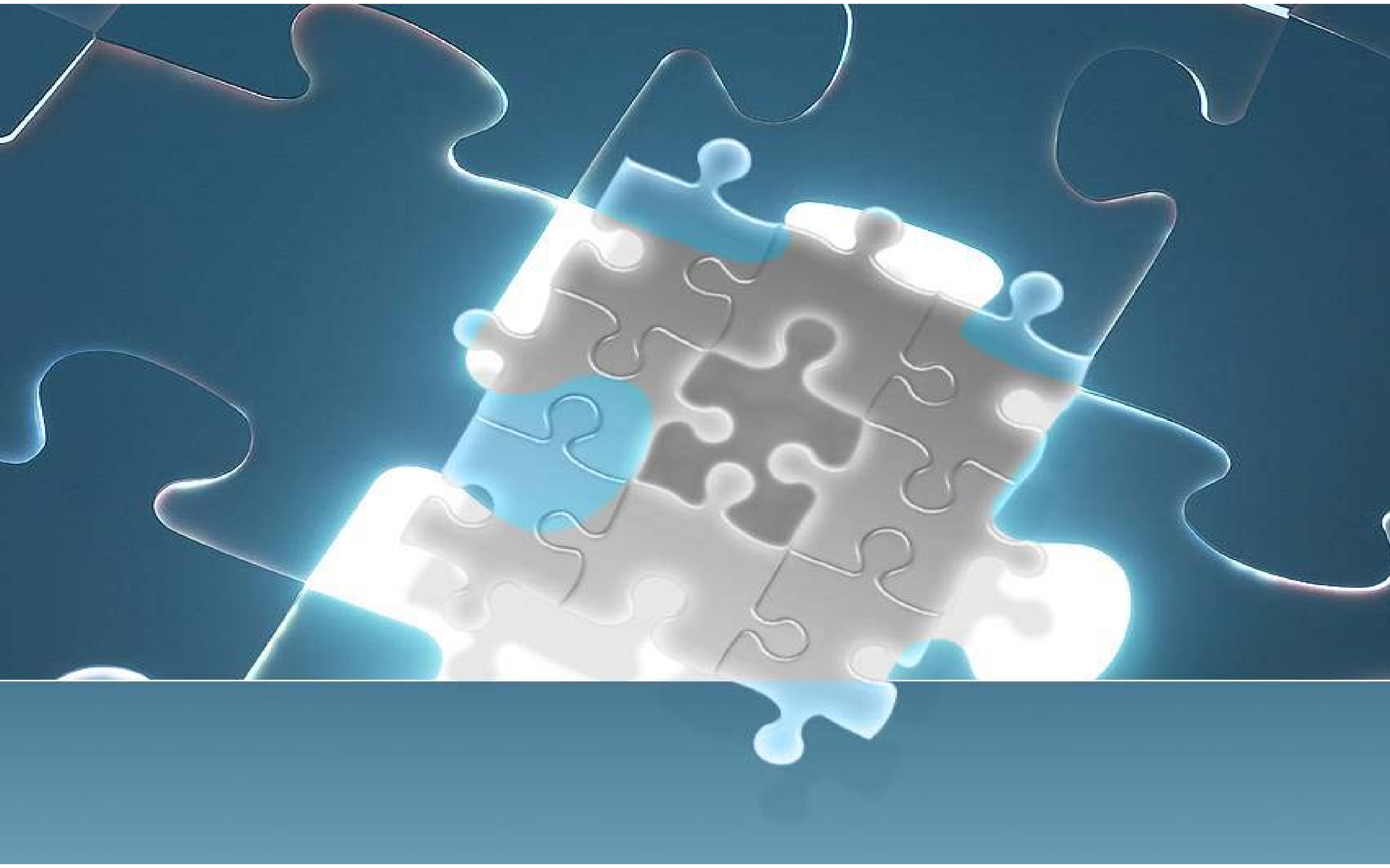

UNSECURED SSH – THE CHALLENGE OF MANAGING SSH KEYS AND ASSOCIATIONS

By Greg Kent (Senior Vice President Commercial Services) and
Bhavna Shrestha (Senior Consultant)

UNSECURED SSH – THE CHALLENGE OF MANAGING SSH KEYS AND ASSOCIATIONS

INTRODUCTION

The Secured Shell (SSH) service is widely deployed to provide secured connectivity between systems. In other words, SSH is the secured alternative for telnet or ftp services, which are clear text and could expose user credentials and sensitive network traffic to eavesdroppers. SSH provides an encrypted tunnel through which users can enter commands, transfer files, or even use an X Windows graphical users interface. For many years, auditors have been advocating wide deployment of SSH as a cost-effective solution to the security problem of clear text network transports. OpenSSH is the most commonly deployed implementation of the SSH protocol. The price is right – it's free – and it does not require the complexities of a Public Key Infrastructure (PKI) for generating keys. However, many organizations that have large OpenSSH deployments have found that SSH can introduce new security problems that can be as significant as the problem of clear text transmissions.

SSH provides a wide-range of authentication mechanisms. SSH supports password authentication, host-based authentication (which is generally avoided), and authentication via public and private key pairs for users. The user key-based authentication option played a big part in the wide adoption of SSH. Although some of the support for SSH keys was driven by a sincere belief that keys provide greater security than traditional passwords, it was the ease-of-use argument that helped make the switch to SSH palatable to system administrators and other users. Users would not have to remember passwords anymore, but could jump from machine to machine using keys – what could be easier than that. Although keys helped make it easier to sell SSH, the uncontrolled deployment of SSH keys has been the cause of significant security problems for large organizations.

Since keys are used for authentication, good security practices require that organizations definitively know who can use the keys to connect to accounts. Questions like "who does this key belong to?" or "who can use this key to connect to that account?" are critical, but often very difficult to answer. Establishing management controls over SSH keys was too often neglected when organizations began deploying keys. As a result, many organizations with legacy SSH deployments with thousands of keys and millions of associations spread across hundreds of servers face a huge security problem that exposes them to a significant amount of risk. In order to properly secure and manage SSH, traditional controls need to be applied to key associations, including centralized provisioning and de-provisioning, periodic recertification, and centralized tracking of user entitlements. These reasonable steps can help organizations manage and control the deployment of OpenSSH keys and key associations.

BACKGROUND ON SSH KEYS AND ASSOCIATIONS

SSH uses asymmetric keys, that is, a public-private key pair, to authenticate users. There is a mathematical relationship between the private key and the public key. In SSH, public keys are published on remote systems to indicate users that are allowed to connect. Private keys are controlled by and should be accessible only to the key owner. When a user attempts to connect, he provides a private key. SSH authenticates users by verifying the relationship between the published authorized public key and the private key provided by the user.

OpenSSH was designed to provide ease-of-use for organizations that do not have an enterprise PKI capability. In most instances when public/private key pairs are widely used in an organization, PKI provides the mechanism for issuing, tracking, managing, renewing, and suspending keys. However, establishing a PKI capability can be quite challenging and expensive. As an alternative, OpenSSH supports a self-service model for key deployment, which makes it easy to adopt and implement. SSH provides a key generating utility that can produce key pairs as needed. In other words, the default implementation of OpenSSH allows users to create their own keys and key associations. This self-serve key management model is the root cause of the problem. Any security process that is self-serve can quickly degrade into an uncontrolled and unmanaged mess, which is precisely what many organizations have painfully experienced with SSH keys.

When considering key management with SSH, it is important to maintain the distinction between the following two components:

- First, there is a key identity, which identifies the user who the key belongs to. This consists of a private key that is created for a user. (The public key is stored as part of the private key and can be extracted/generated from the private key on demand.) In terms of an SSH connection, the key identity is related to the source account on the source system, which we will call "From Account" and "From System" in the rest of this article, respectively. Conceptually, the key identity represents the key's owner. By default, the identity key resides within the user's home directory, so the key's owner is the owner of the home directory.
- The second key management component is called an authorized key. One or more authorized keys define the public keys that are authorized to connect to a system as a particular user. The authorized key defines an association (or trust relationship) via the public key(s). Any user that submits a private key associated with an authorized public key is allowed to logon. In terms of an SSH connection, an authorized key is related to the destination account on the destination system, or as we will call them, the "To Account" and "To System", respectively.

The following table summarizes the elements of a key association:

Key components	Purpose	Default Location	Associated SSH Configuration Element
Key Identity	A private key for a "From Account" on a "From System" that is generally located in the home directory of the "From Account"	~/.ssh/id_rsa or ~/.ssh/id_dsa	IdentityFile in SSH client configuration
Authorized Key	A public key that is trusted by a "To Account" on a "To System," such that any user with the corresponding private key is authorized to log on as the "To Account"	~/.ssh/authorized_keys	AuthorizedKeysFile in SSH server configuration

A key association, which essentially defines a trust relationship, always involves two systems (a "From System" and a "To System") and two users (a "From Account" and a "To Account"):

- The connection is initiated by the "From Account" on the "From System."
- The destination of the connection is the "To System" using the security context of the "To Account."
- The "To Account" controls who is authorized to connect to its account through authorized keys. The public keys that are authorized to connect to the "To Account" are listed in the authorized keys file.
- If the private key of the "From Account" corresponds to an authorized public key, then the connection is allowed.

The path name of the file containing authorized keys is specified by the AuthorizedKeysFile sshd configuration parameter. The authorized keys file contains one line per key and can include additional restrictions on how users can connect and what they are allowed to do once they have connected. The public keys contained within an authorized keys file define a key association, which is a relationship between a "To Account" and a public key.

It is important to note that the authorized key association is defined only via a public key, not to a specific "From Account." Anyone in the network who has a private key that corresponds to the authorized public key can use the association to connect through SSH. The fact that keys, rather than specific user names, are defined in the authorized keys file makes it difficult to determine exactly which users are trusted to connect[1]. Good security practice requires that organizations know who has access to connect to their system, and this is a common sense question that any auditor would ask. It is not particularly helpful or reassuring to answer that anyone with a corresponding key (whoever that might be) can access the system. However, many organizations know little more than this because the default implementation of SSH does not provide sufficient visibility to provide a more definitive answer to the question of who has access. This itself is a significant problem.

ISSUES WITH USING SSH KEYS ASSOCIATIONS

User's Control over Key Creation

The default implementation of OpenSSH allows users to create their own keys with an easy-to-use utility called ssh-keygen. The private key file, called an identity file, is stored in a directory within the key owner's home directory. By default, a user's private SSH key is stored in the ~/.ssh/id_rsa or ~/.ssh/id_dsa file within the user's home directory, depending on whether RSA or DSA public/private key pairs are used. The corresponding public key is stored as ~/.ssh/id_rsa.pub or ~/.ssh/id_dsa.pub.

Users can modify this default directory and define where their key identities are stored. The location of the user's private key identity is defined through either configuration parameters of the SSH client or

[1] The ssh-keygen utility that creates key pairs can include a comment field showing user@host at the end of public keys. When this field exists, the user@host does not enforce any restrictions on access – it is just a comment. The comment is added when the key is created, so it is not changed if a key is copied to another computer or sent to another user. For this reason, the comment is not helpful in identifying who is using the key or where they are connecting from. In addition, the comment can be changed easily (with the –C option) and therefore the comment is often inaccurate. Therefore, the comment in public keys should not be relied upon as providing any meaningful security protection. It may provide a helpful clue as a starting point for investigating the key, but is not reliable beyond that.

run-time parameters. Configuration parameters for the SSH client program can reside in two places: a centralized system-wide configuration file located at /etc/ssh/ssh_config, or a user-specific configuration file ~/.ssh/config stored in each user's home directories. The IdentityFile <identity_file_name> parameter can define one or more identities that are checked in sequence. There is also a run-time option (e.g., -I <identity_file_name>) that can be used to specify an identity when the SSH client is invoked.

This provisioning model puts control in the hands of the user, and there is little that can be done about it. Anything that is stored in user's home directories is ultimately accessible by the user. Users can issue themselves keys, even multiple keys, and store them anywhere they want. It is, of course, possible to make changes to the system-wide ssh_config file, but users can override those settings by specifying run-time options or using a ~/.ssh/config file, both of which are completely under the user's control. The bottom line is that users can create their keys (e.g., "self-serve") and store them anywhere in the file system. Because keys can be distributed throughout the file system (at least in users' home directories and possibly in other directories as well), it is difficult to find all of the SSH identities that exist on a host. This creates a significant challenge when trying to determine the identity of a particular key's owner. The identity of the corresponding private key could be located throughout the file system of any host in the network that runs an SSH client.

Protecting Private Keys from Unauthorized Use
To provide a reliable authentication through keys, SSH private keys need to be protected from unauthorized use. Two mechanisms are available for protecting keys (passphrases and security permissions), but the "self-serve" model of deploying keys can undermine both of these controls. Passphrases are generally used as a second level of control for key-based authentication: security by what you have (e.g., a key) and by what you know (e.g., a passphrase). When a passphrase is entered, SSH encrypts the private key with 3DES to protect it against compromise. Each time the user runs the SSH client to initiate a connection, the system will prompt for the user's passphrase in order to decrypt the private key. If the user does not provide the correct passphrase, then SSH blocks the connection. The OpenSSH utility for creating public/private key pairs always prompts users to enter a passphrase that must be entered in order to use the private key. However, the key generating utility accepts null passphrases, that is, no passphrase.

Since a passphrase is entirely optional, as can be imagined, most users who create their own keys do not use one. A small minority of users who intend to follow good security practices might set a passphrase, but often use phrases that are too short or otherwise too easy to crack. The documentation for SSH key generation program (called ssh-keygen) contains sound guidance for setting a passphrase: "Good passphrases are 10-30 characters long, are not simple sentences or otherwise easily guessable, and contain a mix of upper and lowercase letters, numbers, and non-alphanumeric characters." Few users will follow that advice. A key with no passphrase is more susceptible to compromise.

When no passphrase exists, the only control to protect private keys from unauthorized use is access controls. The permissions on the private key file should be restrictive to allow only the key's owner (and root) to have any access. Permissions are especially important for keys that are stored in the default locations – these are known targets for attackers looking to round up unprotected keys.

It is also worth pointing out that there is no way to recover a lost passphrase. If a user loses or forgets their passphrase, their existing keys are unusable. In this event, the user must generate a new key pair and copy the public key to the authorized_keys files on target SSH servers. This can result in the existence of multiple pairs of keys for the same user. In the "self-serve" model, it is unlikely that users will clean-up the environment by removing the unusable keys, so key associations can quickly get cluttered.

Protecting Keys Used by System Accounts
System accounts are used by operating systems, middleware, databases, and applications for running processes and allowing system components to communicate with each other. For example, an application may use a system account (applprod) to run an application process, which uses another system account (appldb) to access a backend database, whose DBMS processes are running under an "oracle" account. Each of these system accounts is likely to have sensitive levels of access to system resources. The "applprod" account, for instance, is likely to have the ability to create, delete, and update files related to the application.

Keys assigned to system accounts present a special challenge since there is no human to type a passphrase when the SSH client is initiated. One alternative is to encrypt the private key with a passphrase and store the passphrase in the file system so a system process can access it when it needs to use the key to connect via SSH. Although this might sound more secure than using an unprotected key with no passphrase, in fact both solutions provide similar levels of security. Either way, file permissions are the only control to protect the private key. Authentication controls will be ineffective if an attacker is able to compromise the file system security or obtain a copy of the files off-line (through a backup tape).

A better solution may be to use the ssh-agent utility, which is a program that keeps private keys in memory so that they can be accessed whenever private keys are needed. With an agent, passphrases are loaded into memory once when the system boots using the ssh-add command, and after that, no further prompting for passphrases occurs. Although the passphrases could theoretically be hacked from system memory, use of an agent is more secure because it is harder to attack the memory of a running process than to get access to a file on the file system. The major downside of using an SSH agent is that manual intervention is required at boot-time to restart the agent and load passphrases, so unattended system reboots are not possible.

Compromise of Identify File (Private Key)
Unlike for system keys that often support production processes, typically less attention is given to securing keys that are assigned to human users. Human users with unprotected keys – keys without a passphrase - sometime expose their keys accidently. Since keys are stored in users home directories, users have the ability to place their private keys in jeopardy. Actions such as loosening permissions so that other users can read their private keys, copying private keys to unsecured locations, even knowingly sharing keys with other users can undermine the integrity of the authentication mechanism used by SSH. The SSH client program will not use an IdentityFile that is accessible to users other than owner when the client is invoked. However, this does not address the risk of an IdentityFile private key that had been exposed in the past. Even if an IdentifyFile is properly secured now, an attacker may have

obtained a copy of the key previously when the file was unsecured. Anyone who can access a copy of the private key can authenticate and connect through SSH.

Compromise of Authorized Keys File (Public Key Associations)
The authorized keys files have similar issues in the default implementation, but can be more tightly secured. The authorized keys file for an account ("To Account") contains the list of all public keys that are authorized by the account owner to connect using his account. By default, authorized keys files are stored within the user's home directory as ~/.ssh/authorized_keys. The location of the authorized keys file is defined by an sshd configuration setting called AuthorizedKeysFile.

This default configuration allows users to "self-serve" and set-up new associations with their own public keys or another user's public keys. In the default deployment of SSH, authorized key files are accessible by their owners. This enables the account owners to add new associations at will. Since the files are stored in user home directories, it is impractical to attempt to make files inaccessible to users. Even if you try to change file permissions and ownership, the account owner still has access to the parent directory and can delete the file and add a new one with the same name. Furthermore, using the default configuration and storing authorized_keys within individual users' home directories causes key associations to be stored throughout the file system, wherever home directories are located. Because key associations are distributed throughout the file system, it is difficult to produce an inventory of key associations or provide oversight of them.

The authorized keys file is sensitive – if an unauthorized user can obtain write access to the file, they could add their own public key and be able to connect. There is a configuration setting in the sshd_config file (StrictModes yes) that can help protect against a poorly secured authorized keys file. If a user's authorized keys file or home directory is writeable by anyone other than the user (and root), then this setting will keep the SSH server from using the file until the user tightens the restrictions. This is a good feature to use so that users are warned about poor permissions and can fix them. However, it does not itself address all of the risk. In the intervening period, an unauthorized user could have created an association by adding keys to the authorized keys file. If the permissions of the authorized keys file are ever compromised, reviewing and validating each and every installed public key needs to be performed. However, it is highly unlikely that users go through that extra step.

Challenges of Using Keys to Access System Accounts
The OpenSSH model is designed to allow users to add new key associations, and this can create risk, especially for system accounts. IT support personnel may obtain temporary access to system accounts (presumably through a controlled access method) in order to fix production outages or other problems. While logged on as the system account, those users could create new associations to the system accounts by adding new keys to the system account's AuthorizedKeysFile. This may make access easier the next time that there is an outage, but also may allow an alternate access method via SSH keys that circumvent other controls for limiting access to the system account. As a simple example, it is common to restrict knowledge of passwords for system accounts and to change those passwords when support personnel no longer require access. The intent of these password controls is to limit access to the system account except under authorized circumstances. These controls, however, can be circumvented through the use of SSH key based authentication. Regardless of any password change, if a user has a key association defined to a system account, they will still be able to connect as the system account.

This also introduces a new challenge to auditors who are trying to determine who can access system accounts with sensitive access. The old question “who knows the password to the account?” is no longer sufficient. Another question is equally important: “who can use these SSH keys listed in the authorized keys file?” Furthermore, over time the number of keys that have associations with an account could become quite large and unmanageable. Remember that the authorized keys file only list public keys and does not have a mechanism to track the identities (or “From accounts”) of users that can use the key to connect. This creates uncertainty when considering a very basic security question like: “who can connect to this system?” Many organizations with OpenSSH implementations are not able to provide a definitive answer to this question.

Transitive Attacks

Even worse, SSH key associations can create a highly complex mesh of trust that creates the risk of transitive attacks. A transitive attack uses two or more explicitly defined trust associations to obtain access that was not directly assigned. For example, if UserA’s public key is authorized to associate with UserB, and UserB’s public key is authorized to associate with UserC, then effectively UserA can connect as UserC. (This is true unless UserB’s private key has a good passphrase defined, but that is hardly ever the case.) Because the mesh of key associations is so dense, it is likely that transitive attacks could quickly spread throughout many servers in the environment and could potentially reach root-level access. Unfortunately, the key associations that allow for transitive attacks are even harder for organizations to identify and control.

CONTROLLING KEY ASSOCIATIONS

Inappropriate and unknown key associations can provide an access control back-door that bypasses controls that focus on passwords. Consequently, managing key associations is an essential part of having a well-controlled SSH environment. Key associations can quite often get out of control. In the context of authentication controls, it is essential to be able to determine exactly which accounts on the source systems have access to connect to which accounts on the target systems. In the SSH architecture, it is the key associations, also called trust relationships, that provide this information.

The Identity File Dilemma

In the default OpenSSH implementation, getting a handle on key associations can be quite challenging. Imagine that Information Security or Internal Audit identifies a key association with a particular public key. In order to maintain the security of the system, knowing the identity of that public key or who that key belongs to is essential. Given the asymmetric key architecture, anyone who can use the private key that corresponds to the public key will be authenticated and be allowed to use the association. But how do you tell who owns or has access to that key? The possibilities are quite mind-boggling. A particular key in a “To Account’s” authorized key file on a particular “To System” could be related to any “From Account” on any “From System” in the network.

The logical first step would be to address the default implementation. This means examining every computer in the network and looking under every user’s home directory within their ~/.ssh directory to try to find an identity file that contains a matching public key. That is a huge effort, but even that isn’t enough. Users could have overridden the defaults and stored their identify keys files elsewhere in the file system. A zealous Information Security engineer might try to scan ~/.ssh/config files for non-default IdentityFile parameters to try to identify the non-default directories and file names. However, even this is not comprehensive because identity files can be specified dynamically at run time on the SSH client, and there is no way to easily track that. In other words, it may not be practical to look beyond the default locations. This is not ideal because the risk is probably greater for non-default identity files. The most technically savvy, sneaky, or even malicious user would be the least likely to use the default directory location and file names. A malicious user who has access to sensitive SSH keys, someone with the keys to the kingdom, would probably want to conceal that fact and therefore is not likely to use the default IdentityFile. It would be very difficult, however, to track down these non-default locations.

Controlling Authorized Keys Files
If the identity side of the key association is challenging, at least the authorized keys component is easier to control. The only way to prevent account owners from being able to setup new associations is to move the authorized keys file out of users' home directories and into a directory with more sensible permissions. A centralized directory under /etc such as /etc/ssh/authorized_keys is often used. If this directory and the files contained within it are properly secured (e.g., owned by root and writeable only by root), then account owners will not be able to access the file and will be unable to set-up up new key associations on their own. In other words, key associations will no longer be "self-serve," which provides a means to enforce provisioning controls. This also has the advantage of centralizing all key associations in a single location where they are more easily accessible for analyzing and tracking associations between users. The AuthorizedKeysFiles sshd configuration parameter can be used to identify the location of a centralized directory, such as /etc/ssh/authorized_keys/%u. In this example, each user would have a file called /etc/ssh/authorized_keys/<userID> where authorized public keys are stored.

Centralized Provisioning of SSH Keys and Associations
Centralizing and blocking user access to authorized keys files provides another benefit with respect to provisioning. The restrictive security permissions can help support centralized provisioning controls over the creation of key associations. For instance, the Information Security department can be used to provision key associations after obtaining proper approvals and documentation. Since key associations provide access to systems and can subvert existing security structures, it only makes sense that standard security provisioning controls be applied to requests for new key associations. It is only by centralizing the provisioning of new key associations through the standard processes (including formal request and approval by account owners) that SSH keys can be controlled.

Restricting SSH Connections through the Authorized Keys Files
Besides listing the keys that are authorized to connect as each particular user, the authorized keys files can also define additional authorization options that can further restrict keys. These options can be quite useful for helping to control and prevent abuse and misuse of keys that are authorized to connect to, for example, system accounts or accounts used for SFTP connections. Authorization options are defined for each key, so they support a high degree of key-specific granularity. Some of the more useful options are described below:

- command="<some_predefined_command" – Whenever this key is used to connect, this forced command (and only this command) is executed. Any commands requested by the client will be

ignored. This option is useful for connections that provide single operations, such as backups, and nothing else.

- from="<list_of_authorized_host_names_or_IP_addresses> - Only connections from the specified client hosts are permitted for this particular key. To connect, users must have the private key and connect from this designated source location. This makes it harder for an unauthorized user to use a stolen key. This option can be used when one system account has to connect to another system account. Restricting the connection to specific production servers can help limit the risk of abuse.

- no-pty – Prevents the allocation of a pseudo-terminal for interactive login sessions. Requests for an interactive login session will fail. Clients can connect with the key only to execute non-interactive commands.

Restricting SSH Connection through Server-wide Configurations
Although not providing the same level of granularity, some SSH server sshd_config parameters can be used to provide limited server-wide restrictions on permissible key associations. By default, any account on the system can receive connections on an SSH server. Two server-wide configuration settings, which are rarely used, can restrict access to SSH connections using rules based on "To Accounts" and "From Systems." (Note that a "From Account" cannot be used in rules since the SSH servers have no idea of the "From Account's" identity, only that he has the corresponding private key to authenticate.)

- The AllowUsers option permits only the specified "To Accounts" to receive inbound SSH connections; connections to all other accounts are blocked. Wildcard can be used in account names (e.g., ? = a single character and * = any sequence of characters). Accounts can also specify a host name, which is actually the name of the "From Server", following the "@" symbol. The setting "AllowUsers timk@secureit.com" means that local user timk is allowed to receive connections from the remote server secureit.com. This would have the same effect as the "from" option in the AuthorizedKeysFile.

- There is also a DenyUsers setting which denies access to all specified users and permits all others. If both DenyUsers and AllowUsers are defined, then any user that is denied will be blocked and only users that are allowed are permitted.

There are also AllowGroup and DenyGroup options available that work in similar ways based on group memberships. In some circumstances, these configuration options may be helpful. Due to the lack of granularity, oftentimes these server-wide configurations are not adequate for effectively securing key associations.

The PermitRootLogin parameter, however, is frequently useful. A value of "no" will block any connections directly to the root account over SSH, which is recommended. This will require users to first connect as another account and then su to root, providing an audit trail of who assumes root's identity. "Without-password" allows key-based authentication to root, which can present some problems, as will be seen later.

TRACKING AND REVIEWING KEY ASSOCIATIONS

If an organization has a legacy SSH implementation with hundreds of thousands or even millions of key associations already in existence, it isn't enough to merely implement a controlled provisioning process to manage new associations. The provisioning process will provide controls over new associations that will be added to the environment, but something still has to be done about the existing inventory of associations that have accumulated over the years. If OpenSSH operates in the default "self-serve" mode for several years, the resulting inventory of key associations is likely to contain a large number of inappropriate, unauthorized, or otherwise risky associations. Identifying these inappropriate key associations and removing them is key to cleaning up the SSH environment.

Even after the initial clean-up has been completed, additional tracking and reviewing of key associations should be performed. Effective security requires periodic review and recertification to ensure the entitlements are appropriate and continue to be required and authorized. Due to SOX and other regulatory requirements, most organizations have implemented quarterly and semi-annual access recertification processes. Ideally, reviews of SSH key associations should fit into those existing recertification cycles.

In order to validate the existing inventory of SSH key associations and recertify on an ongoing basis, organizations need a process to discover key associations. Most organizations perform this key association discovery task via scripts that run regularly on all systems in the environment. There are usually three steps to the process:

1. The first step is to extract information about the public keys that are stored in users' AuthorizedKeysFiles on the SSH servers. After files are centralized within a single directory (e.g., /etc/ssh/authorized_keys) on each server, it is trivial to pull this information from systems. Usually, instead of tracking the full public keys (which can be quite long), it is preferable to track public key fingerprints. Key fingerprints are unique MD5 hashes of the public key files produced by the built-in ssh-keygen program to differentiate public keys. At this step of the discovery process, the following information is captured in a repository of key associations: "To Account," "To Server", and fingerprints of the authorized public keys. If additional authorization options are defined in the authorized keys files (e.g., forced commands, from restrictions, or no-pty), then it would be useful to capture those options as well within the repository. This information should be documented for periodic review and validation.

2. The second step is to capture information about the identity of the public keys on the SSH clients. At a minimum, this requires scanning every user's home directory on every system for the default identity files. Keep in mind that an SSH client can be run on virtually any system (server or workstation) in the network. In addition, user home directories can be scattered throughout the file system. This is a lot of information to scan! After an identity file is found, the script typically generates a public key signature file and determines who the key belongs to by identifying the owner of the home directory in which the identity file was located. Through this step of the discovery process, the following information is tracked: "From Account", "From Server", and fingerprints of public keys from identity files. As mentioned previously, user identity files can be stored on directories other than the default location. Instead of only

looking for the default identity file location, it is important to also examine user-level ~/.ssh/config files (also stored in user home directories) to identify any non-standard locations. Including these user-defined IdentityFiles within the discovery process will increase the coverage of keys, but even this will not detect all SSH keys in the environment because run-time options can be used to specify other directories. As mentioned earlier, it is not practical to identify every single SSH private key identity in the network, especially if a malicious user is determined to try to hide them. Using the authorization options on keys can help to manage the risk of an unidentified identity. To be absolutely sure that there is no risk, it is necessary to delete the association with the old keys and generate new keys. Periodic re-generation of keys is an accepted security practice for helping to protect keys from unauthorized use.

3. The third step of the discovery process is to correlate the information to produce a complete mapping of the key associations. Both the authorized key information and the identity file information include the public key fingerprint, which serves a key field that can be used to join the "From" information (from the SSH client) and the "To" information (from the SSH server). The resulting information defines a complete key association trust relationship ("From Account," "From Server," "To Account," and "To Server") for a public key fingerprint.

Usually, information about the key association is stored in a registry for analysis. This registry can be integrated into the provisioning process so that new associations are added to the registry as they are created in the environment. It is also possible to re-scan the environment through the key discovery process described above on a regular basis and rebuild the key association registry for analysis purposes. Once complete, the registry of key associations can be very useful in assessing the security risk of SSH key-based authentication and managing the security of the SSH environment.

Typical and recommended uses of an SSH key association registry to provide effective SSH key management include the following:

- Analyzing the existing inventory for risky or unauthorized associations that should be removed. This analysis can be based on high-level categories of associations. For example, any association that allows a user account to connect to a system account (an individual "From Account" and system "To Account") presents risk of unauthorized access and should be avoided. Risks of these and other categories of key associations are discussed in the next section of this paper. Of course, analysis can also be performed for specific accounts. For instance, if the organization has highly sensitive systems, system accounts for those systems warrant additional scrutiny and review.

- Reviewing new associations within authorized keys files that have bypassed the authorized provisioning process. Moving the authorized keys file under /etc prevents most users from provisioning keys, but system administrators (or others with access to root) could still access these files. Reconciling the associations within the authorized keys files against the registry of authorized associations that went through the provisioning process may help identify any rogue backdoors that were created directly by root users. Doing this sort of validation serves as a check on administrative users to guarantee that no associations bypassed the authorization and provisioning process.

- Performing periodic recertification of associations to validate their continuing need and appropriateness. The "To Account" owner should review the associations and validate that they are still required and appropriate. Given the likely volume of associations, it may be appropriate to focus the recertification only on those that present risk. Therefore, key associations where the "To Account" is identical to the "From Account" may not require recertification. These associations with matched accounts present minimal risk because users are allowed to access only their own accounts.

- De-provisioning keys and key associations when users terminate or access is no longer needed. When employees leave the organization or system accounts no longer require access, good security practice mandates that keys/associations be disabled to prevent misuse. At a minimum, all key associations where the "From Account" belongs to a terminated user or retired system account should be removed. This is achieved by logging on to each "To Server" where the association is defined, accessing each "To Account's" authorized keys file, and removing the public key that matches the "From Account's" key fingerprint. After all the associations have been removed from the authorized keys files, no additional action is necessary, although the private keys could be removed from the user's home directory on the "From System". Usually, however, the user's home directory will be removed via other de-provisioning activities, and this will consequently delete the private key from the environment (assuming the user's keys were stored there).

- Rotating keys for risky associations. Periodic rotation of encryption keys, which is basically creating new keys to replace older ones, is a generally accepted practice to maintain the security of cryptography over time. The registry of all key associations in the environment provides the necessary information to enable key rotation. Newly generated keys would need to be placed in the "From User's" home directory on each of the "From Servers." Then that key would need to be installed in the authorized keys files of the "To Users" on each of the "To Servers." As can be imagined, rotating SSH keys can require a significant amount of work. Therefore, it may not always be feasible to adopt a one-size-fits-all approach to key rotation. It may be acceptable to apply a tiered approach to key rotation, whereby only the highest risk keys are regenerated every year or two and others keys may have longer lives.

- Identifying key associations involving keys with unknown owners. In all likelihood, some of the public keys that occur within authorized keys files on SSH servers will not be found through the scanning of SSH client identity files. Although these keys are authorized to connect through SSH authorized keys file, the owners of the keys have not been identified and no information exists regarding who can use the keys to connect. These unknown keys can occur due to several circumstances. For instance, keys could be stored in non-standard locations that were not scanned in the discovery process, or the keys may have been deleted and may no longer be accessible. Although it is difficult to determine exactly how to interpret these unknown keys, an excessively high number of them could indicate that the key discovery process and key association registry are not working effectively. In addition, associations with unknown keys could represent significant risk. Since the owners and users of these keys are unknown, it is not possible to determine if these associations introduce risk through unauthorized access. These associations warrant additional investigation and review. For high-risk "To Accounts," keys with unknown owners should be carefully monitored to determine if they are being used. If not, then these key associations should be removed.

It should also be noted that the key generation utility of OpenSSH supports an optional comment field that can be used to provide the identity of the user to whom the key belongs and information such as a change request number. If key provisioning is centrally provisioned by Information Security, this comment field can also be used to help provide a mechanism to track the identity of keys that is stored in the Authorized Keys File itself.

RISKS OF KEY ASSOCIATIONS

The risks of key associations need to be considered when a new association has been requested, as well as in analyzing and validating existing "legacy" associations that have accumulated over the years. Although each association presents its own unique risks, there are broad categories of associations that share common themes of risk and have implications for control. It is advisable for organizations to make policy decisions about these categories of key associations, defining which are acceptable and which are not. In most cases, at least some of the key associations should be prohibited by policy. The risks and some suggested approaches for handling these associations are discussed in the table below:

Account-based Risks:

Type of Association	Discussion of Risk and Other Concerns
Person Account to System Account	This association is risky since it allows people to assume the identity of system accounts. Passwords for system accounts are generally tightly controlled to prevent abuse and misuse of system accounts. Access through SSH key associations can circumvent these controlled processes and allow unauthorized users to obtain/retain access. This is especially a risk if security associations are not well-controlled (e.g., validated from time-to-time) and private keys not well protected. These associations should be avoided.
System Account to Person Account	It is hard to imagine the circumstances under which such a key association would be required. Systems accounts should only be used for production processes and these should never perform functions in the security context of individual person accounts. These associations can also be used for transitive attacks. For example if PersonA can associate with System1 and System1 can associate with PersonB, then essentially PersonA can connect as PersonB. Transitive relationships can get very complicated very quickly, so it is best to avoid them. The ideal solution is to segregate Person Accounts and System Accounts and avoid any associations between those two categories.
System Account to same System Account	Since both the "From Account" and the "To Account" are the same system account used by automated processes, the risk of these associations is negligible. In many cases, a centralized authentication repository (such as LDAP) provides a single authentication source throughout the environment. In this case, there is no risk for key associations with the same account name.

	However, any association where the “From Account” is a system account potentially presents risk in terms of protecting the keys. Keys used by system accounts often do not have passphrases defined. The risk and possible solutions for this issue were discussed above in this white paper (e.g., using an ssh agent). In addition, appropriate authorization options to limit commands, source addresses, and interactive logons should also be considered since these options can greatly minimize the impact if a key is stolen.
System Account to different System Account	Since both the “From Account” and the “To Account” are system accounts used by automated processes, the risk of these associations may be acceptable, provided that there is a legitimate business need for the association.
Any Account to Root Account	Any account, system or person that can connect to the root account introduces risk. Any remote connections to root should be avoided. In keeping with good security practices, systems should require users to connect locally as a lower-privileged account and then SU to root in order to provide accountability and audit trail. There is an sshd_config parameter that will disallow SSH connections to root. A setting of “PermitRootLogin no” will prevent all root logons. Settings of “yes” or “without-password” allow connections via ssh keys, which should be avoided.
Person Account to same Person Account	When key associations are between like accounts (e.g., To Account is FredW and From Account is FredW), there is little risk of the association itself since users can only connect to their own accounts on different systems. However, person-to-person associations can be a potential headache because there can be a lot of these associations. A single user might want to be able to connect to dozens or even hundreds of systems throughout the network, which means a new key association must be setup for each target system--a significant burden on the provisioning operation. In addition, there are other risks and challenges for keys assigned to person accounts, such as users compromising the security of their private keys by having keys without passphrases or sharing keys with other users. If privacy of the private key cannot be assured, then key-based authentication may be less reliable than traditional password controls. At least passwords are generally stored securely on the system and are not susceptible. For this reason, some organizations decide to disallow person-to-person associations and require users to connect to SSH using password authentication. In this model, key associations are permitted only for system accounts, which can be more easily managed and controlled. If the number of person associations is not overwhelming, the Information Security provisioning function could create the keys and use high-quality passphrases during key generation.
Person Account to different Person Account	By definition, these associations allow one human user to assume the identity of another human user. This causes a loss of accountability and undermines access controls. These types of associations should never be permitted.
Multiple “From Accounts” (Person	Use of the same key by multiple “From Accounts” indicates that the key has been shared and that the privacy of the private key has been compromised. At a

or System) for the same key	minimum, it is necessary to remove all known instances of "From Accounts" with unauthorized keys. That is, delete the private key from the home directories of all "From Accounts" whose logon ID does not correspond to the "To Account." However, since the key was compromised, this approach does not address the risk that unauthorized copies of the private key may be stored elsewhere. The safest approach is to delete all instances of the key, remove the key association entirely, and generate a new key and key associations. After the old association is removed, no one with an unauthorized copy of the private key can connect. Then create a new key that can be protected from the beginning and create new key associations that are required so that each "From Account" uses a different key.
Unknown Keys occur in the authorized keys files	Although these keys are authorized to connect through SSH authorized keys file, the owners of the keys have not been identified and no information exists regarding who can use the keys to connect. Although it is difficult to determine exactly how to interpret these unknown keys, an excessively high number of them could indicate that the key discovery process and key association registry are not working effectively. In addition, associations with unknown keys could represent significant risk. Since the owners and users of these keys are unknown, it is not possible to determine if these associations introduce risk of unauthorized access. It is possible, for example, that some of the unknown keys may exist because malicious users have intentionally hidden the private key in order to preserve their access to system accounts. Because of the potential risk, the unknown keys should be removed from the AuthorizedKeysFiles for system accounts. However, there is also a Denial of Service (DoS) risk with removing these associations since the keys could also be used by system "From Accounts" that are required. Therefore, careful analysis is required before associations are removed. Perhaps associations of unknown accounts to person "To Accounts" could be removed more easily since there would be no impact on production and users could still authenticate through passwords. Associations to system "To Accounts" may require extensive logging in order to identify if the public keys no known owner are dormant or are still being used. Obviously, if unknown public keys have been used for an extended period of time, they can be safely deleted without impacting any processing.

System-based Risks:

Type of Association	Risks and Other Concerns
Low Risk Systems to High Risk Systems	Some organizations have different levels of control applied to different systems. For instance, systems supporting financially significant applications may be subject to tighter restrictions and control practices than systems related to lower risk non-financial applications. Or systems in a testing environment may be more loosely secured than systems in production. If this is the case, then associations between these categories of systems should be avoided. If low risk systems can be compromised more easily by attackers or provide greater access

	to humans, these systems could be the means through which attacks or unauthorized access spreads to higher-risk systems.
External Systems to Internal Systems	Although session encryption capabilities of SSH seem ideally suited to connections that span across networks, there are risks associated with keys that need to be considered. Once keys are sent outside the network, they are no longer in the control of the organization. Practices that might be enforced internally may not be enforced outside the network. If a key is compromised from an external system, then internal systems may be at risk. Reasonable measures should be taken to help protect keys from compromise (such as generating keys internally with strong phrases and securely distributing them to users outside the network). If inbound SSH sessions cannot be completely blocked at the perimeter, then at least firewall restrictions on the source and destination of SSH connections should be used to restrict access as much as possible.

STANDARDIZED CONFIGURATION

A final element of effective management of SSH keys and trust relationships is having standardized SSH configurations across the environment. Configurations can be defined for the both the SSH client and the SSH server. As mentioned earlier in this whitepaper, the configuration of the SSH client can be quite challenging, since in the OpenSSH model, users can override any standard configuration. Fortunately, there are relatively few highly significant configuration settings on the client side of SSH. The IdentityFile parameter (as discussed earlier) and the StrictHostKeyChecking parameter (which concerns SSH host keys and is beyond the scope of this whitepaper) are the most significant settings.

There are several SSH server configurations in the /etc/ssh/sshd_config file that are significant and should be assessed. The typical controls for managing configurations should be deployed, including baseline configuration standards/builds and periodic validation of server settings in production to detect non-compliance and configuration drift. Some of the critical settings that should be assessed are described in the table below:

sshd_config Parameter	**Defaults**	**Recommended Value**
AuthorizedKeysFile – can be used to store authorized keys files in a centralized directory that blocks user update access, thereby providing enforcement for an authorized provisioning process	Default ~/.ssh/authorized_keys	/etc/ssh/authorized_keys/%u or some other central directory outside of users' home directories and properly secured to deny write access
Ciphers – used to define the ciphers and key lengths that are supported.	Ver 2 only Default aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour128, arcfour256,arcfour,aes192-	Default or only 128+ bit ciphers, or stronger if required

	cbc,aes256-cbc,aes128-ctr, aes192-ctr,aes256-ctr. Default for Sun is aes128-ctr, aes128-cbc, 3des-cbc, blowfish-cbc, arcfour	
HostBasedAuthentication – can be used to refuse or enable host based authentication.	Default no	Default or No (If set to Yes, then at least IgnoreRhosts should be Yes) [2]
IgnoreRhosts – can be used to disable used of .rhosts and .shosts files	Default yes	Default or Yes (This has no affect if HostBasedAuthentication is No) [2]
LogLevel – can be used to specify the level of logging that is performed by the SSH server	Default info	Default or info or (for the paranoid) verbose
MaxAuthTries – prevents password brute forcing by setting a cap on failed attempts.	Default 6	Default or 6 or less
PermitEmptyPasswords – can be used to disallow empty password	Default no	Default or No
PermitRootLogin – can be used to prevent access to the root account	Default yes	No [3]
Protocol – can be used to allow limit access to specific SSH protocols	Defaults 2,1	2 [Version 1 is susceptible to some security attacks.]
PubKeyAuthentication – can be used to enable public keys	Default yes	Default or Yes

[2] If HostBasedAuthentication is used, then there are two things to consider. First, each host must have a unique host key (e.g., multiple hosts should not share the same key). SSH uses host keys to validate the identity of systems, so sharing a host key makes it easier to defeat the host authentication process. If host authentication is used, then host keys should be reviewed to make sure that keys are not shared across systems. Second, IgnoreRhosts should be enabled. Host based authentication uses the standard "r" services trust files (/etc/hosts.equiv and ~/.rhosts) and SSH-specific trust files (/etc/shosts.equiv and ~/.shosts) to define the trust relationships that are permitted. Essentially, these 4 files define the same sort of trust associations that are defined in the authorized keys file for user keys. Users will be able to access the ~/.rhosts and ~/.shosts files in their home directories, and can therefore create new trust relationships at will. In order to lock-down the trust relationships so that they can be properly controlled, SSH must be configured to ignore those trust files that are directly accessible to the end user, which is what the IgnoreRhosts option does. When this option is set, then all trust relationships must be defined within either the hosts.equiv and shosts.equiv files, both of which are locked-down under the /etc directory. This approach eliminates "self-serve" trust relationships and enables host based trust relationships to be managed and controlled in a similar way as is described in this paper for user keys.

[3] Setting the configuration option to prevent direct access to the root account via keys helps to avoid the risk of unauthorized key associations for root access.

StrictModes – can be used to ensure that the SSH server program will not execute if the authorized keys file and home directory is insecure	Default yes	Default or Yes
UsePrivilegeSeparation – can be used to prevent privilege escalation attacks by spawning least-privileged processes	Default yes	Default or Yes

CONCLUSION

SSH is a widely deployed facility for establishing encrypted connections. Although it was intended to solve a security problem, the default OpenSSH implementation follows a self-serve key management model that can introduce security risk and result in inappropriate access through the use of key associations. In order to properly secure and manage SSH, traditional controls need to be applied to key associations, including centralized provisioning and de-provisioning, periodic recertification, and centralized tracking of user entitlements. To accomplish these objectives, it is necessary first to centralize authorized keys files into a directory that is not accessible to users. Then it is useful to scan the environment and aggregate information about authorized keys on SSH servers and user identities on SSH clients. A registry of SSH key associations can provide valuable information to help identify inappropriate or otherwise risky associations that are defined in the environment. These reasonable steps can help organizations that do not have a formal PKI infrastructure to manage and control the deployment of OpenSSH keys and key associations. Since key-based authentication is critical feature of SSH, effective key management is required in order to have a reliable authentication mechanism and prevent keys from being used to circumvent controls that exist for password-based logons. More fundamentally, key management is needed so organizations can know who has access to their systems and restrict access to known, authorized users.

ABOUT SECUREIT

SecureIT is a professional and technical services firm focusing on information security and risk management. SecureIT helps corporations, Federal government agencies, and other non-government organizations manage risks in business processes, technology and contracted services through services and solutions in the areas of Cybersecurity, Information Assurance, Governance, Risk & Compliance, IT Audit, and Security Training. Founded in 2001, the company sits on the board and is active in Washington DC area chapters of information security organizations such as ISACA, ISSA and IIA. SecureIT serves clients such as FINRA, Ernst & Young, CSC, E*TRADE, The Washington Post, Department of Treasury, Overseas Private Investment Corporation (OPIC), Federal Aviation Administration (FAA), and the International Monetary Fund (IMF). For more information, call 703.464.7010, email info@secureit.com or visit www.secureit.com.