NAME

git-fetch - Download objects and refs from another repository

SYNOPSIS

```
git fetch [<options>] [<repository> [<refspec>...]]
git fetch [<options>] <group>
git fetch --multiple [<options>] [(<repository> | <group>)...]
git fetch --all [<options>]
```

DESCRIPTION

Fetch branches and/or tags (collectively, "refs") from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated (see the description of <refspec> below for ways to control this behavior).

By default, any tag that points into the histories being fetched is also fetched; the effect is to fetch tags that point at branches that you are interested in. This default behavior can be changed by using the --tags or --no-tags options or by configuring remote.<name>.tagOpt. By using a refspec that fetches tags explicitly, you can fetch tags that do not point into branches you are interested in as well.

git fetch can fetch from either a single named repository or URL, or from several repositories at once if <group> is given and there is a remotes.<group> entry in the configuration file. (See git-config(1)).

When no remote is specified, by default the origin remote will be used, unless there's an upstream branch configured for the current branch.

The names of refs that are fetched, together with the object names they point at, are written to .git/FETCH_HEAD. This information may be used by scripts or other git commands, such as git-pull(1).

OPTIONS

--all
Fetch all remotes.

-a, --append
Append ref names and object names of fetched refs to the existing contents of .git/FETCH_HEAD. Without this option old data in .git/FETCH_HEAD will be overwritten.

--atomic
Use an atomic transaction to update local refs. Either all refs are updated, or on error, no refs are updated.

--depth=<depth>
Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a shallow repository created by git clone with --depth=<depth> option (see git-clone(1)), deepen or shorten the history to the specified number of commits. Tags for the deepened commits are not fetched.

--deepen=<depth>
Similar to --depth, except it specifies the number of commits from the current shallow boundary instead of from the tip of each remote branch history.

--shallow-since=<date>
Deepen or shorten the history of a shallow repository to include all reachable commits after <date>.

--shallow-exclude=<revision>
Deepen or shorten the history of a shallow repository to exclude commits reachable from a specified remote branch or tag. This option can be specified multiple times.

--unshallow
If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow
By default when fetching from a shallow repository, git fetch refuses refs that require updating .git/shallow. This option updates .git/shallow and accept such refs.

--negotiation-tip=<commit|glob>
By default, Git will report, to the server, commits reachable from all local refs to find common commits in an attempt to reduce the size of the to-be-received packfile. If specified, Git will only report commits reachable from the given tips. This is useful to speed up fetches when the user knows which local ref is likely to have commits in common with the upstream ref being fetched.

This option may be specified more than once; if so, Git will report commits reachable from any of the given commits.

The argument to this option may be a glob on ref names, a ref, or the (possibly abbreviated) SHA-1 of a commit. Specifying a glob is equivalent to specifying this option multiple times, one for each matching ref name.

See also the fetch.negotiationAlgorithm and push.negotiate configuration variables documented in git-config(1), and the --negotiate-only option below.

--negotiate-only
Do not fetch anything from the server, and instead print the ancestors of the provided --negotiation-tip=* arguments, which we have in common with the server.

Internally this is used to implement the push.negotiate option, see git-config(1).

--dry-run
Show what would be done, without making any changes.

--[no-]write-fetch-head
Write the list of remote refs fetched in the FETCH_HEAD file directly under $GIT_DIR. This is the default. Passing --no-write-fetch-head from the command line tells Git not to write the file. Under --dry-run option, the file is never written.

-f, --force
When git fetch is used with <src>:<dst> refspec it may refuse to update the local branch as discussed in the <refspec> part below. This option overrides that check.

-k, --keep
Keep downloaded pack.

--multiple
Allow several <repository> and <group> arguments to be specified. No <refspec>s may be specified.

--[no-]auto-maintenance, --[no-]auto-gc
Run git maintenance run --auto at the end to perform automatic repository maintenance if needed. (--[no-]auto-gc is a synonym.) This is enabled by default.

--[no-]write-commit-graph
Write a commit-graph after fetching. This overrides the config setting fetch.writeCommitGraph.

--prefetch
Modify the configured refspec to place all refs into the refs/prefetch/ namespace. See the prefetch task in git-maintenance(1).

-p, --prune
Before fetching, remove any remote-tracking references that no longer exist on the remote. Tags are not subject to pruning if they are fetched only because of the default tag auto-following or due to a --tags option. However, if tags are fetched due to an explicit refspec (either on the command line or in the remote configuration, for example if the remote was cloned with the --mirror option), then they are also subject to pruning. Supplying --prune-tags is a shorthand for providing the tag refspec.

See the PRUNING section below for more details.

-P, --prune-tags
Before fetching, remove any local tags that no longer exist on the remote if --prune is enabled. This option should be used more carefully, unlike --prune it will remove any local references (local tags) that have been created. This option is a shorthand for providing the explicit tag refspec along with --prune, see the discussion about that in its documentation.

See the PRUNING section below for more details.

-n, --no-tags
By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See git-config(1).

--refmap=<refspec>
When fetching refs listed on the command line, use the specified refspec (can be given more than once) to map the refs to remote-tracking branches, instead of the values of remote.*.fetch configuration variables for the remote repository. Providing an empty <refspec> to the --refmap option causes Git to ignore the configured refspecs and rely entirely on the refspecs supplied as command-line arguments. See section on "Configured Remote-tracking Branches" for details.

-t, --tags
Fetch all tags from the remote (i.e., fetch remote tags refs/tags/* into local tags with the same name), in addition to whatever else would otherwise be fetched. Using this option alone does not subject tags to pruning, even if --prune is used (though tags may be pruned anyway if they are also the destination of an explicit refspec; see --prune).

--recurse-submodules[=yes|on-demand|no]
This option controls if and under what conditions new commits of populated submodules should be fetched too. It can be used as a boolean option to completely disable recursion when set to no or to unconditionally recurse into all populated submodules when set to yes, which is the default when this option is used without any value. Use on-demand to only recurse into a populated submodule when the superproject retrieves a commit that updates the submodule's reference to a commit that isn't already in the local submodule clone. By default, on-demand is used, unless fetch.recurseSubmodules is set (see git-config(1)).

-j, --jobs=<n>
Number of parallel children to be used for all forms of fetching.

If the --multiple option was specified, the different remotes will be fetched in parallel. If multiple submodules are fetched, they will be fetched in parallel. To control them independently, use the config settings fetch.parallel and submodule.fetchJobs (see git-config(1)).

Typically, parallel recursive and multi-remote fetches will be faster. By default fetches are performed sequentially, not in parallel.

--no-recurse-submodules
Disable recursive fetching of submodules (this has the same effect as using the --recurse-submodules=no option).

--set-upstream
If the remote is fetched successfully, add upstream (tracking) reference, used by argument-less git-pull(1) and other commands. For more information, see branch.<name>.merge and branch.<name>.remote in git-config(1).

--submodule-prefix=<path>
Prepend <path> to paths printed in informative messages such as "Fetching submodule foo". This option is used internally when recursing over submodules.

--recurse-submodules-default=[yes|on-demand]
This option is used internally to temporarily provide a non-negative default value for the --recurse-submodules option. All other methods of configuring fetch's submodule recursion (such as settings in gitmodules(5) and git-config(1)) override this option, as does specifying --[no-]recurse-submodules directly.

-u, --update-head-ok
By default git fetch refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for git pull to communicate with git fetch, and unless you are implementing your own Porcelain you are not supposed to use it.

--upload-pack <upload-pack>
When given, and the repository to fetch from is handled by git fetch-pack, --exec=<upload-pack> is passed to the command to specify non-default path for the command run on the other end.

-q, --quiet
Pass --quiet to git-fetch-pack and silence any other internally used git commands. Progress is not reported to the standard error stream.

-v, --verbose
Be verbose.

--progress
Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

-o <option>, --server-option=<option>
Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character. The server's handling of server options, including unknown ones, is server-specific. When multiple --server-option=<option> are given, they are all sent to the other

side in the order listed on the command line.

--show-forced-updates
By default, git checks if a branch is force-updated during fetch. This can be disabled through fetch.showForcedUpdates, but the --show-forced-updates option guarantees this check occurs. See git-config(1).

--no-show-forced-updates
By default, git checks if a branch is force-updated during fetch. Pass --no-show-forced-updates or set fetch.showForcedUpdates to false to skip this check for performance reasons. If used during git-pull the --ff-only option will still check for forced updates before attempting a fast-forward update. See git-config(1).

-4, --ipv4
Use IPv4 addresses only, ignoring IPv6 addresses.

-6, --ipv6
Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>
The "remote" repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<group>
A name referring to a list of repositories as the value of remotes.<group> in the configuration file. (See git-config(1)).

<refspec>
Specifies which refs to fetch and which local refs to update. When no <refspec>s appear on the command line, the refs to fetch are read from remote.<repository>.fetch variables instead (see CONFIGURED REMOTE-TRACKING BRANCHES below).

The format of a <refspec> parameter is an optional plus +, followed by the source <src>, followed by a colon :, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty. <src> is typically a ref, but it can also be a fully spelled hex object name.

A <refspec> may contain a * in its <src> to indicate a simple pattern match. Such a refspec functions like a glob that matches any ref with the same prefix. A pattern <refspec> must have a * in both the <src> and <dst>. It will map refs to the destination by replacing the * with the contents matched from the source.

If a refspec is prefixed by ^, it will be interpreted as a negative refspec. Rather than specifying which refs to fetch or which local refs to update, such a refspec will instead specify refs to exclude. A ref will be considered to match if it matches at least one positive refspec, and does not match any negative refspec. Negative refspecs can be useful to restrict the scope of a pattern refspec so that it will not include specific refs. Negative refspecs can themselves be pattern refspecs. However, they may only contain a <src> and do not specify a <dst>. Fully spelled out hex object names are also not supported.

tag <tag> means the same as refs/tags/<tag>:refs/tags/<tag>; it requests fetching everything up to the given tag.

The remote ref that matches <src> is fetched, and if <dst> is not an empty string, an attempt is made to update the local ref that matches it.

Whether that update is allowed without --force depends on the ref

namespace it's being fetched to, the type of object being fetched, and whether the update is considered to be a fast-forward. Generally, the same rules apply for fetching as when pushing, see the <refspec>... section of git-push(1) for what those are. Exceptions to those rules particular to git fetch are noted below.

Until Git version 2.20, and unlike when pushing with git-push(1), any updates to refs/tags/* would be accepted without + in the refspec (or --force). When fetching, we promiscuously considered all tag updates from a remote to be forced fetches. Since Git version 2.20, fetching to update refs/tags/* works the same way as when pushing. I.e. any updates will be rejected without + in the refspec (or --force).

Unlike when pushing with git-push(1), any updates outside of refs/{tags,heads}/* will be accepted without + in the refspec (or --force), whether that's swapping e.g. a tree object for a blob, or a commit for another commit that's doesn't have the previous commit as an ancestor etc.

Unlike when pushing with git-push(1), there is no configuration which'll amend these rules, and nothing like a pre-fetch hook analogous to the pre-receive hook.

As with pushing with git-push(1), all of the rules described above about what's not allowed as an update can be overridden by adding an the optional leading + to a refspec (or using --force command line option). The only exception to this is that no amount of forcing will make the refs/heads/* namespace accept a non-commit object.

Note
When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the + sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.

--stdin
Read refspecs, one per line, from stdin in addition to those provided as arguments. The "tag <name>" format is not supported.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp, and ftps can be used for fetching, but this is inefficient and deprecated; do not use it).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- ssh://[user@]host.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path foo:bar could be specified as an absolute path or ./foo:bar to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support ~username expansion:

- ssh://[user@]host.xz[:port]/~[user]/path/to/repo.git/

- git://host.xz[:port]/~[user]/path/to/repo.git/

- [user@]host.xz:/~[user]/path/to/repo.git/

For local repositories, also supported by Git natively, the following syntaxes may be used:

- /path/to/repo.git/

- file:///path/to/repo.git/

These two syntaxes are mostly equivalent, except when cloning, when the former implies --local option. See git-clone(1) for details.

git clone, git fetch and git pull, but not git push, will also accept a suitable bundle file. See git-bundle(1).

When Git doesn't know how to handle a certain transport protocol, it attempts to use the remote-<transport> remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- <transport>::<address>

where <address> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See gitremote-helpers(7) for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual url base>"]
        insteadOf = <other url base>
```

For example, with this:

```
[url "git://git.host.xz/"]
        insteadOf = host.xz:/path/to/
        insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual url base>"]
        pushInsteadOf = <other url base>
```

For example, with this:

```
[url "ssh://example.org/"]
        pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

REMOTES

The name of one of the following can be used instead of a URL as <repository> argument:

- a remote in the Git configuration file: $GIT_DIR/config,
- a file in the $GIT_DIR/remotes directory, or
- a file in the $GIT_DIR/branches directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote(1), git-config(1) or even by a manual edit to the $GIT_DIR/config file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
        url = <url>
        pushurl = <pushurl>
        push = <refspec>
        fetch = <refspec>
```

The <pushurl> is used for pushes only. It is optional and defaults to <url>.

Named file in $GIT_DIR/remotes

You can choose to provide the name of a file in $GIT_DIR/remotes. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL format
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by git push and Pull: lines are used by git pull and git fetch. Multiple Push: and Pull: lines may be specified for additional branch mappings.

Named file in $GIT_DIR/branches

You can choose to provide the name of a file in $GIT_DIR/branches. The URL in this file will be used to access the repository. This file should have the following format:

```
<url>#<head>
```

<url> is required; #<head> is optional.

Depending on the operation, git will use one of the following refspecs, if you don’t provide one on the command line. <branch> is the name of this file in $GIT_DIR/branches and <head> defaults to master.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

HEAD:refs/heads/<head>

CONFIGURED REMOTE-TRACKING BRANCHES

You often interact with the same remote repository by regularly and repeatedly fetching from it. In order to keep track of the progress of such a remote repository, git fetch allows you to configure remote.<repository>.fetch configuration variables.

Typically such a variable may look like this:

```
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
```

This configuration is used in two ways:

- When git fetch is run without specifying what branches and/or tags to fetch on the command line, e.g. git fetch origin or git fetch, remote.<repository>.fetch values are used as the refspecs--they specify which refs to fetch and which local refs to update. The example above will fetch all branches that exist in the origin (i.e. any ref that matches the left-hand side of the value, refs/heads/*) and update the corresponding remote-tracking branches in the refs/remotes/origin/* hierarchy.

- When git fetch is run with explicit branches and/or tags to fetch on the command line, e.g. git fetch origin master, the <refspec>s given on the command line determine what are to be fetched (e.g. master in the example, which is a short-hand for master:, which in turn means "fetch the master branch but I do not explicitly say what remote-tracking branch to update with it from the command line"), and the example command will fetch only the master branch. The remote.<repository>.fetch values determine which remote-tracking branch, if any, is updated. When used in this way, the remote.<repository>.fetch values do not have any effect in deciding what gets fetched (i.e. the values are not used as refspecs when the command-line lists refspecs); they are only used to decide where the refs that are fetched are stored by acting as a mapping.

The latter use of the remote.<repository>.fetch values can be overridden by giving the --refmap=<refspec> parameter(s) on the command line.

PRUNING

Git has a default disposition of keeping data unless it's explicitly thrown away; this extends to holding onto local references to branches on remotes that have themselves deleted those branches.

If left to accumulate, these stale references might make performance worse on big and busy repos that have a lot of branch churn, and e.g. make the output of commands like git branch -a --contains <commit> needlessly verbose, as well as impacting anything else that'll work with the complete set of known references.

These remote-tracking references can be deleted as a one-off with either of:

```
# While fetching
$ git fetch --prune <name>

# Only prune, don't fetch
$ git remote prune <name>
```

To prune references as part of your normal workflow without needing to remember to run that, set fetch.prune globally, or remote.<name>.prune per-remote in the config. See git-config(1).

Here's where things get tricky and more specific. The pruning feature

doesn't actually care about branches, instead it'll prune local <--> remote-references as a function of the refspec of the remote (see <refspec> and CONFIGURED REMOTE-TRACKING BRANCHES above).

Therefore if the refspec for the remote includes e.g. refs/tags/*:refs/tags/*, or you manually run e.g. git fetch --prune <name> "refs/tags/*:refs/tags/*" it won't be stale remote tracking branches that are deleted, but any local tag that doesn't exist on the remote.

This might not be what you expect, i.e. you want to prune remote <name>, but also explicitly fetch tags from it, so when you fetch from it you delete all your local tags, most of which may not have come from the <name> remote in the first place.

So be careful when using this with a refspec like refs/tags/*:refs/tags/*, or any other refspec which might map references from multiple remotes to the same local namespace.

Since keeping up-to-date with both branches and tags on the remote is a common use-case the --prune-tags option can be supplied along with --prune to prune local tags that don't exist on the remote, and force-update those tags that differ. Tag pruning can also be enabled with fetch.pruneTags or remote.<name>.pruneTags in the config. See git-config(1).

The --prune-tags option is equivalent to having refs/tags/*:refs/tags/* declared in the refspecs of the remote. This can lead to some seemingly strange interactions:

```
# These both fetch tags
$ git fetch --no-tags origin 'refs/tags/*:refs/tags/*'
$ git fetch --no-tags --prune-tags origin
```

The reason it doesn't error out when provided without --prune or its config versions is for flexibility of the configured versions, and to maintain a 1=1 mapping between what the command line flags do, and what the configuration versions do.

It's reasonable to e.g. configure fetch.pruneTags=true in ~/.gitconfig to have tags pruned whenever git fetch --prune is run, without making every invocation of git fetch without --prune an error.

Pruning tags with --prune-tags also works when fetching a URL instead of a named remote. These will all prune tags not found on origin:

```
$ git fetch origin --prune --prune-tags
$ git fetch origin --prune 'refs/tags/*:refs/tags/*'
$ git fetch <url of origin> --prune --prune-tags
$ git fetch <url of origin> --prune 'refs/tags/*:refs/tags/*'
```

OUTPUT

The output of "git fetch" depends on the transport method used; this section describes the output when fetching over the Git protocol (either locally or via ssh) and Smart HTTP protocol.

The status of the fetch is output in tabular form, with each line representing the status of a single ref. Each line is of the form:

```
 <flag> <summary> <from> -> <to> [<reason>]
```

The status of up-to-date refs is shown only if the --verbose option is used.

In compact output mode, specified with configuration variable fetch.output, if either entire <from> or <to> is found in the other string, it will be substituted with * in the other string. For example, master -> origin/master becomes master -> origin/*.

flag
A single character indicating the status of the ref:

(space)
for a successfully fetched fast-forward;

+
for a successful forced update;

-
for a successfully pruned ref;

t
for a successful tag update;

*
for a successfully fetched new ref;

!
for a ref that was rejected or failed to update; and

=
for a ref that was up to date and did not need fetching.

summary
For a successfully fetched ref, the summary shows the old and new values of the ref in a form suitable for using as an argument to git log (this is <old>..<new> in most cases, and <old>...<new> for forced non-fast-forward updates).

from
The name of the remote ref being fetched from, minus its refs/<type>/ prefix. In the case of deletion, the name of the remote ref is "(none)".

to
The name of the local ref being updated, minus its refs/<type>/ prefix.

reason
A human-readable explanation. In the case of successfully fetched refs, no explanation is needed. For a failed ref, the reason for failure is described.

EXAMPLES

- Update the remote-tracking branches:

```
$ git fetch origin
```

The above command copies all branches from the remote refs/heads/ namespace and stores them to the local refs/remotes/origin/ namespace, unless the branch.<name>.fetch option is used to specify a non-default refspec.

- Using refspecs explicitly:

```
$ git fetch origin +seen:seen maint:tmp
```

This updates (or creates, as necessary) branches seen and tmp in the local repository by fetching from the branches (respectively) seen and maint from the remote repository.

The seen branch will be updated even if it does not fast-forward, because it is prefixed with a plus sign; tmp will not be.

- Peek at a remote's branch, without configuring the remote in your local repository:

```
$ git fetch git://git.kernel.org/pub/scm/git/git.git maint
$ git log FETCH_HEAD
```

The first command fetches the maint branch from the repository at git://git.kernel.org/pub/scm/git/git.git and the second command uses FETCH_HEAD to examine the branch with git-log(1). The fetched objects will eventually be removed by git's built-in housekeeping (see git-gc(1)).

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn't required to send the content of X because the victim already has it. Now the victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)

2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

BUGS

Using --recurse-submodules can only fetch new commits in already checked out submodules right now. When e.g. upstream added a new submodule in the just fetched commits of the superproject the submodule itself cannot be fetched, making it impossible to check out that submodule later without having to do a fetch again. This is expected to be fixed in a future Git version.

SEE ALSO

git-pull(1)

GIT

Part of the git(1) suite