

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

In its soul, IOS Authentication, Authorization, and Accounting (AAA) is about access control. This month, in part two of the advanced TACACS+ series, we are going to examine how we can become even more overbearing and oppressive to our network administration staff and at the same time help ease our own network manager paranoia, by implementing access-control using the "A" in the middle -- Authorization. While the commands for authorization are documented -- what is not well covered in the Cisco documentation is that its use and value is unknown. One of the pains of authorization is to be used effectively; it is dependent on using options available in the AAA suite that is not normally dealt with in detail. The hope here is to expand your understanding on how and why to use authorization, specifically command and EXEC shell authorization to control your management and configuration environment more effectively.

Why Do I Need Command Authorization?

Out of the box, IOS based Network Access Devices (NAD) i.e., routers, switches and the like, support a two-tier -- single user authentication model. Once AAA services are enabled the security posture of the IOS device changes to a multi-user model. Multi-user operating systems have two types of users: *Administrative and Executive Users*. An administrative user (AU) has complete access and control over the system's configuration. AU's can add, delete or change the access rights of any user, file or directory along with making alterations to system configuration and service operations. Essentially, if it can be changed on the system an AU can do it. An executive user (EU), despite the name has more restricted access to the system. EU's can execute commands, but are restricted in terms of file and directory creation and modification to only those they have been authorized for.

An AU comes in two forms; first there is the AU user account. On Windows systems this is the *Administrator* account, on UNIX systems it is the *root* account. While these user accounts exist and function just like any executive user account. The common wisdom is that it's not a good idea to "login" as the AU. Since the lack of restrictions on the actions associated with these accounts can make a simple mistake result in un-repairable damage to the system. The preferred and far safer approach is to grant an executive user certain administrative powers through the use of special administrative group memberships, the methodology followed in the Windows environment. Alternatively, on UNIX systems a user can either "become" the AU through membership to the *wheel* group using the *su* command or have the right to use certain administrative command using *sudo*.

So now why do you need NAD command authorization? Think of what your routers configuration looks like out of the box -- the IOS in its default state provides no restrictive command usage controls whatsoever. Single sign on login authentication is available to control access to the built-in serial TTY lines (which do not require a password to be set) and the Virtual TTY (VTY) ports (which do.) As a small aside, a TTY in case you're wondering is an UNIX term holdover, for *Teletype* or what is better known as "serial" terminal. A VTY or in UNIX parlance a PTTY, *pseudo-teletype* is used by Telnet or SSH to provide a "virtual terminal" on the NAD. Operationally, both line types behave identically, as a full duplex serial line. Users access the NAD's command line interface (CLI) over the TTY line interfaces, through an interactive shell known as the *EXEC* shell. Since all "access" lines provide access to an exec shell, the importance of controlling access to the NAD's CLI and by extension the devices configuration, cannot be underestimated.

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

The reason for implementing AAA, over the single-user access model, is to control and monitor access to NAD's, it is not because the model itself is weak from a security standpoint (at least if it is implemented properly.) Rather it is because it's a poor model for a multi-user environment that cannot be scaled without jeopardizing the security integrity of the NAD. An argument could be made that a network with a single router and one network administrator does not need more than the default *single-user* access model supported out of the box. I would counter that argument, that more times than not, administrators fail to configure even the minimal security options in these types of environments -- in part to fear (not knowing enough about the IOS or how to configure it) or laziness. The 'no one will access this but me syndrome,' is no excuse for a production component. As for a network of any reasonable size, with multiple administrators and different requirements for access, the basic IOS security is just not adequate. AAA authentication and accounting addresses the need for multi-user authentication and user accountability, but unless command authorization is enabled there is no way to control access to the configuration.

Think about this: Which sounds better to your management team? Knowing that someone made a mistake by changing something they should never have or preventing them from even being able to make the change at all. I believe most managers would prefer the latter, and to have that ability command authorization must be configured and enabled. Lets just end this discussion with this thought (this is how you sell implementing authorization to management, by the way): authorization affords the network administrator the ability to customize environments to meet the access requirements of specific user groups. So that if only those who need access have access, two avoidable problems that Large Scale Network Environments (LSNE) face can be avoided. First, mistakes made by those who should not be there in the first place. Second, actions taken by cowboy administrators who act before they think (but I am sure you don't have the problem, do you?)

Creating A Multi-Level Authorization Model

Implementing authorization provides the capability to create a highly customizable command and control environment. Improved functionality however, comes with a price. Mainly, there is some work involved with setting up command authorization, but the effort will (hopefully) yield a securer, more controlled and predictable network environment (and you a better night sleep.)

The work starts before implementing any authorization scheme (or any security policy for that matter, period.) By developing a clear understanding of security requirements and goals. Once developed, they need to syndicate and be approved by management, and most importantly the folks who will have to live with it -- the support and operations teams. Once defined (at this point you still have not touched the router) the requirements and goals that have been developed need to be translated into an access-model, with IOS commands assigned according to the needs of the user access groups within that model.

Here is a 1000-foot level example of a five-tier access-model: (L1) Basic Admin User: Needs to be able to access the device and perform basic troubleshooting tasks i.e., ping, traceroute, look at routing table, interface status and status collection counters. No configuration access. (L2)Admin User: Needs all of the basic admin commands, plus the ability to see key parts of the configuration, interface addressing and filtering, routing policy configuration. Can originate outbound sessions from the router. (L3) Senior Admin User: Has the ability to make changes only to parts of the routers

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

configuration, usually those that the Admin user can view. (L4) Security Administrator: Can create and delete users on the NAD and audit the security configuration. (L5) Domain Administrator: Has full access to router configuration - can make any configuration change.

Your access-model should reflect the different tasks performed by different personal -- there is no "right way" with this. Access-models will vary from enterprise to enterprise because they are largely based on the support hierarchy of the support environment. In some environments the security team may be responsible for all user accounting, the remote access administrators may need to configure model pools, client VPN and VPDN, and all of the site-to-site links. There is definitely some homework on the part of the designer. Communication with and between the different user populations is a must. It cannot be emphasized enough -- implementing authorization can quickly alienate user groups if the model is poorly thought out and if prevents them from doing their jobs. Effective communication and syndication will go a long way to avoid users pounding on your door with letter openers in hand. With your access-model complete, configuration and security-server configuration testing will need to ensue. After you have locked yourself out of the test environment a few times you will get the hang of it and be ready for a user pilot. If successful with a lot of user feedback, you will be able to implement authorization, improve on your network SLA's and get a big raise (wouldn't be great if it worked that way.) As you start your development effort, keep this in mind: The overall benefits of implementing authorization far outweigh the potential pit falls, and the best way to avoid them is proper planning, testing, and review.

Configuring Authorization (IOS) Phase 1

At the 1000 foot level, implementing authorization is like any other IOS feature -- there's a set of commands and an order to follow. Authorization has one little hitch to be effective: It must be used in conjunction with a security server. Luckily this article is part of a series on implementing TACACS+ so this should not be a problem. While authorization and authentication are discrete functions they are interdependent when authorization is enabled. The construction of a multi-level authorization scheme is implemented in three phases. Phase one is the configuration of the NAD's authorization policy using the IOS. Phase two is the assignment of IOS commands and users into the command access model groups, this involves configuration on both the IOS and the TACACS+ security server. Phase three is the creation of the authorization policy on the TACACS+ security server.

Authorization is available for the following IOS functions:

Auth-Proxy: An in-line authentication method for controlling access in or out of the network using dynamic access control lists. Commands: Requires authorization for commands associated with a specific privilege level (0-15), defined as part of the configuration a separate statement is needed for each level requiring authorization. Config-commands: Enables or disables authorization for configuration commands. This option uses a specific configuration command *<aaa authorization config-commands>*, preceding the statement with a "no" disables the option. EXEC (The IOS Shell) : Authorization is required on initial login for starting a command shell Network: Authorization is required for starting any network service i.e, PPP, SLIP, ARA, PPTP Reverse-access: Authorization is required for starting a reverse telnet session.

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

With the exception of *config-command* authorization all of the authorization method functions are configured with the configuration command `<aaa authorization {function} {default | method list} {group TACACS+ | local | if-authenticated | none}>`. Here is a basic configuration syntax example (including the Authentication and Accounting statements):

```
!  
aaa authentication login default group tacacs+ local  
aaa authorization exec default group tacacs+ local  
aaa authorization commands 1 default group tacacs+ local  
aaa authorization commands 15 default group tacacs+ local  
aaa accounting commands 1 default start-stop group tacacs+  
aaa accounting commands 15 default start-stop group tacacs+
```

Just looking at the syntax, configuring authorization seems straightforward enough, particularly if you already are using AAA authentication and accounting. You may notice that there are a few things in the configuration example that are not exactly clear. For instance what does the keyword `default` mean? Those numbers after the `commands` keyword and the `local` keyword after the `tacacs+` declaration, what's up with that. Those innocuous keywords, their purpose and relation to implementing authorization are what we will be focusing on now.

Authentication Redux And Understanding Method Lists

For starters, the secret big of AAA is that all AAA services are tied to the username. This is an important point to keep in mind for two reasons. First, while authorization and accounting are dependent on authentication in order to function, authentication, authorization and accounting can be decoupled and originate from independent resources. This allows for a great deal of flexibility in creating a security model that works in any environment. For example, if you're a Windows 2000 shop, user authentication could be handled by Win2000 RADIUS, user privilege levels (what is a privilege level you ask -- we will get to it in a moment) could be set using the NAD's local user database and TACACS+ could be used for authorization and accounting. The only requirement is that the username is configured identically on each database. The second thing is that using the username as the only attribute can be exploited into a security breach, so control over security servers running TACACS+ or RADIUS and NAD's local user database's needs to be regulated, monitored and auditable.

So how does the model work with all of these different AAA service? All accounting and authorization transactions contain the username attribute. When a user logs into a NAD, the first function performed is the memory allocation to store the users profile data. Initially this allocation is made with NULL data entries, which are later filled with valid data once the authentication request is successfully completed. Once the memory allocation is complete the LOGIN call is processed. The first step of the LOGIN call, is to determine the *method* assigned to the line interface to validate the user. This is accomplished by consulting the method list configured for the line. A method list, also referred to as a "named method list," is the *id tag* used by an AAA service definition to describe the available methods of user validation, hence the name -- method list. The validation methods, in the method list, are listed in the preferred sequence, when the first listed method fails the next is tried, and so on. If no valid or

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

working method is found the users are denied access. The IOS supports several user validation methods:

TACACS: The IOS preferred method for all AAA services. **RADIUS:** Supports Authentication and Authorization but no Accounting support. Local user accounts and privilege level profiles are stored locally. Great for creating "backdoor" access accounts, if your TACACS/RADIUS servers have failed. Remember to use a strong ASCII password, (preferably generated by a password generator) and enable the `<service password-encryption>` option to encrypt the password in the configuration. **Kerberos:** Like RADIUS support for Authentication and Authorization, but no Accounting. Plus you have to maintain an open-source MIT Kerberos v5 (or compliant) environment, no Windows K5 support.

Each physical or virtual TTY is capable of having its own custom AAA policy definition using a named method list. Additionally, a default method list, named "default" exists and is applied to all access interfaces (the "default" list has been use in earlier examples in this series) since applying a custom policy on each TTY is not only a pain -- it leave the potential for mistakes to be made. So when we examine the configuration statement:

```
aaa authentication login default group tacacs+ local
```

The method list default, which defines that tacacs+ is the preferred method and the local user database is secondary, is used for validating user authentication for NAD login requests. When creating your AAA policy it is a good idea to first define the general policy that can be applied to all of the TTY interfaces, then use a named method list to "override" the base policy. Here is an example where the default policy (using TACACS+) is applied -- with custom policy for the local serial TTY's (using the local database only):

```
CC2217919-C#config terminal
CC2217919-C(config)# aaa authentication login default group tacacs+ local
CC2217919-C(config)# aaa authentication login su local
CC2217919-C(config)#aaa authorization commands 15 default group tacacs
local
CC2217919-C(config)#aaa authorization exec default group tacacs local
CC2217919-C(config)#aaa authorization commands 15 su if-authenticated
CC2217919-C(config)# aaa authorization exec su if-authenticated
CC2217919-C(config)#line aux 0
CC2217919-C(config-line)#login authentication su
CC2217919-C(config-line)#authorization commands 15 su
CC2217919-C(config-line)#authorization exec su
CC2217919-C(config-line)#exit
CC2217919-C(config)#line con 0
CC2217919-C(config-line)#login authentication su
CC2217919-C(config-line)#authorization commands 15 su
CC2217919-C(config-line)#authorization exec su
CC2217919-C(config-line)#exit
CC2217919-C(config)#username su privilege 15 password superuser
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

```
CC2217919-C(config)# ^ Z
CC2217919-C#
```

This example configures the NAS to use the default list (TACACS+, LOCAL) for any authentication and/or authorization request on the VTY interfaces. The named list "su" (LOCAL) is applied to the physical console and aux ports, requiring anyone who logs in on those interfaces to have a local account.

Before we move on, one last point about method ordering, when multiple TACACS+ to RADIUS servers are configured they are "tried" in the order they have been configured in. Both servers need to fail before the next method (usually *local* for last resort.) If the primary server fails and the secondary becomes active, the NAD will continue to use the server until it fails and then try the primary again. This is not a major issue if you keep user authentication data in sync. It can become an issue as far as collecting accounting data is concerned. A work around for this issue is available in later versions of IOS 12.x and 12.2, which is a broadcast option, and sends accounting data to all of the servers defined on the NAD. Here is a configuration syntax example:

```
CC2217919-C# aaa accounting commands 15 default start-stop broadcast group
tacacs+
```

Once the user has successfully logged in to the NAD, the username and privilege level (there it is again) are written into the memory buffer allocated when the user initialized the login process and the user is assigned a random client ID, for use during the duration of the login session. As the user issues commands in the CLI shell, using these cached credentials the NAD in turn processes these requests (using the credentials) according to the authorization and accounting definitions.

Understanding Privilege Levels: Step 2

The IOS, in single or multi-user mode runs with authorization disabled by default and by design. In its default state the IOS effectively emulates the standard UNIX "user / super user" paradigm. This two-tier security model is accomplished using two (technically, three of 16 possible) *Privilege Levels*. Privilege levels are grouping abstracts used by the IOS to provide a layered security model. Their function is similar to the concept of group's, used in Windows and UNIX as a means of controlling access to assign access rights to files and commands to a group of users. The IOS command line interface (CLI) is commonly described in modal terms (i.e., user-exec mode, enable-exec mode, and configuration-exec mode.) This "model" refers to the IOS's default security configuration. The *idea* of changing modes is actually just that, an idea. These "modes" are created using privilege level. Users experience the process of traversing privilege levels, as changing modes, when in actuality they remain in the same EXEC shell they logged into, unlike the UNIX paradigm where the user actually executes into a new command interpreter shell with new environmental parameters.

Privilege level assignments can be assigned to users, TTY lines, and commands. In the default IOS security environment, users start in the user-exec shell on privilege level 1, which provides very basic command access. Privilege level 1 commands include outbound telnet (and ssh) *<enable>* and *<disable>* and some limited status visibility. IOS privilege levels operate on the security premise of incremental inherited hierarchy. Users start at a base that provides them basic system access and the

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

tools to increase their access if they have the proper credentials (i.e., a password.) Each privilege level supports password protection. As a user moves through the privilege levels (users can also jump from one to another, say level 1 to level 10) they gain access to new commands and retain the access to the commands they were granted in the lower privilege levels. Movement between privilege levels is facilitated using the all too familiar `<enable {1-15}>` command and the *not so familiar* `<disable {0-14}>` command. Most of us are familiar with the `<enable>` command's default behavior, which is to authenticate against the privilege level 15 password (set with the command `<enable password>`) or encrypted secret. The less familiar command `<disable>` allows users to return to lower privilege levels. In an environment without authorization, there is little need to "go back", but in authorized environments commands may be configured so they can only be executed at a certain privilege level.

Command and login (exec) authorization is dependent on the use of privilege level assignments. Privilege level assignments for users and/or NAD *line* interfaces determine what privilege level a user-exec shell "starts" on. Privilege level assignments for commands determine the privilege level a command is executable on, provided that they have the authorization credentials.

User Privilege Level Assignments

A common question asked about authorization is, "Can authorization work using only the NAD's local user database?" The answer is a qualified yes, because when authorization is configured to use only the local database, only the privilege level is considered for validation. This effectively nullifies the value of authorization, since there is no mechanism to assign access rights to individual users, so the system behaves as if authorization was disabled. For authorization to be truly effective it needs to be used in conjunction with a security-server such as TACACS+. By assigning users and commands to distinct privilege groups, versus leaving commands in their default privilege levels, the continuity of the access model can be maintained even (at least in a broad scope) if the security server is inaccessible.

When implementing authorization it's a good idea when using multiple privilege level based command groups to create a local "backdoor" account for each group. This approach addresses two issues, first when creating custom command groupings, if a command is assigned to a privilege level, higher than that of the user the command will not be accessible. Discrete local accounts for each privilege group assures that command access for each group is maintained, a single back door user will either have not enough to much access to the device. Which leads us to the second issue; the security server when running should disable local group accounts. This can easily be accomplished by creating parallel user accounts on the security-server with no access privileges. The reason you do not want users using the group local accounts is because you will be unable to track users activity, leaving you blind if someone does something they should not have. Authorization, limits access to users who should have it, it does not protect against mistakes. Local group accounts insure that in the event that the security-server is unavailable the users in the group can still access the NAD, within the context of the security policy. That is however all they should be for.

Local user privilege level assignments are set using the `<privilege>` command argument of `<username {name} privilege {priv 1-15} password {pass}>` configuration command. Here is a syntax example creating a local user definition, privilege level and password:

```
TestRTR(config)#username fudd privilege 8 password elmer
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

Without a privilege level definition the command assigns the user to the default privilege level, 1. Alternatively, `<username>` command can also assign only a privilege level to a user in the NAD's local database:

```
TestRTR(config)#username fudd privilege 8
```

For privilege level only assignments, no password needs to be specified (assuming that TACACS or RADIUS is being used for authentication). However, when setting the privilege locally, since the local database has a user with no password, any authentication attempt made using that username, against the local user database will fail. To set only the users privilege level, the method list for authentication must define the security-server as the primary method, with authorization using the local user database as the primary and TACACS+ as secondary. Here is the configuration syntax:

```
aaa authentication login default group tacacs+ local
aaa authorization exec default local group tacacs+
```

Technically, while authorization is dependent on privilege levels, a users privilege level assignment is actually part of their authentication profile. The privilege level serves the function, of defining the user's access to the commands, actual authorization (the right to execute) while dependent on access, is defined and managed using the security server. The dependency on the security server's function as an external valuator is why the use of authorization with only the local user database has questionable value. Since the only method of validation is the privilege level. The reason it's value is questionable, is most evident when using exec authorization. Because success is dependent only on the users privilege level, mainly that they have one. Lets look at two examples (You should try these examples in your lab, to get a real feel for it). The example uses only the local user database, here is configuration syntax:

```
aaa authentication login default group tacacs+ local
aaa authorization exec default local
!
username emmet privilege 10
username zed password secret
```

In this example, user Emmet's user-exec shell will start at privilege level 10, the user Zed will start at the default level 1. Both will be successfully authorized for a login shell, Emmet will start at level 10 and Zed at the default level 1. The second example uses the local database as the primary and TACACS+ as the secondary:

```
aaa authentication login default group tacacs+ local
aaa authorization exec default local group tacacs+
!
username emmet privilege 10
username zed password secret
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

The user's that are configured locally be authorized. However, the TACACS+ users will either require a local privilege level definition, like the user Emmet. Or they will need there privilege level set as part of their TACACS+ user profile.

User privilege levels are set on the security server as an exec shell service attribute. On the tac_plus TACACS+ implementation the service/exec attribute value pair is used to set the users privilege level. Here is a basic syntax example of a user profile using skey for authentication with a privilege level assignment of 15:

```
user = martin {
    login = skey
    service = exec {
        priv-lvl = 15
    }
}
```

RADIUS, an alternative authentication service to TACACS+ supports privilege level assignments as part of its user authentication profile. In a Livingston/MERIT RADIUS compatible environment (Interlink, Cistron, FreeRADIUS) the configuration syntax looks like this:

```
martin Authentication-Type = Unix-Pw
      Service-Type = Shell-User
      Cisco-avpair = shell:priv-lvl=15
```

RADIUS mind you does not support Accounting or Enable authentication support. In order to support these services a parallel TACACS+ is required. Regardless of the method used to set a users privilege level, users can verify their current privilege level, using the exec shell command `<show privilege>`. Which prints out the users current privilege level. Here is an output example of the `<show privilege>` command:

```
CC2217919-C#sh privilege
Current privilege level is 9
CC2217919-C#
```

Line privilege Level Assignments

Line privilege level assignments provide a more blanket approach, since the level is applied to any user who accesses the port without any definition. The line's privilege level is set as part of its custom AAA policy using the configuration line command `< privilege level {0-15}>`. Line based privilege level settings come in handy when configuring custom AAA schemes for local serial or rotary based TTYs for device access in the event of server failures or in situations where the device has no network access except out of band.

Command Privilege Level Assignments

IOS command reassignments to new or previous privilege level are accomplished with the global configuration command `<privilege {root command}{priv lvl (0-15) | reset} {specific command}>`.

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

Reassigning a command's privilege level can at times be a little tricky. The most common problem occurs when not enough command syntax is provided. All IOS commands have two parts, the root command and the command arguments. Depending on the IOS and feature set there are a number of root IOS commands to base privilege assignments on. Some of the more common root commands are:

Exec: (Shell Commands) i.e., show, clear, copy, enable
Configure: (Shell Command) i.e., configure-terminal, configure-network
Line: (Config Command) i.e., Serial line and VTY configuration
Crypto-map: (Config Command)
IPSec Configuration
Router: (Config Command)
Routing Protocol Configuration i.e., OSPF, BGP
Interface: (Config Command) interface parameters

Exec commands are pretty straightforward, mainly because the syntax is simple. Here is the configuration syntax from the ping example above:

```
CC2217919-C(config)#privilege exec level 8 ping
```

Where privilege level reassignment becomes tricky is when the commands have to do with configuration (Notice that most of the root commands in the list above have to do with configuration). Since many of the configuration commands have optional arguments, a command may require more than one privilege level definition for it operate correctly. For example, the <access-list> command could require up to six privilege level definitions to support all of the argument variations (the list ID is a place holder only, any ID can be used):

```
privilege configure level 8 access-list 1
privilege configure level 8 access-list 100 permit tcp any any
privilege configure level 8 access-list 100 permit tcp any any eq log
privilege configure level 8 access-list 100 permit tcp any any lt log
privilege configure level 8 access-list 100 permit tcp any any gt log
privilege configure level 8 access-list 100 permit tcp any any established
log
```

The first two statements define standard and extended IP access-lists (the list # are only place holders) the remaining statements define optional arguments. When assigning command privilege levels for authorization, a trick for getting the syntax correct is to enable authorization debugging <debug aaa authorization> and use the debug output to get all of the arguments correct. Here is an example of the debug output for the <username> command:

```
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd=username
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=david
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=privilege
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=15
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=password
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=apassword
2w3d: AAA/AUTHOR/TAC+: (1359073811): send AV cmd-arg=<cr>
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

Looking at the debug output, reports the parts of the syntax are considered the root (cmd) command and which parts are arguments (cmd-arg). This method helps, but in some cases a *command arguments*, may need to be defined by itself, in order to get the command to work properly. So expect some playing around with command syntax. Command re-definition is one of the few areas of IOS configuration that is more of an art than a science, each hardware platform and IOS version behaves a little differently. So don't just assume a set of privilege redefinitions that works on one platform will work on another. Testing is required -- there is no way around it.

The one command users have the most trouble with is the exec command, *<show running configuration>*. Often it is one of the first command administrators reassigned, with the thought being, "I want the support team to see how the device is configured, but not necessarily be able to change the configuration." It makes a lot of sense, but to there dismay it does not work. Executing the reassigned *<show running-config>* command yields only this output:

```
CC2217919-C#sh run
Building configuration...
```

```
Current configuration : 153 bytes
!
! Last configuration change at 23:02:44 EDT Fri Feb 22 2003 by martin
! NVRAM config last updated at 21:38:08 EDT Feb 24 2003 by martin
!
!
!
end
```

This output result is not a bug; it's actually a security feature. The fickle nature of privilege levels and IOS commands is discussed in Cisco a technical brief (not in any of the main line documentation, mind you) explaining why, the *<show running-configuration>* command would return a blank configuration if assigned to a lower privilege level, but *<show starting-configuration>* would function normally, if reassigned (except for displaying SNMP configuration data, which can be used to configure the router if read-write access has been enabled.)

The tech brief explains that commands such as *<write terminal>*, *<copy running-configuration starting-configuration>* and *<show running-configuration>* only function in respect to the commands that the users privilege level and authorization policy permits. Any command that can alter the security posture of the device will only return/save information on configuration data accessible to commands belonging to a privilege level of equal or lesser value. So for example, if a users privilege level allows them access to make configuration changes to access-lists, routing protocol configuration and interface IP addressing, then the *<show running-configuration>* command will return only those parts of the configuration:

```
CC2217919-C#sh run
Building configuration...
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

```
Current configuration : 1779 bytes
!
! Last configuration change at 23:02:44 EDT Fri Feb 22 2003 by martin
! NVRAM config last updated at 21:38:08 EDT Feb 24 2003 by martin
!
interface Ethernet0
 ip address dhcp client-id Ethernet0
 access-class 144 in
!
interface FastEthernet0
 ip address 172.30.71.1 255.255.255.0
 access-class 100 out
!
!
router ospf 2
 network 172.30.10.0 0.0.0.255 area 0.0.0.0
 network 172.30.71.0 0.0.0.255 area 0.0.0.0
!
access-list 100 permit tcp any any log
access-list 100 permit udp any any log
access-list 100 permit icmp any any

access-list 144 permit tcp any any eq 22
access-list 144 permit tcp any any established
access-list 144 permit udp any any
access-list 144 permit icmp any any
!
end
```

At this point we leave IOS configuration and move on to the TACACS+ security server configuration, where the authorization policy is defined.

Configuring Authorization (TACACS+): Step 3

External validation is the primary roll of the security server in authorization. With command level and EXEC shell authorization enabled, once the user has authenticated, the opening of shell and each command executed in it require the NAD to query the security server for validation before the command can be executed.

User and command privilege level groupings create the access structure. The security-server allows you customize to a granular level the commands the users assigned to those groups can and cannot execute, these definitions are part of the users profile on the security server. The TACACS+ servers default authorization stance is to deny any authorization request. When implementing authorization makes sure that you have created at least one local account with privilege level 15 accesses and that the last method in the authorization definition is local. Alternatively, the command augment *<if-*

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

authenticated> can be used, both approaches achieve the same result but with *<if-authenticated>* no local account is required.

A users authorization stance can be implicitly or explicitly defined. The implicit approach uses the root A /V pair default service defined as part of a user or group profile.

```
user = emmet {
  default service = permit
acl = Ne_acl
login = skey
service = exec {
priv-lvl = 4
}
```

The default service definition must be the first clause in the user/group profile definition. The explicit approach requires that each command and its arguments be defined as clauses in their user profile. This can be accomplished in one of two ways, directly as part of the user profile or indirectly as part of a group definition, to which the user is a member of. Either way the command clauses are written using root A/V pair cmd and the parameter pairs permit and deny. TACACS+ command definition rules are not as stringent as the IOS privilege level definitions only the root commands are required. Support for defining command arguments is available using *permit* or *deny* rules. To permit or deny all arguments use the ". *" regx meta characters. Here's a tac_plus authorization syntax example:

```
user = emmet {
acl = Ne_acl
login = skey
service = exec {
priv-lvl = 4
}

cmd = traceroute {
  permit .*
}

cmd = ping {
  permit .*
}

cmd = show {
  permit interfaces
  permit ip
  permit arp
  deny .*
}
}
```

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

This syntax example has the user Emmet, assigned to privilege level 4, and is authorized to execute; <traceroute>, <ping> and the <show> commands with *the interface, ip and arp* arguments. Here is the same tac_plus definition example configured with a group profile definition.

```
user = emmet {
acl = Ne_acl
login = skey
member = ladmin
}

group = ladmin {
service = exec {
priv-lvl = 4
}

cmd = traceroute {
permit .*
}

cmd = ping {
permit .*
}

cmd = show {
permit interfaces
permit ip
permit arp
deny .*
}
}
```

Each of these approaches has advantages and disadvantages. The *default service* approach provides no granularity, but if used in conjunction with static (no enable) privilege level rule set or for an all-access user it can save you some typing. The user vs. group approach works along the same lines. User level definitions work well if you have a small group of administrators with diverse command access requirements or if you wish to have only specific commands run by an individual who has been assigned to privilege level defined command set. If there are no users specific command requirements, then the group approach is far easier administratively and scales well. The approach you chose, in the end should be the one best suited for your environments need and what can be administered effectively. In the end, if the authorization system is to burdensome on users or to hard to effectively and securely administrate then it will be circumvented or not used at all. KISS may mean *Keep It Simple Stupid*, but it also means *Keep It Simply Secure*.

Configuring Advanced TACACS+ - Implementing Authorization

Michael Martin

Conclusion

Well that's about it for authorization (and some AAA points as well) the hope here was to provide you an introduction to how to go about designing and implementing a multi-level command access model using AAA authorization. This article technically concludes the advanced TACACS+ series. Next month we will look at creating tools to manage TACACS+ environments that handle user management, account reporting and service checking. I hope that you have made good use of your time here today and learned something useful that can improve the reliability and security of your own networking environment. So be sure to tune in again, same bat time, same bat channel.