Grsecurity/The RBAC System - Wikibooks, open books for an open world

Archived: 2026-04-05 13:10:59 UTC

A role-based access control (RBAC) system is an approach to restricting system access to authorized users. You need an RBAC system if you want to restrict access to files, capabilities, resources, or sockets to *all* users, including root. This is similar to a Mandatory Access Control (MAC) model. The other features of grsecurity are only effective at fending off attackers trying to gain root, so the RBAC system is used to fill in this gap. Least privilege can be granted to processes, which, in turn, forces attackers to reevaluate their methods of attack, since gaining access to the root account no longer means that they have full access to the system. Access can be explicitly granted to processes that need it, in such a way that root acts as any other user. Though grsecurity and its RBAC system are in no means perfect security, they greatly increase the difficulty of successfully compromising the system.

In grsecurity, the RBAC system is managed through a **policy** file which is essentially a system-wide set of rules. When the RBAC system is activated with `gradm`, the policy file is parsed and checked for security holes, such as granting the default role access to certain sensitive devices and files like the policy file itself. If a security hole is found, `gradm` will refuse to enable the RBAC system, and will give the user a list of things that need to be fixed. The policy file is protected when the RBAC system is active, and only the admin role may access it during that time. To make it easier to create a secure policy, `gradm` has the ability to learn how the system functions, and build a least-privilege policy based on the collected data (see Learning Mode).

So as not to contribute further to the false sense of security many have regarding access control systems (whether they be grsecurity's RBAC, SELinux, RSBAC, SMACK, TOMOYO, AppArmor, etc.) it's important first to describe the limitations of any access control system.

There is a fundamental architectural limitation to the kind of guarantees an access control system can provide when the policy decision-making code resides alongside the Operating System's kernel. A compromise of the Operating System can easily result in compromise of the access control system, and it is common practice for exploits which compromise the kernel to disable any active security systems.

Grsecurity is in no way immune to this fundamental limitation, though it does contain several features to help prevent exploitation of the kernel in the first place and furthermore to make the kernel a more hostile environment to an attacker if they do manage to exploit certain types of bugs. The project will continue to make adding similar protections one of its main goals.

Specifically, the following features are involved in kernel self-protection and increasing the difficulty of kernel exploitation:

```
GRKERNSEC_MODHARDEN
GRKERNSEC_HIDESYM
```

```
GRKERNSEC_RANDSTRUCT
GRKERNSEC_KSTACKOVERFLOW
PAX_MEMORY_SANITIZE
PAX_MEMORY_UDEREF
PAX_MEMORY_STACKLEAK
PAX_MEMORY_STRUCTLEAK
PAX_CONSTIFY_PLUGIN
PAX_SIZE_OVERFLOW
PAX_KERNEXEC
PAX_RANDKSTACK
PAX_USERCOPY
PAX_REFCOUNT
PAX_RAP
```

There also exist some features of grsecurity which are always active (and thus have no configure-time option) which aid in the above goals. These include the read-only and non-executable vsyscall page (and its shadow page) on amd64, hardening of the BPF interpreter buffers, and many more.

Though these features have been successful at preventing previous vulnerabilities from being exploited (and surely will continue to do so) there have still been many vulnerabilities it did nothing to prevent exploitation of, and there are entire classes of vulnerabilities (such as missing capability checks, some race conditions, etc.) that it can likely never do anything to prevent exploitation of.

It's partially due to this fundamental limitation of any access control system that grsecurity's RBAC system was designed as it was: to be as automated as possible, to provide a sufficient level of access control, to have easily editable human-readable configurations, and to enforce secure base policies to eliminate some administrator error.

Neither grsecurity's RBAC system nor any other access control system should be used to separate classified information from unclassified information on the same machine. There is no virtual replacement for a physical air-gap.

The policy is made up of roles, subjects and objects. **Role** is an abstraction that encompasses traditional users and groups that exist in Linux distributions and special roles, that are specific to grsecurity. **Subjects** are processes or directories, and **objects** are files, capabilities, resources, PaX flags, and IP ACLs. The location of the main policy file is */etc/grsec/policy*.

To see a small example policy, look at the default */etc/grsec/policy* file that is installed with `gradm`. In a nutshell, RBAC policies have the following structure:

```
role <role1> <rolemode>
<role attributes>
subject / <subject mode>
<subject attributes>
    / <object mode>
    <extra objects>
    <capability rules>
```

```
    <IP ACLs>
    <resource restrictions>
 subject <extra subject> <subject mode>
 <subject attributes>
    / <object mode>
    <extra objects>
   ...
 role <role2> <rolemode>
...
```

Using the default policy as an example:

```
 role admin sA
 subject / rvka
        / rwcdmlxi

 role default G
 role_transitions admin
 subject /
        /               r
        /opt            rx
        /home           rwxcd
        /mnt            rw
        /dev
        /dev/grsec      h
...
```

There exist some features of the RBAC system to aid in simplification and generalization of policies. One of these is the recently added "replace" rule. The replace rule allows you to assign a string to a variable, and then use that variable within any subject or object pathname to have it replaced with the string. The syntax of replace rules are:

```
 replace <variable name> <replace string>
```

So for example:

```
 replace CVSROOT /home/cvs
```

The defined variable can then be used as follows:

```
 replace CVSROOT /home/cvs
 replace PUBHTML public_html
```

```
subject $(CVSROOT)/bin/test o
      $(CVSROOT)/grsecurity r
      /home/spender/$(PUBHTML) r
     ...
```

The variables defined with replace rules can be reassigned at any location in the policy. All rules in the policy until another redefinition of the variable will use that new assigned value for the variable. For example:

```
replace CVSROOT /home/cvs
$(CVSROOT)/grsecurity r
replace CVSROOT /var/cvs
$(CVSROOT)/test r
```

would cause the following object rules to be created:

```
/home/cvs/grsecurity r
/var/cvs/test r
```

There are some special cases you should know about when writing policies for the RBAC system.

There exist some unique accesses to filesystem objects that require specific object modes. For instance, a process that connects to a unix domain socket (*/dev/log* for example) will need "rw" set as the object mode for that socket.

Adding the setgid or setuid flag to a path requires the "m" object mode.

Creating a hard-link requires at minimum a "cl" object mode. The remaining object flags must match on the target and the source. So for instance, if a process is creating a hard-link from /bin/bash to /bin/bash2, example rules would be:

```
/bin/bash rx
/bin/bash2 rxcl
```

Creating a symlink requires the "wc" object mode.

One very useful feature of the RBAC system is the support of wildcards in objects. The "*" character matches zero or more characters, "?" matches exactly one character, and "[]" can be used to specify an inclusive or exclusive list or range of characters to match. Depending on how these wildcard characters are used, they have different effects. Here are four examples of the use of wildcards:

```
/dev/tty*      rw
/home/*/bin    rwx
/dev/tty[0-9]  rw
/dev/tty?      rw
```

The first example would match */dev/ttya*, */dev/tty0*, */dev/ttyS0*, etc. Since a '*' at the end of a path can match the '/' character as well, if a '/dev/tty/somefile' path existed, the first example would match it also.

The second example would match */home/user1/bin*, */home/user2/bin*, etc. Note that this rule would not match the path */home/user1/test/bin* as the wildcard characters will not match '/' unless it appears at the end of a path. To use the particular wildcarded object for this example, a */home* object must exist as an "anchor" for the wildcarded object. If you forget to add one, **gradm** will remind you.

The third example would match */dev/tty0*, */dev/tty1*,..., */dev/tty9* and nothing else.

The fourth example would match */dev/ttya* and */dev/tty0* just like the first example, but would not match */dev/ttyS0* since only one character can match the '?' wildcard.

Wildcards are evaluated at run-time, providing a powerful way of specifying and simplifying policy. Since wildcard matching is based off pathnames and not inode/device pairs though, they aren't intended to be used for objects which are known to be hardlinked at policy enable time.

Roles exist essentially as a container for a set of subjects, put to use in specific scenarios. There exist user roles, group roles, a default role, and special roles. See Flow of Matches to see how a role gets matched with a particular process.

In a simplified form, user roles are roles that are automatically applied when a process either is executed by a user of a particular UID or the process changes to that particular UID. In the RBAC system, the name of a user role must match up with the name of an actual user on the system.

A user role looks like:

```
role user1 u
```

As with user roles, group roles pertain to a particular GID. The name of the group role must match up with the name of an actual group on the system. Note that this is tied only to the GID of a process, not to any supplemental groups a process may have. Group roles are applied for a given process only if a user role does not match the process' UID.

A group role looks like:

```
role group1 g
```

If neither a user or group role match a given process, then it is assigned the default role. The default role should ideally be a role with nearly no access to the system. It is configured in such a way if full system learning is used.

A default role looks like:

```
role default
```

Special roles are to be used for granting extra privilege to normal user accounts. Some example uses of special roles are to provide an "admin" role that can restart services and edit system configuration files. Special roles can also be provided for regular users to keep their accounts more secure. If they have their own *public_html* directory, the user role for the user could keep this directory read-only, while a special role to which the user is allowed to transition could allow modification of the files in the directory.

Special roles come in two flavors, ones that require authentication, and ones that do not. On the side of special roles that require authentication, the RBAC system supports a flag that allows PAM authentication to be used for the special role. See Role Modes for a list of all these flags.

Special roles by themselves won't do anything unless there exist non-special (user, group, or default) roles that can transition to them. This transitioning is defined by the **role_transitions** rule, described in the Role Attributes page.

To authenticate to a special role, use `gradm -a <rolename>`. To authenticate with PAM to a special role, use `gradm -p <rolename>`. To transition to a special role that requires no authentication, use `gradm -n <rolename>`.

Special roles look like:

```
role specialauth s

role specialnoauth sN

role specialpamauth sP
```

With domains you can combine users that don't share a common group ID as well as groups so that they share a single policy. Domains work just like roles, with the only exception being that the line starting with "role" is replaced with one of the following:

```
domain somedomainname u user1 user2 user3 user4... usern
domain somedomainname g group1 group2 group3 group4... groupn
```

Example:

```
domain somedomain u daemon bin www-data
subject /
    /    h
```

As it is with user and group roles, all domain members must exist, and if they're not, an error is raised.

Subjects can describe directories, binaries or scripts. Regular expressions are currently not permitted for subjects. The ability to place a subject on a script is unique, as it permits one to grant privilege to a specific script instead of generally to the associated script's interpreter. For this to function properly, make sure the script's interpreter directive does not use *#!/usr/bin/env* but rather the full path to the interpreter.

When no capability restriction rules are used for a given subject, all capabilities that the system grants normally to processes within that subject are allowed to be used. An exception to this is if the subject involved uses policy inheritance. In that case, the capability restrictions would come from the subject(s) being inherited from. Capability rules have the form +CAP_NAME or -CAP_NAME. CAP_ALL is a pseudo-capability meant to describe the entire list of capabilities. It's mainly used to remove all capability usage for a subject, or in conjunction with a small number of rules granting the ability to use individual capabilities. Provided below are some example scenarios of capability restriction usage, along with an explanation of how the policy is interpreted.

Scenario #1: In this scenario, we're removing all capabilities from `su` but CAP_SETUID and CAP_SETGID.

```
...
 subject /bin/su o
  ...
   -CAP_ALL
   +CAP_SETUID
   +CAP_SETGID
```

Scenario #2: In this scenario, we're making use of policy inheritance. Note that the default subject allows CAP_NET_BIND_SERVICE and CAP_NET_RAW. In our `ping` subject, we're removing CAP_NET_BIND_SERVICE, but since we're inheriting from the default subject (note the lack of the *o* subject mode on the `ping` subject), we are still allowed CAP_NET_RAW. Granting important capabilities to default subjects is not something allowed by the RBAC system, so this is just an example.

```
...
 subject /
  ...
   -CAP_ALL
   +CAP_NET_RAW
   +CAP_NET_BIND_SERVICE
 subject /bin/ping
  ...
   -CAP_NET_BIND_SERVICE
```

Auditing and Suppression: Auditing of attempted capability use and suppression of denied capability usage is possible as well. Capability auditing and suppression supports the same policy inheritance rules as normal capability rules. The below example demonstrates auditing the use of CAP_NET_RAW and the suppression of CAP_NET_BIND_SERVICE denials:

```
...
 subject /
  ...
   -CAP_ALL
```

```
    -CAP_NET_BIND_SERVICE suppress
    +CAP_NET_RAW audit
```

For a full listing of the capabilities available, see: Capability Names and Descriptions. Note that not all of the capabilities listed may be supported by your particular version of the Linux kernel.

One of the features of grsecurity's ACL system is process–based resource restrictions. Using this feature allows you to restrict things like how much memory a process can take up, how much CPU time, how many files it can open, and how many processes it can execute. Also in this section, we will discuss a "fake" resource implemented in grsecurity's ACL system called "RES_CRASH" that helps guard against bruteforce exploit attempts, which is necessary if you're using PaX.

A single resource rule follows the following syntax:

```
<resource name> <soft limit> <hard limit>
```

An example of this syntax would be:

```
RES_NOFILE 3 3
```

This would allow the process to open a maximum of 3 files (all processes have 3 open file descriptors at some point: stdin (standard input), stdout (standard output), and stderr (standard error output)).

To clarify what the soft limit and hard limit are, the soft limit is the limit assigned to the process when it is run. The hard limit is the maximum point to which a process can raise the limit via `setrlimit(2)`, unless they have CAP_SYS_RESOURCE. In the case of RES_CPU, when the soft limit is overstepped, a special signal is sent to the process continuously. When the hard limit is overstepped, the process is killed.

A person who is less familiar with Linux should stick to setting limits on the number of files, the address space limit, and number of processes. Of course, you can always use the learning mode of grsecurity to set the resource limits for you. The RES_CPU resource is the only one that accepts time as limits. The time defaults to units of milliseconds. You can also append a case sensitive unit to your limit.

Some examples would be:

- 100s – 100 seconds
- 25m – 25 minutes
- 65h – 65 hours
- 2d – 2 days

The other resources either operate on a number itself or on a size, in bytes. For these you can use the following units: K, M, and G, like:

- 2G – 2 billion
- 25M – 25 million

- 100K – 100 thousand

If you don't want any restriction for the soft or hard limit for a resource, you can use "unlimited" as the limit. Here are some more examples to help you understand how this works:

```
subject /bin/bash
       /               r
       /opt            rx
       /home           rwxcd
       /mnt            rw
       /dev
       /dev/grsec      h

       RES_CPU 25m 30m
       RLIMIT_AS 5M 5M
       RLIMIT_NPROC 2 2
       RLIMIT_FSIZE 5K 10K
...
```

For a list of accepted resource names and units, see System Resources.

This "fake" resource limit is expressed by using the name "RES_CRASH" and has the following syntax:

```
RES_CRASH <number of crashes> <amt. of time>
```

For example, if you wanted to allow the program to crash once every 30 minutes, you would use the following:

```
RES_CRASH 1 30m
```

What happens when this threshold is reached? Well, the only way to ensure that the process won't crash again is to keep it from being executed. If the process is a suid/sgid binary run by a regular user, we kill all processes of that regular user and keep them from logging in for the amount of time, specified as the second parameter to the RES_CRASH resource. So for the above example, the user would be locked out of the system for 30 minutes. If the process is not a suid/sguid binary, we simply keep the binary from being run again for the amount of time specified as the second parameter to the RES_CRASH resource, after killing all processes of that binary.

The RBAC system supports policies on what local IP addresses and ports can be reserved on the machine, as well as what remote hosts and ports can be communicated with. These two different accesses are abstracted to *bind* and *connect* rules, respectively. The syntax for the rules is:

```
  connect <IP/host>/<netmask>:<port/portrange> <socket type 1>... <socket type n> <proto 1>... <proto
  bind <IP/host>/<netmask>:<port/portrange> <socket type 1>... <socket type n> <proto 1>... <proto n

  or:
```

```
connect disabled
bind disabled
```

"proto" can be any of the protocol names listed in */etc/protocol* or "any_proto" to denote any protocol. "socket type" is most commonly "ip", "dgram", or "stream", but can also be "raw_sock", "rdm", or "any_sock" to denote any socket type. Most of the parameters for these rules are optional, particularly the netmask and port or port range. If a port is supplied, then at least an IP address of 0.0.0.0/0 needs to be supplied.

As with capability restrictions, resource restrictions, and many other RBAC features, if the socket policies are omitted for a given subject, then the subject is allowed to bind or connect to anything normally allowed by the system. Note though that if a connect rule is given, then at least one bind rule must also be specified. Older versions of `gradm` (before the 9/16/09 2.1.14 release) will treat the unspecified rule as a "disabled" rule, whereas new versions will generate an error on such policies.

Unlike with file objects and capabilities, policy inheritance has not been implemented for socket policies. Therefore, the socket policies for a given subject are solely determined by that subject alone.

Here are some example rules:

```
subject /usr/bin/ssh o
...
connect 192.168.0.0/24:22 stream tcp
connect ourdnsserver.com:53 dgram udp
```

In this example, `ssh` is allowed to connect to ssh servers anywhere on the class C 192.168.0.X network. It is also allowed to do DNS lookups through the host specified. The hostname is resolved at the time the RBAC system is enabled.

```
subject /usr/bin/nc o
...
bind 0.0.0.0/0:1024-65535 stream tcp
connect 22.22.22.22:5190 stream tcp
```

In this example, `netcat` is allowed to listen on ports 1024 through 65535 on any local interface for TCP connections. It is also able to connect to TCP port 5190 of the 22.22.22.22 host.

```
subject /bin/strange o
...
bind disabled
connect 192.168.1.5:6000-6006 stream tcp
```

This example illustrates how you can have bind disabled but still specify connect rules, or conversely, have connect disabled and only specify bind rules.

As you can see from the examples above, you can have as many socket policies as you wish for a given subject, and as you'll read below there are some powerful extensions to the socket policies.

Rules such as:

```
bind eth1:80 stream tcp
bind eth0#1:22 stream tcp
```

are allowed, giving you the ability to tie specific socket rules to a single interface (or by using the inverted rules mentioned below, all but one interface). Virtual interfaces are specified by the <ifname>#<vindex> syntax. If an interface is specified, no IP/netmask or host may be specified for the rule.

Rules such as:

```
connect ! www.google.com:80 stream tcp
```

are allowed, which allows you to specify that a process can connect to anything except to port 80 of www.google.com with a stream TCP socket. The inverted socket matching also works on bind rules.

In more recent versions of the RBAC system, PaX flags have been changed from single-letter subject modes to more closely resemble how capabilities are handled within the policy. Therefore, PaX flags can now be fully controlled on or off for any given subject by adding +PAX_<feature> or -PAX_<feature> within the scope of a subject. For a full listing of the PaX flags available, see: PaX Flags.

Each process on the system has a role and a subject attached to it. This section describes how a process is matched to a role and subject, and how matches are calculated against the objects and capabilities they use. Understanding the flow of matches is necessary for manually creating policies.

When determining a role for a process, the RBAC system matches based on the following role hierarchy, from most specific to least specific:

```
 user -> group -> default
```

Both user and group roles are permitted to have the `role_allow_ip` attributes. When checking the UID or GID against the user or group role, respectively, the `role_allow_ip` attributes come into play. Imagine the following policy:

```
role user1 u
role_allow_ip 192.168.1.5
...
```

If someone attempted to log in to the machine as user1 from any IP address other than 192.168.1.5, they would not be assigned the user1 role. The matching system would then fall back on trying to find an acceptable group role, or if one could not be found, fall back to the default role.

Hierarchy for subjects and objects involves matching a most specific pathname over a less specific pathname. So, if a */bin* object exists, and a */bin/ping* object exists, and a process is attempting to read */bin/ping*, the */bin/ping* object would be the one matching. If */bin/su* were being accessed instead, then */bin* would match.

The path from most specific to least specific pathname isn't linear however, particularly in the case of subjects using policy inheritance. Imagine the following policy:

```
role user1 u
  subject /
    / r
    /tmp rwcd
    /usr/bin rx
    /root r
    /root/test/blah r
   ...
  subject /usr/bin/specialbin
    /root/test rw
   ...
```

If */root/test/blah* was being accessed by */usr/bin/specialbin*, it would not be able to write to it. The reason for this is that when going from most specific to least specific for a given path (which involves stripping off each trailing path component and attempting a match for the resulting pathname), the matching algorithm will look (in order from most specific to least specific) in each of the subjects the current subject inherits from. In this case, the algorithm saw that no object existed for */root/test/blah* in the */usr/bin/specialbin* subject, so upon checking the subject for */* it found a */root/test/blah* object, thus resulting in the read-only permission.

When going from most specific to least specific, a globbed object such as */home/** is treated as less specific than */home/blah* (if the requested access is for */home/blah*). Globbed objects are matched in the order in which they're listed in the RBAC policy. So in the following example:

```
role user1 u
  subject /
     / r
     /home r
     /home/* r
     /home/test* rw
    ...
```

If a process were accessing */home/testing/somefile* it would only be allowed to read it, since the */home/** rule was listed first. It was likely that the policy writer didn't intend this behavior (because the */home/test** rule would never match) so the */home/test** object should be swapped to the line the */home/** object is on.

When determining whether a capability is granted or not, the RBAC system works from most specific subject to least specific (in the case of policy inheritance). The first subject along that path that mentions the capability in question is the one that matches. To illustrate:

```
role user1 u
  subject /
   ...
    -CAP_ALL
    +CAP_NET_BIND_SERVICE
  subject /bin
   ...
    -CAP_NET_BIND_SERVICE
  subject /bin/su
   ...
    +CAP_SETUID
    +CAP_SETGID
```

In this example, */bin/su* is able to use only CAP_SETUID and CAP_SETGID. A lookup on CAP_NET_BIND_SERVICE would fall back to the */bin* subject, since */bin/su* inherits from it and did not explicitly list a rule for CAP_NET_BIND_SERVICE. The */bin* subject specifies that CAP_NET_BIND_SERVICE be disallowed. Matching against another capability, CAP_SYS_ADMIN for instance, would end up falling back to the / subject, where it would match -CAP_ALL and be denied.

Try to remove as many capabilities from default subjects as possible. The more you remove, the closer root comes to acting as a regular user. The more capabilities you remove, however, the more subjects you will have to create for programs that need those capabilities. The RBAC system will enforce that a minimum level of capabilities be removed from all default subjects.

Use full system learning. It will generate a better policy than you would have generated by hand. Make sure you're making full use of the */etc/grsec/learn_config* file to specify the files and directories particular to your system that you want protected. `gradm` will do all the heavy lifting of creating privilege boundaries for processes that access or modify important data.

Administrative programs, such as shutdown or reboot, should require authentication instead of giving everyone the capabilities to run them.

Always inspect your kernel logs. The RBAC system provides a great amount of human-readable information in every kernel log. Of particular importance is what role and subject were assigned to the process causing an alert. If you think that the alert doesn't match up with what you expect from your policy, make sure that the role and subject actually match. If they don't, then you may have issues with a *role_allow_ip* rule that's preventing the proper role from being applied.

Familiarize yourself with Linux's capabilities and what they cover. A full listing of them is available here: Capability Names and Descriptions.

Avoid using policy inheritance until you understand fully how it forms the policy for a given subject. Even then, use it sparingly, reserving it generally for cases where a default subject is configured least privilege, with no readable/writable/executable objects and no capabilities.

Wherever possible, avoid granting both write and execute permission to objects. This gives a potential attacker the ability to execute arbitrary code. Similar to how PaX prevents arbitrary code execution within a given process' address space, one of your goals in creating policies is to prevent this on the file system as well.

Be careful using the suppression ('s') object flag, especially when applying it to / to ignore accesses a program does not really need to operate correctly. A change in glibc or another library the subject uses could cause the application to fail in a way that will be difficult to debug (unless your first step is to remove the suppression flag).

Below is the sample policy provided with a `gradm` installation:

```
role admin sA
subject / rvka
        / rwcdmlxi

role default G
role_transitions admin
subject /
        /               r
        /opt            rx
        /home           rwxcd
        /mnt            rw
        /dev
        /dev/grsec      h
        /dev/urandom    r
        /dev/random     r
        /dev/zero       rw
        /dev/input      rw
        /dev/psaux      rw
        /dev/null       rw
        /dev/tty?       rw
        /dev/console    rw
        /dev/tty        rw
        /dev/pts        rw
        /dev/ptmx       rw
        /dev/dsp        rw
        /dev/mixer      rw
        /dev/initctl    rw
        /dev/fd0        r
        /dev/cdrom      r
        /dev/mem        h
        /dev/kmem       h
        /dev/port       h
        /bin            rx
```

```
        /sbin           rx
        /lib            rx
        /usr            rx
# compilation of kernel code should be done within the admin role
        /usr/src        h
        /etc            rx
        /proc           rwx
        /proc/slabinfo  h
        /proc/kcore     h
        /proc/modules   h
        /proc/sys       r
        /root           r
        /tmp            rwcd
        /var            rwxcd
        /var/tmp        rwcd
        /var/log        r
# hide the kernel images and modules
        /boot           h
        /lib/modules    h
        /etc/grsec      h
        /etc/ssh        h

# if sshd needs to be restarted, it can be done through the admin role
# restarting sshd should be followed immediately by a gradm -u
        /usr/sbin/sshd

        -CAP_KILL
        -CAP_SYS_TTY_CONFIG
        -CAP_LINUX_IMMUTABLE
        -CAP_NET_RAW
        -CAP_MKNOD
        -CAP_SYS_ADMIN
        -CAP_SYS_RAWIO
        -CAP_SYS_MODULE
        -CAP_SYS_PTRACE
        -CAP_NET_ADMIN
        -CAP_NET_BIND_SERVICE
        -CAP_NET_RAW
        -CAP_SYS_CHROOT
        -CAP_SYS_BOOT

#       RES_AS 100M 100M

#       connect 192.168.1.0/24:22 stream tcp
#       bind    0.0.0.0 stream dgram tcp udp

# the d flag protects /proc fd and mem entries for sshd
```

```
# all daemons should have 'p' in their subject mode to prevent
# an attacker from killing the service (and restarting it with trojaned
# config file or taking the port it reserved to run a trojaned service)

subject /usr/sbin/sshd dpo
        /               h
        /bin/bash       x
        /dev            h
        /dev/log        rw
        /dev/random     r
        /dev/urandom    r
        /dev/null       rw
        /dev/ptmx       rw
        /dev/pts        rw
        /dev/tty        rw
        /dev/tty?       rw
        /etc            r
        /etc/grsec      h
        /home
        /lib            rx
        /root
        /proc           r
        /proc/kcore     h
        /proc/sys       h
        /usr/lib        rx
        /usr/share/zoneinfo r
        /var/log
        /var/mail
        /var/log/lastlog        rw
        /var/log/wtmp           w
        /var/run/sshd
        /var/run/utmp           rw

        -CAP_ALL
        +CAP_CHOWN
        +CAP_SETGID
        +CAP_SETUID
        +CAP_SYS_CHROOT
        +CAP_SYS_RESOURCE
        +CAP_SYS_TTY_CONFIG

subject /usr/X11R6/bin/XFree86
        /dev/mem        rw

        +CAP_SYS_ADMIN
        +CAP_SYS_TTY_CONFIG
        +CAP_SYS_RAWIO
```

```
        -PAX_SEGMEXEC
        -PAX_PAGEEXEC
        -PAX_MPROTECT

subject /usr/bin/ssh
        /etc/ssh/ssh_config r

subject /sbin/klogd
        +CAP_SYS_ADMIN

subject /sbin/syslog-ng
        +CAP_SYS_ADMIN

subject /usr/sbin/cron
        /dev/log rw

subject /bin/login
        /dev/log rw
        /var/log/wtmp w
        /var/log/faillog rwcd

subject /sbin/getty
        /var/log/wtmp w

subject /sbin/init
        /var/log/wtmp w
```

Below is a full user role policy that covers the behavior of `cvs-pserver` when run as the non-root cvs user, providing anonymous read-only CVS repository access.

```
role cvs u
   subject /
      /     h
      -CAP_ALL
      connect   disabled
      bind      disabled

   subject /usr/bin/cvs
      /
      /etc/fstab         r
      /etc/mtab          r
      /etc/passwd        r
      /proc/meminfo      r
      /dev/urandom       r
      /dev/log           rw
```

```
      /dev/null          rw
      /home/cvs          r
      /home/cvs/CVSROOT/val-tags  rw
      /home/cvs/CVSROOT/history   ra
      /tmp               rwcd
      /var/lock/cvs      rwcd
      /var/run/.nscd_socket       rw
      /proc/sys/kernel   r
      /var/run
```

Here's all that's needed for an unprivileged sshd account:

```
role sshd u
   subject /
      /  h
      /var/run/sshd r
      -CAP_ALL
      bind disabled
      connect disabled
```
