Even in spite of SSH's port forwarding capabilities, we still sometimes need to use graphical applications. If you have a fast connection or a simple GUI, passing the -Y flag to SSH could be enough to allow you run the application on your local desktop. Unfortunately, this often is a very poor user experience, and it does not work well with screen (a GUI application started in a screen session dies when you detach from the screen session).

The time-tested Unix solution to this problem is VNC. This is effectively a combination of screen and a graphical environment

**Tips for remote Unix work:  
SSH, screen, and VNC**

**I am not where the work is**

If you are anything like me, you have programs running on all kinds of different servers. You probably have a [github](http://github.com) account, a free [Heroku instance](http://www.heroku.com/pricing#1-0), a work desktop, a couple website instances, and maybe even a home server. The best part is that using common Unix tools, you can connect to all of them from one place.

In this post, I will review some of the more interesting aspects of my workflow, covering the usage of SSH, screen, and VNC, including a guide for getting started with VNC. I'll give some quick start information and quickly progress to advanced topics (like SSH pipes and auto-session-creation) that even experienced Unix users may not be aware of.

**SSH to rule them all**

By now you've almost certainly used SSH. It's the easiest way to login to a remote machine and get instant command line access. It's as easy as ssh user@example.com. You type in your password, and you're in! But you might not know that it can be even easier (and more secure) than that.

**Logging in via SSH without a password**

We have only recently seen websites start to offer solutions for logging in without a password. SSH has provided a secure mechanism for this (based on [public-key cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography)) since its inception. It's pretty easy to setup once you know how it works.

**1. Generate a public-private key pair**

If you haven't already, run ssh-keygen on your laptop, or whatever computer you will be doing your work from. You can just continue pressing Enter to accept the defaults, and you can leave the password blank (if you secure your laptop with encryption, a locking screensaver, and a [strong password](http://passphra.se), your SSH key doesn't require a password). This will generate a public key at ~/.ssh/id\_rsa.pub and a private key at ~/.ssh/id\_rsa. **The private key should never leave your computer.**

**2. Copy the public key to each computer you connect to**

For each computer that you connect to, run the following command (edited thanks to a suggestion by [rmc](http://news.ycombinator.com/item?id=3364149) in the [Hacker News comment thread](http://news.ycombinator.com/item?id=3364025)):

ssh-copy-id user@example.com

*Note that you can specify -p PORT or any other SSH arguments before the user@example.com portion of the above command*

This should be the last time you ever have to type your login password when connecting to the remote server. From now on, when you SSH to the remote server, its sshd service will encrypt some data using the public key that you appended to authorized\_keys, and your local machine will be able to decode that challenge with your private key.

**3. There is no step 3**

It's that easy! Don't you wish you had set this up a long time ago?

**SSH and pipes**

If you take a look at the ssh-copy-id script, you'll see a line that roughly translates to:

cat ~/.ssh/id\_rsa.pub | ssh user@example.com "umask 077; test -d ~/.ssh || mkdir ~/.ssh ; cat >> ~/.ssh/authorized\_keys"

When you ran ssh-copy-id above, here's what that line did:

1. The contents of ~/.ssh/id\_rsa.pub were piped into the SSH command.
2. SSH encrypted that data and sent it across the network to your remote machine.
3. Everything in double quotes after the host is a single argument to ssh; this specified that instead of giving you an interactive login, you instead wanted to run a command.
4. The first portion of that command (umask 077; test -d ~/.ssh || mkdir ~/.ssh ;) created a .ssh directory *on the remote machine* if it did not already exist, ensuring that it had the proper permissions.
5. The second portion (cat >> ~/.ssh/authorized\_keys) received the standard input via the SSH tunnel and appended it to the authorized\_keys file *on the remote machine*.

This avoids the need to use SCP and login multiple times. SSH can do it all! Here are some more examples to show you some of the neat things you can do with SSH pipe functionality.

**Send the files at ~/src/ to example.com:~/src/ without rsync or scp**

cd && tar czv src | ssh example.com 'tar xz'

**Copy the remote website at example.com:public\_html/example.com to ~/backup/example.com**

mkdir -p ~/backup/

cd !$

ssh example.com 'cd public\_html && tar cz example.com' | tar xzv

**See if httpd is running on example.com**

ssh example.com 'ps ax | grep [h]ttpd'

**Other SSH tunnels**

If piped data were the only thing that could be securely tunneled over SSH connections, that would still be useful. But SSH can also make remote ports seem local. Let's say that you're logged into example.com, and you're editing a remote website that you'd like to test on port 8000. But you don't want just anyone to be able to connect to example.com:8000, and besides, your firewall won't allow it. What if you could get a connection to example.com, localhost:8000, but from your local computer and browser? Well, you can!

**Create an SSH tunnel**

ssh -NT -L 9000:localhost:8000 example.com

Using the -L flag, you can tell SSH to listen on a local port (9000), and to reroute all data sent and received on that port to example.com:8000. To any process listening on example.com:8000, it will look like it's talking to a local process (and it is; an SSH process). So open a terminal and run the above command, and then fire up your browser locally and browse to localhost:9000. You will be whisked away to example.com:8000 as if you were working on it locally!

Let me clarify the argument to -L a bit more. The bit before the colon is the port on your local machine that you will connect to in order to be tunneled to the remote port. The part after the second colon is the port on the remote machine. The localhost bit is the remote machine you will be connected to, *from the perspective of example.com*. When you realize the ramifications of this, it becomes even more exciting! Perhaps you have a work computer to which you have SSH access, and you have a company intranet site at 192.168.10.10. Obviously, you can't reach this from the outside. Using an SSH tunnel, however, you can!

ssh -NT -L 8080:192.168.10.10:80 work-account@work-computer.com

Now browse to localhost:8080 from your local machine, and smile as you can access your company intranet from home with your laptop's browser, just as if you were on your work computer.

**But my connection sucks, or, GNU screen**

Have you ever started a long-running command, checked in on it periodically for a couple hours, and then watched horrified as your connection dropped and all the work was lost? Don't let it happen again. Install [GNU screen](http://www.gnu.org/s/screen/) on your remote machine, and when you reconnect you can resume your work right where you left off (it may have even completed while you were away).

Now, instead of launching right into your work when you connect to your remote machine, first start up a screen session by running screen. From now on, all the work you are doing is going on inside screen. If your connection drops, you will be detached from the screen session, but it will continue running on the remote machine. You can reattach to it when you log back in by running screen -r. If you want to manually detach from the session but leave it running, type Ctrl-a d from within the screen session.

**Using screen**

Screen is a complex program, and going into everything it can do would be a series of blog posts. Instead, check out this great [screen quick reference guide](http://aperiodic.net/screen/quick_reference). Some of screen's more notable features are its ability to allow multiple terminal buffers in a single screen session and its scrollback buffer.

**What happened to Control-a??**

Screen intercepts Control-a to enable some pretty cool functionality. Unfortunately, you may be used to using Control-a for readline navigation. You can now do this by pressing Ctrl-a, a. Alternatively, you can remap it by invoking screen with the -e option. For example, running screen -e ^jj would cause Control-j to be intercepted by screen instead of Control-a. If you do this, just replace references to C-a in the aforementioned reference guide with whatever escape key you defined.

**Shift-PageUp is broken**

Like vim and less, screen uses the terminal window differently from most programs, controlling the entire window instead of just dumping text to standard output and standard error. Unfortunately, this breaks Shift-PageUp and Shift-PageDown in gnome-terminal. Fortunately, we can fix this by creating a ~/.screenrc file with the following line in it:

termcapinfo xterm ti@:te@

And while you're mucking around in .screenrc, you might as well add an escape ^jj line to it, so that you can stop typing in -e ^jj every time you invoke screen.

**Starting screen automatically**

It's pretty easy to forget to run screen after logging in. Personally, any time I am using SSH to login and work interactively, I want to be in a screen session. We can combine SSH's ability to run a remote command upon login with screen's ability to reconnect to detached sessions. Simply create an alias in your ~/.bashrc file:

alias sshwork='ssh -t work-username@my-work-computer.com "screen -dR"'

This will automatically fire up a screen session if there is not one running, and if there is one running, it will connect to it. Detaching from the screen session will also logout of the remote server.

**Remote graphical work**

Even in spite of SSH's port forwarding capabilities, we still sometimes need to use graphical applications. If you have a fast connection or a simple GUI, passing the -Y flag to SSH could be enough to allow you run the application on your local desktop. Unfortunately, this often is a very poor user experience, and it does not work well with screen (a GUI application started in a screen session dies when you detach from the screen session).

The time-tested Unix solution to this problem is VNC. This is effectively a combination of screen and a graphical environment. Unfortunately, it has several drawbacks.

* It can be tricky to setup reasonably
* It is inherently insecure, with unencrypted data and a weak password feature
* Its performance on a sub-optimal connection is less-than-stellar
* It doesn't transfer sounds over the network

I'm going to help you solve all of these problems, except the sound one. Who needs sounds, anyway?

**VNC installation and setup**

On the remote machine, you'll need to install a VNC server and a decent lightweight window manager. I chose fluxbox and x11vnc:

sudo apt-get install x11vnc fluxbox

The programs that are started when you first start a VNC session are controlled by the ~/.vnc/xstartup file. I prefer something a bit better than the defaults, so mine looks like this:

#!/bin/sh

[ -x /etc/vnc/xstartup ] && exec /etc/vnc/xstartup

[ -r $HOME/.Xresources ] && xrdb $HOME/.Xresources

netbeans &

gnome-terminal &

fluxbox &

Modify this to suit your own needs; I only invoke netbeans because it's the only reason I ever use a remote GUI at all. NB: Although it may seem counterintuitive, it's typically best to put the window manager command last.

You can start a VNC server with the following command **(this isn't the way you should normally do it! Read on...)**:

vncserver -geometry WIDTHxHEIGHT

where WIDTHxHEIGHT is your desired resolution. For me, it's 1440x900. The first time you run this, it will ask you to create a password. We are going to ensure security through other means, so you can set it to whatever you want. Running the above command will give a message like, "New 'remote-machine:1 (username)' desktop is remote-machine:1." The :1 is the display number. By adding 5900 to this, we can determine which port the VNC server is listening on. At this point, we can connect to remote-machine:5901 with a vncviewer and log in to the session we've created. We don't want the entire Internet to be able to connect to our poorly-secured session, so let's terminate that VNC server session:

vncserver -kill :1

**Securing the VNC server**

Remember how we tunneled ports with SSH? We can do the same thing with VNC data. First, we'll invoke our VNC server slightly differently:

vncserver -localhost -geometry WIDTHxHEIGHT -SecurityTypes None

This causes the VNC server to only accept connections that originate on the local machine. It also indicates that we will not need a password to connect to our session; simply being logged in locally as the user who created the session is enough. You should now have a VNC server running on a remote machine listening on localhost:5901.

On your local machine, install a VNC viewer. I personally use gvncviewer, though I don't particularly recommend it. Now, to connect to that remote port, you'll need to start an SSH tunnel on your local machine:

ssh -NT -L 5901:localhost:5901 remote-machine.com

We can now run the VNC viewer on our local machine to connect via the tunnel to our VNC session:

gvncviewer :1

**Speeding up VNC?**

When starting an SSH tunnel, we can compress the data it sends by including the -C flag. Depending on your connection speed, it may be worth including the flag in your tunnel command. Experiment with this option and see what works best for you.

If you are really having problems, you might also want to check out the -deferUpdate option, which can delay how often display changes are sent to the client. For more information, man Xvnc.

**Automatically starting and connecting to your VNC session**

Putting everything together, we can create a script that does all of this for us. Modify the GEOMETRY and SSH\_ARGS variables appropriately (or change the script to accept them as command line arguments).

#!/bin/bash

set -e

GEOMETRY=1440x900

SSH\_ARGS='-p 22 username@remote-server.com'

# Get VNC display number. If there is not a VNC process running, start one

vnc\_display="$(ssh $SSH\_ARGS 'ps\_text="$(ps x | grep X[v]nc | awk '"'"'{ print $6 }'"'"' | sed s/://)"; if [ "$ps\_text" = "" ]; then vncserver -localhost -geometry '$GEOMETRY' -SecurityTypes none 2>&1 | grep New | sed '"'"'s/^.\*:\([^:]\*\)$/\1/'"'"'; else echo "$ps\_text"; fi')"

port=`expr 5900 + $vnc\_display`

ssh -NTC -L $port:localhost:$port $SSH\_ARGS &

SSH\_CMD=`echo $!`

sleep 3

gvncviewer :$vnc\_display

kill $SSH\_CMD

The vnc\_display line is pretty gross, so I'll give some explanation. It uses SSH to connect to the remote server and look for a running process named Xvnc: this is the running VNC server. If there's one running we extract the display number. Otherwise, we start one up with the specified geometry and grab the display number from there. This all happens within a single command executed by ssh, and the resulting output is piped across the network back into our vnc\_display variable.

Either way we get the value, we now know which port to connect to in order to reach our VNC server. We start our SSH tunnel and get the resulting PID. Finally, we invoke the vncviewer on that tunneled local port. When the VNC viewer exits, we automatically kill our SSH tunnel as well.

**Concluding remarks**

One of the best parts of Unix is that it was built to be run remotely from Day One. Just about anything you can do on your local computer can also be done on a remote one. By leveraging tools like SSH, screen, and VNC, we can make remote work as easy and convenient as local work. I hope this post gave you some ideas for how you can create a productive workflow with these very common Unix tools.