Cgit, Nginx & Gitolite: A Personal Git Server

nginx + cgit + gitolite = $5/month

published: 12 January 2021

I've been on a *"own my online presence"* kick for more than a year now. So for this (overly protracted) essay, I thought I'd publish my notes on how I created my own Git server.

There are many open source projects like *GitTea* or *GitLab* to make hosting your own git projects effortless; however I wanted a much more simple (read: old school) setup. I ended up with something that uses many of the same projects that the *Linux Organization* uses to publish the *Linux Kernel*

The server (as of this writing) uses *Ubuntu's 20.04.1 LTS (Focal Fossa)* running on *Digital Ocean's* hardware (*referral-link*). I wholeheartedly support and recommend you chose a different setup. Diversity in people and in tech stack is always and will always be a great thing.

What lies below can be broken into 3 main topics:

1. *The Start* prepares a newly minted server for git hosting duties. Creating a new admin user, locking down the OpenSSH daemon, and installing fail2ban.
2. *Gitolite* installs and configures the server to allow us (and colleagues) to have more fine-grained control over who has access to `git push|fetch` on the server.
3. And *Cgit*, *fcgiwrap*, and *Nginx* to create a web-server to view our published projects.

In the end, you'll have a server much like *this one*.
Enjoy!

The Start

I often find security *"best practices"* are a lot like driving down the highway. Some people speeding past you are *"obviously"* just moments away from a major data breach, while the others you're passing are *"clearly"* so worried about the entire data-center burning down, they couldn't possibly get anything else done. Everyone thinks everyone else has lost their marbles.

So with that in mind, here are a few steps I took to secure my newly minted server. Please feel free to use only the *"best practices"* you deem appropriate for your mission.

Or just *skip to the "installing Gitolite"* part directly.

Admin User

For whatever reason, be it for security or protecting the server from my stupidity, one of the first things I do when creating a new server is add a new user for my general admin tasks.

Adding a new user is remarkably easy to do on a Ubuntu system:

```
$ adduser limb
```

You'll be prompted to answer a few questions, including creating a new UNIX password. This will be the password you'll need to `sudo -s` and gain `root` permissions, so make it a good one, or use tools like *Pass*, or *BitWarden* to help you remember.

Then give our new `limb` user `sudo` permissions:

```
$ usermod -G sudo limb
```

I've also largely eliminated all password based authentication when signing into servers, relying on open source smart cards like *Nitrokey* for authentication. If interested, this requires we setup `.ssh/authorized_keys` for our `limb` user:

Just replace `key` with your public ssh key:

```
$ mkdir /home/limb/.ssh
$ echo "key" > /home/limb/.ssh/authorized_keys
```

Next, set the `.ssh` directory's file permissions so the `ssh` daemon can read the files:

```
$ chown -R limb:limb /home/limb/.ssh
$ chmod 700 /home/limb/.ssh
$ chmod 644 /home/limb/.ssh/authorized_keys
```

> **Keep in Mind:**
> If the `authorized_keys` file or the `.ssh` directory's permissions are set too permissively (eg: `0777`), the SSH daemon will refuse to load the files.

If everything worked, after you restart the `ssh` daemon (`service sshd restart`) you will now be able to login as the administrator user:

```
$ ssh limb@host
```

OpenSSH

Git and Gitolite (*installed in the next sections*) will need us to keep port 22 open, allowing us to `git push` from anywhere on the internet. This open port will eventually attract *"a lot"* of attention from bots who endlessly scour the internet looking for vulnerable servers, mindlessly stuffing passwords, hoping one password will eventually let them in.

We can eliminate all worry about weak or compromised passwords by disabling all password based authentication, relying solely on *asymmetric cryptography*, or *"ssh keys"*. Just use your favorite text editor to open `/etc/ssh/sshd_config` and ensure these lines exist somewhere in it:

```
PubkeyAuthentication yes
PasswordAuthentication no
```

> **New to ssh keys?**
> Digital Ocean has a nice write-up on *how to get started* with ssh keys.

While we're here, a large majority *[1]* of these bots are interested in logging in as the `root` user. If you created a new admin account in *the previous section* and ensured you can login using your public key, you can also disable `root` logins entirely with this line in the config:

```
PermitRootLogin no
```

[1] Some simple *"bash-fu"* on my `/var/log/auth.log` shows ~93.58% of the roughly 15,000 login attempts since I started this server, tried to login as `pi`. Second place was the user `ol` (including legitimate logins) at ~1.82%.

If you uploaded your public key to your VPS provider, most of these changes should have already been configured for you. But in the off chance you had to make some changes, restart the `ssh` service to load the new config changes in:

```
$ service ssh restart
```

Uncomplicated FireWall

Depending on your VPS provider, they may also have a firewall system built into their admin panel allowing you to apply rules simply by adding tags to a server. However, I enjoy keeping all my firewall rules inside each box, if only for the same reason I *keep all my socks on the left hand drawer*, so everything stays organized and in the same place.

You can install `ufw` using the Advanced Packaging Tool:

```
$ apt install ufw
```

Right now, the only thing we have enabled is `ssh` which uses port 22. To allow port 22 through `ufw` just use the following command:

```
$ ufw allow ssh
```

and then turn the firewall on:

```
$ ufw enable
```

and **viola!** You have a firewall.

fail2ban

Even though we've turned off password based authentication *in a previous section*, we will still receive a significant amount of bots wasting our compute cycles trying to login. And while the likelihood of this being successful is *zero* when rounded to any order of magnitude, the bots will nevertheless continue to pilfer a non-zero amount of CPU if given the opportunity.

To stop the most brazen of these bots, tools like *Fail2Ban*, which creates temporary firewall rules to block IP address who repeatedly fail to authenticate with `ssh` are a great compromise between usefulness and annoyance.

The Advanced Packaging Tool can again help us install `fail2ban`:

```
$ apt install fail2ban
```

Once installed, the `ssh` "jail" will come pre-enabled for you. If you wish to make any changes, you will need to make a copy of the `fail2ban` config file:

```
$ cp /etc/fail2ban/jail.conf /etc/fail2ban/jail.local
```

Then add your changes to `jail.local` so they will persist after an upgrade.

`fail2ban` does a great job documenting what each option does in the config file. Some of the changes I made are:

- because I use `ufw` to manage my firewall, I changed `banaction = ufw`
- enabled `bantime.increment` to increase the duration of a ban based on how many times the IP address has been banned previously.
- enabled `bantime.rndtime` to *"randomize"* the length of a ban, preventing bots from knowing exactly when they can resume their assault.
- enabled `bantime.maxtime` so I won't need to unban IP addresses (if you're unfortunate enough to share an IP with a bot).
- lowered `bantime`, `findtime` and `maxretry` allowing me to issue small bans that increase in severity as the IP address continues to antagonize.

Once you're satisfied with your changes, start `fail2ban` using `systemd`:

```
$ systemctl enable fail2ban
$ systemctl start fail2ban
```

And **Done!**

> **Keep in Mind:**
> Any *"script-kiddie"* (read: jackass) that runs a pen-test script they found on Reddit from a large network (eg: college, Starbucks) could ban everyone on that network from your server. Make sure your ban rules have a way to forgive, unless you enjoy playing sys-admin.

Depending on how *"popular"* you are on the internet, you should start to see `NOTICE` lines in `/var/log/fail2ban.log` of misbehaving bots and the equivalent firewall rules in `ufw`:

```
$ cat /var/log/fail2ban.log | grep 'NOTICE' | tail -1
... [125553]: NOTICE  [sshd] Ban 156.155.159.161

$ ufw status
Status: active

To                         Action      From
--                         ------      ----
Anywhere                   REJECT      156.155.159.161
22/tcp                     ALLOW       Anywhere
22/tcp (v6)                ALLOW       Anywhere (v6)
```

Gitolite

Installing *Gitolite* is amazingly simple, there are no binaries to compile or daemons to monitor.

At its core, Gitolite is just a collection of *Perl* scripts that run after someone signs into the server using the `ssh` daemon *we configured in the previous sections*. Once installed, Gitolite will give us more fine-grained-control over who has `git push|fetch` permissions to each repository. I encourage you to checkout *Gitolite's amazing documentation* if only to see how capable Gitolite can be.

Step: 1 - Create The Git User

Before we install Gitolite, we'll need to create a new user for everyone to log into and to run Gitolite's Perl scripts. I typically use the username `git` for this, feel free to replace `git` with the username that you feel is more appropriate.

```
$ adduser --system --group --disabled-password --home /var/lib/git git
```

This creates a new system-user on the server called `git`. Because this is not a *"normal"* user, there will be no aging information in `/etc/shadow` which is convenient when nobody will be monitoring this account.

We also used the `--home` option to set the `$HOME` variable to `/var/lib/git`. This is where we will eventually put Gitolite's configuration files and our Git repositories. Feel free to adjust this to

where you prefer, I've seen many use `/home/git`

I included the `--disabled-password` to disable any password based access into our new user. In *the previous sections*, we've disabled all password based authentication into the server and Gitolite requires ssh keys for authentication, so disabling passwords for our user is a smart move.

Step: 2 - Install Gitolite

Because Gitolite is just a bunch of Perl scripts, I prefer to install Gitolite from *the source*. As we will see, installing Gitolite from source also has the benefit of making upgrades and adding custom patches in the future extremely easy.

This also means we'll need to install Gitolite's dependencies ourselves:

```
$ apt install perl git
```

When we (or a colleague) signs into the server, using the `git` user, we will automatically run Gitolite's Perl scripts, which means these scripts must be executable by our `git` user. So, to make managing file permissions easier, we'll use our `git` user for the rest of the installation process.

Log into our `git` user with the "substitute user" command: (assuming you're the `root` user)

```
$ su - git
```

Then clone Gitolite's source code into the `$HOME` directory: (this should be `/var/lib/git` unless *you changed it* when we setup the `git` user above)

```
$ git clone https://github.com/sitaramc/gitolite
```

> 🤓 **The More You Know:**
> If you want to use a particular version of Gitolite or want to add custom patches, `cd` into the `$HOME/gitolite` directory and `git checkout` the desired tag, branch, or commit. All changes will be picked up immediately the next time someone logs in.

Step: 3 - Setup Gitolite

To setup Gitolite on the server, we'll need to assign Gitolite an admin that will have full control over editing Gitolite's configuration repository. This will most likely be you.

Still as the `git` user, use *your favorite text editor* to create a new file with your desired username in the `$HOME` directory and copy your **public** ssh key into it. For example, my file would be called `bryanbrattlof.pub` and look like this:

```
$ cat $HOME/bryanbrattlof.pub
ssh-ed25519 AAAAC3NzaC1I ...
```

Then run Gitolite's sanity checks and setup script to finish the installation:

```
$ $HOME/gitolite setup -pk bryanbrattlof.pub
```

Färdig!

If everything went well, you should now be able to clone Gitolite's configuration repository:

```
$ git clone git@host:gitolite-admin
```

I recommend consulting *Gitolite's incredible documentation* to understand how to properly configure access and add hooks to all of your projects.

> 🤓 **Uh Oh:**
> If you're asked for a password when your try to clone `gitolite-admin` then something has gone wrong. This is usually a permission issue. Again, consult *Gitolite's superb documentation* for some of the more common troubleshooting advice.

Step: 4 - Add ~/.profile

While not technically needed for Gitolite to function properly, I find adding gitolite to our `git` user's `$PATH` is a great quality of life improvement on the rare days I need to play system administrator.

First, as our `git` user, create a new `$HOME/bin` directory:

```
$ mkdir -p $HOME/bin
```

Next, create the `$HOME/.profile` file and add the `$HOME/bin` directory to our `$PATH`

```
PATH=$HOME/bin:$PATH
```

The *born again shell* will automatically run `$HOME/.profile` when someone starts a new session.

Now, we can use Gitolite's `install` script to add a symbolic link inside our `$HOME/bin` folder:

```
$ gitolite/install -ln $HOME/bin
```

Done!

Logout and back in to the `git` user (or use `source $HOME/.profile`) to pick up the changes. If everything was done correctly, you won't need to type the full path to Gitolite anymore.

```
$ whereis gitolite
gitolite: /var/lib/git/bin/gitolite
```

Cgit

Cgit is a script (written in C) that uses the *Common Gateway Interface (CGI)* specification to give people a web view of our projects. Convenient when you don't have access to your terminal or just want to lookup (or showoff) some changes to a project.

It operates as a back-end (much like *PHP*) to a webserver (we'll install Nginx *in the next sections*) that will parse our repositories and return a web-page for our webserver to distribute.

To get an idea for what Cgit will look like, some of the more popular projects that use Cgit are the *Linux* and *FreeBSD* kernels, along with *Wireguard* and *Cgit* itself.

Step: 1 - Install Cgit

Just like with Gitolite, I prefer to install Cgit *from source* so I can add personal patches and quickly change what version is running on the server. This also means we'll need to install the dependencies ourselves:

```
$ apt install libc6 liblua5.1-0 zlib1g \
  python3-docutils python3-markdown python3-pygments
```

We'll also need the `build-essential` packages to install the `gcc` and `make` tools needed to compile Cgit after we've cloned the project:

```
$ apt install build-essential
```

Then, as the `root` user in the `/root` directory clone the Cgit project:

```
$ git clone https://git.zx2c4.com/cgit
```

Because Cgit uses parts of Git's source code, (included as a *submodule*) we'll need to use `git submodule` to download the remaining code from the Git project.

After you `cd` into `cgit`

```
$ git submodule init   # register the git submodule in .git/config
$ git submodule update  # clone/fetch and checkout correct git version
```

Step: 2 - Build Cgit

With a full copy of Cgit on the server, we can now create some patches to customize it for our use-case. We'll start with creating `cgit.conf` inside the `cgit` project we just cloned, to tell `make` where we want to install the Cgit binaries.

```
CGIT_SCRIPT_PATH = /var/www/html/cgit/cgi
CGIT_CONFIG = /var/www/html/cgit/cgitrc
CACHE_ROOT = /var/www/html/cgit/cache
prefix = /var/www/html/cgit
libdir = $(prefix)
filterdir = $(libdir)/filters
```

Because this is a version controlled project, we can commit our changes to save our work:

```
$ git add -f cgit.conf
$ git commit -m "installation path changes"
```

Some additional changes I made:

- Updated the `cgit.png` and `favicon.ico` icons
- Changed the `pygments` highlighting style to *"algol_nu"*
- Removed the *Data URI* icons from the tab menu
- Limited the `max-width` of readme pages to `95ch`
- Added `padding: 1em;` to code-blocks

When you're satisfied with your changes, use `make` to compile and install Cgit:

```
$ make && make install
```

If everything went well, when you execute Cgit from the terminal, a web-page should print out:

```
$ ./cgit
Content-Type: text/html; charset=UTF-8
Last-Modified: Tue, 12 Jan 2021 22:35:43 GMT
Expires: Tue, 12 Jan 2021 22:40:43 GMT

<!DOCTYPE html>
```

And **Done!**

We can further customize Cgit's behavior using the `/cgitrc` file located at `/var/www/html/cgit/cgitrc`. Feel free to check out *the man page* for a complete description of what every option does.

Some of the options I used:

- set `scan-path` to the location of our repositories `/var/lib/git/repositories`
- set `project-list` to the location of the `projects.list` file Gitolite creates, adding descriptions and categories to the list of repositories on Cgit's index page

FastCGI Wrapper

Cgit, which uses code from Git, was designed to let users run a command (eg: `git push`) then exit, allowing our computers to reclaim the used resources between each call. Nginx uses a faster protocol (*FastCGI*) which calls the same program multiple times without exiting.

However because Cgit was designed to exit after every run, it will never give back its used resources and will continue to take more, quickly exhausting all of the computer's available resources. This is why we need `fcgiwrap`

Thankfully this is easy to install. The Advanced Packaging Tool can, once again, help:

```
$ apt install fcgiwrap
```

Then just enable and start the service:

```
$ systemctl enable fcgiwrap
$ systemctl start fcgiwrap
```

And that's a **wrap!**

Nginx

With *Cgit* and the *FastCGI Wrapper* installed, we can now turn our attentions to Nginx, which can be installed using the Advanced Packaging Tool:

```
$ apt install nginx
```

Next, create a new configuration file in the `/etc/nginx/sites-enabled` directory, replacing `git.bryanbrattlof.com` with your domain. The minimum configuration file you'll need for Cgit to work will look something like this:

```
server {
    server_name  git.bryanbrattlof.com;

    listen [::]:80;
    listen 80;

    access_log  /var/log/nginx/cgit-access.log;
    error_log   /var/log/nginx/cgit-error.log;

    root /var/www/html/cgit/cgi;
    try_files $uri @cgit;

    location @cgit {
        include         fastcgi_params;
        fastcgi_param   SCRIPT_FILENAME /var/www/html/cgit/cgi/cgit.cgi;
        fastcgi_pass    unix:/run/fcgiwrap.socket;

        fastcgi_param   PATH_INFO    $uri;
        fastcgi_param   QUERY_STRING $args;
        fastcgi_param   HTTP_HOST    $server_name;
    }
}
```

Feel free to add more to this, I've added a custom *5xx page*, caching headers, as well as recommendations from *Mozilla Observatory*.

Once satisfied, start the Nginx service and open port 80 in the firewall:

```
$ service nginx start
$ ufw allow http
```

If something went wrong, or if you ever change the configuration file, you can use `nginx -t` to check the configuration for errors and `nginx -s reload` to restart the Nginx server.

```
$ nginx -t && nginx -s reload
```

Wrapping Up

By now we should have a working git server. However like most creative things *"90% done ... 90% left to go."* There is truly an endless supply of things you can and should add or configure to make your server more secure and accessible. If you're the type that likes to learn, then you'll likely find this as fun and rewarding experience as I did.

Some of the extra things I added:

- Installed *certbot* to install and manage a free SSL certificate from *Let's Encrypt*. This has largely been mandatory for any public server for around 5 years now
- Created a *Borg* based backup script with a Borg specific subscription to *rsync.net* to backup my projects. Useful when *that jackass we talked about* finds a 0-day
- Created a *Git Daemon* service to allow people to clone my projects using the `git://` protocol, if they prefer
- Placed a bunch of *Healthchecks* Pings in the scripts and service required to keep everything running. Fail2Ban, Borg Backup, Certbot, all will alert me when `cron` or `systemd` fail over

All of which should be given their own essay as they can be used in all your setups, not just in this fully open-source and free (as in libre) git server.