



GITLAB SETUP: INSTRUCTIONS FOR GETTING STARTED

Kathleen Hrenchir, CRMDA <katieh@ku.edu>
Paul Johnson, CRMDA <pauljohn@ku.edu>
Benjamin Kite, CRMDA <bakite@ku.edu>



Guide No: 34

Keywords: Git, GitLab

Mar. 26, 2018

See <https://crmda.ku.edu/guides> for updates.

GitLab is a Web program that makes it easier to manage teams of users and many Git repositories. GitLab makes it easy to create and manage repositories. It is especially convenient to regulate user access to particular projects. From the user perspective, GitLab is a place to find projects and request permission to participate in them. In other words, it is a “front page” for Git. The CRMDA Git guide, *Git it Together!* is available separately (CRMDA Guide 31).


This write-up describes how to establish a user account on the KU CRMDA GitLab server, which is available at <http://gitlab.crmda.ku.edu>. The advice here will apply to other GitLab servers as well.

1 Access your GitLab Account on the CRMDA Server

Browse to <http://gitlab.crmda.ku.edu>, select the **LDAP** login tab, and sign in using your KU online ID (e.g. a012b345) and password. You will not have access to projects until the administrator adds you as a member.

Inspect the GitLab Web interface. There will be tools that appear in the left and the top. In the top of the page, there is a navigation ribbon.



On the top left there is a project list, and on the top right there is a main user menu . That main user menu holds the all-important User Settings item that we discuss below.

In the main display, there may be some larger panels, one of which is “Explore Public Projects”. If you find that and open it, there should be a tab named “ALL”. That tab displays all of the open projects on the server. These are ones that are open to all participants. Some of them are available more broadly to users anywhere.

In the ALL list, we have some test repositories, one of which is “spr2018”. We are using that public repository for Git teamwork training. Feel free to clone it and practice your branch skills.

The security protocols will allow you to clone the repo using the https protocol, but in order to push changes back to the server, it will be necessary to use SSH security keys. In the next section, we describe the process of creating an SSH key and troubleshooting the (seemingly inevitable) hiccups that arise.



2 Configure an SSH Key

An SSH key pair is required to interact with the server. A key pair is composed of two files stored in the `~/ssh` folder of your user account. (Recall that “~/” means the user HOME folder, which is likely to be `/Users/your-name` on Macintosh, `C:\Users\your_name` on Windows, or `/home/your_name` on Unix/Linux.) The public key file has the suffix “.pub”, while the private key, which must never be shared to anyone, has no suffix. By default, these will be named “id_rsa.pub” and “id_rsa”. Below we suggest custom-naming your key, so the files might be “pj_gitlab_20180228.pub” and “pj_gitlab_20180228”. The “*.pub” is to be shared with servers, which then identify you by matching the private and public key parts.

To create your SSH key

1. Open a terminal. (In Windows, use Git BASH.)
2. Use the following command to create a new SSH key.

```
$ ssh-keygen -b 4096 -C "youremail@ku.edu" -f ~  
/.ssh/YourName_GitLab_YYYYMMDD
```

We recommend you create a custom-named key by including “-f `~/ssh/YourName_GitLab_YYYYMMDD`”.

Otherwise, your key files will be named `~/ssh/id_rsa.pub` and `~/ssh/id_rsa`. In the course of your work, it may be necessary to create several keys for several different purposes. Give each one a unique name. Choose any name you like.

3. Add a passphrase. Do not leave this field empty.

```
$ ssh-keygen -b 4096 -C "youremail@ku.edu" -f ~  
/.ssh/YourName_GitLab_YYYYMMDD  
Generating public/private rsa key pair.  
Enter passphrase (empty for no passphrase):
```

No text will appear in the terminal when you type your passphrase. You will be asked to retype your passphrase once.

4. The terminal will display a confirmation.


```
$ ssh-keygen -b 4096 -C "youremail@ku.edu" -f ~  
/.ssh/YourName_GitLab_YYYYMMDD  
Generating public/private rsa key pair.  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
5 Your identification has been saved in  
  /home/yourname/.ssh/YourName_GitLab_YYYYMMDD.  
Your public key has been saved in  
  /home/yourname/.ssh/YourName_GitLab_YYYYMMDD.  
The key fingerprint is:  
SHA256:FRImSmfxh/CXWcXKB7BIECeGaJDCFM0mkdG0eTDi4yc  
  youremail@ku.edu
```

```

10 The key's randomart image is:
+---[RSA 4096]----+
|+OX=..@+*.o..o. |
|+++**= X + *. . |
|. +oo.. = B. o |
|. . . + o . |
15 | E . S . |
| o |
| |
| |
20 +-----[SHA256]-----+

```

3 Upload the SSH key

1. From GitLab, open User Config  on the top right and access **Settings**.
2. Within Settings, select **SSH Keys**. This may be represented by an icon of a key.

There will be an open text box into which the key should be pasted. If you are careful, a simple copy/paste from the public key file into the server window will succeed. However, users often have errors in pasting due to line breaks. Because of some bad experiences, we are following the lead of the GitHub and GitLab documentation, which suggests a platform specific approach.

- (a) Copy the public key. This can be done from the terminal.

- i. Windows (Git BASH)

```
cat ~/.ssh/YourName_GitLab_YYYYMMDD.pub | clip
```

- ii. Mac

```
pbcopy < ~/.ssh/YourName_GitLab_YYYYMMDD.pub
```

- iii. Linux (using xclip package)

```
xclip -sel clip < ~/.ssh/YourName_GitLab_YYYYMMDD.pub
```

These will work only if your system has the required clipboard software. Linux does not have xclip installed by default. One can either install it, or try the old fashioned method. Open the public key file in an editor that does not impose linebreaks (e.g., Gedit, Emacs), then copy the key in its entirety.

- (b) Paste your key into the Key field of the GitLab form. How to paste? Any of the usual ways seem to work. A right-click -> paste works in any Web browser we have tried. Give your key a descriptive title, then select **Add Key**. The title is never used, except to remind yourself about which key is being pasted in. We suggest the custom name of your custom-named key for this, but anything will work.

4 Access GitLab projects

After the key is accepted, you are a fully functioning member of the community on <http://gitlab.crmda.ku.edu>. Our GitLab server defaults to protect the master branch from commits by all users except the owner or master. Users are required to create branches, which they are allowed to push, and for which they can file merge requests.

As a test of your setup, try to clone the project named “test/spr2018”. Open a terminal and run

```
$ git clone git@gitlab.crmda.ku.edu:test/spr2018.git
```

If you are lucky, and your operating system cooperates, all is well if you see:

```
Cloning into 'spr2018'...
X11 forwarding request failed on channel 0
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
5 remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

On the other hand, if you see this:

```
git@gitlab.crmda.ku.edu's password:
```

it means that the SSH key setup needs some correction.

5 One Last Gotcha: The ssh-agent

Here is the symptom. The Git server responds like this to any git command (clone, pull, push, or fetch):

```
git@gitlab.crmda.ku.edu's password:
```

Typing a password will not help. The SSH key setup needs fixing.

We'll guide you through some temporary adjustments and, if they work, we'll make them permanent.

1. Double-check that the SSH key was uploaded correctly. Sometimes users accidentally insert a line break in the SSH key. Open the key's .pub file in an editor. Notice it is one really long line. Check the server, make sure there are no accidental line breaks.
2. Supposing you did not err at 1, then it is likely the ssh-agent program is not running, or it does not know about the key you want to use. We can fix that.
3. Supposing you did not err at 1 or 2, then you have the “too many keys” problem. We can fix that.

This problem escaped our understanding for quite a long time because it arises intermittently on different computers. On a Linux system, the ssh-agent will generally be running. However, in Windows and Macintosh it is more likely to be a problem. On Linux under the XFCE4 desktop, users need to activate the option to “Launch Gnome services on startup” in the desktop settings (Sessions -> Advanced).

Checking on the ssh-agent problem

To read more about this, see “[Working with non-default SSH key pair paths](#)” on the GitLab documentation site. The same is discussed in the GitHub documentation, see “[Generating a new SSH key and adding it to the ssh-agent](#)”.

The following are temporary changes that you run *within a single terminal* session. They will not be remembered if you start a new terminal.

First, launch the ssh-agent program:

```
$ eval $(ssh-agent -s)
```

Second, tell SSH about your custom-named key.

```
$ ssh-add ~/.ssh/YourName_GitLab_YYYYMMDD
```

That works in Linux and Windows. On a Macintosh computer, the “-K” flag is needed:

```
$ ssh-add -K ~/.ssh/YourName_GitLab_YYYYMMDD
```

Type your SSH key’s passphrase when requested.

Here’s an example of a success on a Windows computer:

```
$ eval $(ssh-agent -s)
Agent pid 6276

$ ssh-add ~/.ssh/Paul_Johnson-windowsvm-20180318
5 Enter passphrase for
  /c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318:
Identity added:
  /c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318
  (/c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318)
```

Again, these changes are temporary. We will make them permanent, *if they work*.

Testing the ssh-agent

Run an ssh function that attempts to interact with the server as the user “git”. Try the following:

```
$ ssh -T git@gitlab.crmda.ku.edu
Welcome to GitLab, Paul E. Johnson!
```

That’s a success. We understand what went wrong and, in section “[Make a Permanent Setup for ssh-agent](#)” below, we will explain how to make a permanent solution.

On the other hand, the problem is not fixed if you again see:

```
$ ssh -T git@gitlab.crmda.ku.edu
git@gitlab.crmda.ku.edu's password:
```

Use Ctl-c to break out of that.

The “Too Many keys” Problem

If your key was uploaded correctly, and your ssh-agent is configured correctly, you may have too many keys. Do you have several keys? SSH uses an odd sequential key checking process. Your system offers your keys one-by-one, as if it is a drunk trying every key on the ring to open the front door. If your custom-named key is at the end of the list of keys, the server may disconnect before your key is tried. The server gives up asking for keys. The administrator determines how many tries are allowed.

Here’s how to fix it. Create a file named `config` in the `~/ssh` folder. Put a stanza in there like this

```
Host gitlab
  HostName gitlab.crmda.ku.edu
  IdentityFile ~/.ssh/YourName_GitLab_YYYYMMDD
  KeepAlive yes
  IdentitiesOnly yes
```

Save that file. On Linux and Macintosh, permissions on the `.ssh` folder should be 700 (no read-/write/execute for the group or the other users) and the SSH engine may reject your config file. To set permissions, run

```
$ chmod 700 config
```

On Windows, we notice the permissions are not adjustable from the shell, but the SSH engine does not seem to mind.

This config file is not a supplement for running ssh-agent and ssh-add as described above. It is step which is done in addition.

Suppose that did not fix it. Try this!

At some point, we have to say “sorry”. We recommended a custom-named key, but some problem we don’t understand still remains. Almost certainly, if you had created a key with the default name, this would have worked by now. Sorry again. Like the noble leader says in Animal House, “you ‘screwed’ up. You trusted us.”

There is good news. It is not necessary to go generate a new key named `id_rsa` and upload it again. Go into the `~/ssh` folder and copy the custom key parts to “`id_rsa`” and “`id_rsa.pub`”. Don’t alter the key files, just copy and rename them. After that, test git or “ssh -T” again. If the ssh connection succeeds, then we know the custom-named key is the problem.

After that, it still does not work? Troubleshooting

Get some diagnostic output. Run:

```
$ ssh -Tvv git@gitlab.crmda.ku.edu
```

That will generate many lines (perhaps 150 or 200). Let us see that. We can check whether your SSH folder, `~/ssh`, was found, whether your config file was found, and we can also see if keys were offered.

Make a Permanent Setup for ssh-agent

Holy cow! I don't want to re-launch ssh-agent every time I open a terminal. The change can be integrated into the login environment for the user account.

In our Linux systems, the desktop usually has some startup code that launches the ssh-agent in the background. That's in Gnome Startup Services in the session managers. It even remembers the passphrases between sessions. On the Macintosh systems after Sierra, the same is true (the ssh-agent is launched automatically and it remembers passphrases after the first use).

Adjustments in Windows can achieve the same benefits. In Windows, the user account environment can be corrected by inserting a file in the user home directory named `~/profile` (see [Auto-launching ssh-agent on Git for Windows](#)). We have tested this with success. We use the standard program, with **one major exception**. Our custom-named key must be named. Twice! Notice lines 17 and 19 in the following.

```
1 env=~/.ssh/agent.env
3 agent_load_env () { test -f "$env" && . "$env" >| /dev/null ; }
5 agent_start () {
6     (umask 077; ssh-agent >| "$env")
7     . "$env" >| /dev/null ; }
9 agent_load_env
11 # agent_run_state: 0=agent running w/ key;
12 # 1=agent w/o key; 2= agent not running
13 agent_run_state=$(ssh-add -l >| /dev/null 2>&1; echo $?)
15 if [ ! "$SSH_AUTH_SOCK" ] || [ $agent_run_state = 2 ]; then
16     agent_start
17     ssh-add ~/.ssh/YourName_GitLab_YYYYMMDD
18 elif [ "$SSH_AUTH_SOCK" ] && [ $agent_run_state = 1 ]; then
19     ssh-add ~/.ssh/YourName_GitLab_YYYYMMDD
20 fi
22 unset env
```

Start a fresh Git BASH terminal and the user passphrase for the custom key will be requested:

```
Enter passphrase for
  /c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318:
Identity added:
  /c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318
  (/c/Users/pauljohn32/.ssh/Paul_Johnson-windowsvm-20180318)
```

That setup will ask for the passphrase every time you log in and try to use Git BASH terminal. When new terminals are launched, the passphrase will not be requested again (until the user logs in to start a new session). As a security feature, perhaps it is best to leave it that way. However,

there may be a way to cause it to remember the passphrase between sessions. One might try [SSH Agent Helper](#), but we have not tried that.

For Macintosh systems, there are threads about the problem for Macintosh users, one of which we started:

1. [Macintosh Git SSH key setup](#)
2. [Mac OS X 1012: ssh-agent todes not automatically load](#)

We'll pin down the Mac details, but we are certain the problem can be corrected by adjusting the user account's environment.

Summary of this sub-section

If the server asks for a password for a user named "git", it means that there is a configuration error in your computer. Was the public key file uploaded without alteration? Perhaps the ssh-agent program is not running. Did you ssh-add the custom key? If you have too many other keys in `~/ssh` folder, you need to create an SSH config file to designate which key is used with the server. In our experience, over 5 years, this is an exhaustive list of the problems and we believe they can all be overcome.