Alternative Authentication

Here we discuss alternative means of managing custom authentication without using PAM.

Custom Challenge-Response Hashing Functions

This approach, called "Scheme #1" below, can provide much better over the transmitted credentials, but requires custom coding to integrate it properly with the site password database.

Background: There are two basic password schemes for remote login, consider their benefits and risks according to your site's needs.

Scheme #1 - Challenge/Response on Hashed Passwords

> - *Best practice basic protection*
> - *No passwords transmitted over the network*
> - *Usually requires a custom account/password database*
> - *Requires implementation of matching hash on server and clients*

This approach has the advantage that it does not send a recoverable password over the network, and protects against replay attacks (in which someone captures the network traffic of a successful login, and replays that sequence later to gain access). The main disadvantage is that it requires a special "account" database to be created and stored on the server side. It will also likely require coding to implement a matching hash function on the server and for each type of client.

The scheme works like this: Prior to using the system, a hashed version of each user's password (see below) is stored into a server-side database, which might be a simple file on disk. During login, a fresh one-time random string, called the challenge, is sent to the UI, and a copy of that string is kept by the server. The UI prompts the person for their login name and password, then the UI performs two hash operations, in sequence. The first hash is identical to the one used to create the user's account entry on the server. Then the output of the first hash is combined with the server's random challenge string and hashed again. The output of this second hash is then sent to the server for validation, along with the plain-text of the user's login name. The server validates the login by redoing its own computation of the second hash, with inputs consisting of the one-time challenge that it previously sent to that client and its own existing pre-hashed entry for that user from the password database. It compares the result of its own computation to the one sent from the UI, and if they match then access is granted.

The hash used to create the initial password database, and also the client's first hash, is a one-way cryptophraphic-quality function applied to the string of characters formed by the combination of the user's login name, their chosen password, and a permanent site-wide "realm" string. The realm string, such as the company name or website name or any other locally unique string, helps to ensure that attackers can't use look-up tables of pre-computed hashed passwords to trivially crack the password database. For example, if the user "baker" chooses the password "Cherry3.417" then the input to the hash function might be "baker|Cherry3.417|example.com". After hashing with MD5 or a stronger function, this input string will be converted to a jumble of new output characters. The hash isn't random, it will always generate a repeatable output when the same inputs are given. A good hashing function will never be "reversable" -- so the input can't be recovered by running some function on the hashed output. Furthermore, a good hash will make it very (very) unlikely that two different input strings will map to the same output.

The initial hash ensures that user passwords are not stored on disk in a recoverable form, and that they are not sent in plain-text form over the network. The hashed value can only be used in a comparison to another hash computation. The second challenge-based hash ensures that the transmitted string is itself not "password-equivalent" -- meaning that it will not match a string copied out of the server's password database file. Since the challenge string varies on every login attempt, the second hash also helps to ensure that login access cannot be gained by replaying a previously captured login transaction.

The main difficulty here is in creating, maintaining, and accessing the storage scheme for the server-side password hash. LDAP might be a handy candidate for this purpose. Secondarily, it can sometimes be difficult to find (and correctly implement) a hashing algorithm that behaves identically on the server and all possible clients.

Scheme #2 - A Basic "/etc/passwd" Server-Side Approach

> - *Hashing performed on the server only*
> - *Passwords are sent from the client in encoded, but recoverable form*
> - *A classic, but long obsolete unix tradition*

This approach is easier to implement given an existing simple /etc/passwd style database of password hashes. The disadvantage is that network snooping can be used to recover the password which is likely to also be the user's main desktop/system password. This scheme is more secure if the network connection between the client and server is itself trusted, for example when using SSL.

Here is the sketch: After the user is prompted for a login name and password, The client-side handler for the dialog encodes the password in some way, and then sends the login request to the server. The server receives the request and decodes the message, thus recovering the plaintext password and user name that the user just entered. The server then runs a one-way hash on the password and compares it to a previously stored version of that user's password hash (hashed using the same algorithm, of course). The main weakness stems from the fact that the transmitted password encoding is required to be recoverable. it should also be noted that the client-side encoding itself might be implemented in source-available context, such as JavaScript functions easily visible within a web page, thus attackers could easily see the encoding algorithm being used.

Note that this scheme is *not* improved by having the client, rather than the server, perform the one-way hash of the password before sending it to the server. Doing so causes the transmitted hash to become "password equivalent" -- as if unhashed passwords were being stored on the server -- making replay attacks trivially effective. However, if replay attacks are not a concern, then having the client do the hash might allow Tractor to use the normal system /etc/passwd database without transmitting those passwords over the wire. However, in the case of /etc/passwd specifically, the Dashboard client code would need an external means of acquiring the existing per-user "salt" values in each passwd entry in order to compute the correct crypt-style hash. For a small number of users these might be built directly into the trSiteDashboardFunctions script in a potentially automated way.

Enabling Tractor Password Validation

- Password validation occurs using a **pair** of site-defined functions, one that sends credentials from the web-browser in the Dashboard client, and the other that validates the credentials on the tractor-engine server.
- The pair of functions work together to implement the validation, scheme, examples include the two approaches outlined above.
- The client side code must be inserted into the file trSiteDashboardFunctions.js located in the Tractor "config" directory. Within that file is a function called trSitePasswordHash which must be modified to return an appropriate password hash or encoding. The actual computation of the hash or encoding can be in this file or it can be elsewhere, as appropriate to each site's plug-in availability and external code conventions. If trSitePasswordHash returns the JavaScript value null then password processing is ignored by the Dashboard. Otherwise the function should return the new computed string, and the result will be sent to the Engine, along with the login name entered by the user.
- Upon receipt of the login credential strings, the Engine invokes an external validation program, passing in the strings as input on stdin. The program to be invoked is specified in the file crews.config with the dictionary key SitePasswordValidator.
- An example validator script, written in python, is supplied in the config directory. Here is an example crews.config entry to invoke it:

> *"SitePasswordValidator": "python ${TractorConfigDirectory}/trSiteLoginValidator.py"*

- The exit code of the invoked program determines whether the password and user name pair are valid. The program should exit with 0 (zero) to indicate success, and non-zero to indicate that login is not allowed.
- The example program, trSiteLoginValidator.py illustrates computing a one-way hash on a plaintext password in the style of /etc/passwd crypt(), and then comparing the hash with the value retrieved from the unix call "getpwnam()", which can access simple /etc/passwd or NIS (yp) entries. **NOTE** this simple example assumes that the Dashboard client-side function trSitePasswordHash() has simply returned the inbound plaintext password as its output -- for the simplicity of illustration -- a real implementation should either strongly encrypt the password for transmission, or switch to a challenge/response scheme using a tractor-specific database of passwords ("Scheme #1" above).