NAME

git-tag - Create, list, delete or verify a tag object signed with GPG

SYNOPSIS

git tag [-a | -s | -u <key-id>] [-f] [-m <msg> | -F <file>]
<tagname> [<commit> | <object>]
git tag-d <tagname>...
git tag[-n[<n um>]] -l [--contains <commit>] [--points-at <object>]
[--column[=<options>] | --no-column] [<pattern>...]
[<pattern>...]
git tag-v <tagname>...

DESCRIPTION

Add a tag reference in refs/tags/, unless -d/-l/-v is given to delete, list or verify tags.

Unless -f is given, the named tag must not yet exist.

If one of -a, -s, or -u <key-id> is passed, the command creates a *tag* object, and requires a tag message. Unless -m <msg> or -F <file> is given, an editor is started for the user to type in the tag message.

If -m <msg> or -F <file> is given and -a, -s, and -u <key-id> are absent, -a is implied.

Otherwise just a tag reference for the SHA-1 object name of the commit object is created (i.e. a lightweight tag).

A GnuPG signed tag object will be created when -s or -u <key-id> is used. When -u <key-id> is not used, the committer identity for the current user is used to find the GnuPG key for signing. The configuration variable gpg.program is used to specify custom GnuPG binary.

Tag objects (created with -a, -s, or -u) are called annotated tags; they contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature. Whereas a lightweight tag is simply a name for an object (usually a commit object).

Annotated tags are meant for release while lightweight tags are meant for private or temporary object labels. For this reason, some git commands for naming objects (like git describe) will ignore lightweight tags by default.

OPTIONS

-a, --annotate
: Make an unsigned, annotated tag object

-s, --sign
: Make a GPG-signed tag, using the default e-mail address's key.

-u <key-id>, --local-user=<key-id>
: Make a GPG-signed tag, using the given key.

-f, --force
: Replace an existing tag with the given name (instead of failing)

-d, --delete
: Delete existing tags with the given names.

-v, --verify
: Verify the gpg signature of the given tag names.

-n<num>
: <num> specifies how many lines from the annotation, if any, are printed when using -l. The default is not to print any annotation lines. If no number is given to -n, only the first line is printed. If the tag is not annotated, the commit message is displayed instead.

-l <pattern>, --list <pattern>

List tags with names that match the given pattern (or all if no pattern is given). Running git tag without arguments also lists all tags. The pattern is a shell wildcard (i.e., matched using fnmatch(3). Multiple patterns may be given; if any of them matches, the tag is shown.

--sort=<type>
Sort in a specific order. Supported type is refname (lexicographic order), version:refname or v:refname (tag names are treated as versions). Prepend - to reverse sort order. When this option is not given, the sort order defaults to the value configured for the *tag.sort* variable if it exists, or lexicographic order otherwise. See **git-config(1)**.

--column[=<options>], --no-column
Display tag listing in columns. See configuration variable column.tag for option syntax.--column and --no-column without options are equivalent to *always* and *never* respectively.

This option is only applicable when listing tags without annotation lines.

--contains [<commit>]
Only list tags which contain the specified commit (HEAD if not specified).

--points-at <object>
Only list tags of the given object.

-m <msg>, --message=<msg>
Use the given tag message (instead of prompting). If multiple -m options are given, their values are concatenated as separate paragraphs. Implies -a if none of -a, -s, or -u <key-id> is given.

-F <file>, --file=<file>
Take the tag message from the given file. Use - to read the message from the standard input. Implies -a if none of -a, -s, or -u <key-id> is given.

--cleanup=<mode>
This option sets how the tag message is cleaned up. The <*mode*> can be one of *verbatim*, *whitespace* and *strip*. The *strip* mode is default. The *verbatim* mode does not change message at all, *whitespace* removes just leading/trailing whitespace lines and *strip* removes both whitespace and commentary.

<tagname>
The name of the tag to create, delete, or describe. The new tag name must pass all checks defined by **git-check-ref-format(1)**. Some of these checks may restrict the characters allowed in a tag name.

<commit>, <object>
The object that the new tag will refer to, usually a commit. Defaults to HEAD.

CONFIGURATION

By default, *git tag* in sign-with-default mode (-s) will use your committer identity (of the form Your Name <your@email.address>) to find a key. If you want to use a different default key, you can specify it in the repository configuration as follows:

[user]
signingkey = <gpg-key-id>

DISCUSSION

On Re–tagging

What should you do when you tag a wrong commit and you would want to re-tag?

If you never pushed anything out, just re-tag it. Use -f to replace the old one. And you're done.

But if you have pushed things out (or others could just read your repository directly), then others will have already seen the old tag. In that case you can do one of two things:

1. The sane thing. Just admit you screwed up, and use a different name. Others have already seen one tag-name, and if you keep the same name, you may be in the situation that two people both have version X, but they actually have *different* Xs. So just call it X.1 and be done with it.
2. The insane thing. You really want to call the new version X too, *even though* others have already seen the old one. So just use *git tag -f* again, as if you hadn't already published the old one.

However, Git does **not** (and it should not) change tags behind users back. So if somebody already got the old tag, doing a *git pull* on your tree shouldn't just make them overwrite the old one.

If somebody got a release tag from you, you cannot just change the tag for them by updating your own one. This is a big security issue, in that people MUST be able to trust their tag-names. If you really want to do the insane thing, you need to just fess up to it, and tell people that you messed up. You can do that by making a very public announcement saying:

Ok, I messed up, and I pushed out an earlier version tagged as X. I
then fixed something, and retagged the *fixed* tree as X again.

If you got the wrong tag, and want the new one, please delete
the old one and fetch the new one by doing:

git tag -d X
git fetch origin tag X

to get my updated tag.

You can test which tag you have by doing

git rev-parse X

which should return 0123456789abcdef.. if you have the new version.

Sorry for the inconvenience.

Does this seem a bit complicated? It **should** be. There is no way that it would be correct to just fix it automatically. People need to know that their tags might have been changed.

On Automatic following

If you are following somebody else's tree, you are most likely using remote-tracking branches (refs/heads/origin in traditional layout, or refs/remotes/origin/master in the separate-remote layout). You usually want the tags from the other end.

On the other hand, if you are fetching because you would want a one-shot merge from somebody else, you typically do not want to get tags from there. This happens more often for people near the toplevel but not limited to them. Mere mortals when pulling from each other do not necessarily want to automatically get private anchor point tags from the other person.

Often, please pull messages on the mailing list just provide two pieces of information: a repo URL and a branch name; this is designed to be easily cut&pasted at the end of a *git fetch* command line:

Linus, please pull from

git://git..../proj.git master

to get the following updates...

becomes:

$ git pull git://git..../proj.git master

In such a case, you do not want to automatically follow the other person's tags.

One important aspect of Git is its distributed nature, which largely means there is no inherent

upstream or downstream in the system. On the face of it, the above example might seem to indicate that the tag namespace is owned by the upper echelon of people and that tags only flow downwards, but that is not the case. It only shows that the usage pattern determines who are interested in whose tags.

A one-shot pull is a sign that a commit history is now crossing the boundary between one circle of people (e.g. people who are primarily interested in the networking part of the kernel) who may have their own set of tags (e.g. this is the third release candidate from the networking group to be proposed for general consumption with 2.6.21 release) to another circle of people (e.g. people who integrate various subsystem improvements). The latter are usually not interested in the detailed tags used internally in the former group (that is what internal means). That is why it is desirable not to follow tags automatically in this case.

It may well be that among networking people, they may want to exchange the tags internal to their group, but in that workflow they are most likely tracking each other's progress by having remote-tracking branches. Again, the heuristic to automatically follow such tags is a good thing.

On Backdating Tags

If you have imported some changes from another VCS and would like to add tags for major releases of your work, it is useful to be able to specify the date to embed inside of the tag object; such data in the tag object affects, for example, the ordering of tags in the gitweb interface.

To set the date used in future tag objects, set the environment variable GIT_COMMITTER_DATE (see the later discussion of possible values; the most common form is YYYY-MM-DD HH:MM).

For example:

$ GIT_COMMITTER_DATE=2006-10-02 10:31 git tag -s v1.0.1

DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables support the following date formats:

Git internal format
: It is <unix timestamp> <time zone offset>, where <unix timestamp> is the number of seconds since the UNIX epoch. <time zone offset> is a positive or negative offset from UTC. For example CET (which is 2 hours ahead UTC) is +0200.

RFC 2822
: The standard email format as described by RFC 2822, for example Thu, 07 Apr 2005 22:13:13 +0200.

ISO 8601
: Time and date specified by the ISO 8601 standard, for example 2005-04-07T22:13:13. The parser accepts a space instead of the T character as well.

  Note

  In addition, the date part is accepted in the following formats: YYYY.MM.DD, MM/DD/YYYY and DD.MM.YYYY.

SEE ALSO

git-check-ref-format(1). **git-config(1)**.

GIT

Part of the **git(1)** suite