Using `ssh` Authentication and `git`

Brian C. Ladd

23 January 2012

Note: Your homework directories should *not* be subtrees of the class source code. You should create new directories for your assignments that are *not* under the directory where **`cis405-src.git`** was cloned. [If you don't understand this, don't worry. Page 6 explains it. It is important enough to call your attention to before you read the document.]

1 SSH Keys

`ssh` (secure shell) uses public key encryption for user authentication and secure communication in an unsecured network.[1] Public key encryption requires *two* different keys: the *public* key which you share with anyone from whom you wish to receive information and the *private* key that is used to decrypt the information sent using your public key. Each **`ssh`** key is actually a *keypair*.

Public key encryption is "backward" from shared-secret encryption: using a shared secret you send a key to those *to whom* you wish to send messages; using public key encryption you share the public key with those *from whom* you wish to receive messages. The public key can also be used by remote systems to authenticate who you are (by checking that you have access to the private half of the key).

On Linux, the keys are generated using the **`ssh-keygen`** program. By default they are stored in the **`~/.ssh`** directory. Current versions of the **`ssh`** programs generate keys using the RSA algorithm by default.

[1] Want to drive Dr. Ladd to apoplexy? Use "insecure" when talking about an unsecure network. An *insecure* network needs affirmation to improve its self-image; an *unsecure* network needs encryption to keep prying eyes from reading things they should not.

Generating SSH Key Pairs

An example command-line session to generate a key pair looks something like this (numbers in the left-hand margin are reference labels; the left-hand edge of the prompt is the left-hand edge of the shell):

```
    laddbc@cs:~$ ssh-keygen
      Generating public/private rsa key pair.
01    Enter file in which to save the key (/home/laddbc/.ssh/id_rsa):
02    Enter passphrase (empty for no passphrase):
      Enter same passphrase again:
      Your identification has been saved in /home/laddbc/.ssh/id_rsa.
      Your public key has been saved in /home/laddbc/.ssh/id_rsa.pub.
      The key fingerprint is:
03    3b:af:85:f8:d3:f6:fd:6f:7d:92:3f:d2:7b:2f:28:b6 laddbc@cs
      The key's randomart image is:
04    +--[ RSA 2048]----+
      |                 |
      |                 |
      |                 |
      |                 |
      |        S        |
      |       . o       |
      |      . +..  ....|
      |       ..+= .o+o=|
      |        o=E+. oB%|
      +-----------------+

    laddbc@cs:~$
```

Explanation

`01` – The default location for the files is the **`~/.ssh`** folder; the default names for the files are **`id_rsa`** and **`id_rsa.pub`**. Press enter to accept the default (or type an alternative name/location if you want). An existing keypair will not be overwritten without a prompt.

`ssh` (and related tools) is easiest to use when the keypair is in the default location; otherwise an identity parameter is required specifying where the keypair resides. See **`ssh --help`** or

man ssh for more on command-line parameters.

02 – ssh-keygen is asking for a password/phrase to secure the private half of your key. You will have to provide this password *every time* you use **ssh** (or **git** which uses the **ssh** protocol).

Just hitting enter means that there is no passphrase and you will not be prompted for one when using **ssh**; this can be very convenient at the cost of some security. Anyone who can access your account can authenticate as you to any remote system using your keypair without guessing a second password. I don't use a passphrase; I will not be able to debug your **ssh** problems without you sitting there if you use a passphrase.

03 – This is a fingerprint for the key, a way of confirming what version of the key another person has without having to transmit the entire key. It also simplifies advertising your public key version to make the system less susceptible to man-in-the-middle attacks.

04 – Similarly, this is a bit pattern encoded in ASCII art that represents your key. You could include it in an e-mail signature to permit correspondents to confirm they are using the current version of the key.

Checking the Results of ssh-keygen

You can see the generated files in the **~/.ssh** directory:

```
laddbc@cs:~$ cd .ssh
laddbc@cs:~/.ssh$ ls -1
  authorized_keys
  id_rsa
  id_rsa.pub
  known_hosts
laddbc@cs:~/.ssh$
```

(The -1 ("dash one") tells ls to display one file name per line.) The two files we're interested in are id_rsa (the **private key**) and id_rsa.pub (the **public key**). **authorized_keys** is a file containing public keys for users who can remotely authenticate to my account without providing a password (you may not have such a file at all) and **known_hosts** is a list of key fingerprints for different remote machines; when a machine's OS or hardware changes, the fingerprint changes and I can be wary if, for some reason, I suspect that someone evil has subverted or replaced the machine.

Sharing Your Public Key

Copy Your Public Key

Copy the **id_rsa.pub** file (in the **.ssh** folder) to a file named **<login>.pub** where **<login>** is your *login name on the lab Linux machines*:

```
laddbc@cs:~/.ssh$ cd ..
laddbc@cs:~$ cp .ssh/id_rsa.pub laddbc.pub
```

E-mail the Copied File to Dr. Ladd

Now send an e-mail, attaching a copy of **laddbc.pub** (your login name replaces mine), to `laddbc@potsdam.edu` so that you can authenticate to the **git** server.

How? You can run **firefox** on the lab machines and use any Webmail account you have (`bearmail.potsdam.edu` anyone?) and attach the file. Or you could copy the file to a USB thumbdrive and mail it from you own computer. Or use **scp** to copy the file to another computer from which to mail it.

Note: You *must* rename the file you send to Dr. Ladd: receiving thirty files named **id_rsa.pub** makes saving them error-prone. The subject of your message should be the CIS courses you are taking with Dr. Ladd: this is so that your name is properly added to the right lists (again, handling the lists for multiple classes with each student taking a different subset of courses...you can see where this might be difficult).

You created an **ssh** keypair, copied, and e-mailed the *public* half of the key so that you can use **git**, a version control system. The departmental **git** server is secured using **ssh** authentication; your key must be in the list of authorized users for the server. The next section describes how to use **git** once you are properly authenticated.

2 Using git

What Is git?

git (the name of the program is written in lowercase) is a *version control system*. Consider it a large-grained undo facility along with a way that I can look at how you work. Note that some courses are about process as much as substance [2] so being able to see *how* you program is as important as seeing *what* you finally program.

[2] *i.e.* software engineering

A version control system is a database of snapshots of the contents of files. With such a database you can reset your project (or any subset of the files in your project) to *any* recorded snapshot. You must **actively** add snapshots to the repository: this should be done regularly (on the order of halves of hours rather than halves of days), whenever you complete a feature.

`git` Users

With your public key, named **`<login>.pub`**, I create a user, known to `git@cs-devel.potsdam.edu`, named **`<login>`**. That user can create homework projects in course space as well as download the course sample code. You are registered in each course you take with me when I get the public key.

How Is `git` Used in Class?

`git` repositories contain all sample source code for classes and are used by students to turn in all programming assignments.

Retrieving Sample Code

Class sample code for a class, say CIS405, is in a repository named for the class with **`-src`** appended: **`cis405-src`**. Take note of the capitalization. The name of the class repository is part of the repository's URL as in

```
git@cs-devel.potsdam.edu:cis405-src
```

The first part of the name is the user/server pair where a **`git`** server is running. After the colon is the path name of the repository. Class repos reside in the base directory of our **`git`** server; your homework and team repos reside in class named folders (as explained below). To clone any **`git`** repository to which you have read access, use the **`git clone`** command:

```
01  laddbc@cs:~/tmp$ git clone git@cs-devel.potsdam.edu:cis405-src
02    Initialized empty Git repository in
        /home/laddbc/tmp/cis405-src/.git/
      remote: Counting objects: 37, done.
      remote: Compressing objects: 100% (34/34), done.
      remote: Total 37 (delta 7), reused 0 (delta 0)
      Receiving objects: 100% (37/37), 28.56 KiB, done.
      Resolving deltas: 100% (7/7), done.
    laddbc@cs:~/tmp$
```

01 – **git** has many subcommands; the word after **git** determines what **git** does. **git clone** takes a **git** URL to a repository as a parameter and copies (clones) the entire repository history to the local machine. An optional second parameter names the local directory to clone to; the name of the repository is used for the directory name if the parameter is omitted.

If the intended directory exists and is non-empty, the **clone** command fails with an error message; **git** is typically careful about clobbering your work.

02 – **git** initializes an empty history database in the local clone directory. The **.git** directory is hidden (Linux **ls** command ignores "dot-files" by default) in the root of a **git** directory tree.

The empty history database is then populated from the source repository on the server. Finally **git clone** copies the most recent version of all files into the clone directory (the working tree).

git log displays the complete history of a **git** repository.

```
laddbc@cs:~/tmp$ cd cis405-src/
laddbc@cs:~/tmp/cis405-src$ git log
  commit 3ed6004b1426e8715dd3d7af79ab599145c16a84
  Author: Brian C. Ladd <laddbc@potsdam.edu>
  Date:   Tue Aug 16 11:27:53 2011 -0400

      Added the sierpinski triangle program.

      Built separately from that in the book's source so it is not
      strongly related.

  commit 0774bddaacabba69b210ac4f901b64afd0c3d0d1
  Author: Brian C. Ladd <laddbc@potsdam.edu>
  Date:   Tue Aug 16 11:26:57 2011 -0400

      GL00 - testing open GL

      Added the GL00 project/source files.

...
laddbc@cs:~/tmp/cis405-src$
```

Updating Sample Code

It is possible to **clone** the class database as many times as you want. Since you have read access to the repository, any machine with **git** installed that can reach **cs-devel.potsdam.edu** can clone the repository.

Re-cloning the repository onto the machine we cloned it onto above is annoying and a waste of time: **git** won't overwrite existing directories with a **clone** so the first clone must be removed or renamed; all of the material that was already downloaded is *on* the local machine – there is no reason to copy it across the wire again.

git clone associates the original, cloned repository with a short name, called a remote name, of **origin**. The short name for the associated URL makes it easier to refer to whence the repository came.

```
01  laddbc@cs:~/tmp/cis405-src$ git remote
    origin
02  laddbc@cs:~/tmp/cis405-src$ git remote show origin
    * remote origin
      Fetch URL: git@cs-devel.potsdam.edu:cis405-src.git
      Push  URL: git@cs-devel.potsdam.edu:cis405-src.git
      HEAD branch: master
      Remote branch:
        master tracked
      Local branch configured for 'git pull':
        master merges with remote master
      Local ref configured for 'git push':
        master pushes to master (up to date)
    laddbc@cs:~/tmp/cis405-src$
```

01 **git remote** with no parameters lists all the remote sites defined in this repository. The only one here is **origin**.

02 **git remote show** shows the current status of the named remote repository after connecting to the remote site and comparing to the local repository.

The report here shows the URL for the remote repository, lists the branches being used locally (**master**) and remotely (**remote master**). It will also list how many extra commits are on the local or remote repository if they are out of sync.

You can just type **git pull**, short for **git pull origin master**, to pull (fetch from the remote and copy over the working directory) any updates from the remote site.

Because **git** will not overwrite changes in the local working tree, you never want to make changes directly in the source tree you download for any class. If you do you might have problems pulling updated source code.

In particular: **Never** put assignment directories anywhere inside the class source code directory tree. If you're extending class code create a *new* directory and *copy* the appropriate code and sub-directories from the class source into your new assignment folder.

Submitting Programs

Submitting a program requires several steps:

1. Create a **git** repository for the project
   Done in the root directory of the source tree for the project; all files you intend to turn in should be in the subtree rooted at that folder. This is done on your personal account/computer.

2. Add all required files to **git**
   Stages the file(s) to be added (or updated in) the local repository. This is done on your personal account/computer in the source subtree.

3. Commit the changes staged in the previous step into the local **git** repository
   Moves the added changes from staging to the local repository. No unstaged or previously committed changes are affected. This is done in the source subtree on your personal account/computer.

4. Push the modified repository into the assignment repository on the central server.
   Places a copy of the repository where I can clone (or pull) it. You can repeat steps 2–3 as many times as necessary, updating the local repository. You can perform step 4 once at the end of the assignment or, if you want me to look at your code, do it earlier and then push you changes over it to turn the final changes in. This is done on your personal account/computer while connecting to the **git** server.

(For seasoned **git** users here at Potsdam, the process changed a little bit in Fall 2011. The naming conventions for assignments has changed (the names of repositories are specified in the assignment) and the new `push –all` syntax means you don't have to set up a remote connection to the server. Using the new name, however, you can set up a standard remote connection if that is more convenient.)

1. Change directory to the root of the project and initialize a git repository. **Note:** Your homework directories should *not* be inside the directory tree containing class source code. **Read that last sentence again! Go read the first sentence in this document. Now you understand!**

```
laddbc@cs:~/tmp/$ mkdir triangles
laddbc@cs:~/tmp/$ cd triangles
laddbc@cs:~/tmp/triangles/$ git init
  Initialized empty Git repository in /home/laddbc/tmp/triangles/.gi
laddbc@cs:~/tmp/triangles/$
```

This creates the local database (the tree rooted at **`./.git`** contains the database). Remember, the assignment directory can be named anything you want *as long as it is **not*** anywhere in the course source code directory tree.

The local database contains no files (note the word "`empty`" in the **`git`** response line. The initialization of the repository happens once per repository; we will use a separate repository for each assignment so you will initialize a repository once per assignment. (In general: if one project extends another, then you will extend the existing repository, possibly `push`ing the repository to a different location on the server.)

The assignment folder, the root of the assignment's directory tree, **cannot** be anywhere within the directory tree of the class's source code. You can name the assignment folder whatever you want.

2. Add the files belonging to the assignment to **`git`**. **`git`** stages them for committing to the local database. That means you can easily remove them if you need to.

```
laddbc@cs:~/tmp/triangles/$ git add draw_triangle.h
laddbc@cs:~/tmp/triangles/$ git add draw_triangle.cpp
laddbc@cs:~/tmp/triangles$ git status
  # On branch master
  #
  # Initial commit
  #
  # Changes to be committed:
  #   (use "git rm --cached <file>..." to unstage)
  #
  #        new file:   draw_triangle.cpp
  #        new file:   draw_triangle.h
  #
laddbc@cs:~/tmp/triangles/$
```

`git status` provides a view of the *working tree*, the *staging area*, and the local *repository*. The working tree is your copy of all of the files. What is currently in the assignment

source code tree is compared to the repository and any differences that are not staged are reported; there are none in the above status. The staging area is the set of changes that have been added to **git** but are not yet committed. The files in the working tree that are new or changed but have been added to the staging area; two new files are listed above. The repository is the database (in **./.git**). Any files that are the same as the most recent addition to the database are *not* listed in the status. In **git**, no news is good news.

3. The staged commits need to be committed to the local database.

```
laddbc@cs:~/tmp/triangles/$ git commit
```

git commit requires a commit message. A commit message serves as a label for the changes to the code.

In order to display compactly, the message is formatted as a single line of text (the short message, the *thesis statement*), a single blank line, and then as many lines (using blanks to break paragraphs) as necessary to explain the reason for the commit.

A commit should be a coherent, unified whole, summarizable in a single sentence. Think of a commit as a paragraph and the first line of the commit message as the thesis sentence. If you have problems coming up with a thesis sentence, consider whether your commit is either incomplete or is actually composed of multiple commits. Adjust it as necessary.

git brings up an editor for you to write a commit message.[3] The bottom of the edit buffer contains the **git status** results printed above. Any lines in the message buffer that begin with a **#** are ignored when **git** records the message.

Save the file from the editor (after filling it in with non-comment lines) and the program announces the successful commit:

(continued from above...)

```
        [master (root-commit) 5dd3389] Initial commit
 Committer: Brian Ladd <laddbc@cs-devel.(none)>
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 draw_triangle.cpp
 create mode 100644 draw_triangle.h
laddbc@cs:~/tmp/triangles/$
```

Notice the first line of **git**'s response: it has the short message that I typed in the editor ("`Initial commit`"), the branch that **git** is currently using ("`master`"), and the identifier

[3] *Which* editor can be configured. Google "git default editor" and you should have plenty of tutorials to choose from.

for the commit (`5dd3389`). The identifier for the commit is from the secure hash of the state of the repository after the changes are committed. You don't need to worry about how it is calculated but can assume there are seldom collisions between states of the repository.

The hash (visible with the **`git log`** and some other commands) is 40 hex digits long. The first several (or last several) digits differentiate commits from one another.

4. Assuming **`triangles`** is CIS405 program 1, turning it in would mean pushing it to the repository **`cis405/laddbc/p001`** (the leading zeros are required). **`git push`** can take the full URL of the new repository. (Note that "\" means continue line.)

```
laddbc@cs:~/tmp/triangles/$ git push \
    --all git@cs-devel.potsdam.edu:cis405/laddbc/p001
  Counting objects: 4, done.
  Delta compression using up to 2 threads.
  Compressing objects: 100% (2/2), done.
  Writing objects: 100% (4/4), 336 bytes, done.
  Total 4 (delta 0), reused 0 (delta 0)
  To git@cs-devel.potsdam.edu:cis405/laddbc/p001
   * [new branch]      master -> master
laddbc@cs:~/tmp/triangles/$
```

The **`--all`** flag tells **`git`** to push all references to branches and tags along with the actual database.

The **`git ls-remote`** command checks the remote version, showing the commit ID of each.

```
laddbc@cs:~/tmp/triangles/$ git ls-remote \
    git@cs-devel.potsdam.edu:cis405/laddbc/p001
  5dd33896dad3a29449d89b627edd27d6f51561b4      HEAD
  5dd33896dad3a29449d89b627edd27d6f51561b4      refs/heads/master
laddbc@cs:~/tmp/triangles/$
```

The repository URL is passed to **`ls-remote`** and the result is the commit ID of the **`HEAD`** (pretty much the "current" pointer in **`git`**) and the commit ID of any branches there (**`master`** is the default starting branch; we will not make much use of branches in any classes). Notice that both match the commit ID reported by **`git`** when the triangles program was committed. This verifies that my changes were pushed up to the server.

Restoring Your Own Repositories

Catastrophe has happened: you clobbered your class assignments directories. And you didn't have anything like a current backup.[4] How can you get **p001** back again? The answer is **git clone**:

```
laddbc@cs:~/restored_cis405$ git clone \
    git@cs-devel.potsdam.edu:cis405/laddbc/p001.git
  Initialized empty Git repository in
    /home/laddbc/restored_cis405/p001/.git/
  remote: Counting objects: 37, done.
  remote: Compressing objects: 100% (34/34), done.
  remote: Total 37 (delta 7), reused 0 (delta 0)
  Receiving objects: 100% (37/37), 28.56 KiB, done.
  Resolving deltas: 100% (7/7), done.
laddbc@cs:~/restored_cis405$
```

Now a complete copy of the repository is in the p001 directory and the **HEAD** commit is checked out (the files are all in the state of the last commit).

What Repositories Can I Access?

To find the names of repositories on **cs-devel.potsdam.edu** that you can access (or ones you can create), you can send the special **info** command to **git@cs-devel.potsdam.edu** using **ssh**:

```
laddbc@cs:~$ ssh git@cs-devel.potsdam.edu info
   hello laddbc, the gitolite version here is v2.0.3-3-gbfbd887
   the gitolite config gives you the following access:
        R W       cis356-src
        R W C     cis356/laddbc/p\d\d\d
        R W       cis380-src
        R W       cis405-src
        R W C     cis405/laddbc/p\d\d\d
        R W       cis405/laddbc/p001
laddbc@cs:~$
```

[4] As a computer professional (or one in training), you should know how to backup your work. You should have a system for backing up your work. Heck, you should *automate* backing up your work so that it happens (and is verified) without human intervention and the copies are moved off the hardware with the original. Failure to heed this warning implicitly gives Dr. Ladd the right to point *and* laugh.

The returned value shows that I authenticated (using **ssh**) as **laddbc**. I have read and write access (**R** and **W**) access to the repositories for several classes that I teach. You will only have read access to classes you are in.

The directory lines ending in regular expressions, the **p\d\d\d** lines, are "wildcard repositories". The **\d\d\d** is a regular expression for three decimal digits. So the **cis405** wildcard repository would match any of the following

```
cis405/laddbc/p002
cis405/laddbc/p123
cis405/laddbc/p999
```

When you run **info**, the name **laddbc** will be replaced by the name by which **gitolite** knows you (the name of the public key file sent). You can see that I pushed the **p001** repository so it is shown specifically as well as the form of the wildcard repositories that I can create.

There are no repositories that I can share with others on the system. This is on purpose: you cannot give others permission to read the repositories you are using to turn in code in class. When we begin working in teams you will use team repositories.

Team Repositories

One prime reason to use **git** is when code is shared among multiple programmers. This can be done out of a central repository (as we will in this class) or by having a "central" user and issuing pull requests from personal repositories (more typical in github style open source setups).

When you run the **info** command, you may see a line (in the wildcard repos near the top of the list) like the following:

```
...
 R W C  cis405/team..*
...
```

This means that in addition to the **program...** repos, you can create your own team repository. The **..*** at the end of the line means that you must have at least one extra character (you cannot call the team repository **cis405/team.git**) and that you can have as many as you want.

You "create" a team repository just like a program repository: create a local repo and push it to the appropriate URL. So, if I create a local directory, **teamAlpha**, create at least a **README** file in it, and commit the change, I can push the repository up to the **cs** box:

```
laddbc@cs:~/tmp/teamAlpha/$ git push \
    --all git@cs-devel.potsdam.edu:cis405/teamAlpha
```

```
      Counting objects: 4, done.
      Delta compression using up to 2 threads.
      Compressing objects: 100% (2/2), done.
      Writing objects: 100% (4/4), 336 bytes, done.
      Total 4 (delta 0), reused 0 (delta 0)
      To git@cs-devel.potsdam.edu:cis405/teamAlpha
       * [new branch]      master -> master
    laddbc@cs:~/tmp/teamAlpha/$
```

The only problem here is that only I have access to the repository. Assuming Drs. Fossom (**fossumtv**) and Haller (**hallersm**) were my team members I would like to give both of them read/write access to the repository. The one assumption made here is that they are both already members of the class **cis405** with registered ssh keys.

Managing Team-member Access The creator of the repository can use an administrative tool similar to the **info** tool described above for finding the list of repositories available to you. The command (run through **ssh**) is **perms**. The command lists its own usage when called with the **-h** parameter:

```
laddbc@cs:~/$ ssh git@cs-devel.potsdam.edu perms -h

Usage:  ssh git@host perms -l <repo>
        ssh git@host perms <repo> - <rolename> <username>
        ssh git@host perms <repo> + <rolename> <username>

List or set permissions for user-created ("wild") repo. The first
usage shown will list the current contents of the permissions file. The
other two will change permissions, adding or removing a user from a role.

Examples:
    ssh git@host perms foo + READERS user1
    ssh git@host perms foo + READERS user2
    ssh git@host perms foo + READERS user3

----
There is also a batch mode useful for scripting and bulk loading.
Do not combine this with the +/- mode above.  This mode also accepts
```

```
an optional ”-c” flag to create the repo if it does not already exist
(assuming $GL_USER has permissions to create it).

Examples:
    cat copy-of-backed-up-gl-perms | ssh git@host perms <repo>
    cat copy-of-backed-up-gl-perms | ssh git@host perms -c <repo>
```

To add read and write access for my partners, I can just use the + syntax (note: there are four lines in the following; the “\” at the end of a line means “line continues”).

```
laddbc@cs:~/$ ssh git@cs-devel.potsdam.edu perms \
                  cis405/teamAlpha + READERS fossumtv

laddbc@cs:~/$ ssh git@cs-devel.potsdam.edu perms \
                  cis405/teamAlpha + WRITERS fossumtv

laddbc@cs:~/$ ssh git@cs-devel.potsdam.edu perms \
                  cis405/teamAlpha + READERS hallersm

laddbc@cs:~/$ ssh git@cs-devel.potsdam.edu perms \
                  cis405teamAlpha + WRITERS hallersm
```

There is no need to modify my own access (as the creator of the repo), nor is there any need to grant access to the instructor of the course (by other security rules for **`git`** on **`cs`**, the instructor has read access to a team repository when it is created).

After giving access, my teammates should clone the repository so that they can modify it as they see fit. I may want to clone it (or set the remote origin) for ease of pulling and pushing changes.

This document, by definition. only covered the most basic uses of **`git`** here at SUNY Potsdam. You should invest some time in learning a bit more, in particular about how to reset a file or a set of files when you want to throw your changes away and how to merge repositories when there is a merge conflict in pulling down changes.