`sudo` users

Mark Bools

December 10, 2020

Last Modified: February 25, 2021

Abstract

Protecting your system from accidental or malicious change is made simpler by allowing users only limited access. This especially means limiting users who have `root` or 'superuser' access as these users have effectively unlimited power. When it is necessary for a user to have this access it is best that they have a normal (unprivileged) account for day-to-day activities with the ability to temporarily promote themselves to 'superuser' to run commands that require this elevated access.

First, some basics of Linux access control.

Linux accounts all have two basic identities that the operating system uses to control access to facilities like which files the account holder can access and what programs the user can execute; the user id (`uid`) and the group id (`gid`). You can easily see the `uid` and `gid` for an account using the `id` command.

```bash
id vagrant
```

This will show the `vagrant` account's current `uid` and `gid` along with a list of 'groups' to which that account belongs.

```
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plug
```

Issuing the command `id` without specifing a username will return these details for the current account.

What does this all mean?

Access to files is specified on three axes; `user`, `group`, and `other`. For each axis we can specify `read` access, `write` access, and `execute` access. All of this information can be seen using the `ls` command.

```bash
ls -l
```

```
1  total 64
2  drwxr-xr-x  7 vagrant vagrant 4096 Nov 18 15:33 .
3  drwxr-xr-x 37 vagrant vagrant 4096 Dec  3 09:01 ..
4  -rw-r--r--  1 vagrant vagrant  419 Nov 18 09:23 404.html
5  -rw-r--r--  1 vagrant vagrant  412 Nov 18 09:23
   about.markdown
6  drwxr-xr-x  3 vagrant vagrant 4096 Nov 18 09:23 assets
7  -rw-r--r--  1 vagrant vagrant  672 Nov 18 09:23
   books.markdown
8  -rw-r--r--  1 vagrant vagrant 2352 Nov 18 09:23
   _config.yml
9  -rw-r--r--  1 vagrant vagrant 1131 Nov 18 09:23 Gemfile
10 -rw-r--r--  1 vagrant vagrant 1869 Nov 18 09:23
   Gemfile.lock
11 drwxr-xr-x  8 vagrant vagrant 4096 Nov 18 15:33 .git
12 -rw-r--r--  1 vagrant vagrant   56 Oct 10 13:36
   .gitignore
13 drwxr-xr-x  2 vagrant vagrant 4096 Nov 18 09:23 _includes
14 -rw-r--r--  1 vagrant vagrant  178 Nov 18 09:23
   index.markdown
15 drwxr-xr-x  3 vagrant vagrant 4096 Oct 10 13:41
   .jekyll-cache
16 -rw-r--r--  1 vagrant vagrant  629 Nov 18 09:23
   saltyvagrant.markdown
17 drwxr-xr-x  4 vagrant vagrant 4096 Nov 18 09:23 _sass§
```

Looking at line 3 we see that the `404.html` file has its access 'mode' set to `-rw-r--r--`. We read the first character as a special indicator (the first, in this example `-`, meaning 'not set'). The remaining nine indicators are read in groups of three. Each group contains; `r` read, `w` write, `x` execute, or `-` for 'not set'. The three groups represent `owner`, `group`, and `other`.

Reading the `404.html` file mode (`-rw-r--r--`) we see; special is not set (`-`), user (`vagrant` as indicated by the entry in the third column of output) can read and write but not execute (`rw-`), Any account in the group (`vagrant` as indicated by the entry in the fourth column of output) can read this file but can neither write not execute it (`r--`). The final three entries show that any account on the machine can read but neither write nor execute it (`r--`).

Looking at line 5 we see that `assets` is a directory, indicated by the `d` special indicator (the first character on the line). User `vagrant` can read, write, and execute the `assets` directory. What does 'execute' mean on a directory? Any account with execute access `x` can list the directory content (in this example any account can list the `assets` directory content as indicated by the `x` in the final position).

Under normal circumstances only the account `vagrant` would be able to add a file to the `assets` directry or make changes (write) to the `404.html` file.

Superusers however are above these restrictions and are able to do aything they please to these files.

Who are superusers?

The special `root` account is always a superuser. Generally it is good practice though to prevent direct login access to the `root` account. Instead we provide non-privileged accounts with a way to run privileged commands but only when some special action is taken. This need to take some special action to run commands as a superuser will (hopefully) reduce the potential for mistakes.

The special action required is called `sudo`, short for superuser do. This is a special command, normally used as a prefix to whatever command we want to run with superuser privilege.

Why is it bad practice to allow direct login access to `root` but instead use `sudo`? Four reasons;

1. Using `sudo` requires an explicit action to perform privileged actions (actions that may have significant effects on your system).
2. All `sudo` actions are logged, making it simpler to find who actually performed each privileged action.
3. `sudo` actions can be controlled such that the account may run only certain commands with superuser rights.
4. The right to run `sudo` commands can be granted and revoked very simply.

Logged in to the non-privileged account `vagrant` we can look at `ls -l /etc/hosts`, the local hosts file.

```
1  -rw-r--r-- 1 root root 254 Oct 14 13:39 /etc/hosts
```

This tells us that only the `root` user has write access to this file. Suppose the `vagrant` user needs to edit this file.

First we need to tell the operating system that `vagrant` is allowed to run elevated commands using `sudo`. This is very simple, the `vagrant` account must be added to the `sudo` *group*. Typically, any account that is a member of this `sudo` group is permitted to run the `sudo` command, and by extension is allowed to run commands as a superuser. I say 'typically' because the `sudo` facility allows much more subtle configuration of what each account or group may do. We could, for example, create a group `reboot` that would allow member of the group to issue a command to reboot the server but have access to no other privileged operation. Similarly, we could change the `sudo` group to allow limited access, but as I say, it is more common to leave `sudo` as all powerful and create other more specialised groups as required.

As it happens our `vagrant` account is already a member of the `sudo` group (we can see this in the output of the `id vagrant` command shown above).

If the `vagrant` account was not already a member of this group we could add it using the command `usermod -g -G sudo vagrant`. As you might expect

this command requires superuser privilege (otherwise any account would be able to add itself to the `sudo` group, making a mockery of our security). So the command we want is `sudo usermod -g -G sudo vagrant` run by an account that is already a member of the `sudo` group (alternatively we could use the `root` account without using `sudo` but as we said above, direct login access to the `root` account is to be avoided). The obvious point here being that when we initially set up our system to use `sudo` we must have at least one account as a member of the `sudo` group.

```bash
sudo vi /etc/hosts
```

Prefixing the edit comand with `sudo` allows the account to edit the file regardless of the normal access rights on the file.