Secure, Non-Interactive Privileged SSH Authentication Using Static Agents

Jonathon Anderson
jonathon.anderson@kaust.edu.sa
King Abdullah University of Science and Technology

22 May 2010

Abstract

Hostbased authentication provides a convenient mechanism for non-interactive authentication of user accounts during SSH connections; however, its dependence on non-encrypted private keys makes it unsuitable for use with privileged system accounts. The security implications of root authority necessitate interaction, but this requirement stands in opposition to the pervasive automation that makes large-scale system administration possible. A network of static SSH agents, coupled with an automated configuration management system, can be used to mitigate the need for interactivity while protecting the unencrypted private key. Furthermore, the use of an agent removes the need to store even the encrypted private key on target hosts beyond the initial configuration of the agent. With the existing OpenSSH implementation, individual client applications can be configured to use the system agent, but future SSH implementations could be updated to provide more generalized support with the addition of a new configuration directive.

1 Hostbased authentication

Hostbased authentication is an SSH[1] authentication method that allows a target host to trust the prior authentication of a user as carried out on a remote client host.

Every host providing the SSH service possesses a cryptographic "private" key. Clients that are in possession of the matching "public" key authenticate

[1] *The Secure Shell (SSH) Authentication Protocol*, Internet Engineering Task Force http://www.ietf.org/rfc/rfc4252.txt .

the target host using a challenge-response protocol before providing account credentials. Likewise, when the target host is in possession of the public key matching the client's private key, the SSH server can authenticate the identity of the client host.

Once the server has been able to verify the identity of the client host it can make assertions of trust regarding properties of that host. In particular, the server might trust that the client has already sufficiently authenticated the client user. In such cases, the target host can accept the authenticated identity of the client host to be sufficient to authorize access to the same account on the server. This host equivalency relationship is defined (for the OpenSSH[2] implementation) by the file `/etc/hosts.equiv`, which contains a list of names identifying trusted hosts.

With a properly configured hostbased authentication system, users have the convenience of being able to access their accounts on any host in the equivalency group without having to interactively re-authenticate (i.e., without having to type a password or passphrase). Additionally, the decreased frequency of password entry provides a number of security benefits.

- The password is less likely to be intercepted.[3]
- A secure password is more likely to be used.[4]

2 Privileged accounts

The ability to access remote hosts without interactive authentication is as useful for system accounts as it is for user accounts. However, OpenSSH explicitly invalidates hostbased authentication when targeting the `root` account.

Hostbased authentication is based on the integrity of a private key that is stored unencrypted on disk. Any compromise of a trusted host implies the compromise of the private key, granting access to any account that can be authenticated via that key. Though sufficient datacenter security might make this an acceptable risk for user accounts, privileged system accounts often cary a greater expectation of security.

OpenSSH can be configured to override the default behavior to allow hostbased authentication for the `root` account.[5] Provided that the configu-

[2] *OpenSSH* http://www.openssh.com/ .

[3] Florêncio, D. and C. Herley. *How To Login From an Internet Café Without Worrying About Keyloggers*, SOUPS '06.

[4] Grubb, M. and R. Carter. *Single Sign-On and the System Administrator*, LISA '98.

[5] http://help.ubuntu.com/community/SettingUpGPFSHowTo .

ration directive `IgnoreRhosts` is set to "no", hosts listed in `~root/.rhosts` (or `~root/.shosts`) will be trusted for hostbased authentication for the `root` account in the same way that hosts listed in `/etc/ssh/hosts.equiv` are trusted for authentication of other accounts. Though this solution addresses the superficial barrier to non-interactive authentication of the `root` account, it remains susceptible to the security pitfalls of privileged account authentication based on a non-encrypted private key: compromise of any host implies the compromise of all hosts that trust the compromised host.

Alternatively, an account-level public/private key pair may be used. With a utility like OpenSSH's own `ssh-keygen`, a key pair can be generated for use in the "publickey" authentication method. The public key is distributed to any host that should trust the key for authentication, and is usually stored in the file `~root/.ssh/authorized_keys`.

Private key generation solicits a "passphrase" used to encrypt the private key on disk: however, such a passphrase must be entered whenever accessing the private key (e.g., when authenticating against a server), preventing its use in non-interactive cases. "Passphraseless" private keys may be generated, but are susceptible to the same pitfalls mentioned for hostbased authentication.

3 Using `ssh-agent`

Raw public/private keys cannot provide appropriately secure non-interactive authentication: they require either the manual entry of a decryption key or the handling and storage of secret data "in the clear" on a physical medium. Public-key authentication can be made non-interactive, however, using a tool called an "agent": a resident process that provides access to a decrypted private key via a socket. In contrast to a "passphraseless" private key, the decrypted private key is stored only in-memory, and is not retrievable via the socket, making compromise of the key much less likely.

Agent processes are typically ephemeral, used primarily for convenience when authenticating user account credentials; but an SSH agent can also be used to provide the non-interactive authentication necessary for automated privileged operations without sacrificing the security of the authentication secret. We use Puppet[6], a declarative configuration management system, to initialize a per-host agent process with a socket defined at a static path. With an agent in place, the decryption passphrase need only be entered once per host. (Puppet can also serve as a convenient distribution mechanism for the public key.)

[6] *Puppet*, Puppet Labs http://www.puppetlabs.com/ .

```
class s_remote_shell::root
{
    file
    { "/root/.ssh/authorized_keys":
        owner => "root",
        group => "root",
        mode => "600",
        source => "puppet:///modules/s_remote_shell/root/ssh/id_rsa.pub",
    }

    exec
    { "root-ssh-agent":
        command => "/usr/bin/ssh-agent -a /root/.ssh/agent.default",
        creates => "/root/.ssh/agent.default",
        require => File['/root/.ssh'],
    }

    file
    { "/root/.ssh":
        ensure => directory,
        owner => "root",
        group => "root",
        mode => "700",
    }
}
```

The static definition of the SSH agent socket, used by the SSH client during public key authentication, permits supporting SSH clients to be configured to access to the cached, decrypted private key without the need to store the decrypted private key on disk. This allows system activities and utilities (examples of which are described in the section *Application-specific configuration*) to execute privileged commands on remote hosts without risking exposure of the credentials that provide privileged access.

4 Private key distribution

OpenSSH provides the command `ssh-add` as an interface to the agent process, allowing private key identities to be added to a local agent. As with many agent-supporting applications, `ssh-add` interacts with the agent via the socket specified in the environment variable `SSH_AUTH_SOCK`.

```
SSH_AUTH_SOCK=/path/to/agent.sock ssh-add /path/to/key
```

For encrypted private keys, `ssh-add` prompts for a passphrase. It then decrypts the key and caches it in the agent process via the specified socket.

Even only once per host (per boot), manually adding keys to a sufficiently large number of hosts is tedious. A script makes it relatively trivial.

```
#!/bin/sh
# multi-ssh-add.sh /path/to/key node1 [node2 [...]]

identity_f=$1
shift

read -p "Enter passphrase for identity: " -s passphrase; echo

for node in $@
do
    echo -n "$node: " 1>&2
    remote_identity_f=‘ssh $node mktemp | tail -n 1‘
    ssh <$identity_f $node "cat - >$remote_identity_f"
    ssh <<<$passphrase $node sudo ssh-add $remote_identity_f
    if [ $? -ne 0 ]
    then
        echo 1>&2
    fi
    ssh $node rm $remote_identity_f
done
```

This sample script reads the passphrase into a variable, copies the private key to each trusted node, adds the key to the system agent using the stored passphrase, and removes the key from the remote filesystem. The encrypted key need only be maintained on a single host from which credentials are distributed to other hosts. With only minor inconvenience to sufficiently stable environments, the private key can even be maintained externally to the network: it need only be present on a networked host during the initial distribution of credentials.

(This script assumes that the SSH environment of the `root` account has been configured with knowledge of the static agent, as described in the section *Non-interactive, non-login SSH connections.*)

5 Application-specific configuration

Once a static `ssh-agent` socket has been defined, client applications that require the ability to execute privileged commands on a remote host must be configured to access the agent. Such configuration typically entails defining the variable `SSH_AUTH_SOCK` in the environment of the application.

5.1 Interactive shell profile

The base case for such configuration is the use of the `ssh` command from a `root` account shell.

```
anderbubble@node1:~> sudo -i
node1:~ # ssh node2-a
Last login: Sat Jun 06 14:00:00 2008 from node1.example.com
node2:~ #
```

This is most simply accomplished by defining `SSH_AUTH_SOCK` in the file `~root/.profile`.

```
SSH_AUTH_SOCK=/tmp/ssh-agent/agent.default; export SSH_AUTH_SOCK
```

We can, again, use Puppet to simplify this configuration by defining the profile of the `root` account in our manifest.

```
file
{ "/root/.profile":
    source => "puppet:///modules/s_remote_shell/root/profile",
    owner => "root",
    group => "root",
    mode => "644",
}
```

5.2 `sudo` environment

We use sudo[7] for all privilege escalation, and support its use in an idiom for running a privileged remote command.

```
anderbubble@node1:~> sudo ssh node2 "hostname; id"
node2
uid=0(root) gid=0(root) groups=0(root)
```

[7] *sudo*, GratiSoft, Inc. http://www.gratisoft.us/sudo/ .

`sudo` executes commands (in this example, the SSH client) in the environment of the executing user account, not the targeted user account. Thus, the definition of `SSH_AUTH_SOCK` in the profile of the `root` account is not available.

Sudo version 1.7.0 adds support for the `env_file` configuration directive, allowing `/etc/sudoers` to define a list of environment variables that should be defined in the `sudo` environment. Defining `SSH_AUTH_SOCK` in this way makes the configuration available in all `sudo` contexts.

For versions of `sudo` prior to 1.7.0, users must define `SSH_AUTH_SOCK` in their own environments (e.g., in `~/.profile`), similarly to the definition in `~root/.profile`.

5.3 Agent forwarding for nested `ssh`

Some applications (e.g., the IBM General Parallel File System[8]) require the ability to execute commands via a nested `ssh` connection.

```
anderbubble@node1:~> sudo ssh node2 "ssh node3 'hostname; id'"
node3
uid=0(root) gid=0(root) groups=0(root)
```

The first instance of `ssh` (*node*1 → *node*2) executes in a non-interactive, non-login shell. Some shells (e.g., Bash[9]) do not source any profile scripts (e.g., `~root/.profile`) in this case. As a result, the second instance of `ssh` (*node*2 → *node*3) executes with `SSH_AUTH_SOCK` undefined, leaving it unable to authenticate.

The simplest method for providing access to a privileged agent by a nested `ssh` process uses a feature of the SSH protocol called agent "forwarding." When the `ssh` configuration directive `ForwardAgent` is set to "yes" (e.g., in `~root/.ssh/config`), the target `sshd` process provides access to the source agent socket via a local socket. The path to this ephemeral socket is recorded in the `SSH_AUTH_SOCK` variable in the target environment, providing transparent, secure access to the privileged agent via the SSH connection.[10]

```
file
{ "/root/.ssh/config":
```

[8] *General Parallel File System Administration and Programming Reference, Version 3 Release 3*, IBM.

[9] *bash*, GNU. http://www.gnu.org/software/bash/bash.html .

[10] Hatch, B. *SSH and ssh-agent.* http://www.symantec.com/connect/articles/ssh-and-ssh-agent .

```
    source => "puppet:///modules/s_remote_shell/root/ssh/config",
    owner => "root",
    group => "root",
    mode => "644",
}
```

(Agent forwarding can, alternatively, be activated per-connection with the use of the `-A` option to `ssh`.)

5.4 Remote agent access in non-interactive, non-login shells

Agent forwarding allows remote commands to authenticate using credentials stored in the local agent, but some use cases (e.g., *Private key distribution*) necessitate explicit use of the remote agent.

```
ssh <<<$passphrase $node sudo ssh-add $remote_identity_f
```

`SSH_AUTH_SOCK` must be defined in the remote environment to point to the remote agent socket. This could be accomplished by hardcoding the variable definition in the remote command, but OpenSSH supports a more general solution: per-account environment variables can be included explicitly during any SSH connection by defining them in `~root/.ssh/environment`. By defining `SSH_AUTH_SOCK` in this file, even non-interactive, non-login shells will have knowledge of the remote agent.

```
file
{ "/root/.ssh/environment":
    source => "puppet:///modules/s_remote_shell/root/ssh/environment",
    owner => "root",
    group => "root",
    mode => "644",
}
```

(This solution assumes that the `PermitUserEnvironment` configuration directive is set to "yes" on the target host. Allowing a user-defined environment in `sshd` could allow users to defeat security measures that depend upon dynamically-linked restricted shells.[11])

[11] *rssh security implications*, http://rssh.sourceforge.net/security.shtml .

5.5 Cron

The *crontab* of the `root` account executes under a non-interactive, non-login shell similar to that used to execute forced remote commands; but configuring cron to provide access to a static agent is as simple as defining the variable `SSH_AUTH_SOCK` on its own line in the table.

```
SSH_AUTH_SOCK=/root/.ssh/agent.default
0 0 * * * ssh node2 "echo 'Hello,' `hostname`\!"
```

5.6 Daemons

Some daemons (e.g., xCAT[12]) expect to be able to run privileged remote commands via `ssh`. The environment of such daemons is typically constructed during the execution of the daemon `init` script. Many daemon `init` scripts already provide a configuration mechanism that allows the manipulation of the initialization environment. Red Hat, and its derivates, often source scripts with names of the form `/etc/sysconfig/$daemon`.[13] Defining `SSH_AUTH_SOCK` in such a file injects the necessary definition into the daemon environment.

```
file
{ "/etc/sysconfig/xcat":
    source => "puppet:///modules/xcat/sysconfig/xcat",
    owner => "root",
    group => "root",
    mode => "644",
}
```

Each operating system adheres to its own set of daemon initialization idioms. Check the documentation and `init` scripts for a facility similar to the one described here. In the worst case, it might be necessary to explicitly edit the daemon `init` script to include a definition for `SSH_AUTH_SOCK`.

```
file
{ "/etc/init.d/xcat":
    source => "puppet:///modules/xcat/init.d/xcat",
    owner => "root",
    group => "root",
```

[12] *Extreme Cloud Administration Toolkit*, http://xcat.sourceforge.net/ .
[13] Red Hat Enterprise Linux 5.5 Deployment Guide, Red Hat, Inc.

```
    mode => "744",
}
```

6 Support in future SSH implementations

Deployment of a privileged authentication system based on static, per-host SSH agents is possible with the existing OpenSSH implementation, given the additional work of defining `SSH_AUTH_SOCK` in each relevant environment (as exemplified in *Application-specific configuration*). Deployment would be simplified, however, with the addition of a new configuration directive to the underlying SSH implementation.

The existing OpenSSH implementation provides only one means by which the location of the SSH agent socket can be defined: the `SSH_AUTH_SOCK` environment variable. However, OpenSSH could be modified such that a static socket could be defined as a default agent in the client configuration (e.g., at `~root/.ssh/config`).

```
AgentSocket ~/.ssh/agent.default
```

The implementation of the `AgentSocket` configuration directive would eliminate the need for execution environments to be configured individually, consolidating the configuration to a single file. Such a facility could likely take advantage of existing support for agent forwarding which already provides the ability to define `SSH_AUTH_SOCK` in the remote environment for an ephemeral agent socket.