ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Torino

INFN-12-20/TO
30th november 2012

SSH AUTHENTICATION USING GRID CREDENTIALS

Dario Berzano[1]

[1)] *INFN - Sezione di Torino, Via P. Giuria 1, I-10125 Torino, Italy*

Abstract

SSH is one of the most widely used tools in Unix. Apart from opening remote shells, its most intriguing feature is the capability to tunnel TCP connections, providing for both a secure channel and an authentication mechanism for generic services lacking them. We will show how to setup a very simple infrastructure that temporarily authorizes users presenting valid X.509 credentials, and in particular Grid users, to use SSH by performing the authentication through HTTPS beforehand. We will also compare this method with tools that provide similar functionalities, such as GSI-Enabled OpenSSH. Finally, a brief description of PROOF on Demand for the LHC ALICE experiment is presented as a use case.

Published by ***SIDS–Pubblicazioni***
Laboratori Nazionali di Frascati

CCR-42/2011/P

1 SSH FOR GRID SERVICES

SSH (Secure Shell)[1] is the ubiquitous Unix tool used to establish encrypted connections to remote hosts. Although it is generally known as a secure replacement for Telnet[2], a SSH TCP connection is a generic encrypted tunnel that multiplexes several data channels in parallel, which may carry either a shell or a forwarded network connection like a TCP connection[3]. Thanks to such capabilities there exist services that do not even implement security mechanisms for remote data transmission, as they fully rely on the encryption and authentication infrastructure provided by SSH itself.

Grid users have credentials they use to access Grid services: a personal X.509 certificate, issued by a trusted Certification Authority, and a pair of keys, a public one and the private counterpart. The X.509 certificate is sent over the network during the authentication and has the public key stored within.

Grid services normally accept “proxy” certificates (temporary certificates with a limited validity created using the user’s certificate as issuer) while many Web-based Grid services (for instance the MonALISA[4] web interface of the LHC ALICE Experiment[5]) directly understand the user’s certificate: for this reason, the credentials are often installed in user’s browser too.

There exists one more class of services that require either the capability to open a remote shell, or the encapsulation of data through an authenticated and encrypted connection – most notably, tools used to leverage interactive parallel computing like PROOF on Demand[6].

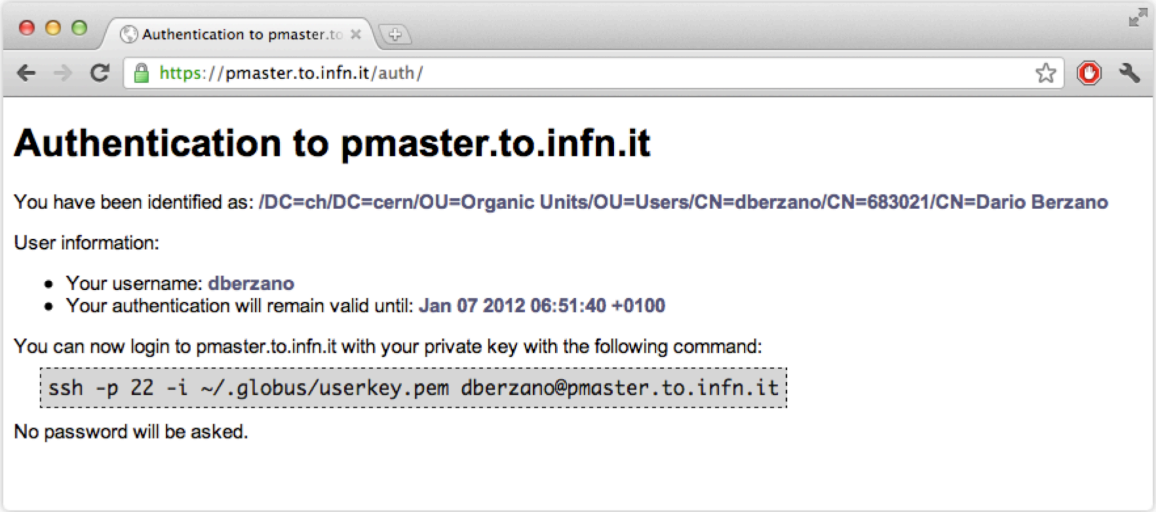
Using SSH by maintaining the usual Grid credentials would be extremely convenient. There actually exists a patched SSH version, called GSI-OpenSSH[7], which features Grid proxy authentication, that requires a custom *sshd* to be compiled on the server and one custom *ssh* client compiled for each user’s machine.

We took however the decision to comply with the mainstream distribution of SSH because security updates for such a critical system component are deployed more quickly and they are easier to apply: moreover, we don’t want to force our users to install and maintain a custom client.

The authentication schema we are going to illustrate in the next paragraphs has been achieved by complying with the constraints of zero additional requirements for the end user and full compatibility with the existing SSH and user’s Grid credentials.

1.1 SSH public key authentication

SSH supports a wide variety of authentication methods. We are particularly interested in the public key authentication[8] mechanism.


Authentication to pmaster.to
https://pmaster.to.infn.it/auth/

Authentication to pmaster.to.infn.it

You have been identified as: /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=dberzano/CN=683021/CN=Dario Berzano

User information:

- Your username: dberzano
- Your authentication will remain valid until: Jan 07 2012 06:51:40 +0100

You can now login to pmaster.to.infn.it with your private key with the following command:

```
ssh -p 22 -i ~/.globus/userkey.pem dberzano@pmaster.to.infn.it
```

No password will be asked.

FIG. 1: The web page presented to the user by the PHP web application when authentication succeeds.

Public key authentication works as follows. A pair of keys (a private key and the matching public key) is generated on the client: the private key is known by the client only, while the public key must be known and authorized by the server.

As soon as the client tries to connect as a certain user, it proves to the server that it has access to the private key, while the server checks if the corresponding public key is in the list of authorized keys for the specified user name.

1.2 Grid credentials and the SSH private key

The Grid private key is an RSA[9)] key generally stored under the user's home directory in PEM[10)] format. Incidentally, *ssh* accepts private keys in this exact format when it tries to authenticate using the public key authentication method. This means that a Grid user could in principle use directly this key file to connect to a remote host by issuing a command similar to:

```
ssh -i ~/.globus/userkey.pem user@host
```

without requiring to create a Grid proxy certificate. We only have the problem to tell the server to authorize the corresponding public key, given that the associated user certificate is valid. The solution we propose is to perform authentication outside of SSH and beforehand, using another standard protocol supporting X.509 certificates natively such as HTTPS.

```
# Certificate and key for this host
SSLCertificateFile /etc/grid-security/hostcert.pem
SSLCertificateKeyFile /etc/grid-security/hostkey.pem

# Certificates of the authorized CAs (from AliEn)
SSLCACertificatePath /etc/grid-security/cadir

# Maximum certificate depth
SSLVerifyDepth  10

<Directory /var/www/html/auth>

  # Require SSL auth
  SSLVerifyClient require

  # Set envvars for CGI scripts to some small data
  # plus the whole encoded certificate
  SSLOptions +StdEnvVars +ExportCertData

</Directory>
```

FIG. 2: Relevant portions of the Apache 2 configuration for the authentication web application.

1.3 Certificate authentication via HTTPS

The HTTPS protocol is a way to perform HTTP communications encapsulated into a SSL/TLS tunnel[11)], and it is practically used to establish an encrypted connection between a web server and a browser.

Our HTTPS-for-SSH authentication works in two steps.

In the first step, the browser tries to authenticate using the user's certificate, and the web server checks its validity: authentication does not succeed if the certificate has been issued by an unknown CA (certification authority), if the certificate is included in a CRL (certificate revocation list) or if the certificate has expired.

If authentication succeeds, the web server sends the user's certificate to a web application written in PHP[12)] which will take all the necessary steps to extract the public key shipped with the certificate and authorize it for a certain user name.

2 SSH X.509 AUTHENTICATION DETAILS

The workflow of this authentication procedure from the user's perspective is rather straightforward.

- The user points the browser to *https://hostname/auth*. Assuming the browser already knows user's Grid certificate, it uses it to authenticate to that web page. In case of success a web page like the one in Fig. 1 is shown to illustrate the procedure to do *ssh* and the expiration date of the authorization.

```
// Default SSH port
$sshPort = 22;

// Authorized keys directory
$sshKeyDir = '/etc/ssh/authorized_keys';

// Maximum token validity, in seconds
$maxValiditySecs = 3600;

// Plugin to retrieve user name from subject
$pluginUser = 'alice_ldap';
```

FIG. 3: Example configuration file *(conf.php)* for the PHP web application.

- The user does *ssh* on the target machine: the private key must be accessible by *ssh* on the filesystem.
- When the authorization expires it will be sufficient to go back to the authentication web page to renew it.

2.1 Web server configuration

Any SSL-enabled web server capable of performing certificate authentication may be used for our purposes. The following guidelines focus on Apache 2 *mod_ssl*[13)].

First of all we have to tell the web server which certificates should be recognized as valid: the *SSLCACertitificatePath* directive can be used to point to a path of Grid CAs, for instance.

Apache 2 can pass information to an underlying CGI web application (such as our PHP script) by means of environment variables: by default no variables containing SSL information are set. By setting the option *SSLOptions* as shown in Fig. 2 we are exporting these variables *(+StdEnvVar)* and the full client certificate in PEM format *(+ExportCertData)*: this last option is particularly important because the public key will be extracted from it.

2.2 *sshd* configuration

By default, *sshd* looks for the authorized keys in a file stored under each user's home directory. Since we do not want users to tamper those keys, even accidentally, we set the *AuthorizedKeysFile* directive in *sshd* configuration file to a directory owned by *root* containing one file for each user like this:

```
AuthorizedKeysFile /etc/ssh/authorized_keys/%u
```

where *%u* is substituted with the user name. For security reasons the SSH server automatically ignores key files stored in a system directory that are not owned by *root*.

Each key file may contain more than one key for each Unix user: this is useful when mapping users to a set of pool accounts.

2.3 Web application configuration and Unix user name

The PHP web application, available on GitHub[14] must be installed under a directory of the web server served by SSL. Any directory can be used.

There is only one configuration file (see Fig. 3) with a few configuration options: the authorized keys directory *$sshKeyDir* must match *sshd*'s *AuthorizedKeysFile* directive (without the final *%u*).

Since the method to retrieve the user name from the user's certificate is not unique, a plugin method has been adopted. The plugin consists of an external PHP script to be saved in the *plugins/user* directory: the desired plugin can be chosen by setting the variable *$pluginUser* to its file name without the *.php* extension.

The file should contain the implementation of a function with the following prototype:

```
function authGetUser($certSubject, &$userName, &$validitySecs, &$errMsg);
```

This function (which is expected to return *true* on success and *false* if authentication shall not continue) takes as arguments:

- *$certSubject* – the certificate's subject;
- *&$userName* – a variable where the Unix user name will be returned – *root* is not considered safe and will be discarded by the caller;
- *&$validitySecs* – a variable where the validity, expressed in seconds, shall be stored – unlimited validity is not allowed;
- *&$errMsg* – an array where to append error strings.

This function (which returns *true* on success and *false* if authentication shall not continue) takes as arguments:

2.4 Public key format conversion and expiration

The public key included in the X.509 certificate needs first to be extracted, then to be converted into the SSH public key format.

For this task we use phpseclib[15], a complete and popular implementation of SSH, SFTP and RSA written natively in PHP.

SSH public key format stores keys in one long ASCII line, and allows for a free field called the “comment” field. We set this field to a special string containing a human-readable expiration date, for instance:

```
Valid until: Jan 06 2012 06:55:52 +0100
```

This field will be used by an external program to identify and prune expired keys.

2.5 Storing and pruning keys

Superuser permissions are needed to access the directory of public keys, however the web server does not run as *root*. A program external to the PHP script has been written to manage the authorized keys: our security policy is to allow the user that runs the web server to invoke this program (and this program only) as *root* without requiring a password by means of *sudo*, resulting in the following *sudoers* configuration entry:

```
webserver-user ALL=(ALL) NOPASSWD: /path/to/www/auth/keyskeeper.sh add
```

The script works in two modes: an “add” mode that adds, or replaces, an existing key for a given user to the authorized keys, and an “expiry” mode that scans all the keys in all files and removes the expired ones. If no more valid keys are left in a file, the file is deleted.

The first mode is invoked with *sudo* by the PHP application to authorize a key. The second mode should be configured in *crontab* to be run periodically by the *root* user, like this:

```
*/10 * * * * /path/to/www/auth/keyskeeper.sh expiry
```

where we have chosen to scan all the keys every ten minutes.

The script features a wait-lock mechanism in order to prevent potential disaster due to concurrent access to the authorized keys repository.

2.6 The client

Although a client is not needed by design, a small and portable Ruby application has been written in order to automatize the authentication procedure. It is a command-line application that contacts the HTTPS authentication server in background, and immediately invokes *ssh* on success.

The application caches the authentication token received and reuses it on subsequent calls, avoiding further calls to the authentication server until the token expires.

When the application contacts the server, it asks for a XML output, instead of the standard human-readable HTML web page.

The user invokes the client by specifying the remote server, plus some optional parameters:

```
httpssh authserver [--cadir dir] [--cert usercert.pem] [--key userkey.pem] [ssh options]
```

The contacted URL is guessed to be *https://authserver/auth*, or it can be directly specified in place of *authserver*.

Optional parameters are:

- *--cadir dir* – directory of the valid CAs to identify *authserver*'s certificate;
- *--cert usercert.pem* – the file containing user's certificate in PEM format;
- *--key userkey.pem* – the file containing user's private key in PEM format.

Further parameters are passed as-is to *ssh*.

3 USE CASE: PROOF ON DEMAND AND ALICE

PROOF on Demand[6] is a tool that simplifies the setup of a dynamic PROOF[16] cluster, an interactive parallel computing framework based on ROOT[17] very popular in the HEP community and in particular in the LHC ALICE experiment[18].

In a static PROOF setup, conventional Grid proxy authentication is performed by a single PROOF daemon that serves many users. On the contrary, in a dynamic PROOF cluster, each user requests PoD to start a personal PROOF daemon: for this reason authentication is constrained to occur beforehand, and this Grid authentication mechanism for SSH becomes convenient. A dynamic setup of PROOF on Demand over a cloud-computing environment which features the Grid SSH authentication mechanism described in this paper has been set up in the Tier-2 computing centre in Torino[19].

Another reason is that every PROOF server requires its own port, while tunneling every connection into SSH like PoD does requires only ports 22/TCP (for SSH) and 443/TCP (for HTTPS) to be opened.

3.1 LDAP for Unix users in ALICE

In the context of the ALICE experiment, we have decided to use no pool accounts for PoD: every ALICE user has an Unix-like user name maintained in a LDAP database.

In order to allow ALICE users to connect via SSH to the target machine, a PHP plugin that maps the user's subject to the proper Unix user name using the ALICE LDAP database has been written: the plugin is distributed along the rest of the code via GitHub[14].

On the SSH target machine, ALICE users are not added manually; instead, system's name service switch has been configured to read users information both in */etc/passwd* and on the ALICE LDAP. Moreover, the PAM[20] *mkhomedir* plugin has been enabled in order to have the home directory created upon the first login.

4 REFERENCES

(1) Ylonen T. and C. Lonvick Ed. RFC 4251 (2006)

(2) Postel J. and Reynolds J.K. RFC 854 (1983)

(3) Ylonen T. and C. Lonvick Ed. RFC 4254 (2006)

(4) Legrand I., Voicu R., Cirstoiu C., Grigoras C., Betev L. and Costan A. Comm. of the ACM 52 (9) 49

(5) http://alimonitor.cern.ch/

(6) Malzacher P. and Manafov A., J. Phys.: Conf. Ser. **219** 072009 (2010) (see also http://pod.gsi.de/)

(7) http://grid.ncsa.illinois.edu/ssh/

(8) Ylonen T. and C. Lonvick Ed. RFC 4252 (2006)

(9) Rivest R.L., Shamir A., Adleman L.M. U.S. Patent 4405829 (1983)

(10) Linn J. RFC 1421 (1993)

(11) Rescorla E. RFC 2818 (2000)

(12) http://www.php.net/

(13) http://httpd.apache.org/docs/2.0/mod/mod_ssl.html

(14) https://github.com/dberzano/sshcertauth/

(15) http://phpseclib.sourceforge.net/

(16) Ballintijn M., Brun R., Rademakers F. and Roland G. Arxiv preprint physics/0306110 (2003) (see also http://root.cern.ch/drupal/content/proof)

(17) Brun R. and Rademakers F. Nucl. Instr. and Meth. in Phys. Res. A **389** 81 (1997) (see also http://root.cern.ch/)

(18) http://aaf.cern.ch/

(19) Berzano D., Bagnasco S., Brunetti R., Lusso S. in proceedings of XIV Advanced Computing And Analysis Techniques – Uxbridge, UK (in press) (2011)

(20) Samar V. and Schemers R. Open Group RFC 86.0 (1995)