GRSECURITY

ACL DOCUMENTATION

V1.5

Brad Spengler

http://www.grsecurity.net

spender@grsecurity.net

April 1, 2003

TABLE OF CONTENTS

I.   Introduction

    A.   What Is an ACL System?

          An ACL (Access Control List) system is software that provides fine–grained access control for your computer.

    B.   Why USE an ACL System?

          You need an ACL system if you want to restrict access to files, capabilities, resources, or sockets to ALL users, including root. This is what is called a Mandatory Access Control (MAC) model. The other features of grsecurity are only effective at fending off attackers trying to gain root, so the ACL 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 ACL system are in no means perfect security, they greatly increase the difficulty of successfully compromising the system.

C. Features

What features does our ACL system offer?

- Process–based ACLs (soon: role-based)
- Process–based resource restrictions (soon: role-based as well)
- Process–based IP ACLs (soon: role-based as well)
- Resource to prevent bruteforce attacking of processes
- Full–featured intelligent learning mode that produces least-privilege ACLs with no configuration
- Full–featured fine–grained auditing
- Configurable process accounting
- Configurable log suppression
- Human–readable configuration files
- Secure and intelligent policy enforcement
- Supports hide, protect, and override subject flags
- Supports the PaX flags
- Shared memory protection
- Integrated local attack response on all alerts
- Supports read, write, append, execute, view, and read-only ptrace object permissions
- Subject flag that ensures a process can never execute trojaned code
- ACLs can be placed on non-existent files/processes
- ACL regeneration on subjects and objects
- Administrative mode to use for regular sysadmin tasks
- ACL system is resealed up admin logout
- /proc/pid file descriptor/memory restriction
- Globbing support on ACL objects
- Support for including additional ACL configurations via an include directive. The argument can be a directory or a file
- Not filesystem dependent
- Scales well: supports as many ACLs as memory can handle
- No runtime memory allocation
- SMP safe
- O(1) time efficiency for most operations

- Administrator mode
- ACL inheritance
- Option to hide kernel processes

II.	Installation

A.	Obtaining grsecurity and gradm

Grsecurity and gradm are both available at http://www.grsecurity.net.
The download page features the current stable and development versions.
The current source for grsecurity and gradm can be viewed at http://cvsweb.grsecurity.net/.
Changelogs are updated nightly, and are available at http://www.grsecurity.net/cvs–changelog and http://www.grsecurity.net/cvs–gradm–changelog for grsecurity and gradm, respectively.

B.	Installing grsecurity

After downloading the patch for grsecurity place it in `/usr/src`. Download the version of the Linux kernel mentioned in the name of the patch (i.e. If the patch name is `grsecurity-1.9.9f-2.4.20.patch`, download the 2.4.20 kernel) and also place it in `/usr/src`.

```
tar -zxf linux-<version>.tar.gz
```

or, if you downloaded a bzipped kernel:

```
tar -jxf linux-<version>.tar.bz2
patch -p0 < grsecurity-<version>.patch
cd linux
make menuconfig
```
or:
```
make xconfig
make dep bzImage modules modules_install install
```

Be sure to update your bootloader if `make install` doesn't.

C.	Installing gradm

After downloading gradm:

```
tar -zxf gradm-<version>.tar.gz
cd gradm
make
make install
```

It will prompt you for the password to be used for administrating the ACL system.

Choose a long password, but one that you will remember (especially if you start `gradm` from an initscript). DO NOT use the same password as your root password.

III.   Configuration and Specification

A.   ACL Structure

ACLs for grsecurity are made up of subjects and objects and are implemented with processes, being subjects and files, capabilities, resources, and IP ACLs, being objects. The location of the main ACL file is `/etc/grsec/acl`. The structure of an ACL is as follows:

```
<path of subject process> <optional subject modes> {
        <file object> <optional object modes>
        [+|-]<capability>
        <resource name> <soft limit> <hard limit>
        connect {
            <ip>/<netmask>:<low port>-<high port> <type> <proto>
        }
        bind {
            <ip>/<netmask>:<low port>-<high port> <type> <proto>
        }
}
```

Some rules for our ACLs:

- All paths must be absolute paths, i.e. `/bin/sh` as opposed to `sh`. A newer feature of the ACL system is globbing support, which will be discussed later.
- Omitting a mode parameter for an object allows FIND access to the object. FIND access encompasses stat(), chdir(), filldir(), etc, but does not allow any other access to the object.
- To include additional ACL files, simply use `include <pathname>`. This must be done outside of any subject ACL definition. For example, if you had several ACL configurations in the directory `/etc/grsec/acls`, and you wanted to include them all, you could use:

```
include </etc/grsec/acls>
```

- Commenting is supported in ACL configurations at any point in the line. Comments start with the "#" character, and end at the end of the line.

- Neither subject nor object paths need to be a specific binary or file, they can also be directories. You'll see why this is important when we discuss ACL inheritance later on in this document.

- You MUST have a subject ACL for `/`. This is also referred to as the default ACL. Gradm will tell you if you do not have a default ACL.

- For all subjects with override set as a subject mode, an object ACL must be present for `/`.

- `CAP_ALL` is not a real capability, but was coded into gradm to represent all capabilities. Therefore to denote dropping of all capabilities, but `CAP_SETUID`, `-CAP_ALL` and `+CAP_SETUID` would be used.

- Though this will be commented on in more depth in section III.C, access to a file is granted only when the file system's Discretionary Access Control (DAC) model grants permission, as well as grsecurity's ACL system. Therefore, the two rules:

```
/root h
/root/secretscript.sh rx
```

are completely valid.

Modes and what they represent:

Subject modes:

- `h` This process is hidden and only viewable by processes with the `v` mode.

- `v` This process can view hidden processes.

- `p` This process is protected; it can only be killed by processes with the `k` mode, or by processes within the same subject.

- `k` This process can kill protected processes.

- `l` Enables learning mode for this process.

- `d` Protect the /proc/<pid>/fd and /proc/<pid>/mem entries for processes in this subject.

- `b` Enable process accounting for processes in this subject.

- `P` DISABLES the PAGEEXEC feature of PaX on this subject

- S   DISABLES the SEGMEXEC feature of PaX on this subject

- M   DISABLES the MPROTECT feature of PaX on this subject

- R   DISABLES the RANDMMAP feature of PaX on this subject

- G   ENABLES the EMUTRAMP feature of PaX on this subject

- X   ENABLES the RANDEXEC feature of PaX on this subject

- O   Override the additional mmap() and ptrace() restrictions for this subject

- A   Protect the shared memory of this subject. No other processes but processes contained within this subject may access the shared memory of this subject.

- K   When processes belonging to this subject generate an alert, kill the process

- C   When processes belonging to this subject generate an alert, kill the process and all processes belonging to the IP of the attacker (if there was an IP attached to the process)

- T   Ensures this process can never execute any trojaned code

- o   Override ACL inheritance for this process. This will be discussed later in section III.B.

Object modes:

- r   This object can be opened for reading.

- w   This object can be opened for writing or appending.

- x   This object can be executed (or mmap'd with PROT_EXEC into a task).

- a   This object can be opened for appending.

- h   This object is hidden.

- t   This object can be ptraced, but cannot modify the running task. This is referred to as a 'read-only ptrace'.

- s   Logs will be suppressed for denied access to this object.

- i   This mode only applies to binaries. When the object is executed, it inherits the ACL of the subject in which it was contained. This will be explained in section III.B.

- R   Audit successful reads to this object

- W   Audit successful writes to this object

- X   Audit successful execs of this object

- A   Audit successful appends to this object

- F Audit successful finds of this object
- I Audit successful ACL inherits of this object

B.   Inheritance

There are two kinds of inheritance that we will discuss regarding the ACL system. The first pertains to the configuration file, and the second pertains to how ACLs are handled in the kernel.

Grsecurity implements a feature called "inheritance" in its ACL configuration. If you are familiar with C++, the idea won't be new to you, as it is similar to class inheritance. Inheritance applies to all ACLs that do not have "o" in the subject mode. There is a simple set of rules that govern inheritance in the ACL configuration. Note that the notion of "parent ACLs" will be explained in the next section.

If an object in the parent ACL does not exist in the current ACL, we are calculating inheritance for, add the object from the parent ACL into the current ACL.

Examples of inheritance: An ACL configuration such as:

```
/ {
        /               rwx
        /etc            rx
        /usr/bin        rx
        /tmp            rw
}
/usr/bin/mailman {
        /tmp            rwx
}
```

would expand to the following, once inheritance was calculated:

```
/ {
        /               rwx
        /etc            rx
        /usr/bin        rx
        /tmp            rw
```

```
        }
        /usr/bin/mailman {

                /               rwx

                /etc            rx

                /usr/bin        rx

                /tmp            rwx

        }
```

As you can see, `/usr/bin/mailman` now has the objects of `/`, and the declaration of an ACL for /tmp in /usr/bin/mailman overrode the declarations in the parent ACL.

The algorithm used in grsecurity does not simply calculate inheritance by the direct parent (e.g. `/bin` for `/bin/su`), but by all parents of the path. As an example, suppose the ACL subject was `/usr/X11R6/bin/XFree86`. That subject would inherit acls from `/usr/X11R6/bin`, `/usr/X11R6`, `/usr`, and `/`.

The reason behind implementing inheritance is to reduce the amount of configuration, necessary for similar binaries. If you wish to explicitly state the files a certain binary can access, you should use the "o" flag in the subject mode. This tells grsecurity not to perform inheritance for that subject.

Inheritance in regards to the kernel's handling of ACLs is a bit different. In this case, it involves copying the ACL of the subject when the object with "i" in its mode is executed. It allows you to grant special permissions to a process only when it is executed through a process of your choosing. An example of this kind of inheritance is:

```
        / {

                /       rwx

                /tmp    rw

        }


        /usr/bin/mozilla {

                /usr/bin/mozilla-bin rxi

                /tmp                    rwx

        }
```

For this example, assume `/usr/bin/mozilla` is a script that executes `/usr/bin/mozilla-bin`. When `/usr/bin/mozilla-bin` is executed by `/usr/bin/mozilla`, it inherits the ACL of `/usr/bin/mozilla`, which allows it to execute in `/tmp`. If `/usr/bin/mozilla-bin` were executed directly by a user, it would not be able to execute in `/tmp` as it would use the ACL for `/`.

As a final note, the "i" mode for objects will only have the desired effect if the application performs a `fork()` and then an `execve()`, or just an `execve()`. If the application executes the program through **`system()`**, you will not get the result you want, because the shell is being executed directly by the application, which in turn executes the program you wanted to have inheritance.

C.   Flow of Matches

To understand how to write correct ACL configurations, you need to understand how the ACL system generates a "match" for a given file, a mapping of how the current process is allowed to interact with that given file. The following is a flow chart that outlines the steps grsecurity takes to determine an ACL match.
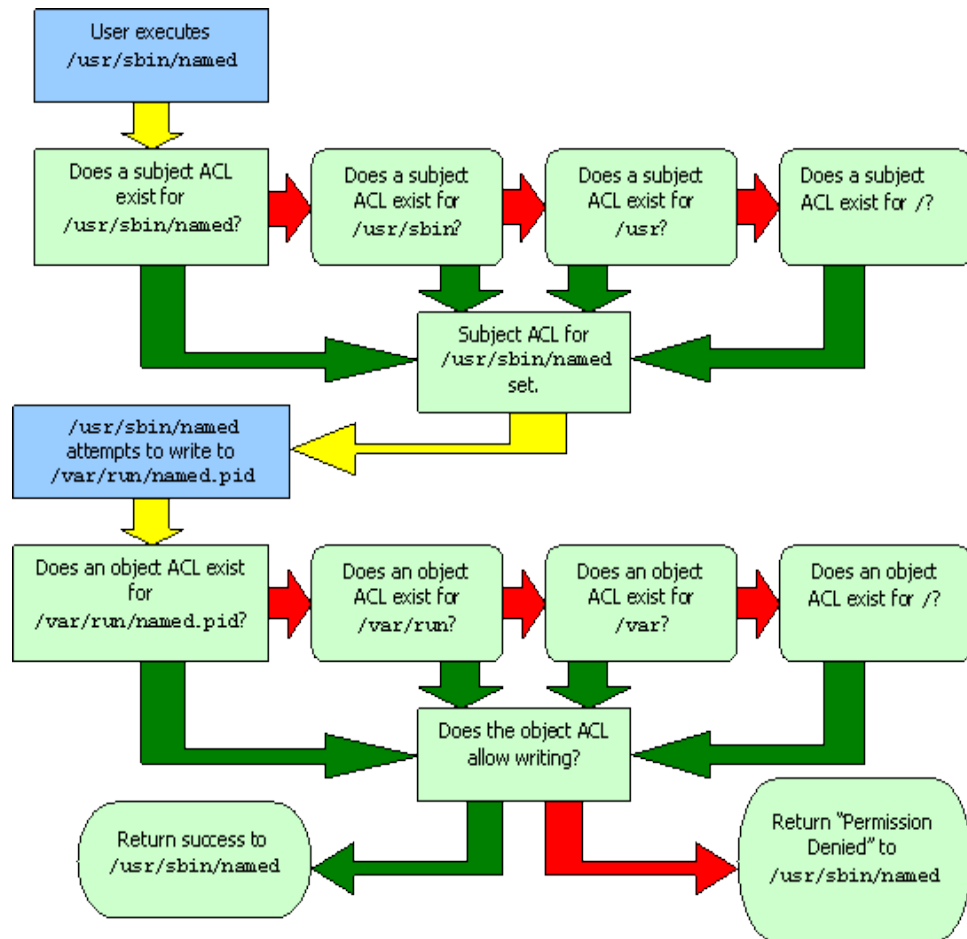
Light blue boxes are user actions; light green boxes are kernel actions. Green arrows indicate a successful operation; red arrows indicate a failed operation. As you can see, to create a "match" for a specific file, grsecurity follows the parent directories of the path until it finds an object ACL for that path. It then treats that path as the rule for the path it was looking for in the first place. This approach is quite different from the DAC model Linux uses. Permission to access a file does not depend on certain access to each parent directory of that file.

D.   Resource Restrictions

One of the newer 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, stdout, and stderr)

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.

The following is a list of accepted resource names (grsecurity supports all the resources Linux supports) and their descriptions:

- RES_CPU – CPU time in milliseconds
- RES_FSIZE – Maximum file size in bytes
- RES_DATA – Maximum data size in bytes
- RES_STACK - Maximum stack size in bytes
- RES_CORE – Maximum core size in bytes
- RES_RSS – Maximum resident set size
- RES_NPROC - Maximum number of processes
- RES_NOFILE – Maximum number of open files

- `RES_MEMLOCK` – Maximum locked–in–memory in bytes
- `RES_AS` – Address space limit in bytes
- `RES_LOCKS` – Maximum file locks

I suggest that a person who is less familiar with Linux 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 learning mode will be explained in section IV.

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:

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

Now on to the "fake" resource limit, implemented in grsecurity. It is expressed by using the name "`RES_CRASH`" and has the following syntax:

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

So 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 `s[u|g]id` 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 `s[u|g]id` 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.

E.   IP ACLs

Another new feature of grsecurity's ACL system is IP ACLs. IP ACLs allow you to control such things as what IPs and ports a process can bind to on a server, as well as what IPs and ports they can connect to remotely. You can also specify what kind of sockets a process is allowed to use (e.g. `stream, dgram, raw`), as well as, what protocols they're allowed to use (e.g. `tcp, udp, icmp`).

The syntax for IP ACLs is:

```
connect {

        <ip>/<netmask>:<low port>-<high port> <type> <proto>
}
bind {

        <ip>/<netmask>:<low port>-<high port> <type> proto>
}
```

If netmask is omitted, it is assumed to be 32. If high port is omitted, the low port will be set as the low and high values of the range. If both low and high ports are omitted, 0 is used as the low port and 65535 is used as the high port. "`type`" can be one of "`sock`", "`dgram`", "`raw_sock`", or "`any_sock`".   "`proto`" can be any of the protocol names listed in `/etc/protocols`, as well as "`raw_proto`" and "`any_proto`". You can have multiple IP ACLs per connect or bind definition. If you want to disable connect or bind for a specific process, instead of using an IP ACL, just use "`disabled`". If you don't specify connect or bind within your process ACL, all socket usage that is normally allowed will be allowed. If connect or bind ACLs are specified, then only accesses matching your rules will be allowed. The following are some examples of valid IP ACLs:

```
connect {
        disabled
}
bind {
        192.168.1.2/24:80     stream tcp
}
connect {
        192.168.1.2/24        stream dgram tcp udp
```

```
           134.55.22.12/24:80   stream tcp
    }
    bind {
           192.168.1.2/24:1024-65535 any_type any_proto
    }
```

Based on your ACLs for connect and bind, an ACL is automatically generated for `socket (2)` calls. If you had two rules that allowed some sort of stream tcp access, and another rule that allowed dgram udp access, any attempt to open a socket with raw set as the socket type, for instance, would fail.

F.   PaX flags and caveats

Another new feature of the ACL system is the support of the binary flags for PaX. More information on PaX is available at http://pageexec.virtualave.net/ and in the help for grsecurity's kernel configuration.

Repeating from earlier in this document, the following are the PaX flags in the ACL system:

- P DISABLES the PAGEEXEC feature of PaX on this subject

- S DISABLES the SEGMEXEC feature of PaX on this subject

- M DISABLES the MPROTECT feature of PaX on this subject

- R DISABLES the RANDMMAP feature of PaX on this subject

- G ENABLES the EMUTRAMP feature of PaX on this subject

- X ENABLES the RANDEXEC feature of PaX on this subject

It bears noting that the flags act in the opposite direction as the default binary flags. By default, PAGEEXEC, SEGMEXEC, MPROTECT, and RANDMMAP are all enabled on ELF binaries on the system. This also means that the PaX flags in the ACL system will never override the PaX flags set on a binary that are changed from the default flags. So if you for instance enable RANDEXEC on sshd with chpax, and you forget to put "X" in the subject mode for sshd, RANDEXEC will still be enabled on sshd.

Why would you want to use the PaX flags in the ACL system? Let's say you have a

developer whose name is Joe. Joe does lots of coding on your server, and he gets a thrill out of debugging his applications. Joe is tired of having to chpax his application every time he recompiles it so that he can debug it. Grsecurity's ACL system comes to the rescue. An ACL like the following:

```
/home/joe PSMR {
}
```

will solve Joe's problem.

The caveat to using the PaX flags in the ACL system is that while the ACL system can be applied at any time while the system is on, and can apply ACLs to already running processes, it cannot apply the PaX flags to already running processes. This is simply because of the way PaX works, and cannot be changed. Be aware of this when you're using the PaX flags in the ACL system.

G.   Object globbing

Yet another new feature of the ACL system is the support of the wildcards * and ? in ACL objects.   The "*" character matches zero or more characters, while "?" matches exactly one character. Depending on how these globbing characters are used, they have different effects. Here are two examples of the use of globbing:

/dev/tty*   rw
/home/*/bin   rwx

In the first example, you'll notice that a globbing character was used in the last path component. On a typical system, this would expand to /dev/tty, /dev/tty1, /dev/tty2, /dev/tty3, etc. Because the globbing character was used in the last path component, /dev/tty* expands to only the files that match the expression at the time the ACL system is enabled.

In the second example, you'll notice that there is no globbing character in the last path component. This is handled differently than the previous case, because we can do a trick here that will make it more useful. Let's say you have many users on your system, and you allow them to have a bin dir within their home directory, where you will allow write and execute access to. Some of the users on your system have this bin directory already set up, but others do not. You don't want to have to reconfigure your ACLs every time a user sets up his bin directory, so you use the /home/*/bin ACL.

The way these work is as follows: The last path component is stripped off, which in this case would leave /home/*. All matches are found for that expression, and then the last path component is appended to each of the directories that result (eg. `/home/user1/bin, /home/user2/bin`, etc), and then each of those objects are added to the compiled ACL list, whether or not they exist yet. So what happens is when the user creates their bin directory, it automatically gets the rwx ACL for the bin directory.

What if you want to place a special ACL on /dev/null, for example, and cover the rest of the files in /dev with a globbed ACL? You must specify the /dev/null rule first, otherwise gradm will report the /dev/null as a duplicate.

H.   Miscellaneous notes

The mmap and ptrace restrictions in grsecurity I believe need some additional clarification.

You may have noticed from time to time while you were trying to set up your ACLs some logs from grsecurity about "attempting to load writable libraries." The reason you are receiving this log is because a process on your system tried to load a library from a location that is allowed to be written to with the default ACL (the subject ACL for /). One of the reasons why this check is in place is because it is possible to leak privileges from privileged processes by using `LD_PRELOAD, LD_LIBRARY_PATH`, and others. To correct the "attempting to load writable library" error, you need to remove write permission from the location in your default ACL that is matching the library being loaded. This check is avoided if the binary doing the mmap is writable itself by the default ACL, since if the binary itself can't be trusted, there's no sense in enforcing that the libraries it loads be trusted.

Ptraces are disallowed across subjects, and subjects other than the default ACL do not allow ptracing of other processes within the same subject.

Both of these additional restrictions can be ignored for a given subject by adding the "O" override flag to its subject mode. The override flag should rarely be used, if ever.

The handling of symlinks needs an additional clarification as well. For correct handling of symlinks in the ACL configuration, whenever an ACL for a symlink is encountered, two ACLs are actually added. An ACL for the symlink itself is added, and the real path of its target gets the same ACL added. What if you want to put one ACL on the symlink, and a different ACL on its target? You must specify the ACL for the target first, otherwise gradm will report a duplicate.

The automatic local attack response works for all alerts in grsecurity that would be signs of an attack, not just those of the ACL system.

I.   ACL Recommendations

Now that you know how the ACL system works, here are some tips to help you create secure ACL configurations.

- Try to remove as many capabilities from the default ACL as possible. The more you remove, the closer root comes to acting as a regular user. The more capabilities you remove, however, the more ACLs you will have to create for programs that need those capabilities.

- Use the ACLs in the `debian_secure_acls` directory of gradm if possible.

- Use the learning mode. It can create ACLs better than you can.

- Administrative programs, such as shutdown or reboot, should require authentication instead of giving everyone the capabilities to run them. You can do this by making an ACL for `/sbin/shutdown`, and making it hidden to all processes by putting `/sbin/shutdown h` as an object in your process ACL for `/`. Then, the only way you can access the special privileges of `/sbin/shutdown` is by entering administration mode with `gradm -a`.

- Familiarize yourself with Linux's capabilities and what they cover. A listing and description of each of the capabilities is present at the end of this document, and also in `include/linux/capability.h` of your Linux source tree.

J.   Sample ACLs

To give you an idea of what actual ACL configurations look like, here are a few sample
ACLs:

```
/usr/sbin/apache oXA {

    /usr/share r

    /etc r

    /etc/grsec h

    /etc/ld.so.cache r

    /tmp rwx

    /lib rx

    /usr/lib rx

    /var/log/apache a

    /var/run/apache.pid w

    /var/www rx

    /dev/null rw

    /bin/bash x

    /usr/sbin/apache x

    /

    -CAP_ALL

    +CAP_DAC_OVERRIDE

    +CAP_KILL

    +CAP_SETGID

    +CAP_SETUID

    +CAP_NET_BIND_SERVICE

    RES_CRASH 1 10m

    connect {
```

```
            0.0.0.0/0:53 dgram udp

    }

    bind {

            0.0.0.0/0:80 stream tcp

    }

}

/usr/X11R6/bin/XFree86 SP {

    /var/log/XFree86.0.log rw

    /var/log/XFree86.1.log rw

    /dev/mem rw

    /dev/nvidiactl rw

    /dev/nvidia0 rw

    +CAP_CHOWN

    +CAP_SETUID

    +CAP_SETGID

    +CAP_DAC_OVERRIDE

    +CAP_SYS_RAWIO

    +CAP_SYS_MODULE

    +CAP_SYS_NICE

    +CAP_SYS_TTY_CONFIG

    +CAP_IPC_OWNER

}

/usr/lib/postfix o {

    /var/spool/postfix rw

    /var/spool/postfix/lib rx

    /var/mail w

    /dev/log rw
```

```
        /dev/null rw

        /dev/urandom r

        /etc/aliases

        /etc/postfix rw

        /etc r

        /etc/grsec h

        /lib rx

        /usr/lib rx

        /usr/share/zoneinfo r

        /var/tmp

        / h

        -CAP_ALL

        +CAP_DAC_OVERRIDE

        +CAP_KILL

        +CAP_SETGID

        +CAP_SETUID

        +CAP_SYS_CHROOT

        connect {

            0.0.0.0/0:53 stream dgram ip tcp udp

            0.0.0.0/0:25 stream ip tcp

        }

        bind {

            disabled

        }

    }

    /bin/login oX {

        /var/mail
```

```
/var/log/btmp

/proc

/home

/root

/var/run/utmp rw

/var/log/wtmp w

/var/log/lastlog rw

/var/log/faillog rw

/usr/share/zoneinfo r

/lib rx

/etc r

/etc/grsec h

/dev/tty1 w

/dev/tty2 w

/dev/tty3 w

/dev/tty4 w

/dev/tty5 w

/dev/log rw

/bin/bash x

/bin/login x

/ h

-CAP_ALL

+CAP_CHOWN

+CAP_FOWNER

+CAP_FSETID

+CAP_SETGID

+CAP_SETUID
```

```
RES_CRASH 1 10m

connect {

    disabled

}

bind {

    disabled

}

}
```

IV.   Using Gradm and the Learning Mode

Gradm is a powerful tool that parses your ACLs, performs the enforcement of a secure base policy, optimizes the ACLs, as well as handles parsing of the learning logs, merges them with your ACL set and outputs the final ACLs.

First, a quick rundown of all the options of gradm:

`-E`	Enable the ACL system
`-R`	Reload the ACL system (only valid while in admin mode)
`-M <filename|uid>`	Remove an execution ban on a given uid or filename, that has been put in place by the `RES_CRASH` resource restriction of the ACL system.
`-L [logfile]`	Parses the learning logs. Accepts an optional argument, which specifies the kernel logfile to scan for the learning logs. Learning logs are logged through syslog with a level of INFO. If the argument is not specified, gradm will scan your `/etc/syslog.conf` file to find a suitable log to scan. This option has to be used with `-O`.
`-O <filename|stream>`	Specifies output mode for learned ACLs. Requires a single argument that can be «stdout», «stderr», or a regular file. Only used with `-L`.
`-D`	Disable the ACL system.
`-T <subject> <object>`	Displays the permissions for object, allowed by subject.
`-P`	Setup the password for the ACL system.
`-a`	Obtain full administrative capabilities (aka admin mode).
`-h`	Display help information.
`-v`	Print version information.

To recap where you're allowed to execute certain options of gradm, here's a table:

	Enabled	Disabled	Admin
Enable	Ignored	Allowed	Ignored
Disable	Allowed	Ignored	Allowed
Reload	Denied	Ignored	Allowed
Segvmod	Allowed	Ignored	Allowed
Test	Denied	Allowed	Allowed
Learn	Denied	Allowed	Allowed
Admin	Allowed	Ignored	Allowed
Passwd	Denied	Allowed	Denied

Now on to the learning mode of grsecurity. The learning mode is different than anything found in other security systems. Our learning mode is process–based. You can use the learning mode on a single process, while the rest of the system remains protected as usual. The learning mode can learn all things that the ACL system supports: files, capabilities, resources, and socket usage.

Using the learning mode is very simple. All you have to do is add "l" to the subject mode of the process, you want to enable learning for. Enable the ACL system with `gradm –E`. Run the application(s) you enabled learning mode for several times. This is important, since the learning mode uses a threshold–based system to determine when access should be given to a file or whether it should be given to a directory. If 4 or more similar accesses are made in a single directory (such as writing to several files in `/tmp`), access is granted to that directory instead of the individual files. This reduces the amount of rules you have and ensures that the application will work correctly after the final ACLs are compiled.

Once you feel you've given the application the normal usage it would see in real life, disable the ACL system with `gradm –D` (or alternatively, go into admin mode with `gradm –a`), and use `gradm –L –O /etc/grsec/acl`. This will place the new learned ACLs at the end of your ruleset. Simply remove the old ACLs and you're ready to go.

What the learning mode does is log every access attempt that would have been denied by your ACL for that process, while allowing the access to occur. Therefore, it should be clear

to you that to create a least privilege ACL you would use the following ACL:

```
/path/to/executable lo {

    / h


    ─CAP_ALL


    RES_FSIZE      0 0

    RES_DATA       0 0

    RES_RSS        0 0

    RES_NOFILE     0 0

    RES_MEMLOCK    0 0

    RES_STACK      0 0

    RES_AS         0 0

    RES_NPROC      0 0

    RES_LOCKS      0 0


    connect {

        disabled

    }


    bind {

        disabled

    }
}
```

V.   FAQ

A.   Where Do I Look for Additional Information?

- If you need help with gradm, use `gradm -h` or `man gradm`.
- If you need help creating ACLs, use the forums at [http://forums. grsecurity.net/](http://forums.grsecurity.net/) or the mailing list, which you can view more information on at http://www.grsecurity.net/mailinglist.php.

B.   I've Found a Bug! What Do I Do?

If you believe, you've found a bug in the ACL system, send an email to [dev@grsecurity. net](mailto:dev@grsecurity.net) with the following information:

- Version of Linux kernel you're using
- Version of grsecurity you're using (old versions will not be supported)
- If you see an error message in your log, please include relevant sections of the log
- Describe what you did to cause the bug
- If the bug caused a kernel oops, please include the oops as well as the output from ksymoops, if possible.

If you submit a patch or a solution to a bug, you will be credited in the changelog.

C.   Program "X" Isn't Working Since I Enabled the ACL system, What Gives?

If a program stops working properly after you enable the ACL system, it is most likely due to improper or incomplete ACLs. Go back and read the section on ACL suggestions and see if that helps. If the program still does not work, ask for help on the mailing list or the forums, and we'll help you fix it.

D.   How Can I Help?

The greatest way, you can help us, is by submitting thorough bug reports or patches,

and feature enhancements. If you are not a programmer, but still wish to help in some way, send an email to dev@grsecurity.net and tell us how you would like to help.

VI.　　　　Appendix

A.　　　Capability Names and Descriptions

0. **CAP_CHOWN** – In a system with the `[_POSIX_CHOWN_RESTRICTED]` option defined, this overrides the restriction of changing file ownership and group ownership.

1. **CAP_DAC_OVERRIDE** – Override all DAC access, including ACL execute access if `[_POSIX_ACL]` is defined. Excluding DAC access covered by `CAP_LINUX_IMMUTABLE`.

2. **CAP_DAC_READ_SEARCH** – Overrides all DAC restrictions, regarding read and search on files and directories, including ACL restrictions, if `[_POSIX_ACL]` is defined. Excluding DAC access covered by `CAP_LINUX_IMMUTABLE`.

3. **CAP_FOWNER** – Overrides all restrictions about allowed operations on files, where file owner ID must be equal to the user ID, except where `CAP_FSETID` is applicable. It doesn't override MAC and DAC restrictions.

4. **CAP_FSETID** – Overrides the following restrictions, that the effective user ID shall match the file owner ID, when setting the `S_ISUID` and `S_ISGID` bits on that file; that the effective group ID (or one of the supplementary group IDs) shall match the file owner ID when setting the `S_ISGID` bit on that file; that the `S_ISUID` and `S_ISGID` bits are cleared on successful return from `chown(2)` (not implemented).

5. **CAP_KILL** – Overrides the restriction, that the real or effective user ID of a process, sending a signal, must match the real or effective user ID of the process, receiving the signal.

6. **CAP_SETGID** –

- Allows `setgid(2)` manipulation;

- Allows `setgroups(2)`;

- Allows forged gids on socket credentials passing.

7. **CAP_SETUID** –

- Allows `set*uid(2)` manipulation (including `fsuid`);

- Allows forged pids on socket credentials passing.

8. **CAP_SETPCAP** – Transfer any capability in your permitted set to any pid,

remove any capability in your permitted set from any pid.

9. **CAP_LINUX_IMMUTABLE** – Allow modification of `S_IMMUTABLE` and `S_APPEND` file attributes.

10. **CAP_NET_BIND_SERVICE** –

- Allows binding to TCP/UDP sockets below 1024;

- Allows binding to ATM VCIs below 32.

11. **CAP_NET_BROADCAST** – Allow broadcasting, listen to multicast.

12. **CAP_NET_ADMIN** –

- Allow interface configuration;

- Allow administration of IP firewall, masquerading and accounting;

- Allow setting debug option on sockets;

- Allow modification of routing tables;

- Allow setting arbitrary process / process group ownership on sockets;

- Allow binding to any address for transparent proxying;

- Allow setting TOS (type of service);

- Allow setting promiscuous mode;

- Allow clearing driver statistics;

- Allow multicasting;

- Allow read/write of device–specific registers;

- Allow activation of ATM control sockets.

13. **CAP_NET_RAW** –

- Allow use of RAW sockets;

- Allow use of PACKET sockets.

14. **CAP_IPC_LOCK** –

- Allow locking of shared memory segments;

- Allow `mlock` and `mlockall` (which doesn't really have anything to do with IPC).

15. **CAP_IPC_OWNER** — Override IPC ownership checks.

16. **CAP_SYS_MODULE** —

    - Insert and remove kernel modules – modify kernel without limit;

    - Modify `cap_bset`.

17. **CAP_SYS_RAWIO** —

    - Allow `ioperm/iopl` access;

    - Allow sending USB messages to any device via `/proc/bus/usb`.

18. **CAP_SYS_CHROOT** — Allow use of `chroot()`.

19. **CAP_SYS_PTRACE** — Allow `ptrace()` of any process.

20. **CAP_SYS_PACCT** — Allow configuration of process accounting.

21. **CAP_SYS_ADMIN** —

    - Allow configuration of the secure attention key;

    - Allow administration of the random device;

    - Allow examination and configuration of disk quotas;

    - Allow configuring the kernel's syslog (printk behaviour);

    - Allow setting the domainname;

    - Allow setting the hostname;

    - Allow calling `bdflush()`;

    - Allow `mount()` and `umount()`, setting up new smb connection;

    - Allow some autofs root ioctls;

    - Allow nfsservctl; Allow VM86_REQUEST_IRQ;

    - Allow to read/write pci config on alpha; Allow irix_prctl on mips (setstacksize);

    - Allow flushing all cache on m68k (sys_cacheflush);

- Allow removing semaphores; Used instead of `CAP_CHOWN` to "chown" IPC message queues, semaphores and shared memory;

- Allow locking/unlocking of shared memory segment;

- Allow turning swap on/off;

- Allow forged pids on socket credentials passing;

- Allow setting readahead and flushing buffers on block devices;

- Allow setting geometry in floppy driver;

- Allow turning DMA on/off in xd driver;

- Allow administration of md devices (mostly the above, but some extra ioctls);

- Allow tuning the ide driver;

- Allow access to the nvram device;

- Allow administration of apm_bios, serial and bttv (TV) device;

- Allow manufacturer commands in isdn CAPI support driver;

- Allow reading non–standardized portions of pci configuration space;

- Allow DDI debug ioctl on sbpcd driver;

- Allow setting up serial ports;

- Allow sending raw qic–117 commands;

- Allow enabling/disabling tagged queuing on SCSI controllers and sending arbitrary SCSI commands;

- Allow setting encryption key on loopback filesystem.

22. **CAP_SYS_BOOT** – Allow use of `reboot()`.

23. **CAP_SYS_NICE** –

- Allow raising priority and setting priority on other (different UID) processes;

- Allow use of FIFO and round–robin (realtime) scheduling on own processes and setting the scheduling algorithm used by another

process.

24. **CAP_SYS_RESOURCE** –

- Override resource limits. Set resource limits;

- Override quota limits;

- Override reserved space on ext2 filesystem;

- Modify data journaling mode on ext3 filesystem (uses journaling resources); NOTE: ext2 honors fsuid when checking for resource overrides, so you can override using fsuid too;

- Override size restrictions on IPC message queues;

- Allow more than 64hz interrupts from the real–time clock;

- Override max number of consoles on console allocation;

- Override max number of keymaps.

25. **CAP_SYS_TIME** –

- Allow manipulation of system clock;

- Allow irix_stime on mips;

- Allow setting the real–time clock.

26. **CAP_SYS_TTY_CONFIG** – Allow configuration of tty devices; Allow `vhangup ()` of tty.

27. **CAP_MKNOD** – Allow the privileged aspects of `mknod()`.

28. **CAP_LEASE** – Allow taking of leases on files.