Contents

NAME

tsshbatch - Run Commands On Batches Of Machines

SYNOPSIS

```
tsshbatch.py [-KNSehktvxy -G 'file dest' -P 'file dest' -f cmdfile -n name
```

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.

You can also use `tsshbatch` to `GET` and `PUT` files from- and to many hosts at once.

`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 server farms.

OPTIONS

`tsshbatch` supports a variety of options which can be specified on either the command line or in the `$TSSHBATCH` environment variable:

-K Force prompting for passwords. This is used to override a prior `-k` argument.

-G spec GET file on host and write local dest directory. `spec` is a quoted pair of strings. The first specifies the path of the source file (on the remote machine) to copy. The second, specifies the destination *directory* (on the local machine):

```
tsshbatch.py -G "/foo/bar/baz /tmp" hostlist
```

This copies `/foo/bar/baz` from every machine in `hostlistfile` to the local `/tmp/` directory. Since all the files have the same name, they would overwrite each other if copied into the same directory. So, `tsshbatch` prepends the string `hostname:` to the name of each file it saves locally.

-H hostlistfile List of hosts on which to run the command. This should be enclosed in *quotes* so that the list of hosts is handed to the -H option as a single argument:

```
-H 'host1 host2 host3'
```

-N Force interactive username dialog. This cancels any previous request for key exchange authentication.

-P spec PUT file from local machine to remote machine destination directory. `spec` is a quoted pair of strings. The first specifies the path of the source file (on the local machine) to copy. The second, specifies the destination *directory* (on the remote machine):

```
tsshbatch.py -P "/foo/bar/baz /tmp" hostlist
```

This copies `/foo/bar/baz` on the local machine to `/tmp/` on every host in `hostlist`.

-S Force prompting for sudo password

-e	Don't report remote host stderr output
-f cmdfile	Read commands from a file. This file can be commented freely with the # character. Leading- and trailing whitespace on a line are ignored.
-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
-t	Test mode: Only show what *would* be done but don't actually do it. This also prints diagnostic information about any variable definitions, the list of hosts, any `GET` and `PUT` requests, and final command strings after all variable substitutions have been applied. This is the default program behavior.
-v	Print detailed program version information and exit
-x	Override any previous `-t` specifications and actually execute the commands. This is useful if you want to put `-t` in the `$TSSHBATCH` environment variable so that the default is always run the program in test mode. Then, when you're ready to actually run commands, you can override it with `-x` on the command line.
-y	Turn on 'noisy' reporting for additional detail on every line, instead of just at the top of the `stdout` and `stderr` reporting. This is helpful when you are filtering the output through something like `grep` that only returns matching lines and thus no context information.

If the `-H` option is not selected, the item immediately following the options is understood to be the name of the `hostlistfile`. This is a file that contains the name of each host - one per line - on which to run the commands. This file can be commented freely with the # character. Leading- and trailing whitespace on a line are ignored.

The last entry on the command line is optional and defines a command to run. `tsshbatch` will attempt to execute it on every host you've specified either via `-H` or a `hostfile`:

```
tsshbatch.py -Hmyhost ls -al /etc
```

This will do a `ls -al /etc` on `myhost`.

Be careful when using metacharacters like `&&, <<, >>, <, >` and so on in your commands. You have to escape and quote them properly or your local shell will interfere with them being properly conveyed to the remote machine.

If you've specified a `cmdfile` containing the commands you want run via the `-f` option, these commands will run *before* the command you've defined on the command line. It is always the last command run on each host.

You can put as many `-f` arguments as you wish on the command line and the contents of these files will be run in the order they appeared from left-to-right on the command line.

`tsshbatch` does all the `GETs`, then all the `PUTs` before attempting to do any command processing. If no `GETs`, `PUTs`, or commands have been specified, `tsshbatch` will exit silently, since "nothing to do" really isn't an error.

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.

`tsshbatch` also supports searching for files over specified paths with the `$TSSHBATCHCMDS` and `$TSSHBATCHHOSTS` environment variables. Their use is described later in this document.

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.

  You may use the # comment character freely throughout a host list file to add comments or temporarily comment out a particular host line.

  You can even use the comment character to temporarily comment out one or most hosts in the list given to the `-H` command line argument. For example:

  ```
  tsshbatch.py -H "foo #bar baz" ls
  ```

  This would run the `ls` command on hosts `foo` and `baz` but not `bar`. This is handy if you want to use your shell's command line recall to save typing but only want to repeat the command for some of the hosts your originally Specified.

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
```

By default, `tsshbatch` uses your login name found in the `$USER` environment variable when logging into other systems. In this example, you'll be prompted only for your password 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
```

This is so common that you may want to set it in your `$TSSHBATCH` environment variable so that keys are used by default. If you do this, there may still be times when you want for force prompting for passwords rather than using keys. You can do this with the `-K` option which effectively overrides any prior `-k` selection.

4) Executing A `sudo` Command

`tsshbatch` is smart enough to handle commands that begin with the `sudo` command. It knows that such commands *require* a password no matter how you initially authenticate to get into the system. If you provide a password - either via interactive entry or the `-p` option - by default, `tsshbatch` will use that same password for `sudo` promotion.

If you provide no password - you're using `-k` and have not provided a password via `-p` - `tsshbatch` will prompt you for the password `sudo` should use.

You can force `tsshbatch` to ask you for a `sudo` password with the `-S` option. This allows you to have one password for initial login, and a different one for `sudo` promotion.

Any time you a prompted for a `sudo` password and a login password has been provided (interactive or `-p`), you can accept this as the `sudo` password by just hitting `Enter`.

5) Precedence Of Authentication Options

`tsshbatch` supports these various authentication options in a particular heirarchy using a "first match wins" scheme. From highest to lowest, the precedence is:

1. Key exchange
2. Forced prompting for name via -N. Notice this cancels any previously requested key exchange authentication.
3. Command Line/$TSSHBATCH environment variable sets name
4. Name picked up from $USER (Default behavior)

If you try to use Key Exchange and `tsshbatch` detects a command beginning with `sudo`, it will prompt you for a password anyway. This is because `sudo` requires a password to promote privilege.

6) File Transfers

The `-G` and `-P` options specify file `GET` and `PUT` respectively. Both are followed by a quoted file transfer specification in the form:

```
"path-to-source-file path-to-destination-directory"
```

Note that this means the file will always be stored under its original name in the destination directory. Renaming isn't possible during file transfer.

However, `tsshbatch` always does `GETs` then `PUTs` *then* any outstanding command (if any) at the end of the command line. This permits things like renaming on the remote machine after a `PUT`:

```
tsshbatch.py -P "foo ./" hostlist mv -v foo foo.has.a.new.name
```

`GETs` are a bit of a different story because you are retrieving a file of the same name on every host. To avoid having all but the last one clobber the previous one, `tsshbatch` makes forces the files you `GET` to be uniquely named by prepending the hostname and a ":" to the actual file name:

```
tsshbatch.py -H myhost -G "foo ./"
```

This saves the file `myhost:foo` in the `./` on your local machine.

These commands do not recognize any special directory shortcut symbols like ~/ like the shell interpreter might. You must name file and directory locations using ordinary pathing conventions. You can put as many of these requests on the command line as you like to enable GETs and PUTs of multiple files. You cannot, however, use filename wildcards to specify multi-file operations.

You can put multiple GETs or PUTs on the command line for the same file. They do not override each other but are *cummulative*. So this:

```
tsshbatch.py -P"foo ./" -P"foo /tmp" ...
```

Would put local file foo in both ./ and /tmp on each host specified. Similarly, you can specify multiple files to GET from remote hosts and place them in the same local directory:

```
tsshbatch.py -G"/etc/fstab ./tmp" -G"/etc/rc.conf ./tmp" ...
```

If any file transfer fails, for any reason, the program is aborted and no further work is done.

Warning

tsshbatch does *not* preserve file permissions when transferring files. Recall that commands are always run *after* file transfers, so you can manually manage permissions like this:

```
tsshbatch.py -P"myfile ./tmp" hostlist chmod 640 ./tmp/myfile
```

This gets pretty clumsy for transferring more than one or two files. A better way to do this is to create a tarball of the source files, GET or PUT the tarball where you want it, and then untar it.

7) Commenting

Both the cmdfile and hostlistfile can be freely commented using the # character. Everything from that character to the end of that line is ignored. Similarly, you can use whitespace freely, except in cases where it would change the syntax of a command or host name.

8) Includes

You may also include other files as you wish with the .include filename directive anywhere in the cmdfile or hostlistfile. This is useful for breaking up long lists of things into smaller parts. For example, suppose you have three host lists, one for each major production areas of your network:

```
hosts-development
hosts-stage
host-production
```

You might typically run different tsshbatch jobs on each of these sets of hosts. But suppose you now want to run a job on all of them. Instead of copying them all into a master file (which would be instantly obsolete if you changed anything in one of the above files), you could create hosts-all with this content:

```
.include hosts-development
.include hosts-stage
.include hosts-production
```

That way if you edited any of the underlying files, the `hosts-all` would reflect the change.

Similarly you can do the same thing with the `cmdfile` to group similar commands into separate files and include them.

`tsshbatch` does not enforce a limit on how deeply nested `.includes` can be. An included file can include another file and so on. However, if a circular include is detected, the program will notify you and abort. This happens if, say, file1 includes file2, file2 includes file3, and file3 includes file1. This would create an infinite loop of includes if permitted. You can, of course, include the same file multiple times, either in a single file or throughout other included files, so long as no circular include is created.

9) Search Paths

`tsshbatch` supports the ablity to search paths to find files you've referenced. The search path for `cmdfiles` is specified in the `$TSSHBATCHCMDS` environment variable. The `hostlistfiles` search path is specified in the `$TSSHBATCHHOSTS` environment variable. These are both in standard path delimited format for your operating system. For example, on Unix-like systems these look like this:

```
export TSSHBATCHCMDS="/usr/local/etc/.tsshbatch/commands:/home/me/.tssh
```

And so forth.

These paths are honored both for any files you specify on the command line as well as for any files you reference in a `.include` directive. This allows you to maintain libraries of standard commands and host lists in well known locations and `.include` the ones you need.

`tsshbatch` will always first check to see if a file you've specified is in your local (invoking) directory and/or whether it is a fully qualified file name before attempting to look down a search path. If a file exist in several locations, the first instance found "wins". So, for instance, if you have a file called `myhosts` somewhere in the path defined in `$TSSHBATCHHOSTS`, you can override it by creating a file of same name in your current working directory.

`tsshbatch` also checks for so-called "circular includes" which would cause an infinite inclusion loop. It will abort upon discovering this, prior to any file transfers or commands being executed.

10) Defining Variables

`tsshbatch` allows you to define variables which will then be used to replace matching strings in both `cmdfiles` and `hostlistfiles`. For example, suppose you have this in a `hostlistfile`:

```
.define DOMAIN=.my.own.domain.com
```

```
host1DOMAIN
host2DOMAIN
host3DOMAIN
```

At runtime, the program will actually connect to `host1.my.own.domain.com`, `host2.my.domain.com`, and so on. This allows for ease of modularization and maintenance of your files.

Similarly, you might want define `MYCMD=some_long_string` so you don't have to type `some_long_string` over and over again in a `cmdfile`.

There are some "gotchas" to this:

- The general form of a variable definition is:

  ```
  .define name = value
  ```

  You have to have a name but the value is optional. `.define FOO=` simply replaces any subsequent `FOO` strings with nothing, effectively removing them.

  Any = symbols to the right of the one right after `name` are just considered part of the variables value.

  Whitespace around the = symbol is optional but allowed.

- Variables are substituted in the order they appear:

  ```
  .define LS = ls -alr
  LS /etc               # ls -alr /etc
  .define LS = ls -1
  LS /foo               # ls -1 /foo
  ```

- Variable names and values are *case sensitive*.

- Variables may be defined in either `cmdfiles` or `hostlistfiles` but they are *visible to any subsequent file that gets read*. For instance, `cmdfiles` are read before any `hostlistfiles`. Any variables you've defined in a `cmdfile` that happen to match a string in one of your hostnames will be substituted.

  This is usually not what you want, so be careful. One way to manage this is to use variables names that are highly unlikely to ever show up in a hostname or command. That way your commands and hostnames will not accidentally get substrings replaced with variable values. For example, you might use variable names like `--MYLSCOMMAND--` or `__DISPLAY_VGS__`.

- Variable substitution is also performed on any host names or commands passed on the command line.

11) Using The Current Hostname In Commands And File Transfers

There are times when it's convenient to be able to embed the name of the current host in either a command or in a file transfer specification. For

example, suppose you want to use a single invocation of `tsshbatch` to transfer files in a host-specific way. You might name your files like this:

```
myfile.host1
myfile.host2
```

Now, all you have to do is this:

```
tsshbatch.py -xH "host 1 host2" -P "myfile.<HOSTNAME> ./"
```

When run, `tsshbatch` will substitute the name of the current host in place of the string `<HOSTNAME>`.

You can do this in commands (and commands within command files) as well:

```
tsshbatch -x hosts 'echo I am running on <HOSTNAME>'
```

Be careful to escape and quote things properly, especially from the the command line, since < and > are recognized by the shell as metacharacters.

There are two forms of host name substitution possible. The first, `<HOSTNAME>` will use the name *as you provided it*, either as an argument to `-H` or from within a host file.

The second, `<HOSTSHORT>`, will only use the portion of the name string you provided up to the leftmost period.

So, if you specify `myhost1.frumious.edu`, `<HOSTNAME>` will be replaced with that entire string, and `<HOSTSHORT>` will be replaced by just `myhost1`.

Notice that, in no case does `tsshbatch` do any DNS lookups to figure this stuff out. It just manipulates the strings you provide as hostnames.

OTHER

Comments can go anywhere.

Directives like `.define` and `.include` must be the first non-whitespace text on the left end of a line. If you do this in a `cmdfile`:

```
foo .include bar
```

`tsshbatch` thinks you want to run the command `foo` with an argument of `.include bar`. If you do it in a `hostlistfile`, the program thinks you're tryingto contact a host called `foo .include bar`. In neither case is this likely to be quite what you had in mind. Similarly, everything to the right of the directive is considered its argument (up to any comment character).

Whitespace is not significant at the beginning or end of a line but it is preserved within `.define` and `.include` directive arguments as well as within commmand definitions.

Strictly speaking, you do not have to have whitespace after a directive. This is recognized:

```
.includesomefileofmine
.definemyvar=foo
```

But this is *strongly* discouraged because it's really hard to read.

`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 as well.

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 hosts 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 host 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 1.136 2014/03/27 23:40:59 tundra Exp $

You can find the latest version of this program at:

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