The following paper was originally published in the
Proceedings of the Sixth USENIX UNIX Security Symposium
San Jose, California, July 1996.

Texas A&M University Anarchistic Key Authorization (AKA)

David Safford, David K. Hess, and Douglas Lee Schales
Texas A&M University

Texas A&M University Anarchistic Key Authorization (AKA)

David Safford, *Texas A&M University (Dave.Safford@net.tamu.edu)*
David K. Hess, *Texas A&M University, (David-Hess@net.tamu.edu)*
Douglas Lee Schales, *Texas A&M University, (Doug.Schales@net.tamu.edu)*

Abstract:

At the Fourth USENIX Security Conference, we presented a paper on SRA Telnet, which was a simple Diffie-Hellman based system to defeat standard password sniffing, without requiring externally validated keys. Since that time, several projects, such as Secure Telnet (stel), have worked to extend this simple Diffie-Hellman model to add data encryption, larger keys, and resistance to man-in-the-middle attack. Other projects, such as SSL and SSH use validated RSA keys for full authentication. This project uses standard PGP keys as the basis of unified authentication, authorization, and encryption, combining both perfect forward secrecy and strong RSA authentication.

1. Introduction:

AKA has several design goals, including:

- RSA authentication based on an individual's standard PGP key
- Perfect forward secrecy
- Unified, powerful authorization across applications
- Encryption of all authentication and application data
- Immunity to both passive and active attacks
- Full mutual repudiation
- Key security (An individual's secret key never leaves the desktop)
- Distinction between an individual and his account(s)
- Interoperability with unmodified telnet and ftp systems
- Socket layer implementation for ease of incorporation
- Compatibility with connection oriented services

The remainder of the paper presents AKA in five major sections:

- The basic cryptographic protocol
- The authorization model and file format
- The software architecture
- Comparison to other systems
- Summary

II. AKA Basic Cryptographic Protocol:

For AKA, we realized that a good way to provide true mutual authentication without shared secrets was with an RSA based public key system with authenticated public keys. In the past, RSA has been avoided because of the difficulty of providing authenticated key service, and because of the patent status of RSA. Both of these issues have been largely overcome by the release of MIT PGP-2.6.2 [8]. AKA's use of PGP keys provides both essential secure authentication functions; mutual authentication by RSA signing of challenges, and secure encryption of authentication data by RSA encryption with the destination's public key. This system is "anarchistic" in that an individual provides and certifies his own keys, and obtains the server's public key. Neither the user nor the server system need trust any third party Certificate Authority (CA), or Secure DNS service, although these services may be used to obtain keys initially. The AKA protocol provides support for selectable levels of security versus convenience in PGP key initialization.

Perfect forward secrecy means that breaking a secret key will at most compromise one data session; breaking an individual's secret PGP key should not allow decryption of previously recorded sessions. Normally Diffie-Hellman is used with one-time keys to provide perfect forward secrecy, although AKA (and to some extent SSH) use one-time RSA keys in a similar fashion.

The basic design of AKA is very simple. An individual has standard PGP key rings with his protected secret key, his public key, and the public key of a remote service. Similarly the remote service has a secret key, public key, and a copy of the individual's public key. These fixed PGP keys are used only for authentication; separate one-time keys are used to RSA encrypt the AKA exchange to provide perfect forward secrecy. A simple bidirectional random challenge-response exchange is made (to prevent replays from either side), with each item signed with the local side's secret key for authentication, and then encrypted with the other side's one-time public key for confidentiality. (PGP actually signs a hash of the item, to avoid a chosen text attack.) The verified random challenges are then xor'ed to form a session key for a traditional data cipher.

Given a user 'A' wishing to authenticate to remote service 'X', the following high level data flow diagram shows the basic interactions:

```
A                          X
-------- RPKa ------------> (exchange one-time
<------- RPKx ------------- public keys)

------- RPKx(Ra) ---------> (challenge to X)
<----- RPKa(Sx(Ra)) ------- (signed response)

<------ RPKa(Rx) ---------- (challenge to A)
------ RPKx(Sa(Rx)) ------> (signed response)

<--- (Rx xor Ra)(data) ---> (idea/des encrypt
                            all data)
```

where RPKn(R) means RSA encryption of 'R' with one-time public key 'n', Sn(R) means PGP signature of R with secret authentication key 'n'. This basic design has several interesting properties. First, authentication is based on the ability to sign a random challenge with the secret key corresponding to a given public key. The public key signature demonstrates possession of the secret key without exposing it to eavesdroppers, so simple sniffing is ineffective. Since both ends already have the other's public key, neither end can be spoofed. As all exchanges are encrypted with the other side's one-time public key, man-in-the-middle or hijacking is impossible without at least one side's secret key, and perfect forward secrecy is preserved. Authentication based on PGP signatures eliminates the need for any other form of identification, including reusable or one-time passwords, and smartcards.

The use of standard PGP key rings puts control of the keys in the hands of the users, and leverages the key management facilities of the PGP package. An individual's secret key never leaves the local host, and need never be subject to compromise on remote systems.

Addition of Key Initialization Support:

The basic protocol given assumes that the necessary validated authentication keys are already in place. One enhancement to the basic protocol adds an initial key exchange phase to provide the option of key initialization. This modifies the data flow to look like:

```
A                          X
-------- RPKa, PKa--------> (exchange one-time
<------- RPKx, PKx--------- public keys)

------- RPKx(Ra) ---------> (challenge to X)
<----- RPKa(Sx(Ra)) ------- (signed response)

<------ RPKa(Rx) ---------- (challenge to A)
------ RPKx(Sa(Rx)) ------> (signed response)

<--- (Rx xor Ra)(data) ---> (idea/des encrypt
                            all data)
```

Given the availability of unauthenticated keys through this exchange, the client and server side are free to:

- Accept only authenticated keys on existing key rings
- Accept keys if they match those obtained from secure DNS
- Accept transmitted user key based on traditional password (once)
- Accept transmitted keys without validation

The third option provides a convenient method of initializing a user's public key on the server side. In this scenario, a user has (or creates) a PGP key pair on his local machine. The administrator of the remote machine creates a typical user account with traditional password. When the first connection is made through AKA, the provided PKa is conditionally accepted, and installed as a validated PKa only after the user provides his traditional plaintext password. The traditional password is not compromised, as it is transmitted only after the AKA protocol has established the conditionally accepted encrypted stream. In any case, once the validated PKa is installed, the plaintext password will no longer be accepted. All subsequent logins would require full AKA authentication.

The fourth option would provide a similar service to the existing unauthenticated key Diffie-Hellman systems, in which encryption provides only confidentiality to some other authentication system.

Modifications of the protocol for efficiency:

The modified base protocol as presented is still not the final design; it needs to be modified for efficiency. An amazing number of protocols are not organized to parallelize the time consuming encryption operations on both ends of the connection, but rather serialize them. The basic AKA design presented so far, does in fact serialize the encryption operations:

```
           A                  X
           ---- RPKa, PKa---->
           <--- RPKx PKx------
(encrypt)
           ----- RPKx(Ra) --->  (decrypt
                               sign, and
                               encrypt)
           <-- RPKa(Sx(Ra)) --
(decrypt &                      (encrypt)
validate)
           <--- RPKa(Rx) -----
(decrypt
Sign and
encrypt)
           --- RPKx(Sa(Rx) -->
                                (decrypt and
                               validate)
           <---(Rx xor Ra)--->
```

Here we have 12 RSA operations, requiring 11 sequential time periods. The basic data flow can be reorganized for significantly faster operation:

```
             A                   X
             ---- RPKa, PKa---->
             <--- RPKx PKx------
(encrypt)                         (encrypt)
             ----- RPKx(Ra) --->
             <--- RPKa(Rx) -----
(decrypt                          (decrypt,
sign and                          sign and
encrypt)                          encrypt)

             <-- RPKa(Sx(Ra)) --
             --- RPKx(Sa(Rx) -->
(decrypt &                        (decrypt and
validate)                         validate)

             <---(Rx xor Ra)--->
```

This results in the same 12 RSA operations requiring only 6 time periods.

Another optimization recognizes that creation of 1024 bit ephemeral RSA keys is a time consuming process (around 10 seconds). While it is necessary to use at least one ephemeral key to obtain perfect forward secrecy, it is not necessary for both sides to use one. AKA therefore has the server side pre-generate an ephemeral key, and does not generate on the client side. This way, the initial key exchange need not wait for generation of the ephemeral keys. (SSH uses a similar ephemeral server key, although it generates a new key only once per hour, rather than once per connection, and the ephemeral key is shorter (768 bits). This weakens the perfect forward secrecy, and may allow replay attacks.) Since the challenge Rx from X to A is now encrypted in A's permanent public key, A need not sign Rx to authenticate; simply decrypting PKa(Rx) is sufficient. Therefore A simply returns the hash of Rx to authenticate.

```
A                   X
------ PKa-------->
<--- RPKx PKx------

----- RPKx(Ra) --->
<---- PKa(Rx) -----

<--- PKa(Sx(Ra)) --
---- RPKx(H(Rx) -->

<---(Rx xor Ra)--->
```

Account Considerations:

This section addresses four extensions to the basic model of the identity of the client 'A' and the server 'X':

- separation of principal from account
- default account mapping
- anonymous access
- multiple service keys

A modification to the basic protocol involves support for a separation of the name of the individual (a) on the client from the name of the account (u) he is requesting on the server. While the PGP key identifies (a) with a combination of User ID and Key ID, it does not provide a desired (u), so this information must be sent across. The problem is that we do not want to send this information across before encryption is established, but the information may be needed for the server to validate the provided PKa. For example, the validated key PKa may be stored in the user (u)'s home directory, so the server will need to know (u) before authentication can complete.

Both goals can be achieved by conditionally accepting PKa, and validating the key one step later, as in:

```
A                   X
------ PKa-------->
<--- RPKx PKx------

---- RPKx(Ra,u) -->
<---- PKa(Rx) -----
                      (now validate
                     PKa based on u)
<--- PKa(Sx(Ra)) --
---- RPKx(H(Rx) -->

<---(Rx xor Ra)--->
```

Given potentially multiple mappings from principal to account, a second extension allows for the concept of a "default" account mapping. In this case 'u' may be flagged as requesting the default map, rather than having to name an account specifically.

A third extension is to support the concept of anonymous connections. In this case, a null PKa is sent, the requested 'u' is flagged as anonymous, and the server to client challenge is omitted. The server still uses a combination of authentication and ephemeral keys, although lacking a client key, the server's contribution toward the session key cannot be sent securely to the client, so the client chooses all of the session key. (This is essentially the normal SSH protocol, but with a new ephemeral key per session.)

A fourth consideration for account mapping is to recognize that AKA servers are free (and encouraged) to use a separate service principal for the purpose of selecting the service key. All servers could use a "root" principal and key, for simplicity, but separate principals would be more secure, as the service's secret key must be available to the server in unencrypted form.

III. The authorization model

Existing authorization mechanisms suffer from two significant problems: they are scattered among an unmanageable number of diverse configuration files, and they are frequently based on unreliable information, such as IP address.

The first problem in most systems and especially Unix is the plethora of different mechanisms that exist for service authorization. The following table is a sample of some of the more common mechanisms.

Service	Authorization
many	/etc/inetd.conf
many	/etc/hosts.allow
	/etc/hosts.deny
many	sshd_config
login	/etc/default/login
	/etc/ttytab
login	/etc/skeykeys
r-cmds	~/.rhosts
r-cmds	/etc/hosts.equiv
FTP	/etc/shells
FTP	/etc/ftpusers
FTP	ftpaccess (wuftpd)
FTP,login,POP	/etc/{passwd,shadow}
SMTP	sendmail.cf
	(trusted users)
NFS	/etc/exportfs
	/etc/dfs/dfstab
NFS	nfs_portmon
	(kernel variable)
HTTP	access.conf
lpd	/etc/hosts.lpd
X Windows	xhost
X Windows	.Xauthority
UUCP	USERFILE,L.cmds,...
portmapper	securenets
NTP	ntp.conf
NIS	ypbind (-broadcast)
NIS	ypserv (securenets)
SNMP	community string
TFTP	chroot

The second problem is that traditional network authorization mechanisms typically center around four different types of information: source host, source port, account requested, and service requested. The principal requesting access has to provide a certain combination of these credentials in order to gain access. The source host and source port information is unreliable, and should not be used.

Trying to develop a coherent security policy out of all these mechanisms is daunting.

Rather than introduce yet another authorization mechanism that adds to the mess, AKA tries to define a philosophy and a mechanism that is flexible enough to meet the authorization needs of most services. It is hoped that over time this mechanism can become the standard for user authorization.

There are some key elements to AKA's authorization philosophy. First, AKA discards the philosophy of authenticating via accounts and instead uses principals, where a principal is uniquely identified by the possession of a given secret key with a specified keyid. An individual authenticates a claim to be a given principal by signing a challenge with the principal's secret key. An authenticated principal is allowed to use any account for which it has been authorized.

Also, since the AKA mechanism is based on principals, the concept of including a host and port as part of the authentication mechanism has been removed. Currently this information is unreliable (though IPv6 addresses this problem) and this information does not contribute anything useful to a strong cryptographic authentication of a principal.

The AKA authorization mechanism works with an authorization tuple of (<principal>, <service>, <accounts>, <restrictions>). <principal> corresponds to the entity requesting the service, <service> is the service being requested, <accounts> are the accounts that principal is authorized to use for this service, and <restrictions> are the service specific authorizations (required chroot paths, allowed operations, etc.).

The authorization mechanism defines special tokens for the special case principals of "anonymous", in the unauthenticated case, and "wildcard", to match any (non-anonymous) principal.

All other information pertaining to a service request is considered to be useful for service management, such as the path to the service executable, execution arguments, etc. This information, which is not related to authorization information should be stored in a separate database or file from the authorization information.

IV. The software architecture

A number of software architectures have been used by other projects, such as replacement library (SSL), service wrapper (tcpwrapper [5]), dedicated daemon (sshd), and fully integrated (SRA telnet and ftp). Each approach has certain advantages and disadvantages. The fully integrated approach of SRA has the advantage of full interoperability between modified and unmodified clients and servers, as the use of the service is enabled within the existing application's option negotiation. On the other hand, integration of a new protocol into an existing application is a lot of work, and some applications may not have existing negotiation mechanisms. Tcpwrapper is the easiest to implement as it requires no modifications to existing executables, but it is limited, as the only authorization information available is based on the IP connection addresses. SSHD has the disadvantage of complete lack of interoperability with existing applications, but this offers it great freedom of implementation. The SSL library approach transparently adds secure socket communications, without modifications to the application code, but does have interoperability problems between modified and unmodified versions.

For AKA, we wanted to satisfy the following architectural goals:

- complete interoperability between modified and unmodified applications.
- little or no modification to application code
- maintain a single point of authorization across all services, whether or not they have been modified.

To satisfy these goals, AKA uses a combination of a replacement socket library and an AKA specific service daemon, which also acts as an ephemeral key service. A client application is linked with the AKA socket library. This library attempts to contact the AKA daemon on the remote server. If there is not an AKA daemon available, the client asks the user if an unsecure connection is ok, and if so attempts to establish a traditional connection.

When a client does successfully contact the AKA daemon, it informs the daemon of the service it is requesting. The AKA daemon then determines how to establish that service. There are three distinct situations that can occur.

The first situation is when the client is attempting to access a service that directly supports AKA, but is not memory resident (it must be started). In this situation, the AKA daemon will behave like **inetd** and will *dup* the socket to the appropriate file descriptors and *exec* the service daemon. The service daemon will request an ephemeral key from the AKA daemon (through a separate channel), and then will execute the AKA protocol. The AKA daemon will be removed from all interaction with the session. This method works when the service daemon is not resident, but in situations where an AKA capable service daemon is itself waiting for network connections, a different technique is necessary.

In this situation, where the service daemon is actively accepting connections, the AKA daemon will not be able to simply use exec to start the service. There are multiple solutions to this. The most efficient is to pass the file descriptor for the session to the service daemon. On many UNIX systems, this can be accomplished through a UNIX domain socket. By passing the file descriptor to the service daemon, the AKA daemon can be removed from having to interact with the session. In situations where this is not possible, the AKA daemon can establish a new connection to the service daemon via a loopback address. The AKA daemon will simply act as a data relayer. The service daemon will have to recognize that connections from loopback are always AKA sessions. When implemented in this manner however, the AKA daemon will remain active in the session, as the data relayer, introducing additional load on the server side.

When the service daemon has not been modified to support AKA, it is still possible to derive some of the benefits of AKA. The AKA daemon will simply connect locally to the service via a loopback connection, and then will sit in front of the service, implementing the AKA protocol. This will introduce additional load on the server side, since an additional process is involved in the communication. Also, the level of AKA support will vary based on the particular service.

Currently the encryption/decryption function will be implemented in either an AKA daemon process, or in the application server process (depending on whether or not the server is AKA modified). It is anticipated that the encryption will eventually be moved into a streams module, or handled by the transport layer (IPv6) for greater efficiency.

V. Comparison to other systems:

Comparison to DES based systems:

Kerberos [4]

In many ways, AKA is the exact opposite of Kerberos. Kerberos depends on a central server with everyone's secret key, while AKA allows users to maintain their own keys in a distributed fashion, with just one copy of the secret key needed on the first machine. Kerberos uses strictly secret key encryption, while AKA uses public keys. Kerberos has difficulties with inter-realm authentication, while AKA does not even have realms.

deslogin [1]

deslogin uses a shared plaintext secret on both ends for one way (user −> server) authentication based on a challenge-response. While it does prevent sniffing and man-in-the-middle attacks, it does not prevent server spoofing, and requires shared plaintext secrets. Since it does not encrypt the full telnet session, it is subject to hijacking or injection.

Comparison to unauthenticated-key Diffie-Hellman systems:

A secure authentication system must do two major things: first it must provide a secure session key for encryption of the service connection, and second it must provide for mutual authentication. Diffie-Hellman, particularly when augmented with the interlock protocol, is a good technique to obtain a common session key. It does nothing, however for authenticating either end of the connection; it simply provides a secure channel over which other authentication methods can be used. The encrypted channel does prevent sniffing, hijack, and man-in-the-middle attacks, but does nothing to prevent client or server spoofing. Authentication of the client (user) to the server must be done with one of the traditional schemes, such as plaintext passwords, or one time passwords (s/key, SecureID...). The problem is that someone could spoof the server to the client and spoof the client to the server, passing the respective challenges and responses between the two separate connections. Diffie-Hellman, even with interlock cannot prevent this "spoof-in-the-middle" form of attack, because it inherently has no way to ensure that it is talking to the right system - it only knows that it is talking securely to whomever is on the other end.

SRA [3]

SRA was a first attempt at a simple Diffie-Hellman based secure authentication system. At the time it was recognized to be susceptible to server spoofing and man-in-the-middle attacks, but these were considered minor compared to the known threat of password sniffing. In addition, its key length was chosen to be compatible with Sun's NIS, which at 192 bits is quite small. Since it does not encrypt the full telnet session, it is subject to hijacking.

Stel [6]

Stel overcomes active attacks, but only at a price; it uses a shared plaintext secret to encrypt the interlock exchange to authenticate both ends to each other, while simultaneously denying man-in-the-middle. Having to have plaintext secrets on all hosts is not very desirable. If you do have them, then why bother with Diffie-Hellman/interlock in the first place? Why not use a simpler and faster challenge/response system directly based on the secret, in a similar (but bidirectional) manner to deslogin?

Comparison to other RSA based systems:

SSL [2]

SSL is based on X.509 (or other) style CA services, which while acceptable for large web based services, is not desirable for individuals, or large numbers of systems, due to the current lack of a standardized CA hierarchy, and the difficulty and expense of registration. In addition, the basic protocol allows the client to choose the session key, which could provide a possible attack. The basic protocol does not have perfect forward secrecy, although version 3 does allow optional Diffie-Hellman. On the other hand, SSL is well suited to the typical secure web model, where many unauthenticated users connect into a relatively small number of certified servers.

SSH [7]

While SSH is similar to AKA in many respects (it is based on individual RSA keys), it does not provide a separation of individual from account, it does not provide a true challenge of the server, it is rsh specific, and does it not interoperate with existing rsh implementations. In addition, its establishment of session key is similar to SSL, in that it allows the client to pick the session key. It attempts to limit key replay exposure to one hour through a periodically changing an ephemeral server public key, but a better approach would be to base the session key on a combination of both client and server information, so that replays were not possible in the first place. The limited duration (one hour) ephemeral key does provide some perfect forward secrecy, although all sessions for the lifetime of this ephemeral key would be vulnerable, rather than the preferred one random key per session. While SSH is very attractive in its ability to forward X11 and other tcp ports, this added power and complexity may open up security holes, and makes SSH difficult to verify.

VI. Summary

AKA provides true mutual authentication that is resistant to sniffing, spoofing, man-in-the-middle, and hijacking attacks, and that does not require shared plaintext secrets. The use of standard PGP keys and key rings makes key management relatively painless. AKA provides fully interoperable services with a single point of authorization for all modified or unmodified services.

In addition to the basic feature of strong distributed authentication, AKA provides some powerful new features:

- Unified authorization.
- Separation of identity from account.
- Key management and security.
- Interoperability with unmodified services.

Unified authorization is an important goal for any security system. One of the problems securing machines is that there are so many different configuration files that must be tailored/checked to ensure security. AKA introduces a unified authorization switch control file, with a unified syntax, to simplify the configuration task. From the application perspective, a single simple API provides all authentication and authorization information through this combined authentication and authorization service.

AKA separates the concepts of identity and host account. An individual is authenticated based on possession of a secret key that matches a given public key. This individual may then be authorized to use one or more host accounts. For example, you do not authenticate the root account, you authenticate “joe”, who may in turn be authorized to telnet (or ftp...) into the root account. Most systems disallow directly telnetting to root, instead insisting on logging in as an unprivileged account, and then su’ing to root. AKA allows direct telnet as root, while logging the principal involved.

AKA provides powerful key management and security features. First, AKA uses standard PGP keys, and thus all of PGP’s key management facilities. An individual’s PGP key pair can be used for both PGP and for all AKA services. An individual’s key is entirely under that individual’s control, and the secret key need never leave the user’s desktop host. Keys are fully distributed - there is no centralized key server.

Unlike SSH, AKA is designed to be implemented to interoperate with existing Telnet and FTP clients and servers. The modified clients and servers negotiate the availability of AKA and can fall back to traditional methods if the other end does not support AKA, and the user/administrator allows. Thus the users need not keep track of which systems support AKA, and use different commands depending on the situation.

Status and Availability:

An initial prototype of AKA is in progress, and further information is expected by the time of the conference. An extension of protocol to handle multi-hop connections is in internal review.

References:

[1] Barrett, D., “Deslogin”, ftp://ftp.uu.net:/pub/security/des/deslogin-1.3.tar.gz

[2] Netscape, “SSL Protocol”, http://home.netscape.com/newsref/std/SSL.html

[3] Safford, D., Schales, D., and Hess, D., “Secure RPC Authentication for Telnet and FTP”, Proceedings Fourth USENIX Security Symposium, October 1993, Santa Clara, CA, pp 63-67.

[4] Steiner, J., “Kerberos: An Authentication Service for Open Network Systems”, Proceedings Usenix Conference, Winter 1988, Dallas TX, pp191-202.

[5] Venema, W., Tcpwrapper, ftp://ftp.win.tue.nl/pub/security/tcp_wrappers_7.4.tar.gz

[6] Vincenzetti, D., “STEL: Secure TELnet”, Proceedings Fifth USENIX Security Symposium, June 1995, Salt Lake City, UT, pp 75-83.

[7] Ylonen, T., “SSH”, http://www.cs.hut.fi/ssh

[8] Zimmermann, P., “PGP”, http://web.mit.edu/network/pgp.html