Contents

NAME

tsshbatch - Run Commands On Batches Of Machines

SYNOPSIS

tsshbatch.py [-ehvk] [-n name] [-p pw] [-H 'h1 h2 ...' | hostlistfile] command arg ...

DESCRIPTION

`tsshbatch` is a tool to enable you to issue a command to many hosts without having to log into each one separately. When writing scripts, this overcomes the `ssh` limitation of not being able to specify the password on the command line.

`tsshbatch` also understands basic `sudo` syntax and can be used to access a host, `sudo` a command, and then exit.

`tsshbatch` thus allows you to write complex, hands-off scripts that issue commands to many hosts without the tedium of manual login and `sudo` promotion. System administrators, especially, will find this helpful when working in large host farms.

OPTIONS

-H 'hostlist' Single quoted list of hosts on which to run the command

-e	Don't report remote host stderr output
-h	Print help information
-k	Use ssh keys instead of name/password credentials
-n name	Login name to use
-p pw	Password to use when logging in and/or doing sudo
-v	Print detailed program version information and exit

ENVIRONMENT

`tsshbatch` respects the `TSSHBATCH` environment variable. You may set this variable with any options above you commonly use to avoid having to key them in each time you run the program. For example:

```
export TSSHBATCH="-n jluser -p l00n3y"
```

This would cause all subsequent invocations of `tsshbatch` to attempt to use the login name/password credentials of `jluser` and `l00n3y` respectively.

USE CASES

1) Different Ways To Specify Targeted Hostnames

   There are two ways to specify the list of hosts on which you want to run the specified command:

   - On the command line via the `-H` option:

     ```
     tsshbatch.py -H 'hostA hostB' uname -a
     ```

     This would run the command `uname -a` on the hosts `hostA` and `hostB` respectively.

     Notice that the list of hosts must be separated by spaces but passed as a *single argument*. Hence we enclose them in single quotes.

   - Via a host list file:

     ```
     tsshbatch.py myhosts df -Ph
     ```

Here, `tsshbatch` expects the file `myhosts` to contain a list of hosts, one per line, on which to run the command `df -Ph`. As an example, if you want to target the hosts `larry`, `curly` and `moe` in `foo.com`, `myhosts` would look like this:

```
larry.foo.com
curly.foo.com
moe.foo.com
```

This method is handy when there are standard "sets" of hosts on which you regularly work. For instance, you may wish to keep a host file list for each of your production hosts, each of your test hosts, each of your AIX hosts, and so on.

2) Authentication Using Name And Password

The simplest way to use `tsshbatch` is to just name the hosts can command you want to run:

```
tsshbatch.py linux-prod-hosts uptime
```

You will be prompted for your username and password one time which `tsshbatch` will then use to log into each of the machines named in `linux-prod-hosts`. (*Notice that his assumes your name and password are the same on each host!*)

Typing in your login credentials all the time can get tedious after awhile so `tsshbatch` provides a means of providing them on the command line:

```
tsshbatch.py -n joe.luser -p my_weak_pw linux-prod-hosts uptime
```

This allows you to use `tsshbatch` inside scripts for hands-free operation.

If your login name is the same on all hosts, you can simplify this further by defining it in the environment variable:

```
export TSSHBATCH="-n joe.luser"
```

Any subsequent invocation of `tsshbatch` will only require a password to run.

HOWEVER, there is a huge downside to this - your plain text password is exposed in your scripts, on the command line, and possibly your command history. This is a pretty big security hole, especially if you're an administrator with extensive privileges. (This is why the `ssh` program does not support such an option.) For this reason, it is strongly recommended that you use the `-p` option sparingly, or not at all. A better way is to push ssh keys to every machine and use key exchange authentication as described below.

However, there are times when you do have use an explicit password, such as when doing `sudo` invocations. It would be really nice to use `-p` and

avoid having to constantly type in the password. There are two strategies for doing this more securely than just entering it in plain text on the command line:

- Temporarily store it in the environment variable:

  ```
  export TSSHBATCH="-n joe.luser -p my_weak_pw"
  ```

  Do this *interactively* after you log in, not from a script (otherwise you'd just be storing the plain text password in a different script). The environment variable will persist as long as you're logged in and disappear when you log out.

  If you use this just make sure to observe three security precautions:

  1) Clear your screen immediately after doing this so no one walking by can see the password you just entered.

  2) Configure your shell history system to ignore commands beginning with `export TSSHBATCH`. That way your plain text password will never appear in the shell command history.

  3) Make sure you don't leave a logged in session unlocked so that other users could walk up and see your password by displaying the environment.

  This approach is best when you want your login credentials available for the duration of an *entire login session*.

- Store your password in an encrypted file and decrypt it inline.

  First, you have to store your password in an encrypted format. There are several ways to do this, but `gpg` is commonly used:

  ```
  echo "my_weak_pw" | gpg -c >mysecretpw
  ```

  Provide a decrypt passphrase, and you're done.

  Now, you can use this by decrypting it inline as needed:

  ```
  #!/bin/sh
  # A demo scripted use of tsshbatch with CLI password passing

  MYPW=`cat mysecretpw | gpg`   # User will be prompted for unlock

  sshbatch.py -n joe.luser -p $MYPW hostlist1 command1 arg
  sshbatch.py -n joe.luser -p $MYPW hostlist2 command2 arg
  sshbatch.py -n joe.luser -p $MYPW hostlist3 command3 arg
  ```

  This approach is best when you want your login credentials available for the duration of *the execution of a script*.

It does require the user to type in a passphrase to unlock the encrypted password file, but your plain text password never appears in the wild.

3) Authentication Using Key Exchange

   For most applications of `tsshbatch`, it is much simpler to use key-based authentication. For this to work, you must first have pushed ssh keys to all your hosts. You then instruct `tsshbatch` to use key-based authentication rather than name and password. Not only does this eliminate the need to constantly provide name and password, it also eliminates passing a plain text password on the command line and is thus far more secure. This also overcomes the problem of having different name/password credentials on different hosts.

   By default, `tsshbatch` will prompt for name and password if they are not provided on the command line. To force key-based authentication, use the `-k` option:

   ```
   tsshbatch.py -k AIX-prod-hosts ls -al
   ```

4) Executing A `sudo` Command

   `tsshbatch` is smart enough to handle commands that begin with `sudo`. It knows that such commands *require* a password even if you used key exchange to initially log in. That's because, once you are logged in - whether via name/password or via key exchange - `sudo` requires your password again to promote your privileges.

   When using name/password authentication, with `tsshbatch` you need do nothing special to run `sudo` commands on your targeted hosts (assuming you have the privilege of doing so there).

   However, when using key exchange-based authentication, if you want to run `sudo` commands, *you will also have to provide a password* by one of the means described previously. That's because, once you are logged into a host, your password is required again to do `sudo` privilege promotion.

OTHER

`tsshbatch` writes the `stdout` of the remote host(s) to `stdout` on the local machine. It similarly writes remote `stderr` output to the local machine's `stderr`. If you wish to suppress `stderr` output, either redirect it on your local command line or use the `-e` option to turn it off entirely.

You will not be able to run remote `sudo` commands if the host in question enables the `Defaults requiretty` in its `sudoers` configuration.

You must have a reasonably current version of Python installed. If your Python installation does not install `paramiko` you'll have to install it manually, since `tsshbatch` requires these libraries.

BUGS AND MISFEATURES

When `sudo` is presented a bad password, it ordinarily prints a string indicating something is wrong. `tsshbatch` looks for this to let you know that you've got a problem and then terminates further operation. This is so that you do not attempt to log in with a bad password across all the servers you have targeted. (Many enterprises have policies to lock out a user ID after some small number of failed login/access attempts.)

However, some older versions of `sudo` (noted on a RHEL 4 server running `sudo` 1.6.7p5) do not return any feedback when presented with a bad password. This means that `tsshbatch` cannot tell the difference between a successful `sudo` and a system waiting for you to reenter a proper password. In this situation, if you enter a bad password, the *the program will hang*. Why? `tsshbatch` thinks nothing is wrong and waits for the `sudo` command to complete. At the same time, `sudo` itself is waiting for an updated password. In this case, you have to kill `tsshbatch` and start over. This typically requires you to put the program in background (`'Ctrl-Z` in most shells) and then killing that job from the command line.

There is no known workaround for this problem.

COPYRIGHT AND LICENSING

AUTHOR

```
Tim Daneliuk
tsshbatch@tundraware.com
```

DOCUMENT REVISION INFORMATION

```
$Id: tsshbatch.rst,v 1.109 2012/01/17 14:58:52 tundra Exp $
```

You can find the latest version of this program at:

http://www.tundraware.com/Software/tsshbatch