Yaksha appears to be equivalent to breaking RSA.

3.3.1 Introduction

In [GANE94b] we provide certain formal arguments for the security of the underlying primitives used in Yaksha. In this dissertation we will present more intuitive, but hopefully more readily comprehensible, arguments for the security of Yaksha. A reader interested in the more formal details is referred to [GANE94b].

What does it mean for Yaksha to be secure? There are at least three levels in which the question can be answered:

- The security of the underlying primitives.
- The security of the protocols themselves.
- The security of the actual implementations.

The first aspect is key; without secure primitives, we have no chance. The security of the protocols relies heavily on the security of the primitives, *but*, it is readily possible to develop an insecure protocol built on secure primitives. This is one of the major reasons why we constantly reuse existing protocols, like Kerberos, which even if lacking "proofs" of security, have at least met our test of "lots of bright people have tried to break the system and failed". The security of the implementation itself is clearly beyond the scope of this thesis, and we are compelled to reiterate, that in our experience[4], this is more likely to be the Achilles Heel of the system, than failures in the protocols itself.

Our approach here will be to discuss the security of the primitives themselves, and show the reader why we believe that breaking Yaksha is synonymous with breaking RSA. Our protocols are believed to be secure because their security is based on (a) the security of the primitives and (b) the reuse of an underlying protocol which is widely believed to be secure.

There are two cases to consider; Yaksha when used with regular length RSA keys, and when used with "short" keys. There are also, at least, two types of attacks to consider. Those mounted by an eavesdropper, and those mounted by one of the parties. Let us consider all these eventualities, in turn.

[4] In addition to having been a practitioner of security in the "real world", the "experience" referred to here includes being ultimately accountable for all aspects of the security of one of the world's largest telecommunications corporations. Historically, telecommunications corporations have been the target of a disproportionate share of attacks. Consequently, the "experience" has been obtained by the harsh lessons of real life.

3.3.2 Attacking Yaksha When Used with Regular Length RSA Keys

In this situation, Alice, and the Yaksha server, each have regular length (say 768 bits[5]) keys, d_{aa} and d_{ay} respectively. The public key (e_a, n_a) is known to all parties, including the attacker. The typical situation is that the user sends the Yaksha server M, encrypted using her key, i.e. $C1 = M^{d_{aa}} \bmod n_a$, (the analogous situation of the Yaksha server sending the user a similar message, is in this case symmetrical). The Yaksha server applies d_{ay} to compute the signature $S = C1^{d_{ay}} \bmod n_a$, and can apply the public key to recover $M = S^{e_a} \bmod n_a$. Let us assume that the user, the attacker and the Yaksha server have seen a history of a large (polynomially bounded) number of triplets $(M_1, C1_1, S_1), (M_2, C1_2, S_2)$....... The questions to ask are, having seen many of these triplets:

- Can the attacker recover M_i for an arbitrary message $C1_i$?
- Can the attacker forge $M_{forge}^{d_{aa}}$ or $M_{forge}^{d_{aa} \times d_{ay}}$ on a message of his choosing?
- Can the Yaksha server launch similar attacks, or determine d_{aa} by conducting computations on these messages?

Answering these questions turns out to be straightforward, namely if the attacker (or the Yaksha server) can break this system, then they can also break RSA. Why do we say this? We refer the reader to our work in [GANE94b] for formal arguments, but there is a more intuitive explanation. Yaksha has three keys, d_{aa}, d_{ay}, e_a related by:

$$d_{aa} \times d_{ay} \times e_a = 1 \bmod \phi(n_a) \quad \text{...(3.3)}$$

We can also represent this in two other forms, namely:

$$x \times y = 1 \bmod \phi(n_a) \quad \text{...(3.4)}$$

where $x = d_{aa}$ and $y = d_{ay} \times e_a \bmod \phi(n_a)$, or as,

$$x' \times y' = 1 \bmod \phi(n_a) \quad \text{...(3.5)}$$

where $x' = d_{ay}$ $y' = d_{aa} \times e_a \bmod \phi(n_a)$.

The main argument for the security of Yaksha, is that the attacker, the user and the Yaksha server, each in effect see a regular RSA system with keys (x, y, n_a) or (x', y', n_a). They know one of the keys, and do not know the other[6]. This is exactly analogous to an RSA

[5] The appropriate number of bits changes with time. Importantly, it is still the case, that each increase in the number of bits, from an efficiency perspective, is to the disadvantage of the attacker. i.e. for an extra bit of key length, the extra work required of users is less than the extra work required by attackers.

[6] To be more precise, the attacker does not even know one of the keys, she simply knows a factor of one of the keys.

system where the public key is known, and the private key is not. When all keys are regular length RSA keys, an algorithm to break Yaksha also results in an algorithm to break RSA! Since the latter is believed to be infeasible, the same applies to the former. See [GANE94b] for a more formal statement of these statements.

3.3.3 Attacking Yaksha When Used with Short User Keys

When the user's key d_{aa} is short, the situation is slightly different. While identical reasoning to the above can be used to argue that the attacker has no added advantage, the similar argument does not hold for the Yaksha server who (unlike the attacker) who knows d_{ay}. We present the formal statements in [GANE94b], and speculate that this additional knowledge provides no useful information to the Yaksha server, but we are unable to prove this statement. The best attack on our system under these circumstances that we know of is a "meet in the middle attack" described to us by Michael Weiner [WEIN94], but this is a $O(\sqrt{n})$ attack, i.e. if our key d_{aa} has 80 bits, then an attacker can mount an attack requiring an exhaustive search of 2^{40} keys. Given that this attack is *only feasible by the Yaksha server itself* (which we hope to guard carefully using tamper proof chips, as described earlier), this should not be an issue in most practical situations.

For these reasons we are comfortable recommending that Yaksha be used with short user keys (80 to 100 bits), until such time that smart cards, and smart card readers, become ubiquitous.

3.3.4 Observations: Madras Attack

This analysis of the Yaksha system, leads us to propose a new way of attacking RSA, which to the best of our knowledge [KALI96] has not been documented elsewhere. We call this the Madras Attack. Namely, given (e,n,d), instead of attempting to find d, directly, let us try and find (d_a,d_b) where $d_a \times d_b = d \bmod \phi(n)$. We can do this by "guessing" a value for d_a, computing $M^{d_a} \bmod n_a$, (where we choose M), and then searching for a d_b such that $(M^{d_a} \bmod n_a)^{e \times d_b} = M$. In fact, after guessing a d_a, we can launch a meet in the middle attack for d_b which[7] as discussed before can work in $O(\sqrt{n})$ steps. So if it happens that we chance upon a d_b of length say 80 to 120 bits, then we are successful. Observe that this attack does not appear to realize in a method of factoring. However, it can be very easily parallelized, and consequently may have some potential.

[7] As observed in our explanation of Yaksha key generation, a d_b will only exist if $\gcd(d_a, \phi(n_a)) = 1$.

4. Using Yaksha for Digital Signatures

The first function we describe is digital signatures, since it is the simplest function to perform within the framework of the Yaksha infrastructure, and is also reused for subsequent functions. The protocol described here was first presented in joint work with Yacov Yacobi [GANE94b].

To sign message M the user i first computes:

$$S1 = M^{d_{ii}} \bmod n_i \quad \text{……………………………………….(4.1)}$$

and sends $(i, M, S1)$ to the Yaksha server.

The Yaksha server first computes:

$$S = S1^{d_{iy}} \bmod n_i \quad \text{……………………………………….(4.2)}$$

and then computes:

$$\overline{M} = S^{e_i} \bmod n_i \quad \text{……………………………………….(4.3)}$$

If $\overline{M} = M$ then the Yaksha server knows that the message was sent by user i (we are using the signature function to authenticate i to the Yaksha server). For this to work the only requirement is that M have some structure (like English text, a standard syntax or have a time stamp, etc.). At this point the Yaksha server returns (S) to the user i .

Naturally, in a practical implementation, messages would be hashed, it is highly likely that the messages and signatures will be encrypted for privacy and so on. Further, the messages will follow syntax standards like the PKCS standards, and the messages may well contain time stamps etc.

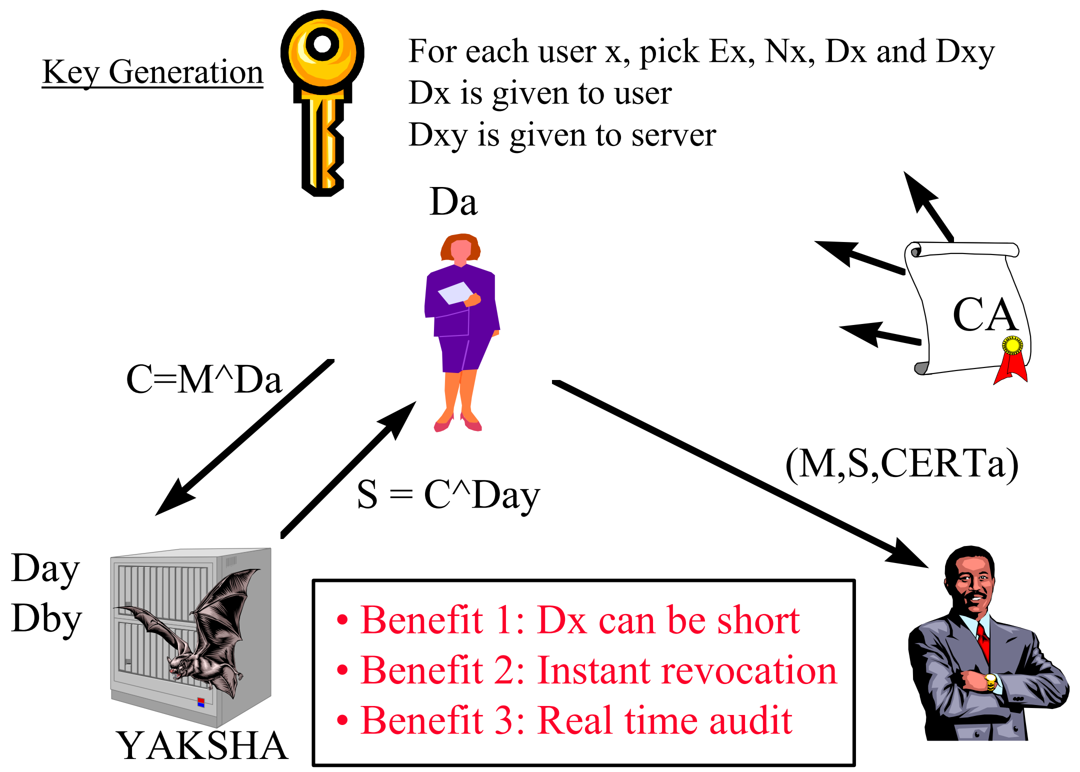
The key points to note about the Yaksha signature system are:

- At completion of signature, the value S obtained is identical in every single respect to the S that would have been obtained using regular RSA. The signing process itself involves interaction with the Yaksha server, but at the end of that, we have a regular RSA signature which is perfectly compatible with any RSA based application.
- It requires both parties to cooperate to complete the signature. Neither the user nor the Yaksha server can create a valid RSA signature on their own. Consequently, if the user's private key is compromised, then the moment the Yaksha server is notified, that key is revoked (instant revocation), and no further signatures using that key can be generated. This is a huge improvement over a cumbersome CRL based approach to revocation.

- Every signature is flowing through a central point; the Yaksha server. This provides a good point for audit trails which may be useful for functions like fraud analysis. In the off-line systems envisaged by some, if a smart card is stolen unbeknownst to the user, then the loss will not be reported, and the smart card may be used over and over again. In Yaksha, the central audit point allows for pattern analysis to detect unusual usage patterns. This technique is the single most effective way for recovering from calling card and credit card fraud today, and the Yaksha system supports this functionality completely.
- The user's key can be a short password. As discussed earlier, this is a huge benefit in an era where smart cards and smart card readers are far from ubiquitous.

It can also be noted that in terms of software or hardware implementation, both the client and server essentially implement straightforward RSA functionality and are hence not complex.

Figure 4.1: Using Yaksha for Digital Signatures. Signing requires interaction with the Yaksha server. Neither the user nor the server can forge signatures on their own. Once created, a signature is identical in all respects to a regular RSA signature.


Key Generation
For each user x, pick Ex, Nx, Dx and Dxy
Dx is given to user
Dxy is given to server
Da
CA
C=M^Da
S = C^Day
(M,S,CERTa)
Day
Dby
YAKSHA
• Benefit 1: Dx can be short
• Benefit 2: Instant revocation
• Benefit 3: Real time audit

Observe that this simple mechanism also lends itself to several possible authentication protocols. Alice can interact with the Yaksha server to generate credentials bearing her signature. Since she cannot generate her signature without the cooperation of the Yaksha server, the recipient of the credentials will know that Yaksha is effectively vouching for her. As it turns out, a more complex scheme based on the Kerberos system has stronger security (dealing with issues like replay attacks), and we will describe a complete Yaksha based authentication system later in this work.

5. Using Yaksha for Key Exchange

As described in Chapter 3, most public key systems are used in conjunction with conventional cryptosystems. The public key systems are used to exchange a *session key* which is then used to encrypt traffic using a conventional cryptosystem. So for instance RSA can be used to perform key exchange, and DES to perform the encryption. From this perspective we must answer two questions. The first question we must answer is how a user and the Yaksha server itself exchange keys. The second question is how the Yaksha infrastructure can be used to perform key exchange between two or more users.

For every user there are three relevant keys, d_{ii}, d_{iy} and e_i. A message encrypted with any one (or two) of these three keys, can always be decrypted using the remaining two (or one) keys. So for a Yaksha server to send a message to the user i it can send either:

$$C1 = M^{d_{iy}} \bmod n_i \quad \text{......(5.1)}$$

or

$$C2 = M^{d_{iy} \times e_i} \bmod n_i \quad \text{......(5.2)}$$

The user will perform:

$$M = C1^{d_{ii} \times e_i} \bmod n_i \quad \text{......(5.3)}$$

or

$$M = C2^{d_{ii}} \bmod n_i \quad \text{......(5.4)}$$

to recover M. Using an identical process, the user can send the Yaksha server any message M securely.

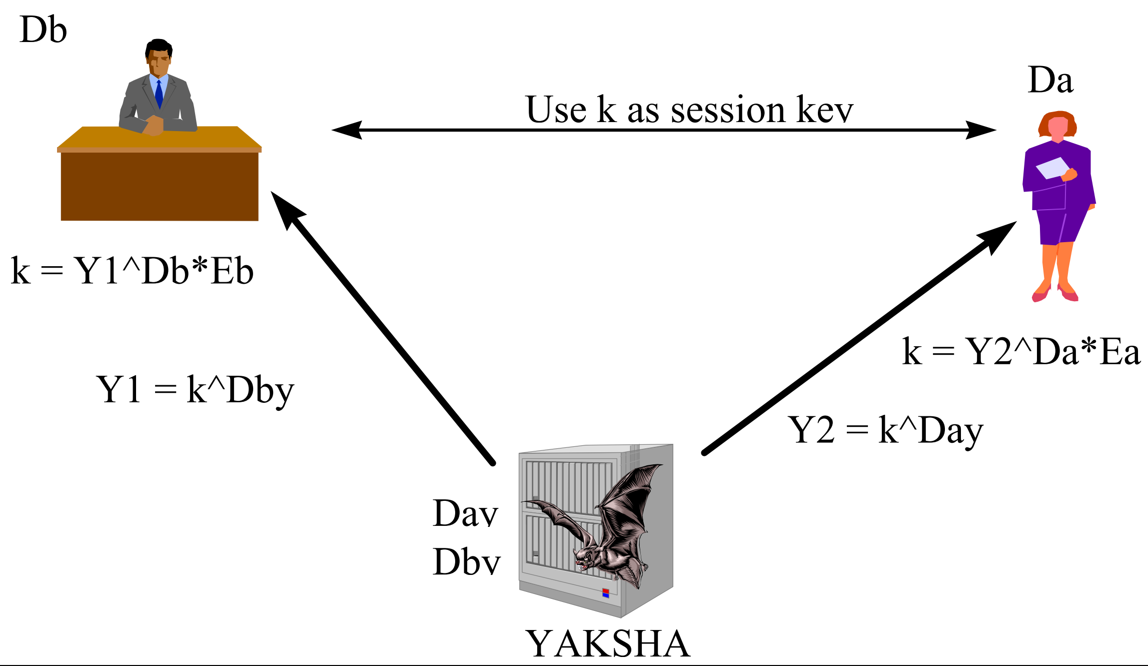
We note that there is a choice here between using equations 5.1 and 5.3 or using 5.2 and 5.4. There are marginal differences in the amount of work done by each party for the two options and the choice for implementation may be made based on this consideration. For the rest of this work we may use one or the other option, and even if not explicitly stated, it will usually be the case that the other option is equally viable.

This simple mechanism provides instant key exchange; all we need to do is send the session key k as the message. In Chapter 3 we reviewed a slightly different protocol proposed in [GANE94b]. Applied to the Yaksha infrastructure, this equates to the user sending the Yaksha server $k1$ using the above mechanism, the Yaksha server sending the user $k2$, and then having both use some combination of $k1$ and $k2$, say, $k = k1 \oplus k2$ as the session key. This version is more symmetrical and may be suited for some applications. However, in the rest of this work we will use the first version since it is more amenable for applications (such as e-mail) where the Yaksha server is used to mediate key exchange between parties who might not necessarily all be on-line at the same time.

Having seen how Yaksha and a user exchange keys, it is straightforward to describe methods to have the Yaksha server mediate key exchange between multiple users. Let us consider two possible systems.

In the first scenario, user i sends a message (this message should be encrypted and signed, though we omit those details for brevity) to the Yaksha server requesting key exchange with users $j,k,l,....$. The Yaksha server generates a random session key k, and sends, $k^{d_{iy}} \bmod n_i$, $k^{d_{jy}} \bmod n_j$, $k^{d_{ky}} \bmod n_k$, $k^{d_{ly}} \bmod n_l$ to users $i,j,k,l...$respectively. Each user can recover k using their private and public keys. Alternately, the Yaksha server can send these messages collectively to the requester i who can then distribute them. This latter approach is needed in applications such as E-Mail where the other users might not be online. Whereas if the Yaksha server were negotiating a telephony call set-up between multiple parties, it is likely the messages would be sent directly to the users. The most critical aspect of this protocol is that the Yaksha server *generates* and *knows* the session key.

Figure 5.1: Using Yaksha for Key Exchange. This illustrates one of several methods of key exchange possible within Yaksha.



Another option (described for the two party case) would be for user i to pick a random key $k1$ and generate:

$$C1 = k1^{e_j} \bmod n_j \qquad (5.5)$$

and then send $C1$ to the Yaksha server, which can compute:

$$C2 = C1^{d_{jy}} \bmod n_j \qquad (5.6)$$

and then send $C2$ to user j. User j can perform:

$$k1 = C2^{d_{jj}} \bmod n_j \qquad (5.7)$$

to recover $k1$. Similarly through a symmetrical process user j can transmit $k2$ to i. They can then use some combination of $k1$ and $k2$ as their shared key. This can be easily extended to the multiple party case. Note again that our explanation is conceptual. In a real implementation, all messages will have predictable structure, the identities of the sender and recipient, expiration times, etc. and it is likely that time stamps will be used. Although we omit them for brevity, these details are not trivial and security against various attacks relies on these details. The most critical aspect of this protocol is that the *Yaksha server does not know the eventual session key.*

Other protocols are also possible, and this wealth of options is evidence of the flexibility and power of the Yaksha infrastructure. Having seen how the Yaksha infrastructure can be used for digital signatures and key-exchange, we now turn to its use for key-escrow.

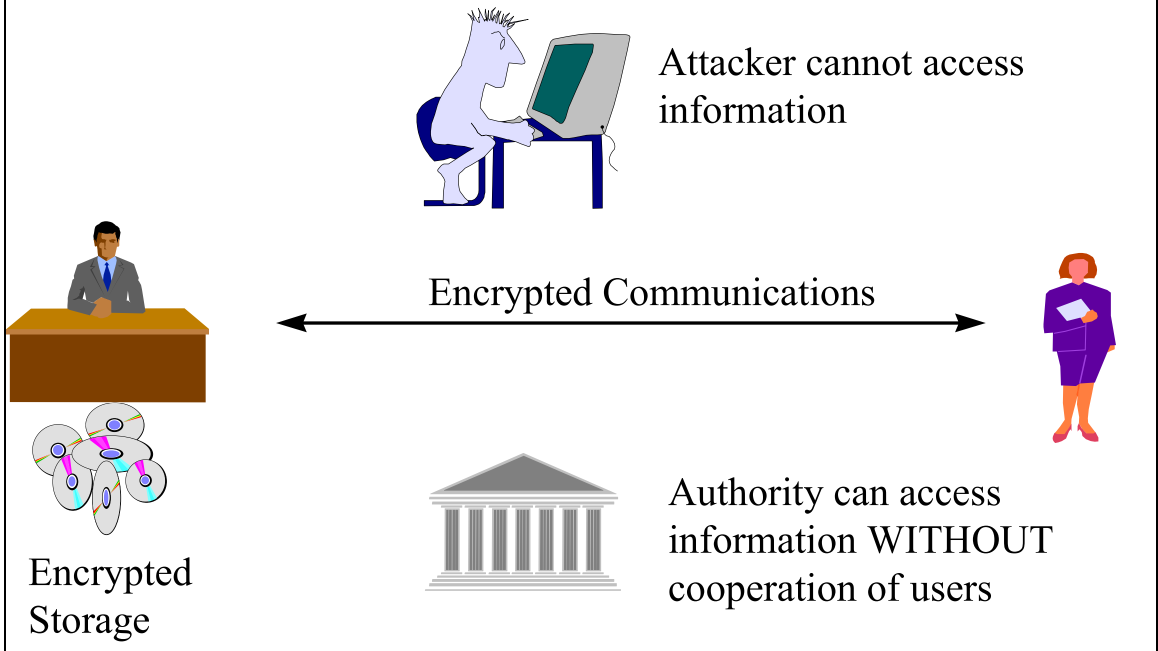
6. Using Yaksha for Key Escrow

In this chapter we examine how Yaksha can be used for key-escrow in various contexts. These results have been published by the ACM in the *Communications of the ACM* in [GANE96].

6.1 Introduction

It is commonly accepted that encrypted communications and data storage are an essential component of our emerging information infrastructure. Somewhat more controversial, is the concept that certain third parties other than those communicating or storing information may have a *legitimate* right to seek access to the information without the active cooperation of the participants. Clearly, encrypting information being communicated or stored could put the third parties at a significant disadvantage. Techniques to provide secure communications and storage with intentional backdoors that allow *legitimate* third parties access to the information, fall into the broad category of what may be described as *key escrow* systems. For the rest of this work we use the term *authority* synonymously with *legitimate third party*.

Figure 6.1: Key-Escrow in General. The generic requirements are that there should be a mechanism for authorities to access encrypted communications and storage, without the cooperation of the participants, but that the attacker should not be able to do the same.


Attacker cannot access information
Encrypted Communications
Encrypted Storage
Authority can access information WITHOUT cooperation of users

When the authority is the government and the participants are citizens, the entire concept is fairly controversial, as has been well documented [DENN93]. In this context the system that dominates the discussion is the so called "EES" or "Clipper" system [DENN96]. An analogous, (and in our opinion, less controversial) situation exists when the information is owned by an organization/corporation, the participants are employees, and the third parties are legitimate organizational/corporate authorities. The term 'corporate key escrow' is used loosely to describe this situation.

Several key escrow systems have been proposed in recent years and Denning and Branstad [DENN96] present an excellent summary and taxonomy of these various systems. Each of these systems approaches the problem using different underlying technical approaches, and from what can be described as different *philosophical stances*. In this work we describe a Yaksha based approach, which has its own philosophical stance, and begins with a different set of assumptions about the requirements than do many of the other systems. We begin with a summary of the key requirements driving the design of this system, describe the general system and show some example applications for telephony, electronic mail and data storage.

6.2 What are the Requirements?

The following two requirements are commonly accepted and in essence define key escrow systems:

1. The system should provide an *authority* the ability to access encrypted information without the cooperation of the participants.
2. The "backdoor" inherent in the system should not be usable by an unauthorized third party.

We view the next few requirements as important.

We believe authorities should only have access to short term session keys, not to long term user secrets.

In most cryptographic systems, each user has a long term private secret. In public-key cryptography [DIFF76], this would be the user's 'private key'. In a third party authentication system [NEED78], this would be the long term secret shared between the user and the third party server. For communications security, these long term private secrets are typically used as a means to negotiate a short lived session key, which in turn is used for encrypting a given session or conversation. If a legitimate authority seeks to eavesdrop on a conversation, one of two things can happen: (a) the key escrow system allows the authority to discover the long term private secret of the user, using which the authority can learn the session key for a given conversation, and proceed to eavesdrop. (b) the key escrow system allows the authority to recover the session key for a particular conversation, but not the long term private secret. The authority can still eavesdrop successfully, but the long term private secret is safe.

We believe that key escrow systems should be designed around the latter approach of only revealing short lived session keys. There are a number of reasons for our belief:

- First, since the long term private secret is never revealed to anyone, it can be reused for other functions, e.g. digital signatures. In systems where an authority can access this long term secret, reusing the long term private secret to generate digital signatures give the authority the power to forge signatures!
- Second, in our opinion, revealing only session keys provides a finer level of granularity of control. For instance, in such a system, one could implement policies such as: "the authority can eavesdrop on all of John Doe's conversations, except those he has with his wife or lawyer" or "the corporation can decrypt all of John Doe's files saved between March, 1994 and September, 1994, but not before or after. To our mind, all escrow systems represent a trade-off between an individual's rights to privacy and an authority's right to eavesdrop. Revealing session keys as opposed to long term private secrets provides more opportunities for an equitable trade-off.
- Third, the compromise of keys is not permanent. Namely, in systems where long term private secrets are revealed, the compromise of the user's secret is permanent. At some point the user must get a new private key (or if the key is embedded in a chip in a cellular phone, a new chip!). On the other hand, revealing session keys does not compromise the long term integrity of the permanent secret. So, once the period of 'legal eavesdropping' is over, the user does not have to be issued a new private secret.

We do note however, that to deliver session keys to an authority requires the escrow server to be on-line. Also, note that we are not suggesting that the escrow agent inspect the contents of any messages, and in fact in a practical system, it is unlikely that the agent would have any access to the actual message stream, and the agent's participation would be limited to playing a role in setting up the parameters for a session.

While the justification for the above requirement is grounded in a debatable philosophical stance, the next requirement is based on something more concrete: money!

It is very desirable that the key escrow system reuse the security infrastructure necessary for other security functions such as key exchange, digital signatures and authentication.

We have already discussed the need for reuse extensively in Chapter 2, but we wish to reiterate that achieving reuse is to our mind critically important. Yaksha is the only system we know of that does not require a separate 'key escrow infrastructure'.

Our next requirement is:

A key escrow system should be implementable in either hardware or software, should apply to both computer communications as well as telephony, and should be usable both for the citizen-government and employee-organization situations. i.e. the system should have universal applicability.

We believe this is important for two reasons. First, as we discussed above, reusing security infrastructures is beneficial and it may not be cost effective to have multiple infrastructures for different types of key escrow. Second, there is a clear convergence between telephony and computer communications, and it will increasingly become impractical to treat these situations differently. There is little logic for instance in treating voice conversations differently from on-line *chat*.

Is it possible to design a system that meets all these requirements? We believe the answer is yes, and we now describe how these requirements can be met with Yaksha.

6.3 Yaksha Key Escrow: General Concepts

In the previous section we considered two separate key exchange systems that are possible within the Yaksha infrastructure. In the first, the Yaksha server generates the random session key and distributes it securely to the relevant parties. The key point in this version of the key exchange system is that the Yaksha server *knows* the session key, and is consequently in a position to reveal it. If the appropriate authorities are also part of the Yaksha system, then they can send a message to the Yaksha server asking that certain session keys be captured and revealed, for a particular period of time. If a court order is required, then one of those making the request should be the judge. If multiple authorities are required to authorize the 'tap' then all of them should make the request. This message to the Yaksha server should be encrypted and signed using the Yaksha system itself, and then the Yaksha server will reveal selected appropriate session keys to the authority.

Consider the advantages of this simple but powerful method:

- The Yaksha server only reveals session keys. It cannot reveal the user's long term private secret since it does not know it! Almost all other [DENN96] systems require compromise of the user's long term private secrets.
- If the nature of the authorities (either the specific identity or the number of authorities) should change, then a simple parameter change at the Yaksha server solves the problem. Contrast this with the Clipper system where every user's private secret is escrowed in two pieces with two pre-determined authorities at key generation time. Imagine the difficulty in changing either the identity, or the number of escrow agents!
- The amount of trust placed in the escrow agents is much less. In the Clipper system the escrow agents keep their portions of the escrowed keys in "safe storage". How safe is safe? Even the best kept secrets, for instance the identity of a nation's spies in another nation, are frequently not kept "secret enough" as recent evidence shows. In the Yaksha system the authorities never have a user's private secret, so there is no question of compromise!
- The Yaksha system provides a fine granularity of control. It is perfectly feasible in our system for a judge to sign a court order asking the server to release only specific session keys. Whereas in most systems, like the Clipper system, the judge has to sign an "all or nothing" request for tapping.
- Finally, beyond some additional rules to implement the functionality of storing certain session keys, and revealing them under appropriate conditions, the existing Yaksha infrastructure requires no changes, and can be completely reused.

We do wish to emphasize that if it is desired to use Yaksha without using key escrow, then that is perfectly possible. Key escrow is an added function which Yaksha provides and can be used if desired. The last big advantage of Yaksha Key Escrow will become readily apparent as we give concrete examples of escrow; namely, it can be used for varied escrow applications from telephony to e-mail to file storage.

6.4 Applications of the Yaksha System for Key Escrow

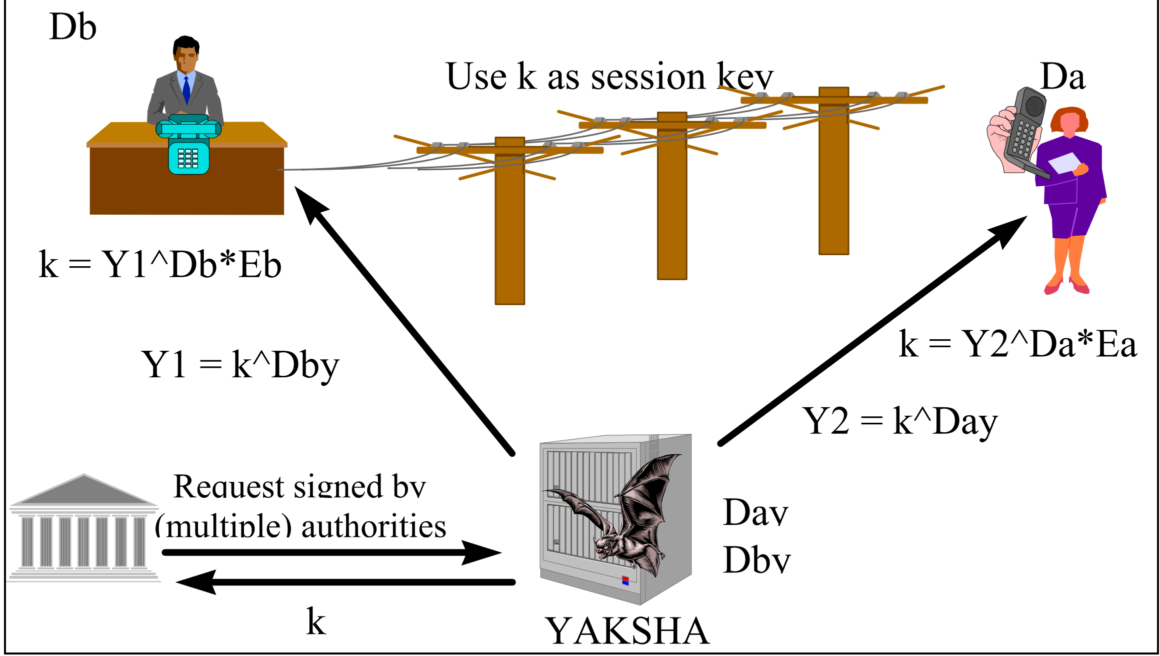
We now describe how three very different key escrow problems can be solved using the same Yaksha infrastructure. As in our discussion so far, our goal is to illustrate the concepts and consequently we do not describe several details, a few of which have significant security implications.

6.4.1 Using Yaksha for Telephony

Our first example is in the world of telephony. Since in this model both parties are on-line at the same time, the key exchange protocol described above can be used exactly as stated. Alice indicates to the Yaksha server, a desire for secure communication with Bob. The Yaksha server distributes $C_a = k^{d_{ay}} \bmod n_a$ to Alice and $C_b = k^{d_{by}} \bmod n_b$ to Bob, who each recover k and use it for encrypting the conversation. In practice the transaction would be transparent to the user who might for instance, simply pick up the phone, dial *007, and then Bob's number, and never notice anything else. The key exchange etc. would be handled as part of call set-up, and the Yaksha server would be just one more of the many intelligent computers now attached to the phone network to provide special services.

When an authority wishes to tap a phone line, a request, R, is signed and sent to the Yaksha server. If it is desired that multiple authorities must cooperate in order to tap a line, then we can require that the request, R, be signed by multiple authorities. Observe that the authorities themselves should be registered in the Yaksha system. Each authority has its own private key, d_{A1}, d_{A2}, ..., and the Yaksha server keeps a corresponding d_{A1y}, d_{A2y}, ..., and corresponding public keys $(e_{A1}, n_{A1}), (e_{A2}, n_{A2})$, ... exist in the system. So for instance, if certain types of taps require the signature of authorities $A1$ and $A2$, then the request sent to the Yaksha server can be of the form $(R^{d_{A1}} \bmod n_{A1})^{d_{A2}} \bmod n_{A2}$. The Yaksha server can authenticate and recover the request using d_{A1y}, d_{A2y}, ..., and the corresponding public keys $(e_{A1}, n_{A1}), (e_{A2}, n_{A2})$. Alternately, separate signed messages can be sent to the Yaksha server. The request R can take on several forms, for instance it may order the Yaksha server to provide session keys for all future conversations Alice carries out, or, it may ask only for certain types of conversations. The key point to note is that the design provides tremendous flexibility and a wide variety of underlying policies can be implemented. Further, the policies can be changed easily without huge changes to the system. For instance, if public policy were to change to requiring four cooperating authorities instead of two, or if the identities of the authorities should change, then minor changes to the system parameters achieve the goal. This may not be of great theoretical interest, but as is often observed in practice, it is mundane issues such as ease of ability to change, on which the security of systems rest.

Figure 6.2: Yaksha Key-Escrow in Telephony. The Yaksha server performs key exchange, and when needed, provides session keys to the authority. Providing session keys, as opposed to long term private secrets, results in a system with much greater flexibility and less risk.



It is worth observing that a part of the system requires Bob's telephone to recover k from C_b in a fashion that ensures proof that C_b was generated by the Yaksha server. This means it is possible to prevent a dishonest (trying to cheat the key escrow system) Alice from carrying out a secure conversation with an honest (playing by the rules) Bob.

6.4.2 Using Yaksha for Electronic Mail

We pick electronic mail as our next example, because it has a fundamental structural difference in the requirements, namely the underlying messaging is of a store and forward nature in which the sender and receiver are not both on-line at the same time. Current systems [SCHN94] for secure e-mail are generally based on the sender, say Alice, sending the receiver, say Bob, the following construct:

$$\{E(M,k), k^{e_b} \bmod n_b, S, Alice's\_Certificate\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(6.1)$$

The construct has four pieces:

- $E(M,k)$: The message M encrypted with a session key k generated by Alice.
- $k^{e_b} \bmod n_b$: The session key k is encrypted with Bob's public key, e_b. On receiving the message Bob will use his private key d_{bb} to recover this session key k and will then use k to recover M from $E(M,k)$.

- S: Next a hash, or fingerprint, of the message, $H(M)$ is signed by Alice, using her private key d_{aa}, to generate her signature, S, i.e. $S = (H(M))^{d_{aa}} \bmod n_a$. The hash of the message is used in lieu of the message itself, for reasons of efficiency.
- $Alice's\_Certificate$: Finally Alice's certificate is enclosed. Bob can retrieve Alice's public-key from her certificate, and then use her public-key to verify her signature on the hash.

In keeping with our general policy of integrating Yaksha with existing systems, as opposed to creating a fresh system from scratch, we attempt to reuse the above constructs to the extent possible. We see the system working as follows:

1. Alice sends the Yaksha server, $S1 = H(M)^{d_{aa}}$ and indicates that the intended recipient is Bob.
2. The Yaksha server computes $S = S1^{d_{ay}} \bmod n_a$ and replies to Alice with the message $\{S, k^{d_{ay} \times e_a} \bmod n_a, k^{d_{by} \times e_b} \bmod n_b\}$. The first portion is simply the completed RSA signature for Alice on the message M. The second portion is decrypted by Alice, using d_{aa} to recover k. Alice will use this k to encrypt the message M, i.e. $E(M,k)$. The third portion is sent on to Bob by Alice without any modification.
3. So the message Alice sends Bob is:

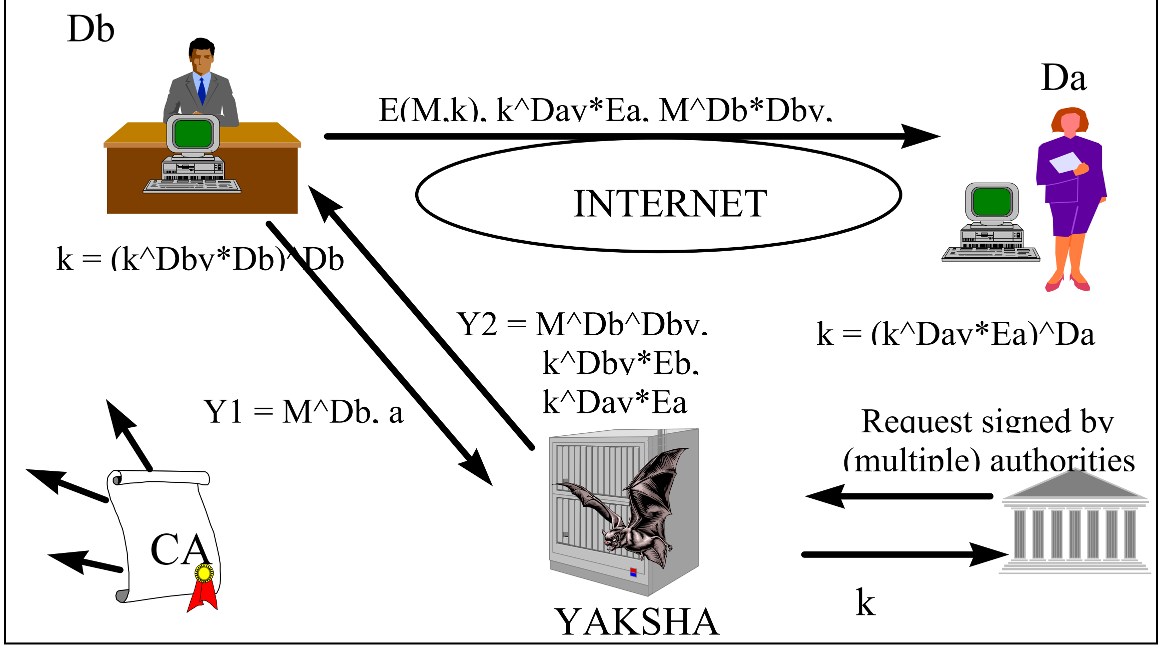
$$\{E(M,k), k^{d_{by} \times e_b} \bmod n_b, S, Alice's\_Certificate\} \text{ ....(6.2)}$$

Except for the second field, this is exactly equivalent to the construct Alice would have sent Bob, in a non-Yaksha system.

4. When Bob receives this message, he verifies Alice's signature exactly as in a non-Yaksha system, but to recover the session key k, he uses d_{bb}, i.e. $k = (k^{d_{by} \times e_b} \bmod n_b)^{d_{bb}} \bmod n_b$.

Since the Yaksha server has the session key k the actual escrow process is identical to that described for the telephony situation. All the benefits of using Yaksha (e.g. short memorizable user passwords) still apply. From the standpoint of message structure the new system is identical to the existing standards [SCHN94]. In fact, it is worth observing that interoperability between Yaksha and non-Yaksha systems is relatively easy. If Bob is not a part of the system, his corresponding Yaksha key, d_{by} is simply set to one. Bob will not notice the difference, and the escrow will still work. We reiterate, that we are glossing over some fine print essential to secure functioning; for instance, the hash sent by Alice to the Yaksha server should be surrounded by specific structure so that the Yaksha server can authenticate Alice before responding.

Figure 6.3: Yaksha Key-Escrow in E-Mail. Unlike most other key escrow systems, the Yaksha system is highly flexible, and can be used both for telephony (real time) and e-mail (store and forward). This flexibility allows the infrastructure to b reused. Even better, the underlying infrastructure for key escrow is the same as that for authentication and signatures.



6.4.3 Using Yaksha for Encrypted File Storage

As with communications, it is increasingly becoming necessary to provide users of computers with access to encrypted file or data storage, and escrow mechanisms are needed. In addition to the usual reasons for an authority to be able to retrieve this data without the user's cooperation, more ordinary reasons like access to a critical file in a co-worker's absence, also come into play. Using Yaksha to meet this requirement is fairly straightforward and one can think of countless variations. We describe one such possibility below, which assumes the existence of a file server process which is an entity independent of the user. The system works as follows:

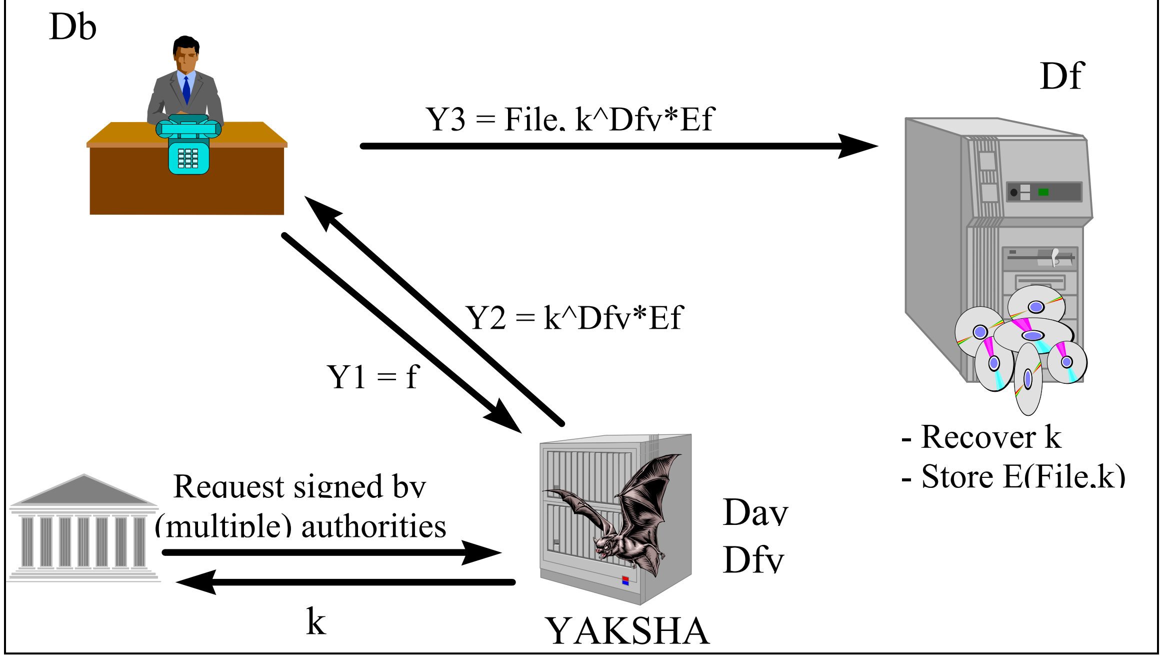
1. Alice sends the Yaksha server the name of the cryptographic file server, F, where she wants to store the file.
2. The Yaksha server sends Alice a storage key k, encrypted with the Yaksha server's key for the file server, i.e. $k^{d_{Fy} \times e_F} \bmod n_F$.
3. Alice sends the file server the file and the encrypted storage key (note Alice does not know the storage key).
4. The file server recovers the storage-key, using its own private key, d_{FF}, encrypts the file with the storage key k, and stores the encrypted file, $E(File, k)$ and the encrypted storage key, $k^{d_{Fy} \times e_F} \bmod n_F$.

5. When Alice wants the file, she simply sends it a signed, and time stamped, request, Q, which she signs by interacting with the Yaksha server. The file server can verify the signature on the request, recover k from $k^{d_{Fy} \times e_F} \bmod n_F$, decrypt the file, and send it to Alice.
6. When an authority wants a file, the authority interacts with the Yaksha server and sends a duly signed request, R, to the file server. The file server uses the authority(s) public key(s) to verify the signature on the request, recovers the session key, decrypts the file, and sends it to the authority.

The basic idea is that the file server will only encrypt files using a key which it gets from Yaksha. It then stores this key in an encoded form, with the file itself. Note that in practice such a system would have provisions for mutual authentication, encrypted communications between users and the file servers, and most likely, require a signed hash of the file to also be stored. All of these functions can of course be achieved by reusing the Yaksha infrastructure. Observe that we do require the file server process to have a long term private secret key, d_{FF}, which it must keep in persistent storage. We anticipate that in a practical system, this key and the functions that are performed with it, happen inside the safe confines of a tamper resistant chip. This is not particularly onerous, especially since we do not require the storage key for each file to be stored inside this chip.

Several variations on this theme are possible. For instance we could create a system where the Yaksha system can be the source of keys for the authorities, or have a system where the user knows the keys used for encryption. Within the basic Yaksha framework there are many ways to achieve escrow functionality, and the system can be tailored to fit the requirements of the application.

Figure 6.4: Yaksha Key-Escrow for File Storage. Further illustrating the flexibility of the Yaksha infrastructure, this shows how the same infrastructure is reused for a completely different application, file storage.



In conclusion we wish to note that the examples above are only a sample of the endless possibilities in which key escrow can be implemented within Yaksha. Given the constraints of a particular application, a designer can take the basic Yaksha infrastructure and design a key escrow system that is specifically tailored to the application. It is this flexibility of Yaksha that makes it so powerful, and leads to reuse.

7. Using Yaksha for Authentication: The Next Generation Kerberos

This chapter describes how the Yaksha infrastructure can be integrated with a Kerberos environment. The resulting system completely removes the major shortcomings of Kerberos. These results have been published by the IEEE in the Proceedings of the Internet Society Symposium on Network and Distributed Systems Security [GANE95].

7.1 Introduction

The Kerberos authentication system [KOHL93] based on the classic Needham-Schroeder authentication protocols [NEED78] with extensions by Denning-Sacco [DENN81], 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 in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE). Consequently, we expect Kerberos to be among the most widespread security standards used in distributed systems over the next several years. (Both IBM for their mainframe MVS operating system and Microsoft for their PC based NT operating system are basing their future security architecture around Kerberos).

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 attack - 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. We began work on Yaksha with a different approach. 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 how within the Yaksha infrastructure, it is possible to create a new generation Kerberos which completely eliminates the shortcomings of the existing Kerberos system, yet keeps changes to the protocol to a surprising minimum.

Before explaining how we achieve this, it is necessary to describe the Kerberos system.

7.2 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. We now describe the messages in further detail.

Message 1, known as *as_req* (request to authentication service) consists of:

$$as\_req\text{: } c, tgs, time\_ex, n \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (7.1)$$

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, $T_{c,tgs}$, $time\_ex$ 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*,

$$as\_rep\text{: } \{K_{c,tgs}, time\_ex, n, \ldots\}K_c, \{T_{c,tgs}\}K_{tgs} \quad \ldots\ldots\ldots\ldots\ldots (7.2)$$

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 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[8] as_req. The attacker can then take the portion of the as_rep encrypted with K_c, and attempt to decrypt it 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.

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

$$tgs\_req{:}s, time\_ex, n, \{T_{c,tgs}\}K_{tgs}, \{ts...\}K_{c,tgs} \quad \text{…………………….(7.3)}$$

This message consists of the name of the server, s, the expiry time, $time\_ex$, 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*,

[8] Version 5 of Kerberos has some methods of preventing this.

$$tgs\_rep: \{K_{c,s}, time\_ex, n, s, ...\}K_{c,tgs}, \{T_{c,s}\}K_s \quad \text{......................(7.4)}$$

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,

$$ap\_req: \{ts, ck, ...\}K_{c,s}, \{T_{c,s}\}K_s \quad \text{....................................(7.5)}$$

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*,

$$ap\_rep: \{ts\}K_{c,s} \quad \text{...(7.6)}$$

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.

7.3 Other Potential Approaches to a Better Kerberos

Yaksha is not the only attempt to fix the problems inherent in Kerberos. Some other efforts are worth mentioning. The SPX system [TARD91] is a full-fledged public-key based authentication system which does not require a trusted on-line server. Its 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. Unlike Yaksha, 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 [MCMO95] 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 this objective.

7.4 Using Yaksha for Authentication: Design Goals

We now discuss our design goals in more detail.

7.4.1 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. We discuss how we protect Yaksha against these attacks in the next Chapter. Further, if each d_{iy} is encrypted with a master key which is stored inside a tamper resistant chip, then even this is not possible.

7.4.2 Removing Vulnerability to Dictionary Attacks

Dictionary attacks are essentially a password-guessing attack. The importance of this attack, and the importance of "good" passwords in general, is sometimes overlooked, and we will discuss this attack and methods of ensuring 'good passwords' in the next Chapter. Suffice to say here that Kerberos is vulnerable to password guessing, and that removing this vulnerability is an important design goal.

7.4.3 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 Yaksha 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.

7.4.4 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 deployment. 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 private keys as smart cards become ubiquitous.

7.4.5 Reuse Authentication Infrastructure for Other Functions

We have described the need for reuse extensively in Chapter 2 and will not repeat ourselves. We do wish to point out that the ideas here are applicable even if reuse is not an objective, and the objective is simply a better Kerberos.

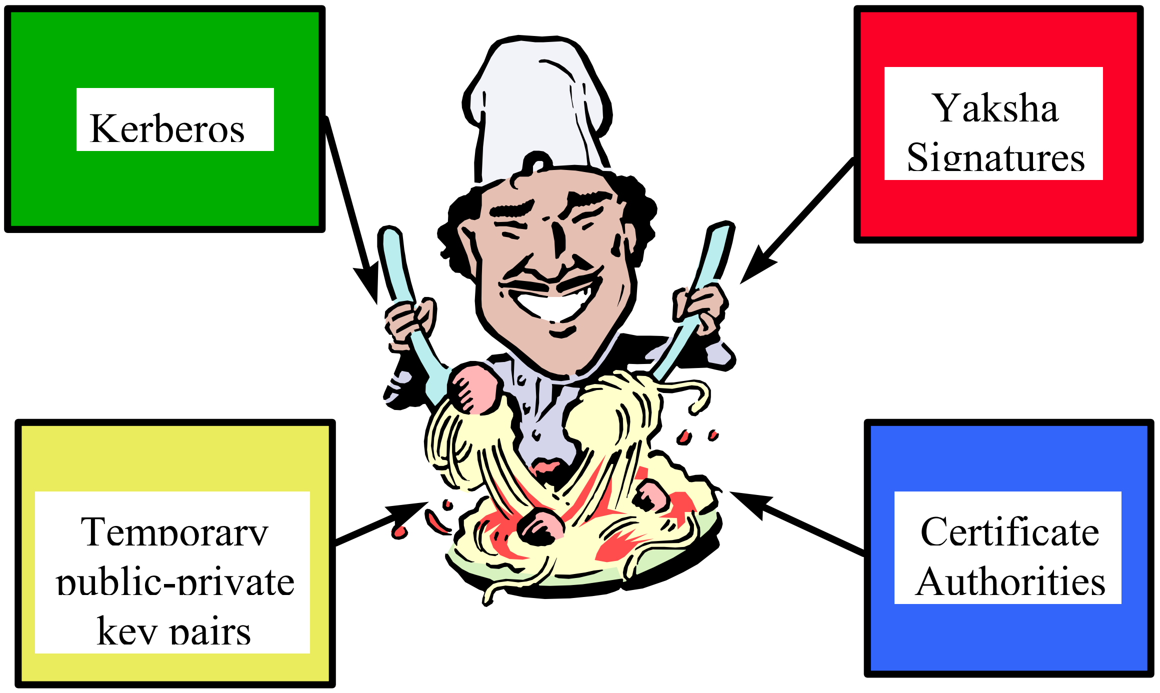
7.5 Using Yaksha for Authentication

We are now ready to describe the basic 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. Also, $[Cert\_c]^{D_{ca}} \bmod n_{ca}$, is the public key certificate for user *c*, signed by the certificate authority, ca. Finally, for the sake of brevity, we often leave the modulus out of our equations. To reiterate, all arithmetic is modular, and the modulus should be obvious from context. For example if we say $a^{e_{ii}}$ or $a^{d_{iy}}$, we mean $a^{e_{ii}} \bmod n_i$ or $a^{d_{iy}} \bmod n_i$ respectively. Also, we occasionally abbreviate the key d_{ii} by d_i. For instance, we write d_{tgs} for the private key of the tgs. Finally, our use of upper case (E, D, N) instead of (e, d, n) is simply for ease of viewing complex equations.

7.5.1 Message Structure Overview

In Yaksha we desire that to authenticate itself to the Yaksha server, the client reveal *knowledge* of his Yaksha private key D_{cc} (which is of course *completely* different from revealing D_{cc} 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 Yaksha 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 server.

Figure 7.1: Yaksha Authentication. The Yaksha infrastructure can provide several methods of authentication, including a next generation Kerberos protocol which solves some of the most critical weaknesses in Kerberos. To achieve this requires a novel way of using temporary public-private key pairs, and a judicious combination of the techniques shown below.



Like in Kerberos, we do not want to store the user's private-key D_{cc} on the computer for more than the barest minimum time. This requires the use of a novel approach; i.e. the use of a temporary RSA private-public key pair which is generated on the fly. In our novel approach, the client and the Yaksha server collaborate to sign a certificate certifying the temporary public-key. Note, that the certificate is signed, and verified using the long term private and public-key. However, all subsequent messages use the temporary keys (for the length of time it is valid). Note that the Yaksha server never sees the private-key portion of the temporary pair.

To summarize:

- An entity's private 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.

7.5.2 The Yaksha *as_req* and *as_rep* Messages

In Kerberos the initial as_req message is:

Kerberos: *as_req: c, tgs, time_ex,n ……………………………….(7.7k)*

The corresponding Yaksha message is:

Yaksha: *as_req: c, tgs, time-ex,* $[[TEMP\_CERT]^{D_{cc}},n]^{D_{cc}}$ *……….(7.7y)*

Here $TEMP\_CERT$ contains $(c, E_{c,temp} N_{c,temp}, time\_ex, etc.)$ 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 $time\_ex$, 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_{cc}. This structure is then concatenated with a random string, n, and again "encrypted" with D_{cc}. This prevents an attacker who later sees $TEMP\_CERT$, from seeing $[TEMP\_CERT]^{D_{cc}}$ and mounting a dictionary attack by taking guesses at D_{cc} and checking if $[TEMP\_CERT]^{guess} = [TEMP\_CERT]^{D_{cc}}$.

Let us denote $[[TEMP\_CERT]^{D_{cc}},n]^{D_{cc}}$ by Y. The Yaksha server performs $[[Y]^{D_{cy}}]^{E_c}$ to recover $Z = [TEMP\_CERT]^{D_{cc}}$ 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:- $as\_req: \{K_{c,tgs}, time\_ex, n, ...\} K_c, \{T_{c,tgs}\} K_{tgs}$ *…………..(7.8k)*

The Yaksha message is:

Yaksha:- *as_rep:*$[K_{c,tgs}, time\text{-}ex, n, ...]^{E_{c,temp}}, [T_{c,tgs}]^{D_{tgsy}}, [[TEMP\text{-}CERT]^{D_{cc}}]^{D_{cy}}$ *….(7.8y)*

Observe that the first two components of the Kerberos and Yaksha messages are identical, except 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 private key. Notice that the user's private key D_{cc} 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_{cc}. Yet, D_{cc} 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_{cc}. Further, a message can be sent securely to the client encrypted under $E_{c,temp}$, by any entity that sees the temporary certificate.

7.5.3 The Yaksha *tgs_req* and *tgs_rep* Messages

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

$$\text{Kerberos:- } tgs\_req{:}s, time\_ex, n, \{T_{c,tgs}\}K_{tgs}, \{ts...\}K_{c,tgs} \text{ ………….}(7.9k)$$

The only Yaksha modifications to this Kerberos message are to (a) attach the temporary certificate to the message, and (b) take the encrypted ticket granting ticket from the Yaksha as_rep message, $[T_{c,tgs}]^{D_{tgsy}}$, 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 ticket granting ticket for a "fake" client. The resulting message is:

$$\text{Yaksha:- } tgs\_req{:}\ s,\ time\text{-}ex,\ n, [\,[T_{c,tgs}]^{D_{tgsy}}\,]^{D_{c,temp}}, \{ts...\}K_{c,tgs}, [\,[TEMP\text{-}CERT]^{D_{cc}}\,]^{D_{cy}} \text{ ..}(7.9y)$$

The ticket granting server:

- retrieves the user's permanent certificate $[Cert\_c]^{D_{ca}}$,
- recovers (E_c, N_c) and uses this to recover the $TEMP\_CERT$,
- uses the temporary public-key in the temporary certificate to retrieve $[T_{c,tgs}]^{D_{tgsy}}$, and
- then uses its private key D_{tgs} and public key (E_{tgs}, N_{tgs}) to recover the ticket, $T_{c,tgs}$, where $T_{c,tgs}=[[\,T_{c,tgs}]^{D_{tgsy}}\,]^{D_{tgs}\times E_{tgs}}$.

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

Kerberos:- $tgs\_rep\colon \{K_{c,s}, time\_ex, n, s, ...\}K_{c,tgs}, \{T_{c,s}\}K_s$ ……………….(7.10k)

almost unchanged, for instance, simply by replacing $\{T_{c,s}\}K_s$ with $[T_{c,s}]^{D_{sy}}$. 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. To avoid this we make the return message contain proof of the tgs's authenticity. To do this we simply make the tgs complete the signature on $[T_{c,tgs}]^{D_{tgsy}}$ message using D_{tgs} and return the result to the client.

Important Note: Taken literally, this would imply that anyone with access to the public key of the tgs can decrypt the completed signature and see the contents of the $T_{c,tgs}$. This would be fatal since this ticket contains the shared session key! What we mean, but neglect to say for the sake of brevity of notation, here and in other similar places, is that in addition to the ticket, a hash (in some valid structure) of the ticket would be signed first by the as and then by the tgs. It is the hash that is returned by the tgs. Observe that the client can regenerate the hash from the ticket, which it knows. Identical comments apply to the ap_rep message.

The resulting tgs_rep message is:

Yaksha:- $tgs\_rep\colon \{K_{c,s}, time\text{-}ex, n, s, ...\}K_{c,tgs}, [T_{c,s}]^{D_{sy}}, [[T_{c,tgs}]^{D_{tgsy}}]^{D_{tgs}}$ ……..(7.10y)

The client retrieves (unless we choose to include it in the tgs_rep) the tgs's long term certificate, ($[Cert\_tgs]^{D_{ca}} \bmod n_{ca}$) uses (E_{ca}, N_{ca}), which is the Certifying Authority's public key, to recover (E_{tgs}, N_{tgs}) and verifies that $[[T_{c,tgs}]^{D_{tgsy}}]^{D_{tgs}}$ is the signature on a valid $T_{c,tgs}$. A compromised as cannot generate a valid signature signed by the tgs, since it does not know D_{tgs}.

7.5.4 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:- $ap\_req\colon \{ts, ck, ...\}K_{c,s}, \{T_{c,s}\}K_s$ ……………………..(7.11k)

Yaksha:- $\{ts, ck, ...\}K_{c,s}, [[T_{c,s}]^{D_{sy}}]^{D_{c,temp}}, [[TEMP\_CERT]^{D_{cc}}]^{D_{cy}}$ ….(7.11y)

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

Kerberos: $ap\_rep\colon \{ts\}K_{c,s}$ ……………………………………..(7.12k)

Yaksha: $ap\_rep\colon [[T_{c,s}]^{D_{sy}}]^{D_{ss}}$ ……………………………….(7.12y)

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.

Figure 7.2: Kerberos Message Structure. The basic message structure is retained in Yaksha, as can be seen with contrast to Figure 7.3.

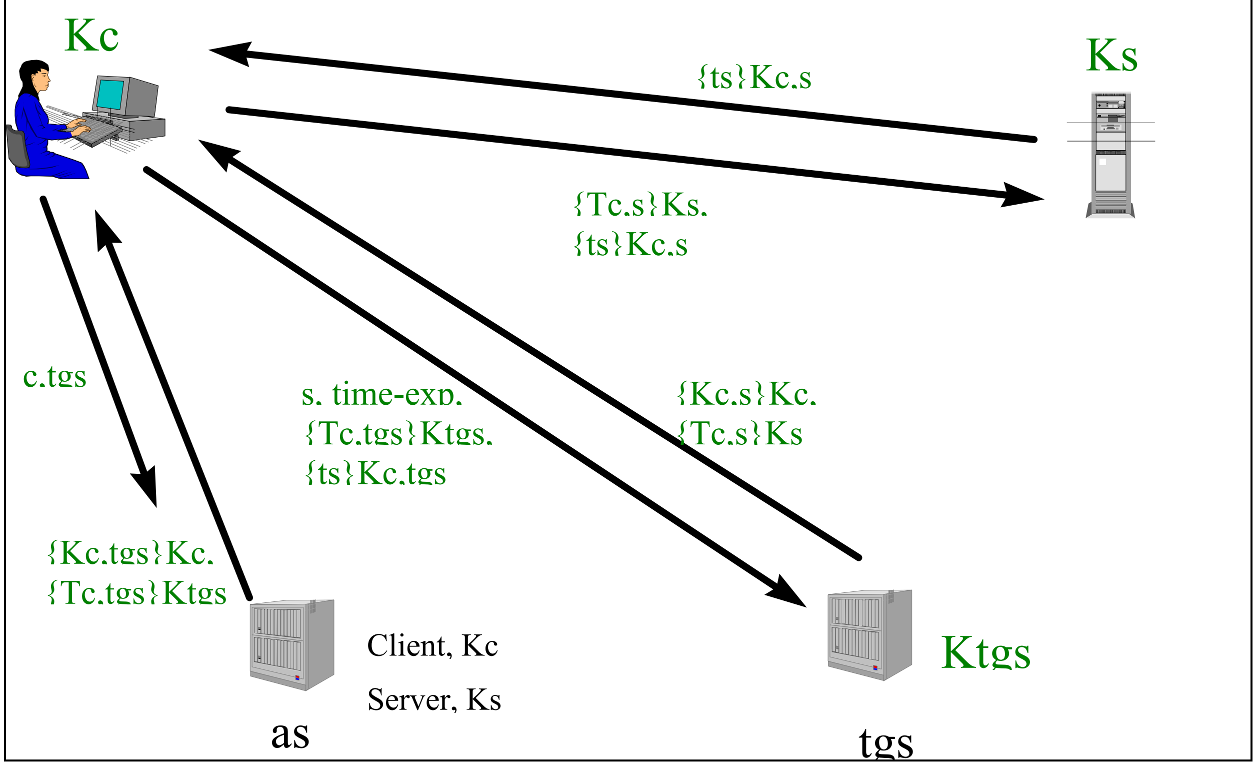
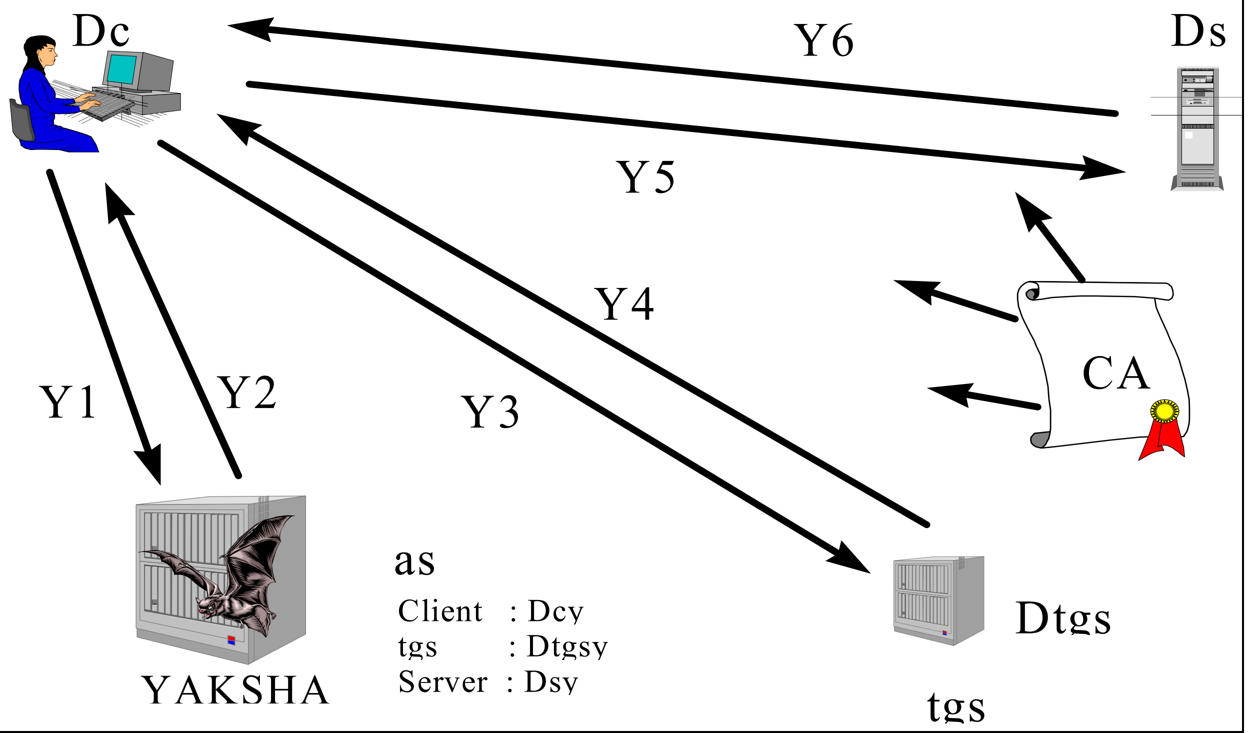


Figure 7.3: Yaksha Authentication Message Structure. When contrasted to Figure 7.2 we see an identical structure. Given the vastly improved security and functionality, it is surprising that the number of changes are relatively small.



7.5.5 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 earlier (with added bells and whistles to remove the potential of dictionary attacks). The messages are:

$$\text{Yaksha: } sign\_req\text{: } c,\ [[H,ts]^{D_{cc}}\ ,\ n]^{D_{c,temp}},\ [[TEMP\_CERT]^{D_{cc}},\ n]^{D_{cc}} \quad \ldots\ldots(7.13)$$

$$\text{Yaksha: } sign\_rep\text{: } [[[H,\ ts]^{D_{cc}}\ ]^{D_{cy}},n]^{E_{c,temp}} \quad \ldots\ldots\ldots\ldots\ldots\ldots(7.14)$$

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, $[H,ts]^{D_{cc}}$, 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 $[[H,ts]^{D_{cc}}, n]^{D_{c,temp}}$. The client sends this and the usual partially signed temporary certificate, $[[TEMP\_CERT]^{D_{cc}}, n]^{D_c}$ 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 the temporary public key to recover $[H,ts]^{D_{cc}}$ and n. The server can then recover $[H,ts]$, by performing $[[H,ts]^{D_{cc}}]^{D_{cy} \times E_c}$. 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 $[[H, ts]^{D_{cc}}]^{D_{cy}}$ 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.

7.6 Observations

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 Yaksha at hand, several of our choices may seem curiously sub-optimal. This results from our strong desire to retain several Kerberos structures and ideas for interoperability purposes, 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 work, but are a natural extension of the ideas contained here.
4. A key idea is the notion of the temporary public-private key pair, with the corresponding certificate signed using the long term keys. It is possible to exploit this idea further for efficiency. One can imagine a system where such temporary certificates are kept on-line in some common database. These keys are then used for authentication, signatures and key-exchange.

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 pre-computed 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. Also, the Yaksha server is likely to require more “horsepower” than a comparable Kerberos (perhaps requiring more servers). However, this may be a small price to pay when weighed against the potential havoc of a catastrophic failure of a Kerberos database.

8. Passwords in the Yaksha Infrastructure

In this Chapter we describe why “good” passwords are necessary, discuss some methods for ensuring their use and point to some related results we have obtained (which have been published elsewhere).

8.1 Introduction

Next to sloppy administration, it is our informal assessment that poorly chosen user passwords are still the single largest source of attacks. Such passwords lend themselves to what are known as dictionary attacks. In a dictionary attack, an attacker guesses passwords from a ‘dictionary’ of bad passwords, which would typically include a natural language dictionary, phone books, common acronyms used in a particular industry, etc. So if passwords are eight characters long, then good key generation principles would dictate that the passwords be uniformly selected from a space of roughly 100 characters, and the total size of the password space, P, should be 100^8, or 10,000,000,000. Unfortunately too many passwords are likely to be selected from a space of a few hundred thousand natural language words or names (which comprise the attacker’s dictionary, D). We loosely define as “bad passwords” those in D.

So for instance, the password *unknown* is a bad password, since it will be present in an English dictionary. What often goes unrealized is that passwords such as *unknown1* are also vulnerable to attack. Most dictionary attackers will be able to capture passwords with one character of random noise, by expending more effort. For instance, the character of ‘noise’ most people are likely to pick is a numeral. There are 10 digits, and having selected one, there are nine ways to stuff an eight character password with a digit. Consequently, there are potentially 90 resulting combinations for each passwords. So, approximately speaking, an attacker by spending about hundred times more effort is likely to capture a large number of ‘bad passwords’ which have been stuffed with a digit. What if instead of digits, the user inserts any character into the password, to generate a password like *unknQown*? Since there are roughly hundred other characters (including lower case alphabets, upper case alphabets, digits and special characters), the extra effort the attacker has to spend to check D is roughly increased by a factor of 1000.

It is unfortunately the case that this extra effort is not a huge constraint, and consequently a bad password with *one character of noise* is still a bad password. We call such passwords as *bad noisy passwords*. It is essential to ensure that users pick neither bad passwords nor bad noisy passwords. Observe that our definition of bad noisy passwords is somewhat arbitrary, and does not include passwords such as *un22known*, which is a bad password with a date in it, clearly a weak password. However, barring such special cases, our definitions are in general useful classifications.

Before discussing ways to prevent users from selecting bad passwords or bad noisy passwords, let us discuss the different types of dictionary attacks in more detail.

8.2 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 developing 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, or an English language password the attacker is guessing
    - 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. We assume that the attacker sees C.
3. The password space P being guessed consists of $|P|$ passwords. The attacker has a dictionary of D with $|D|$ passwords. The attacker will take guesses $p_1, p_2, ..., p_{|D|}$ until he finds a p_i which is equal to k.
4. The function F and its inverse (assuming one exists), 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, since the effort to apply F to a guess is higher in the latter case.

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 D until he discovers (assuming it exists) 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 pre-compute the $F(S1, p_i)$ for all or many p_i and also amortize his attack against several users. This amortization can happen if the attacker has a series of encrypted passwords, $C_1, C_2, C_3, ...$, (for instance the password file on a UNIX system), and can for each guess at a p_i simply compute the encryption and look up the encrypted passwords to see if any match.
- $S2$ type attacks. Here the attacker is typically computing $F^{-1}(C, p_i)$ and is hoping to find an $S2$ which he can recognize. Here the attacker cannot start computations before he captures C. Further, since C is likely to be different for each instance, no amortization of attacks are likely. In some forms, the Kerberos system is vulnerable to such attacks.

The degree of vulnerability depends on the amount of effort the attacker can spend. We urge the reader to see [KARN89] to see what is practical. We then urge the reader to remember that this reference was written close to a decade ago, a period during which the ubiquity of cheap computing has increased manifold!

8.3 Ensuring Good Passwords

There are several methods of ensuring that users pick good passwords. When we use Yaksha with short user memorizable passwords we assume that one of these methods (not all are applicable) is used. A detailed description of all these methods is beyond the scope of this thesis, and our intent here is to describe the general details, and to point to some of our results (published elsewhere) which provide much greater depth on this topic.

Here are some approaches.

8.3.1 Generate Random Passwords for Users

This is the simplest approach, instead of letting the user pick a password, the system randomly picks a password from the entire password space P, and makes the user use it. An example of such a password can be: *G6t*w@u+*. These systems have two properties:

- They have perfect theoretical security against dictionary attacks. You cannot do better.
- In practice they are likely to be resisted stoutly by users who will find them very unfriendly. This is more than just an issue of being kind to users (something which rarely bothers computer scientists!). A hard to remember password is far more likely to be written down, further it more likely to be forgotten. The first is clearly bad security practice, and the second increases overall costs.

In the past we have observed [DAVI93] that such systems are only practical in environments where security administrators can shoot users who violate security policy (by writing down passwords or complaining!). In most situations a difficult to use system is likely to result in the users shooting the perpetuators of the system, and consequently we do not recommend this scheme!

8.3.2 Generate Random *Pronounceable* Passwords for Users

A way to fix the unfriendliness of random passwords is to generate random passwords which are pronounceable. For instance the word *sheblat* is probably easy to remember (since it can be pronounced), but is unlikely to be present in an attacker’s dictionary. This “best of both worlds” scenario is alluring and such schemes are fairly prevalent. In fact there is a National Institute for Standards and Technology (NIST) standard [FIPS93] for such systems.

However, we were successful in developing a new attack against many such schemes, including a scheme used at Sandia National Laboratories and the NIST standard. The attack is so powerful that we were able to claim that the system would be more secure with user chosen passwords! The attack which is described in [GANE94a] essentially exploits a weakness prevalent in all random pronounceable password generators we examined. Namely, while the absolute size of the password space of the generators may be high, it is easy to find sub-spaces which are much smaller which will a disproportionately contain large number of user passwords[9]. We refer the interested reader to [GANE94a] for more details. While we have developed (in other work) a generator which is not vulnerable to our attack, in general we do not like solutions involving random pronounceable passwords, and do not believe it is necessary.

8.3.3 Proactive Password Checkers

Here the model is different, we let users pick their own passwords and we then develop a checker which will pass judgment on the password. The system will interact with the user during password selection and only allow 'good' passwords to be picked[10].

A naïve approach to developing such a system would be to wait until the user picks a password and to then run a dictionary attack against it. Since such attacks can take hours or days, this is impractical. Somewhat less naïve is to maintain large dictionaries of passwords and then check whether the password chosen is in it. This requires considerable space, and if bad noisy passwords are also being filtered out, considerable time. There are some other approaches (which we survey in [DAVI93]), but all of them have various weaknesses, and none of them are useful against bad noisy passwords, which as we discussed before, must be protected against.

In [DAVI93] we describe a proactive password checker, BApasswd, which runs in very fast (practically zero) constant time and space, and does not require the use of large dictionaries. We achieve this result by using Markov models to model bad passwords, and to then run classical hypothesis testing techniques to determine if a given password could be generated by that model. Given that passwords are very short (compared to recognizing lengthy text), a number of 'tricks' are needed to make this idea work. The interested reader is referred to [DAVI93] for details on the effectiveness of our proactive password checker against bad passwords and bad noisy passwords.

The use of such a proactive password checker is our preferred method of ensuring good passwords. The key reason is that this allows users to pick their own passwords, which are more likely to be remembered, and yet protects against bad passwords. For instance, in the usability testing of our system we discovered a user who picked a very good password, *vill84mth*, which sounds hard to remember until the user pointed out that he had graduated from Villonova University in 1984 with a major in Mathematics! He would never forget the password, and no checker is likely to catch the password. It is these sort of passwords that provide greatest security.

[9] Our attack is based in flaws in the generators, not in the predilection of users to pick certain types of pronounceable passwords from the choices that are provided to them. This latter fact is likely to further limit the effective password size.

[10] A related approach is for the administrator to use the same tools attackers use against a system's password files, and to see if a password can be broken, and if it can, to notify the user to change the password. This approach can be inconvenient to users and leaves open a window of vulnerability (until the bad password is detected and fixed). Consequently, we do not recommend this approach.

8.3.4 Other Approaches

Other approaches such as biometrics and the use of hand held authenticators can also be used. Some of our references discuss these in further detail, and a discussion of these is beyond our scope. For the purposes of Yaksha, when short memorizable passwords are used, a proactive password checker provides an excellent solution.

9. Conclusions

This dissertation makes the following contributions:

1. It introduces the new concept of a reusable security infrastructure, and motivates its importance.
2. It presents one complete instance of such an infrastructure, namely the Yaksha system, whose security appears to be equivalent to that of RSA.

The Yaksha system itself contains several improvements over the state of the art in the individual components, namely:

3. It has a digital signature infrastructure that permits short user keys, allows for instant revocation and provides real time audit.
4. It has the ability to do highly flexible key escrow for virtually any application. It also has the significant advantage of allowing only session keys to be revealed.
5. It can be used as the next generation Kerberos, solving the major problems, including vulnerability to catastrophic failure, inherent in the current design of Kerberos.

This work also:

6. Presented a taxonomy for dictionary attacks and a classification for good/bad passwords.
7. Described several approaches to solving the poor password problem, and surveyed some of our other results, which have been published elsewhere.

We are of the firm opinion that the Yaksha system will be eventually deployed and widely used. We expect its amazing flexibility to result in further applications that are based on its reusable infrastructure. We hope that other reusable security infrastructures will also be developed, perhaps based around other public key systems.

The net result of these developments will be vastly increased security for our information infrastructure, at lower cost and with greater user friendliness.

10. Bibliography

[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.

[CCIT88] CCIT. Recommendation X.509: The Directory - Authentication Framework. 1988

[DAVI93] Davies, C. and R. Ganesan, “BApasswd: A New Proactive Password Checker”, In *Proceedings of the 16th National Computer Security Conference*, September 1993.

[DENN81] Denning, D. and G. M. Sacco, “Timestamps in Key Distribution Protocols”, *Communications of the ACM*, Vol. 24, No. 8. August 1981.

[DENN93] Denning, D., “To Tap or Not to Tap”, *Communications of the ACM*, Vol. 36 No. 3, March 1993.

[DENN96] Denning, D. and D. Branstad, “A Taxonomy for Key-Escrow Encryption Systems”, *Communications of the ACM*, Vol. 39, No. 3, March 1996.

[DIFF76] Diffie, W. and M. E. Hellman, “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, Vol. IT-22, No. 6. November 1976.

[FIPS77] FIPS PUB 46, “Data Encryption Standard”, *Federal Information Processing Standards Publication*. 1977.

[FIPS93] FIPS PUB 181, “Automated Password Generator*”, Federal Information Processing Standards Publication*. 1993.

[GANE94a] Ganesan, R. and C. Davies, “A New Attack on Random Pronounceable Password Generators”, In *Proceedings of the 17th National Computer Security Conference*, October 1994.

[GANE94b] Ganesan, R. and Y. Yacobi, “A Secure Joint Signature and Key Exchange System”, *Bellcore TM-24531*, October 1994

[GANE95] Ganesan, R., “Yaksha: Augmenting Kerberos with Public-Key Cryptography”, In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, February 1995.

[GANE96] Ganesan R., “The Yaksha Security System”, *Communications of the ACM*, Vol. 39, No. 3, March 1996.

[KALI96] Kaliski, B., *Personal Communication*, 1996

[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.

[MCMO95] McMohan, P., "SESAME V2 Public Key and Authorization Extensions to Kerberos", *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1995

[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,* Vol. 21, No. 12, Dec. 1978.

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

[RIVE78] Rivest, R., A. Shamir and L. Adelman, "On Digital Signatures and Public-Key Cryptography", *Communications of the ACM,* Vol. 27, No. 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.

11. Index

12. VITA

RAVI GANESAN

EDUCATION

Ph.D. Computer Science, 1996 (to be awarded)
The Johns Hopkins University
Baltimore, Maryland

M.S. Computer Science, 1990
The University of Maryland Baltimore County
Baltimore, Maryland

B.E. Computer Science and Engineering, 1989
Anna University
Madras, India

PROFESSIONAL EXPERIENCE

1990 - Current Vice President, Bell Atlantic
Joined Bell Atlantic's Information Systems and have since worked on various assignments including leading various application implementation projects, leading the operations of the Center of Excellence of Electronic Commerce and then creating and leading a new Center of Excellence for Internet Services. In current position of Vice President - Distributed Operations and Information Technology, lead a team of about 500 employees in implementing and maintaining all aspects of Bell Atlantic's distributed operations, including responsibility for all aspects of security of the computing and telecommunications infrastructure.

1989 - 1990 Instructor and Teaching Assistant, University of Maryland Baltimore County
Taught and/or assisted in the teaching of various courses on Discrete Structures, Databases and LISP.

PUBLICATIONS

Ganesan, R. 'The Yaksha Security System', *Communications of the ACM*, Volume 39, No. 3, March 1996.

Ganesan, R. 'Yaksha: Augmenting Kerberos with Public Key Cryptography', *INTERNET Society Symposium on Network and Distributed Systems Security*, 1995.

Ganesan, R. (with C. Davies) ‘A New Attack on Random Pronounceable Password Generators’, *Proceedings of the 17th National Computer Security Conference*, October 1994.

Ganesan, R. ‘BAfirewall: A Modern Firewall Design’, *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, February 1994.

Ganesan, R. (with Y. Yacobi) ‘A Secure Joint Signature and Key Exchange System’, *Bellcore TM-ARH-55801*. 1994.

Ganesan, R. (with A. Sherman) ‘Statistical Techniques for Language Recognition: An Empirical Study Using Real and Simulated’, *Cryptologia*, Volume XVIII, No. 4, October 1994.

Ganesan, R. (with A. Sherman) ‘Statistical Techniques for Language Recognition: An Introduction and Guide to Cryptanalysts’, *Cryptologia*, Volume XVII, No. 4, October 1993.

Ganesan, R. (with C. Davies) ‘BApasswd: A New Proactive Password Checker’, *Proceedings of the 16th National Computer Security Conference*, September 1993. (Outstanding Paper Award).

Ganesan, R. ‘Securing Systems 2000: A Case Study of Implementing Security in Distributed Systems’, *6th International Conference on Viruses and Computer Security*, February 1993.

Ganesan, R. (with S. Weiss) ‘Scalar Memory References in Pipelined Multiprocessors: A Performance Study’, *IEEE Transactions on Software Engineering*, Volume 18, No. 1, January 1992.

EDITORIAL BOARDS

Member of the Editorial Boards of the following Journals:

- Communications of the ACM
- IEEE Transactions on Computers
- Journal of Electronic Commerce
- IEEE Communications Interactive
- ACM Transactions on Information and Systems Security

GUEST EDITOR

Guest edited following issues:

- IEEE Communications Special Issue on Securing the Information Superhighway, September 1994
- Communications of the ACM Special Issue on Securing Cyberspace, November 1994
- Communications of the ACM Special Issue on Key Escrow, March 1996

INVITED TALKS

Some recent invited talks:

- *Implementing Security in a Distributed Environment*, AT&T Bell Laboratories
- *The Electronic Marketplace: Experiences and Directions*, Keynote at the 1995 CALS EXPO '95
- *Using Cryptography to Architect Distributed Open Systems Security*, Telestrategies Conference on Encrypting Voice and Data
- *Yaksha: Towards Reusable Security Infrastructures*, Supercomputing Research Center

CONFERENCES

Served as Program Chair (1993 and 1994) and General Chair (1996) for the ACM Conference on Computer and Communications Security, and served on the Program Committee (1995) of the Internet Society Symposium on Network and Distributed Systems Security. Served as Session and Panel Chair at various conferences and led discussions on topics such as *Electronic Cash*, *Electronic Commerce Security Technologies*, *Telecommunications Security*, *Corporate Key Escrow*, and *Payment Mechanisms on the Internet.*

PATENTS

Several technical inventions have either already been patented or are in the patent process.

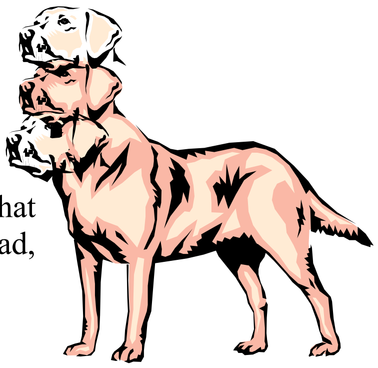
PROFESSIONAL ORGANIZATIONS

Member of the Association of Computing Machinery (ACM) and of the Institute of Electrical and Electronics Engineers (IEEE).

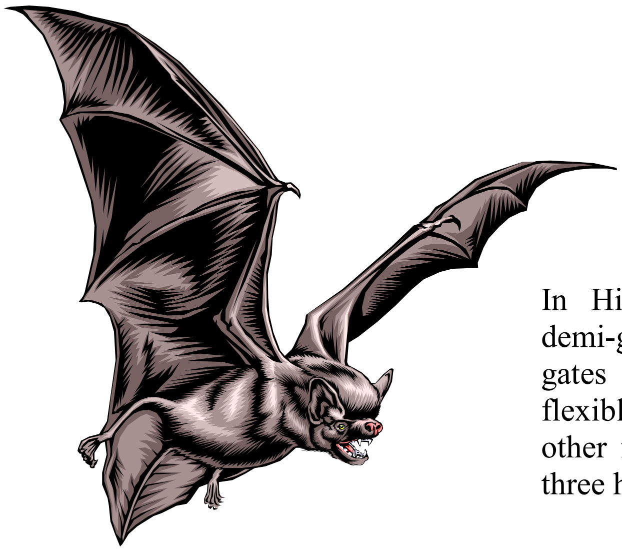
HONORS

- University of Maryland's Graduate Merit Fellowship
- Outstanding Paper Award, for 'BApasswd: A New Proactive Password Checker', *Proceedings of the 16th National Computer Security Conference*, September 1993

13. A Lesson in Mythology…



In Greek mythology, Kerberos is the three headed dog that guards the gates of Hades, “the land of the dead, underworld”.



In Hindu mythology, Yakshas, 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.

Moral…

To guard the “land of the dead” current security technology such as Kerberos will do. To protect something truly precious, like the gates to heaven, one needs Yaksha!