Steve Friedl UNIXWIZ.NET

Steve Friedl's Unixwiz.net Tech Tips

An Illustrated Guide to SSH Agent Forwarding

The Secure Shell is widely used to provide secure access to remote systems, and everybody who uses it is familiar with routine password access. This is the easiest to set up, is available by default, but suffers from a number of limitations. These include both security and usability issues, and we hope to cover them here.

Table of Contents

In this paper, we'll present the various forms of authentication available to the Secure Shell user and contrast the security and usability tradeoffs of each. Then we'll add the extra functionality of agent key forwarding, we hope to make the case that using ssh public key access is a substantial win.

Note - This is not a tutorial on setup or configuration of Secure Shell, but is an overview of technology which underlies this system. We do, however, provide some pointers to information on several packages which may guide the user in the setup process.

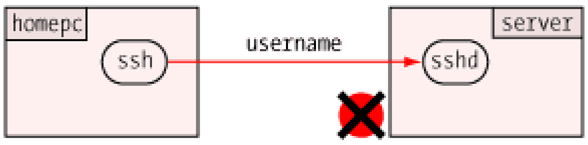
Ordinary Password Authentication

SSH supports access with a username and password, and this is little more than an encrypted telnet. Access is, in fact, just like telnet, with the normal username/password exchange.

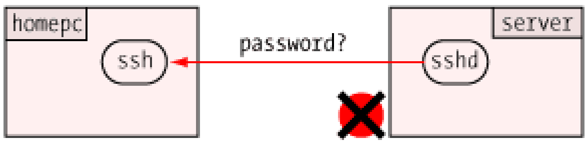
We'll note that this exchange, and all others in this paper, assume that an initial exchange of **host keys** has been completed successfully. Though an important part of session security, host validation is not material to the discussion of agent key forwarding.

All examples start from a user on **homepc** (perhaps a Windows workstation) connecting with PuTTY to a server running OpenSSH. The particular details (program names, mainly) vary from implementation to implementation, but the underlying protocol has been proven to be highly interoperable.

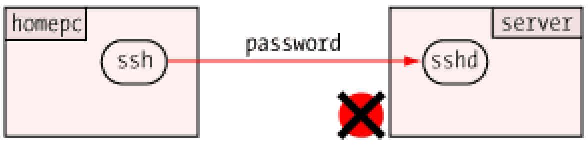
1 The user makes an initial TCP connection and sends a username. We'll note that unlike telnet, where the username is prompted as part of the connected data stream (with no semantic meaning understood by telnet itself), the username exchange is part of the ssh protocol itself.



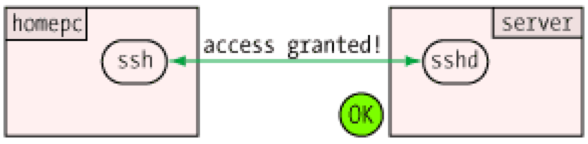
2 The ssh daemon on the server responds with a demand for a password, and access to the system has not yet been granted in any way.



3 The ssh client prompts the user for a password, which is relayed through the encrypted connection to the server where it is compared against the local user base.



4 If the user's password matches the local credential, access to the system is granted and a two-way communications path is established, usually to a login shell.



The main advantage of allowing password authentication is that it's simple to set up — usually the default — and is easy to understand. Systems which require access for many users from many varying locations often permit password auth simply to reduce the administrative burden and to maximize access.

Password Authentication

Pro:	easy to set up

The substantial downside is that by allowing a **user** to enter a password, it means **anybody** is allowed to enter a password. This opens the door to wholesale password guessing by users or bots alike, and this has been an increasingly common method of system compromise.

Con:	allows brute-force password guessing
Con:	requires password entry every time

Unlike prior-generation ssh worms, which attempted just a few very common passwords with common usernames, modern badware has a very extensive dictionary of both usernames and passwords and has proven to be most effective in penetrating even systems with "good" passwords. Only one compromised account is required to gain entry to a system.

But even putting security issues aside, the other downside of password authentication is that passwords must be remembered and entered separately upon every login. For users with just one system to access, this may not be such a burden, but users connecting to many systems throughout the day may find repeated password entry tedious.

And having to remember a different password for every system is not conducive to choosing strong passwords.

Public Key Access

To counteract the shortcomings of password authentication, ssh supports public key access. A user creates a pair of public and private keys, and installs the public key in his **$HOME/.ssh/authorized_keys** file on the target server. This is nonsensitive information which need not be guarded, but the other half — the private key — is protected on the local machine by a (hopefully) strong passphrase.

Note - older versions of OpenSSH stored the v2 keys in **authorized_keys2** to distinguish them from v1 keys, but newer versions use either file.

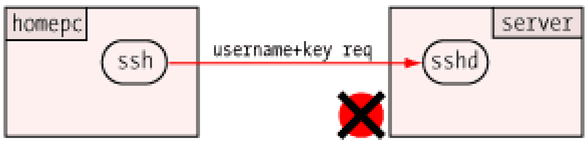
A public key is a long string of bits encoded in ASCII, and it's stored on one long line (though represented here on three continued lines for readability). It includes a type (**ssh-rsa**, or others), the key itself, and a comment:

$HOME/.ssh/authorized_keys

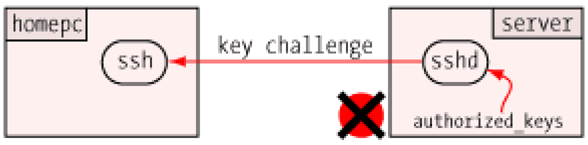
```
ssh-rsa AzAAB3NzaC1yc2EaaaabiWaaaieaX9AyNR7xWnW0eI3x2NGXrJ4gkQpK/EqpkveGCvvbM \
  oH84zqu3Us8jSaQD392JZAEAhGSoe0dWMBFm9Y41VGZYmncwkfTQPFH1P07vDw49aTAa2RJNFyV \
  QANZCbSocDeuT0Q7usuUj/v8h27+PqsUUl9XVQSDIhXBkWV+bJawc1c= Steve's key
```

This key must be installed on the target system — one time — where it is used for subsequent remote access by the holder of the private key.

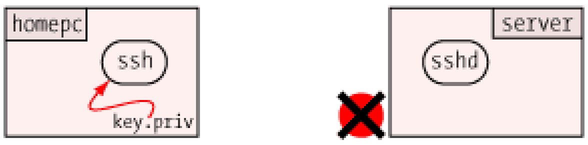
1 The user makes an initial connection and sends a username along with a request to use a key.



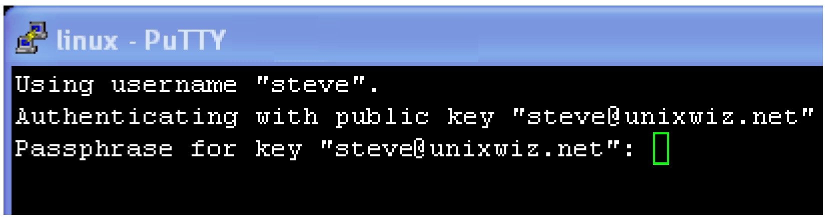
2 The ssh daemon on the server looks in the user's **authorized_keys** file, constructs a challenge based on the public key found there, and sends this challenge back to the user's ssh client.



3 The ssh client receives the key challenge. It finds the user's private key on the local system, but it's protected by an encrypting passphrase. An RSA key file is named **id_rsa** on OpenSSH and SecureCRT, ***keyname*.ppk** on PuTTY. Other types of keys (DSA, for instance) have similar name formats.

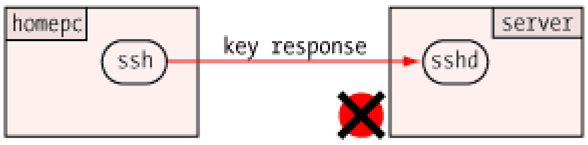


4 The user is prompted for the passphrase to unlock the private key. This example is from PuTTY.

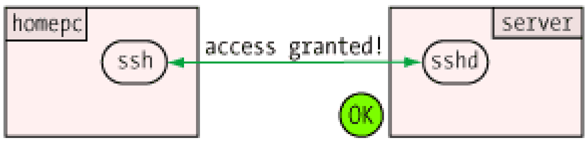


```
Using username "steve".
Authenticating with public key "steve@unixwiz.net"
Passphrase for key "steve@unixwiz.net":
```

5 ssh uses the private key to construct a key response, and sends it to the waiting sshd on the other end of the connection. It does **not** send the private key itself!



6 sshd validates the key response, and if valid, grants access to the system.



This process involves more steps behind the scenes, but the user experience is mostly the same: you're prompted for a **passphrase** rather than a **password**. But, unlike setting up access to multiple computer systems (each of which may have a different password), using public key access means you type **the same** passphrase no matter which system you're connecting to.

This has a substantial, but non-obvious, security benefit: since you're now responsible for just one secret phrase instead of many passwords, you type it more often. This makes you more likely to remember it, and therefore pick a stronger passphrase to protect the private key than you otherwise might.

Trying to remember many separate passwords for different remote systems is difficult, and does not lend itself to picking strong ones. Public key access solves this problem entirely.

Public Key Authentication

Pro:	public keys cannot be easily brute-forced
Pro:	the same private key (with passphrase) can be used to access multiple systems: no need to remember many passwords
Con:	requires one-time setup of public key on target system
Con:	requires unlocking private key with secret passphrase upon each connection

We'll note that though public-key accounts can't generally be cracked remotely, the mere installation of a public key on a target system does not disable the use of passwords systemwide. Instead, the server must be explicitly configured to allow only public key encryption by the use of the **PasswordAuthentication no** keywords in the **sshd_config** file.

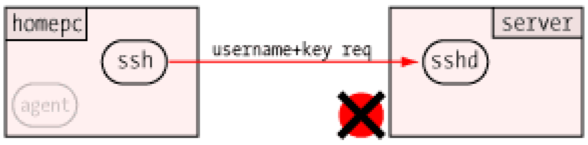
Public Key Access with Agent support

Now that we've taken the leap into public key access, we'll take the next step to enable agent support. In the previous section, the user's private key was unlocked at every connection request: this is not functionally different from typing a password, and though it's the same passphrase every time (which makes it habitual), it nevertheless gets tedious in the same manner.

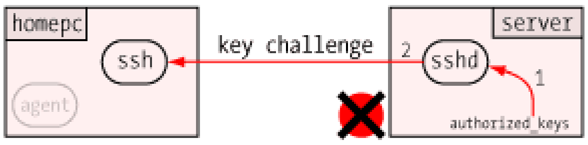
Fortunately, the ssh suite provides a broker known as a "key agent" which can hold and manage private keys on your workstations, and responding to requests from remote systems to verify your keys. Agents provide a **tremendous** productivity benefit, because once you've unlocked your private key (one time when you launch the agent), subsequent access works with the agent **without prompting**.

This works much like the key access seen previously, but with a twist.

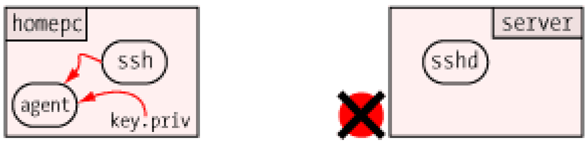
1 The user makes an initial connection and sends a username along with a request to use a key.



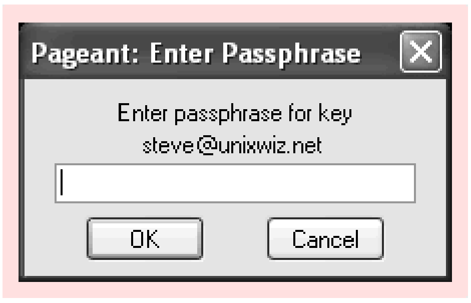
2 The ssh daemon on the server looks [1] in the user's **authorized_keys** file, constructs a challenge based on the key, and sends it [2] back to the user's ssh client.



3 The ssh client receives the key challenge, and forwards it to the waiting agent. The agent, rather than ssh itself, opens the user's private key and discovers that it's protected by a passphrase.

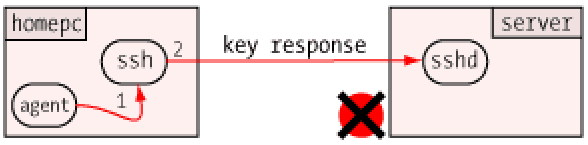


4 The user is prompted for the passphrase to unlock the private key. This example shows the prompt from PuTTY's **pageant**.

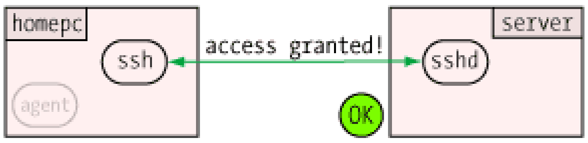

Pageant: Enter Passphrase

Enter passphrase for key
steve@unixwiz.net

5 The agent constructs the key response and hands it back [1] to the **ssh** process, which sends it off [2] to the **sshd** waiting on the other end. Unlike the previous example, **ssh** never sees the private key directly, only the key response.



6 **sshd** validates the key response, and if valid, grants access to the system. Note: the agent still retains the private keys in memory, though it's not participating in the ongoing conversation.



As far as the user is concerned, this first exchange is little different from key access shown in the previous section: the only difference is which program prompts for the private key (**ssh** itself *versus* the agent).

But where agent support shines is at the *next* connection request made while the agent is still resident. Since it remembers the private keys from the first time it was unlocked with the passphrase, it's able to respond to the key challenge immediately ***without prompting***. The user sees an immediate, direct login without having to type anything.

Many users only unlock their private keys once in the morning when they launch their ssh client and agent, and they don't have to enter it again for the rest of the day because the resident agent is handling all the key challenges. It's **wonderfully** convenient, as well as secure.

It's very important to understand that private keys **never** leave the agent: instead, the clients ask the agent to perform a computation based on the key, and it's done in a way which allows the agent to prove that it has the private key without having to divulge the key itself. We discuss the challenge/response in a later section.

Public Key with Agent

Pro:	Requires unlocking of the private key **only once**
Pro:	Facilitates scripted remote operation to multiple systems
Con:	One-time cost to set up the agent
Con:	Requires private key on remote client machines if they're to make further outbound connections

Once agent support is enabled, all prompting has now been bypassed, and one can consider performing scripted updates of remote systems. This contrived example copies a **.bashrc** login config file to each remote system, then checks for how much disk space is used (via the **df** command):

```
# scripted update of several remote systems

for svr in server1 server2 server3 server4
do
        scp .bashrc $svr:~/    # copy up new .bashrc
        ssh $svr df            # ask about disk space
done
```

Without agent support, each server would require **two** prompts (first for the copy, then for the remote command execution). With agent support, there is no prompting at all.

However, these benefits only accrue to outbound connections made from the local system to ssh servers elsewhere: once logged into a remote server, connecting from there to yet a third server requires either password access, or setting up the user's private key on the intermediate system to pass to the third.

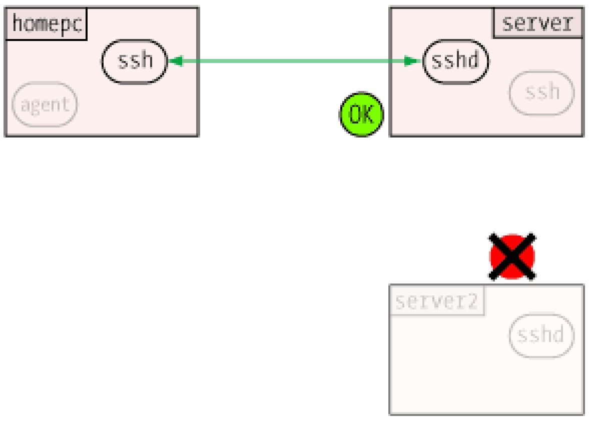
Having agent support on the local system is certainly an improvement, but many of us working remotely often must copy files from one remote system to another. Without installing and initializing an agent on the first remote system, the **scp** operation will require a password or passphrase every time. In a sense, this just pushes the tedium back one link down the ssh chain.

Fortunately, there's a great solution which solves all these issues.

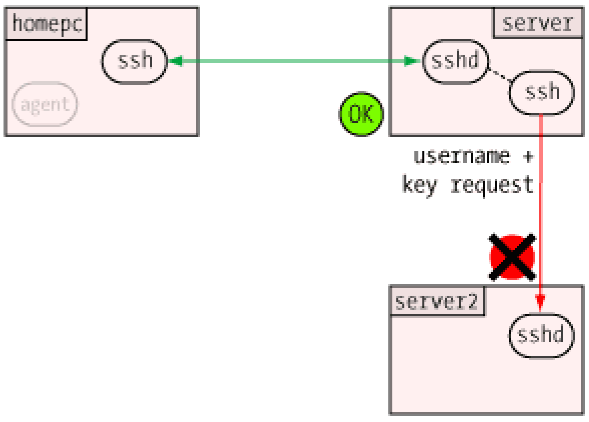
Public Key Access with Agent Forwarding

With our Key Agent in place, it's time to enable the final piece of our puzzle: ***agent forwarding***. In short, this allows a chain of ssh connections to forward key challenges back to the original agent, obviating the need for passwords or private keys on any intermediate machines.

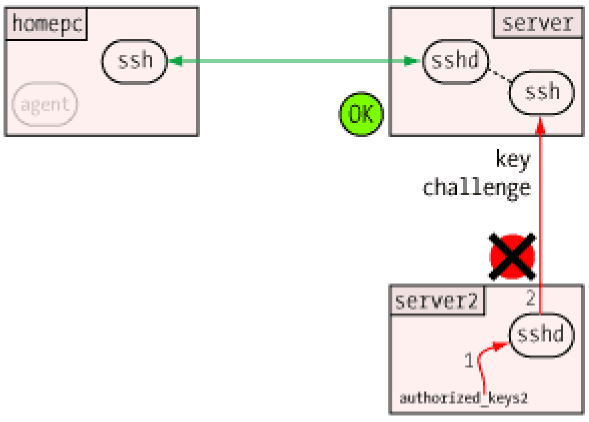
1 This all starts with an already established connection to the first server, with the agent now holding the user's private key. The second server plays no part yet.



2 The user launches the **ssh** client on the first server with a request to connect to **server2**, and this passes the username and a use-key request to the ssh daemon (this could likewise be done with the **scp** secure copy command as well)

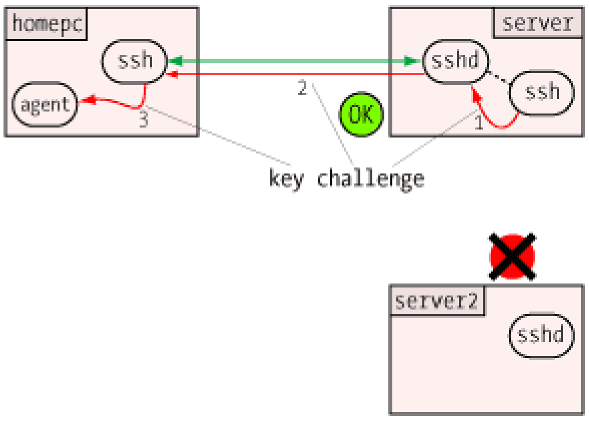


3 The ssh daemon consults the user's **authorized_keys** file [1], constructs a key challenge from the key, and sends it [2] back down the channel to the client which made the request.



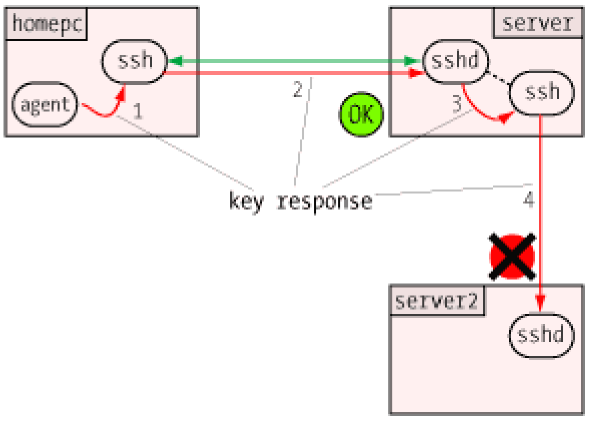
4 This is where the magic occurs: the ssh client on **server** receives the key challenge from the target system, and it forwards [1] that challenge to the sshd server on the same machine acting as a key agent.

sshd in turn relays [2] the key challenge down the first connection to the original ssh client. Once back on **homepc**, the ssh client takes the final step in the relay process by handing the key challenge off [3] to the resident agent, which knows about the user's private key.

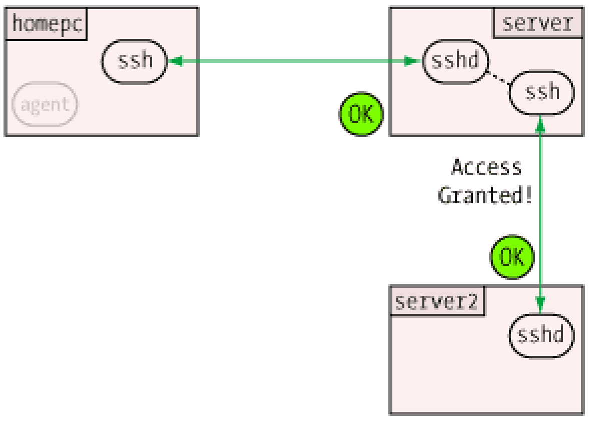


5 The agent running on **homepc** constructs the key response and hands it [1] back to the local ssh client, which in turn passes it [2] down the channel to the sshd running on **server**.

Since **sshd** is acting as a key agent, it forwards [3] the key response off to the requesting ssh client, which sends it [4] to the waiting **sshd** on the target system (**server2**). This forwarding action is all done automatically and near instantly.



6 The ssh daemon on **server2** validates the key response, and if valid, grants access to the system.



This process can be repeated with even more links in the chain (say, if the user wanted to ssh from **server2** to **server3**), and it all happens automatically. It supports the full suite of ssh-related programs, such as **ssh**, **scp** (secure copy), and **sftp** (secure FTP-like file transfer).

This does require the one-time installation of the user's public — **not** private! — keys on all the target machines, but this setup cost is rapidly recouped by the added productivity provided. Those using public keys with agent forwarding rarely go back.

Agent Forwarding

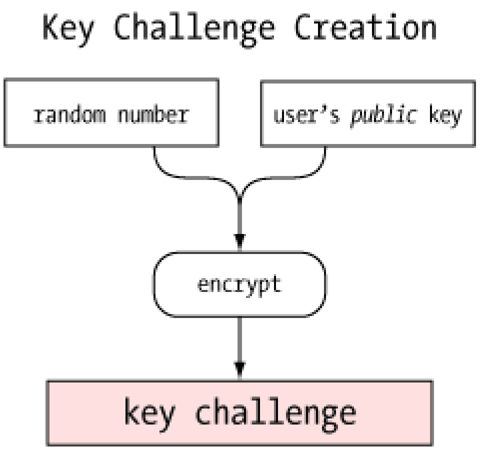
Pro:	Exceptional convenience
Con:	Requires installation of public keys on all target systems
Con:	Requires a Tech Tip to understand
Pro:	An excellent Tech Tip is available 😊

How Key Challenges Work

One of the more clever aspects of the agent is how it can verify a user's identity (or more precisely, possession of a private key) without revealing that private key to anybody. This, like so many other things in modern secure communications, uses public key encryption.

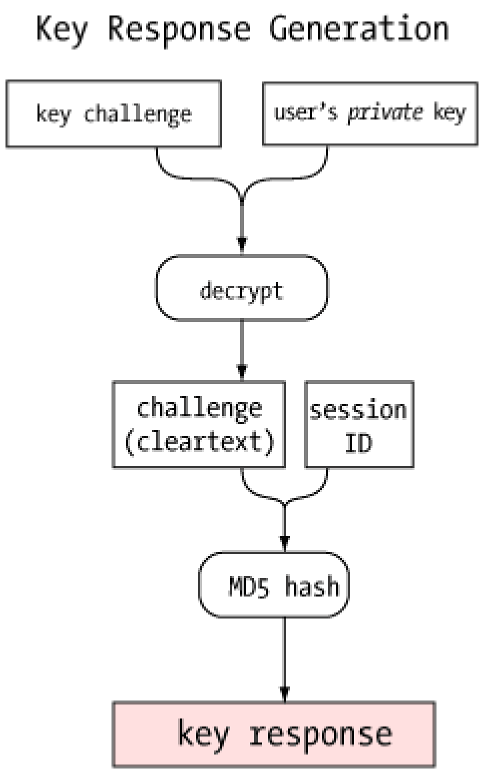
When a user wishes access to an ssh server, he presents his username to the server with a request to set up a key session. This username helps locate the list of public keys allowed access to that server: typically it's found in the **$HOME/.ssh/authorized_keys** file.

The server creates a "challenge" which can only be answered by one in possession of the corresponding private key: it creates and remembers a large random number, then encrypts it with the user's public key. This creates a buffer of binary data which is sent to the user requesting access. To anybody without the private key, it's just a pile of bits.



When the agent receives the challenge, it decrypts it with the private key. If this key is the "other half" of the public key on the server, the decryption will be successful, revealing the original random number generated by the server. Only the holder of the private key could ever extract this random number, so this constitutes proof that the user is the holder of the private key.

The agent takes this random number, appends the SSH session ID (which varies from connection to connection), and creates an MD5 hash value of the resultant string: this result is sent back to the server as the key response.



The server computes the same MD5 hash (random number + session ID) and compares it with the key response from the agent: if they match, the user must have been in possession of the private key, and access is granted. If not, the next key in the list (of any) is tried in succession until a valid key is found, or no more authorized keys are available. At that point, access is denied.

Curiously, the actual random number is never exposed in the client/agent exchange - it's sent encrypted **to** the agent, and included in an MD5 hash **from** the agent. It's likely that this is a security precaution designed to make it harder to characterize the properties of the random number generator on the server by looking at the the client/agent exchange.

More information on MD5 hashes can be found in An Illustrated Guide to Cryptographic Hashes, also on this server.

Security Issues With Key Agents

One of the security benefits of agent forwarding is that the user's private key never appears on remote systems or on the wire, even in encrypted form. But the same agent protocol which shields the private key may nevertheless expose a different vulnerability: agent hijacking.

Each ssh implementation has to provide a mechanism for clients to request agent services, and on UNIX/Linux this is typically done with a UNIX domain socket stored under the **/tmp/** directory. On our Linux system running OpenSSH, for instance, we find the file **/tmp/ssh-CXkd6094/agent.6094** associated with the SSH daemon servicing a SecureCRT remote client.

Caution

This socket file is as heavily protected as the operating system allows (restricted to just the user running the process, kept in a protected subdirectory), but nothing can really prevent a **root** user from accessing any file anywhere.

Sidebar: though the agent uses a socket, it's a *UNIX domain socket*, which is accessible only from the local filesystem. The agents do not listen on any TCP/IP-based socket. This post provides a good discussion of UNIX domain sockets *versus* internet sockets.

If a root user is able to convince his ssh client to use another user's agent, root can impersonate that user on any remote system which authorizes the victim user's public key. Of course, root can do this on the *local* system as well, but he can do this directly anyway without having to resort to ssh tricks.

Several environment variables are used to point a client to an agent, but only **SSH_AUTH_SOCK** is required in order to use agent forwarding. Setting this variable to a victim's agent socket allows full use of that socket if the underlying file is readable. For **root**, it always is.

```
# ls -l /tmp/ssh*      — look for somebody's agent socket
/tmp/ssh-CXkd6094:
total 24
srwxr-xr-x    1 steve    steve          0 Aug 30 08:46 agent.6094=

# export SSH_AUTH_SOCK=/tmp/ssh-CXkd6094/agent.6094

# ssh steve@remotesystem

remote$                     — Gotcha! Logged in as "steve" user on remote system!
```

One cannot tell just from looking at the socket information which remote systems will accept the user's key, but it doesn't take too much detective work to track it down. Running the **ps** command periodically on the local system may show the user running **ssh *remotesystem***, and the **netstat** command may well point to the user's home base.

Furthermore, the user's **$HOME/.ssh/known_hosts** file contains a list of machines which the user has a connection: though they may not all be configured to trust the user's key, it's certainly a great place to start looking. Modern versions

(4.0 and later) of OpenSSH can optionally hash the **known_hosts** file to forestall this.

There is no technical method which will prevent a root user from hijacking an SSH agent socket if he has the ability to access it, so this suggests that agent forwarding might not be such a good idea when the remote system cannot be entirely trusted. All ssh clients provide a method to disable agent forwarding.

Additional Resources

Up to this point, we've provided essentially no practical how-to information on how to install or configure any particular SSH implementation. Our feeling is that this information is covered better elsewhere, and we're happy to provide some links here to those other resources.

- **The Secure Shell: The Definitive Guide, 2 Ed (O'Reilly & Associates).**

  This is clearly the standout book in its class: it covers Secure Shell from A to Z, including many popular implementations. There is no better comprehensive source for nearly all aspects of Secure Shell Usage. A worthy addition to any bookshelf.

- **PuTTY**

  This is a very popular Open Source ssh client for Windows, and it's notable for its economy (it will easily fit on a floppy disk). The next resource provides extensive configuration guidance.

- **Unixwiz.net Tech Tip: Secure Linux/UNIX access with PuTTY and OpenSSH**

  This is one of our own Tech Tips which is a hands on guide to configurating the excellent open source PuTTY client for Windows. Particular coverage was given to public key generation and usage, with plenty of screenshots to guide the way.

- **Unixwiz.net Tech Tip: Building and configuring OpenSSH**

  Though this Tech Tip is mainly concerned with configuration of the server on a UNIX/Linux platform, it also provides coverage of the commercial SecureCRT Windows client from VanDyke Software (which we use ourselves). It specifically details key generation and agent forwarding settings, though briefly.

- **Unixwiz.net Tech Tip: An Illustrated Guide to Cryptographic Hashes**

  Though not central to using SSH Agent Forwarding, some coverage cryptographic hashes may help understand the key challenge and response mechanism. This Tech Tip provides a good overview of crypto hashes in a similarly-illustrated format.

First Published: 2006-02-22

Home | Stephen J. Friedl | Software Consultant | Orange County, CA USA | steve@unixwiz.net |