

Restricting UNIX Users

Anton Chuvakin 2002-05-01

Restricting UNIX Users

by Anton Chuvakin, Ph.D.

last updated May 1, 2002

Stories of cruel system administrators oppressing poor users have been around since the rise of UNIX in the 1970s. Users are inherently limited in what they can do on a UNIX system due to file permissions, passwords and other standard UNIX controls. However, it is often necessary to further restrict system users in other ways, both to protect them from themselves and to protect the system from the malicious or overly "playful" users. This article will discuss ways in which security administrators can limit what users are able to do on a UNIX system, with a particular focus on Linux. Both local and remote users will be considered. However, restricting root users from doing things on the system (while possible) is a somewhat different story and will not be addressed in detail here.

Limiting what users can do is also a good method preventing "insider attacks", which account for majority of losses from infosec-related incidents. (According to the annual CPI/FBI survey, 59% of companies surveyed said they have had one or more attacks reported internally. Almost 8% of those companies reported 60 or more internal incidents. The survey is available [here](#). Great perimeter defenses and network IDS will not stop the console user from wreaking havoc upon a system. Furthermore, organizational security policies might call for increased restrictions for end-users in order to reduce the IT management costs.

Objectives of Restricting Users

It is important to keep in mind the **purpose** of imposing a particular restriction on system users. Different environments imply different goals, which will require different tools to be used. For example, "saving" casual users from "complexities" of the UNIX shell is a different problem from keeping a determined expert attacker from exploiting a buggy SUID binary. In addition, a different degree of restriction might be needed for different environments, such as only allowing certain applications or only denying certain applications. Curiously, the latter is often futile since there are simply too many ways to break loose, just like a "default-allow" policy for the firewall usually provide only little security.

File Permissions

Let's start from the weakest limitation one can impose: use of UNIX permissions. In fact, it is probably sufficient for most environments. While seemingly inflexible, permissions allow control over what binaries users can execute. If all binaries except the ones that need to be executed by the untrusted users are not world executable, and all user-writable partitions such as `/tmp` and `/home` are mounted with `noexec` flag (see `man mount` for the details on that), the security of the set-up is adequate. `noexec` flag is needed to prevent users from downloading or building their own versions of binaries. If there is a pool of users that need the higher privileges, those can be gathered in a special group and binaries are made group-executable.

Linux file attributes, such as immutable, append-only, and others belong to the same groups of controls; they are, however, more useful to restrict root somewhat.

Permissions through PAM

Another simple (and weak) restriction of a different kind can be implemented via Linux system resource limits. It will prevent users from hogging system resources (disk space, RAM, CPU cycles), both in the case of abuse (fork bomb, mail bomb) or accidents (program fills the disk, spawns too many child processes, etc). Linux Pluggable Authentication Modules (PAM) can be used to impose resource limits. There are many guides available for doing this, such as [chapter 5.15 of Securing and Optimizing Linux: RedHat Edition - A Hands-on Guide](#) and [Using Pam](#) by Dave Wreski. For overview of PAM see the SecurityFocus articles [Pluggable Authentication Manuals, Part One](#) and [Part Two](#). The summary is shown below:

Limits for all users (apparently, except "root"):

File: /etc/security/limits.conf

```
*      hard   core    0
*      hard   rss     10000
*      hard   nproc   100
```

Shown above are limits for the size of core files, resident memory for the process and number of owned processes. "Hard" limits cannot be changed by the user, while "soft" can.

and for the specific group "lusers":

```
@lusers  hard   core    0
@lusers  hard   rss     2000
@lusers  hard   nproc   200
@lusers  hard   fsize   100000
@lusers  hard   nofile  100
@lusers  hard   cpu     10
@lusers  hard   priority 5           # same as "nice -n 5" on all processes
```

(in addition, maximum file size, number of open files, maximum CPU time and process priority are set)

Also, adding a line to the file /etc/pam.d/login is needed to make PAM check the above config file:

```
session      required          /lib/security/pam_limits.so
```

It should be noted, that PAM allows even more granular restrictions to be created. The RedHat whitepaper [Enhanced Console Access](#) outlines using PAM to limit (or, rather, to selectively enable) user's access to devices, X Windows applications, shutdown functionality and other privileges for console and remote users.

Restricted Shell

The next degree of limitation is to be a restricted shell. In this case, a version of a normal bash shell will prevent users from changing the directory and environment variables, redirecting output, running commands with absolute pathnames, using `exec` command and some other actions. Restrictions are not enforced for shell scripts. See `man bash` for more details. Combining `rbash` with a restrictive configuration of UNIX permissions can help achieve further security.

`Rbash` is a viable choice if you are trying to somewhat contain trusted users. Its restrictions can be easily overcome (see below).

To test `rbash` restricted shell functionality:

```
# adduser luser
# ln -s /bin/bash /bin/rbash
# echo "/bin/rbash" >> /etc/shells
# chsh -s /bin/rbash luser
# cd ~luser
# su luser
$
```

and then:

```
$ cd /
rbash: cd: restricted
```

Menu-Based Shells

Another solution is a custom application that only provides access to selected system resources. UNIX menu-based shells fall into this category. Let's look at several such solutions. [Pdmenu](#) provides a curses-based color interface for launching applications under Linux. It can safely be used as the user's log-in shell and only allows access to specified console applications such as mail or Web browser. It can also launch an application and then present the output to the user, such as `/bin/ls` for browsing directories. Apparently, users should not be given rights to add or modify the menu. Ideally, the menu shell should not require, or even allow, the user to type anything, but to only allow them choose from the list of presented options.

[Flash](#) is another menu-based Linux, using older and somewhat less-refined interface. It provides similar functionality with menus, commands and hot-keys.

These shells work on remote users (connected via SSH or telnet) and local users not using an X Window system.

Leaving the convenience features of those applications aside, let's concentrate on the extra security they might provide. If used as log-in shells, they limit user to only running applications that can be chosen from the menu. Provided that the normal shell (such as `/bin/bash`, or a `/bin/chsh` command to change a default shell) is not one of the applications and the user cannot modify the command line, there is seemingly no way to start anything else. Apparently, menu shell has no control over what happens after application is launched. And herein lies one of the pitfalls of the menu shells, which will be discussed later in this article, in the section on breaking out.

Similar to `rbash` above, this feature is more of a protection of non-technical users from themselves, rather than a true security mechanism.

Chroot

Another way to restrict users is to confine them to a certain directory via chroot. It does provide a degree of security in case that some conditions are met. For more details, see my analysis of chroot security [Using Chroot Securely](#). Briefly, if there is no way to get root privileges within chroot-ed directory, chroot does provide effective security. To prevent users from getting root within the chrooted directory, the set of application present within their jail should be carefully selected.

Combining restricted shell with a chroot protection increases the security of the whole set-up by providing defense in-depth. If the user manages to break out of the confines of the restricted shell, they would still end up in a directory without a method to get extra privileges. If chroot directory is well designed, the only thing gained after the breaking out is an ability to modify a command line for the allowed applications and not much else.

Moving outside of Linux realm, FreeBSD's jail command (described in detail at Chapter 12 of the FreeBSD Developers' Handbook, [The Jail Subsystem](#)) is a significant enhancement of simple chrooting. Jail is a great way to impose restrictions on remote users. Using both user-level and kernel-space code, jail creates a better confinement than simple chroot call by also restricting inter-process communication, network connectivity and some other system calls.

Kernel-Based Capabilities

Now let's turn to more serious security measures to further restrict users - kernel based capabilities. As outlined in the SecurityFocus article [Linux Kernel Hardening](#), several Linux security kernel patches implement mandatory access controls and role-based security. Using this functionality, one can take away almost any user access rights in various combinations. For example, [LIDS kernel patch](#) can prevent a user from binding network sockets, using or even seeing certain files, devices and processes, changing ownership of files and

using other UNIX functions.

GRSecurity kernel patch (described in detail in the SecurityFocus article [Grsecurity](#)) also has features to restrict users. `CONFIG_GRKERNSEC_TPE` option allows one to create a group for "untrusted" users who can only execute binaries from specified root-owned directories.

Admittedly, kernel-level protection is not fun to configure, unless one is into tweaking multiple ACL lists and tracing applications for allowed system calls. It is also not something that will work out of the box, since there will always be some heavy customization required. Kernel-level protection is the most difficult or even impossible to bypass, provided it is configured properly.

Overall, below the method usability summary is provided.

| | |
|------------------|---|
| Restricted shell | mostly "convenience" feature for limiting the non-technical users |
| Menu shell | provides some security if implemented correctly and applications are audited |
| Chroot | provides good security provided there is no root inside |
| Chroot+menu | combines security and convenience of the above two approaches |
| Kernel | high security, but also hard to configure, can be used to restrict expert users |

Breaking Out of Various Restrictions

Now it is time to touch upon the breaking out of various protective mechanisms.

First, let's look at the restricted shell. There are simply too many ways things can go wrong: modifying environment variables (such as `EDITOR`, `VISUAL`, and others unlocked by `rbash`), starting applications with shell escapes, changing user's shell, simply running the downloaded shell binary and numerous other methods can let the bird fly out of the cage.

A well-written menu shell is a significant improvement. The "breaking focus" is now on the applications that are being launched from the menu shell. The problem lies in the fact that many UNIX console applications have a shell escape. Examples include `pine` (`Ctrl-Z`), `vi` (`:!bash`), `ftp` (`!`), `telnet` (`!`), `emacs` (`M-x shell`), `gdb` (`shell`) and many others. (figures in brackets are the commands that invoke a shell.) It is interesting to note that some of the applications will spawn a new copy of restricted shell and some will use regular `/bin/sh`. It is worth noting that `vi` (`vim`) has a special restricted mode with no shell escape feature. It can be invoked by calling `rvim` in a manner similar to `bash` and `rbash`. To make applications resistant to such an attack, one has to manually remove the shell escapes from the source code, which is probably too time-consuming for most uses.

Even if the launched application does not provide for shell escape, one can crash the application using the

overflow shellcode (`/bin/sh`) typically used for overflowing SUID binaries and gaining root. If shellcode is used for crashing the application or the menu-based shell itself, the clean copy of `/bin/sh` will be launched (provided that one exists, which might not be the case in chroot environment).

For methods to break out of chroot, see [Using Chroot Securely](#). However, breaking out of well-written menu-shell, which only launches source-audited application all running in well-designed chroot environment, appears to be impossible.

Kernel-level protection is the closest to being unbreakable, if not for the bugs in the kernel security patch code (LIDS had a serious bug earlier this year) or configuration errors. The "good news" for the attackers, is that configuration errors are likely, due to complexity of the typical rule setup.

In addition, many scenarios in which the need to restrict shell users appears can be solved using a different technology. For instance, shell e-mail users can be moved to Web mail application and Web site editors can be given a CGI application to update their pages. Some virtual hosting solutions operate this way.

Conclusion

To conclude, many tools are available to restrict what users can do on a Linux system. Depending upon business requirements and other needs, one is able to choose the right combination of security and functionality. As usual, there is a trade-off between security and ease of set-up and use.

[Anton Chuvakin](#), Ph.D. is a Senior Security Analyst with [netForensics](#), a security information management company that provides real-time network security monitoring solutions.

[Privacy Statement](#)

Copyright 2006, SecurityFocus