

TACACS Authentication Methods

Michael Martin

A year ago I started contributing to SearchNetworking.com with a couple articles on Terminal Access Controller Access Control (TACACS). I began with Building a secure TACACS+ environment and followed up with Configuring secure authentication on Cisco routers and switches. This month's article is the first of three additional articles that will complete the series I started a year ago. The previous articles focused on building a TACACS+ server running on Linux and implementing authentication, authorization and accounting (AAA) services on Cisco routers and switches.

We'll continue on this track, discussing the tac_plus source code for readers interested in implementing TACACS for dial services. This month's focus will be examining the authentication methods available using the tac_plus daemon. Then we'll look into implementing TACACS command authorization utilizing the IOS multi-level security model and discuss some ways to extract useful information out of the TACACS accounting data using some simple shell scripts. The final installment on using TACACS jumps around a bit in terms of the material it covers, dealing both with Cisco IOS and TACACS configuration, but the overall theme of the material relates to one central concern: controlling access.

Authentication

Access authentication is a simple concept -- something is out in the open, available to anyone, but only some can actually utilize it. When granting access, the administrator needs to consider three factors: first, a method for controlling access (the means of id for establishing trust, i.e., text password, seed and token, public-key, etc); second, the term of access (how long does this user have access to this system or systems?); and third, the depth of access, including what commands and which systems can be accessed, and the level of control the user has on the configuration of the system.

Method

The tac_plus daemon supports three authentication methods: clear text, Data Encryption Standard (DES -- local and remote), and S/Key. The tac_plus daemon provides authentication services for VTU login, Point-to-Point Protocol authentication via Password Authentication Protocol (PAP) and Challenge/Handshake Authentication Protocol (CHAP), and AppleTalk Remote Access (ARAP) by default. CHAP and ARAP can only utilize clear text, as required by their protocol definitions. Support for Microsoft CHAP (MSCHAP) is available, but support needs to be built in at compile time. Here is the Makefile section for adding MSCHAP support:

```
# Microsoft CHAP extension support. See the user's guide for more
# information.
# MSCHAP = -DMSCHAP
# MSCHAP_DES = -DMSCHAP_DES
# MSCHAP_MD4_SRC = md4.c
```

Tac_plus follows a recursive tree hierarchy when performing authentication lookups. The tree starts with the user. If no authentication method is found and a group is configured, the group is examined. A user can only be a member of one group, but groups can be members of other groups (user defined attributes override those set in a group). This approach provides a great deal of flexibility; groups can be used to build "command and access" sets, which can be nested together for user assignment (we will look at this more later on).

TACACS Authentication Methods

Michael Martin

```
User
/
Group
/
Group
/
Group
```

TACACS+ configuration parameters also follow a hierarchy. Attribute/Value (A/V) pairs (the device employed for defining parameters) use a "nested" hierarchy for defining attributes. This nested hierarchy is built using two classes of A/V pairs: root pairs and parameters pairs. A root pair is used to construct an AAA object definition. For example, the user and group A/V pairs are root pairs; they function like a bucket, containing attribute values. Parameter pairs are those attribute values that define the specific options and rules for the root pair. Here is an example of the A/V hierarchy. A/V sets that have nested A/V use brackets to indicate the beginning and end of the nested attributes:

```
user = moe {
login = skey
}
```

The best way to understand the tac_plus configuration hierarchy is to look at an example. Above, the user A/V pair is the root pair and the login A/V pair is the parameter pair.

When using tac_plus for router and switch authentication the *login = password_method* A/V is used. In Building a secure TACACS+ environment, the sample tac_plus configuration file used S/Key (a seed and token method) for the user password authentication method. DES and clear text passwords are also supported. There are two options for using DES authentication. The first option takes a user's already encrypted password from the /etc/shadow or /etc/password file and pastes into the tac_plus configuration using the *login = des* A/V pair. Alternatively, a user can generate a DES password with the generate_passwd tool (included with the tac_plus source distribution) and send it to an administrator, who can paste it into the configuration file. (Don't laugh; have you ever used SSH with RSA keys?) This approach also uses the *login = des* A/V pair.

The second DES option uses the server's /etc/shadow file (or just a standalone file) and scans the file for the user's DES password using the A/V pair *login = file "file path."* The last method -- simple clear text -- uses the A/V pair *login = cleartext "password."* Like the name states, the clear text method uses a simple text string stored in the tac_plus configuration file to validate users. The username can be up to 31 bytes in length, with a password up to 254 bytes long. The clear text approach provides nothing in terms of password protection for the user. So it is clearly not ideal for situations requiring password security and some assurance that the user is who they appear to be. However, in scenarios where temporary passwords are needed when used in conjunction with a random password generator, this approach may be acceptable.

Here are tac_plus configuration examples using the clear text and local DES options (notice that quotation marks are not used to border the DES password entry):

```
user = god {
login = cleartext "spepass03"
Revised February 19, 2003
```

TACACS Authentication Methods

Michael Martin

```
}  
  
user = god {  
login = des TgrRRYeHXHTzs  
}
```

So why the different password methods? It really comes down to a question of use. In large remote access server (RAS) dial-in, VPN and firewall environments that utilize tac_plus for AAA, there is often a large amount of "user churn." User accounts are being added and removed with regular frequency, and the users need to be able to reset their passwords without (hopefully) administrative assistance. In this scenario, the DES file method is the best approach. In fact, tac_plus provides a default user definition for just this scenario. The core user parameters are defined in the tac_plus configuration, and the password housekeeping function is offloaded to another password mechanism. Following is the A/V configuration for using a default user definition:

```
# Core System Configuration Values  
# Shared Key  
key = secretkey  
# Accounting File  
accounting file = /var/tmp/acctfile  
# Global Defaults  
default authentication = file /etc/shadow  
  
# User and Group Definitions  
  
user = DEFAULT {  
service = exec {  
priv-lvl = 1  
}  
}
```

The above configuration supports TACACS authentication for any user defined in the default authentication file.

As for the "password mechanism," enabling the Linux server to provide support for secure password change/reset is the next step. The setup is quite straightforward. Edit the */etc/shells* file and add */usr/bin/passwd* as a supported shell. User accounts are created using the shell flag (-s) of the */usr/sbin/adduser*, declaring the */usr/bin/passwd*. RAS users connect to the server using Secure Socket Shell (SSH). Users log in with their generic or temp passwords, then are prompted to set a new password and logged out after the password is set. Here is the output from a user "reset" session:

```
BASH>ssh -l testuser pserver.x2net.net  
testuser@pserver.x2net.net password:  
Changing password for user testuser.  
Changing password for testuser  
(current) UNIX password:  
New password: xxxxxxxx  
Revised February 19, 2003
```

TACACS Authentication Methods

Michael Martin

```
Retype new password: xxxxxxxx
passwd: all authentication tokens updated successfully.
Connection to pserver.x2net.net closed.
BASH>
```

For our purposes (to provide AAA services to routers and switches), the use of a single file containing configuration data and passwords has the advantage of being very portable. The "DES paste" (even the clear text) approach works well when the tac_plus service is running on a host that is not under your administrative control. Password authentication is not dependent on an external mechanism (like S/Key or DES file), and while it is clearly not the easiest means for administrating passwords, having a user generate a DES password and send it using PGP/GPG to an administrator who manages the tac_plus service is not overly burdensome to the users or administrator in a small environment. Using TACACS for "admin control" should be "low impact" in terms of administrative overhead. Unlike RAS/VPN environments, password changes are either infrequent or "policy enforced" and occur with scheduled frequency. Plus, user account information and configuration data in one file makes it easy to maintain multi-server environments. Configuration or user modifications can be quickly pushed to the other servers by sending a single file and restarting the service. Here is a simple Expect script that does just that:

```
#!/usr/local/bin/expect
# Command syntax tacconfrep.exp
#
# Set the server login and password
set user "root"
set upas "root"
set host [lindex $argv 0]
set file "/etc/tacacs.conf"
#
# Upload The TAC_PLUS config file
spawn sftp $user@$host
expect ":"
send "$upas "
expect "> "
send "put /etc/tacacs.conf /etc/tacacs.conf "
expect "> "
send "exit "
#
# Restart The TAC_PLUS daemon
spawn ssh -l $user $host
expect ":"
send "$upas "
expect "# "
send "kill -HUP `cat /etc/tac_plus.pid` "
expect "# "
send "exit "
```

Alternatively, if you are working only in a LAN/MAN environment and don't like to move files around, the tac_plus daemon loads the authentication table as a hash file in RAM when it starts, so a centrally

TACACS Authentication Methods

Michael Martin

stored (accessed using NFS, AFS or SMB) configuration file can be used. You will still have to reload the tac_plus daemon when changes are made, but once the service is running -- even if the share is lost -- the tac_plus service will be unaffected.

Term

Question: How long should you keep your password?

Answer: As short as is practical.

Text passwords are by far the most insecure method for controlling user access. Whenever possible, a one-time password (OTP) method like S/Key or SecureID should be utilized. Sadly, these methods are often seen as too burdensome for broad use in most user populations because they require the user to carry a token or password calculator in order to access the system. But if you have to use text passwords, you can at least limit your exposure by implementing password expiration.

Your site security policy should document the acceptable methods for validating authentication, including username format (length), RSA key sizes, acceptable password hashes, minimum password length, password life times, etc. Check with your security team before implementing anything. If you do not have a policy, you should work within your organization to develop one. The lifetime of a password should be gauged by the degree of exposure and liability the environment will be subject to in the event the password is compromised.

If you are issuing passwords for staff use on a non-public network, a lifetime of 60 to 90 days should be sufficient. Passwords for hosts on public networks should be really no more than 30 days. That may sound a little uptight, but look at your exposure, and then consider exactly why your users can use S/Key. Another "community" you should consider is contractors and vendors. If you are providing password for "external user" access, the password lifetime should only be for the term of access needed. Often a general "vendor" password is created for this kind of access. This is also a big no-no. Never to let a vendor piggyback on a "guest" or "general access" password (you really should not have these anyway). A "general use" password will only create problems when trying to track down who did what on the accounting system. They often are abused and are used by staff who forget their own passwords or let them expire. So a good rule of thumb is that when it comes to granting access to anyone outside your organization--as well as inside -- a specific ID and password is the way to go.

The tac_plus daemon supports password expiration for locally stored passwords (local DES and clear text). To set a password expiration date, use the A/V parameter pair *expires* = "*MMM DD YYYY*." The month value is the first three letters of the month name (upper or lower case), and the day and year are expressed in numbers. Refer to this tac_plus configuration file syntax example:

```
user = surferguy {
expires = "Jan 31 1999"
login = cleartext "apassword"
service = exec {
priv-lvl = 15
}
}
```

TACACS Authentication Methods

Michael Martin

The daemon starts a expire notice time 14 days before the password expires; it is displayed each time the user logins in. The notice looks like this:

```
Username: surferrguy
Password:
Password will expire soon
```

TESRTR#

Once the password has expired, the user is informed about why he/she is unable to log in:

```
Username: mjmartin
Password:
Password has expired
```

```
[system:/home/surferguy]>
```

If the "DES File" option is used with the `tac_plus` daemon, the `expire` AV pair is not consulted. Instead, the daemon uses the expiration date in the password file (it looks in the field used for the shell definition). By default, the Linux `/usr/sbin/useradd` command does not set an expiration date for passwords. To see the defaults used by the `/usr/sbin/useradd` command, use the `-D` flag or look at the file `/etc/default/useradd`:

```
[root@orion]# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
[root@orion#
```

To configure a user account with an expiration date, use the `-e` flag followed by the date with the syntax format `YYYY-MM-DD`. Each date value is expressed in Arabic numbers. The syntax looks like this:

```
[root@orion]# useradd auser -e 2002-02-03
```

If you want to set an expiration date as part of the default profile, use the command combination `/usr/sbin/useradd -D -e YYYY-MM-DD`.

Before we move on, let's quickly consider limiting the number of simultaneous sessions a user can have. While this idea may seem odd in our context, when used to control RAS sessions or access control via the authentication proxy service session control, using TACACS does have some advantages. By default, the `tac_plus` daemon places no restrictions on the number of user sessions. Session limiting is a Makefile setting, not a global configuration variable. To enable session limiting, set the `MAXSESS` value in the `tac_plus` Makefile:

TACACS Authentication Methods

Michael Martin

```
# Enforce a limit on maximum sessions per user. See the user's guide
# for more information.
#MAXSESS = -DMAXSESS
MAXSESS =
```

Once the limit is built in, it cannot be overridden. Any change in the session limit requires the daemon to be rebuilt.

Depth

Access control is a phrase that covers a lot of ground. When it comes to authentication, access control has two operational realms: access to the system and access within the system. Access to a system can be enforced through two means. The first is to enforce access control locally on the system itself. The Unix `inetd` super-daemon `tcpd` (TCP Wrappers) or applying IOS VTY access-class filters are examples of ways to enforce access to. The second is to enforce access control centrally as part of a centralized authentication method, such as Microsoft's Active Directory or Novell's NDS. The generic `tac_plus` distribution does not provide a means for centrally controlling access to systems. However, the "shrubbery distribution" `tac_plus` daemon does provide this capability.

The shrubbery distribution provides login access control for users by assigning them to groups. Access is defined using an access control list (ACL) A/V pair as a root pair. The ACL definition can contain one or more "permit" or "deny" parameter pairs. The access statements are written using standard Unix regular expressions. Here is an example ACL using some character set ranges and standard metacharacters:

```
acl = north_am_dev {
permit = 147.225.([12-16] | [33-48]).*
deny = 172.[16-32].*.*
deny = 10.*,*.*$
}
```

ACL filtering is done using the source address. The TACACS server checks to see if the user can access the host by examining the source address of the authentication verification request. The ACL example above permits connections to hosts within the ranges 147.225.12.0 through 147.225.16.255 and 147.225.33.0 through 147.225.48.255 and denies connections to 172.16.255.255 through 172.32.255.255 and 10.0.0.0 through 10.255.255.255. Like IOS access lists, each ACL ends with an implicit deny rule. So the same ACL example can also be written this way:

```
acl = north_am_dev {
permit = 147.225.([12-16]|[33-48]).*
}
```

While not the friendliest method around, the use of regular expressions provides a large degree of control and flexibility. If you are not familiar with using regular expressions, there are a number of articles on the Web. The book *Mastering Regular Expressions*, published by O'Reilly, is a must-have for anyone who is seriously writing scripts.

TACACS Authentication Methods

Michael Martin

Once the ACL has been created it needs to be assigned to a group A/V root pair. This is done using the ACL A/V pair as a parameter pair. Here is a syntax example:

```
Group = north_am_admin {  
Acl = north_am_dev  
}
```

Once the group has been created, the appropriate administrators are added to the group:

```
User = boston-admin {  
Login = skey  
Member = north_am_admin  
}
```

You now have a way to centrally control which administrators can access which system. Now, while you can achieve the same results using access-class VTY filtering, the VTY filter approach involves substantially more administration when changes needed to be made. And while this is a rather broad control example, an administrator could easily institute more granularity between users by nesting groups with more explicit permit or deny lists.

The primary method for controlling (command) access within the system is using the enable password. The IOS has three access modes: exec, enable-exec and configure-exec. By default, of the 15 privilege levels (there are actually 16 -- 0 through 15, but 0 is not really utilized) exec mode is privilege level 1. A user logs in to exec mode (prv-lvl 1) and then supplies an enable password (or secret, which is encrypted at creation and is preferred over an enable password, , unless the service password encryption has been enabled) to move into enable-exec mode (prv-lvl 15). The user can then move to configure-exec to make modifications to the device configuration.

TACACS provides the capability to centrally manage the enable password. To turn on external enable password verification, use the IOS AAA configuration command . TACACS+ is the only method supported to provide enable password verification; while RADIUS is an available option, it will not work. On the tac_plus configuration side, a user called \$enab15\$ is created, using clear text or local DES authentication. Here is the tac_plus configuration:

```
User = $enab15$ {  
Login = cleartext "Iamsuperman"  
}
```

This is a major benefit, in terms of administration. The capability alone is a great reason to implement TACACS. But there is a downside, and I quote from the *TACACS User Guide*, "Be aware that this does have the side effect that you now have a user named \$enab15\$ who can login ..." But never fear -- the Shrubbery distribution is here!

Along with providing access control to the router, the guys at www.Shrubbery.net have also implemented per-user and/or per-group access control within the router. Thank you, Shrubbery guys! To configure unique enable passwords, the parameter pair *enable = {password}* or *enable = file {file path}* is added to the user or group root pair. Here is the configuration:

TACACS Authentication Methods

Michael Martin

```
Group = north_am_admin {  
  Enable = cleartext "Iamsuperman"  
}
```

```
user = boston_admin {  
  enable = cleartext "Iamsuperman"  
}
```

Additionally, the distribution also supports enable access filtering using the parameter pair *enableacl = {acl name}*. Here is an example:

```
user          =          boston_admin          {  
enable        =          cleartext          "Iamsuperman"  
enableacl    =          boston_acl  
}
```

The Shrubbery distribution modifications address some of the shortcomings of the generic `tac_plus` distribution in the area of access control. However, as it turns out, IOS supports more the one method for access control, which brings us to our next installment. We will look at implementing a multi-level security model, named AAA lists, and hardcore command access control using AAA authorization.