# 7.14 Git Tools - Credential Storage

## Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The "cache" mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The "store" mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won't ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you're using a Mac, Git comes with an "osxkeychain" mode, which caches credentials in the secure keychain that's attached to your system account. This method stores the credentials on disk, and they never expire, but they're encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you're using Windows, you can install a helper called "Git Credential Manager for Windows." This is similar to the "osxkeychain" helper described above, but uses the Windows Credential Store to control sensitive information. It can be found at https://github.com/Microsoft/Git-Credential-Manager-for-Windows.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The "store" helper can take a `--file <path>` argument, which customizes where the plain-text file is saved (the default is `~/.git-credentials`). The "cache" helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is "900", or 15 minutes). Here's an example of how you'd configure the "store" helper with a custom file name:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here's what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn't plugged in:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

### Under the Hood

How does this all work? Git's root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let's say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here's a session that uses the "fill" command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill (1)
protocol=https (2)
host=mygithost
(3)
protocol=https (4)
host=mygithost
username=bob
password=s3cre7
$ git credential fill (5)
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

1. This is the command line that initiates the interaction.

2. Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.

3. A blank line indicates that the input is complete, and the credential system should answer with what it knows.

4. Git-credential then takes over, and writes to stdout with the bits of information it found.

5. If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

| Configuration Value | Behavior |
|---|---|
| `foo` | Runs `git-credential-foo` |
| `foo -a --opt=bcd` | Runs `git-credential-foo -a --opt=bcd` |
| `/absolute/path/foo -xyz` | Runs `/absolute/path/foo -xyz` |
| `!f() { echo "password=s3cre7"; }; f` | Code after `!` evaluated in shell |

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is "git-credential-foo [args] <action>." The stdin/stdout protocol is the same as git-credential, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper's memory.
- `erase` purge the credentials for the given properties from this helper's memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn't know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here's the same example from above, but skipping git-credential and going straight for git-credential-store:

```
$ git credential-store --file ~/git.store store (1)
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get (2)
protocol=https
host=mygithost

username=bob (3)
password=s3cre7
```

1. Here we tell `git-credential-store` to save some credentials: the username "bob" and the password "s3cre7" are to be used when `https://mygithost` is accessed.

2. Now we'll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.

3. `git-credential-store` replies with the username and password we stored above.

Here's what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It's just a series of lines, each of which contains a credential-decorated URL.
The `osxkeychain` and `wincred`helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

## A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it's not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let's say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don't want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let's see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we'll just exit cleanly when they're received.

2. The file format of the shared-credential file is the same as that used by `git-credential-store`.

3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we'll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here's the full source code of our new credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' # (1)
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' # (2)
```

```
exit(0) unless File.exists? path

known = {} # (3)
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| # (4)
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] and user == known['username'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

1. Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.

2. This program only responds if the action is `get` and the backing-store file exists.

3. This loop reads from stdin until the first blank line is reached. The inputs are stored in the `known` hash for later reference.

4. This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to stdout and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with "git-", we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

prev | next