git-push(1) Manual Page

NAME

git-push - Update remote refs along with associated objects

SYNOPSIS

git push [--all | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--receive-pack=<git-receive-pack>]
[--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-v | --verbose]
[-u | --set-upstream] [-o <string> | --push-option=<string>]
[--[no-]signed|--signed=(true|false|if-asked)]
[--force-with-lease[=<refname>[:<expect>]]]
[--no-verify] [<repository> [<refspec>...]]

DESCRIPTION

Updates remote refs using local refs, while sending objects necessary to complete the given refs.

You can make interesting things happen to a repository every time you push into it, by setting up *hooks* there. See documentation for git-receive-pack(1).

When the command line does not specify where to push with the `<repository>` argument, `branch.*.remote` configuration for the current branch is consulted to determine where to push. If the configuration is missing, it defaults to *origin*.

When the command line does not specify what to push with `<refspec>...` arguments or `--all`, `--mirror`, `--tags` options, the command finds the default `<refspec>` by consulting `remote.*.push` configuration, and if it is not found, honors `push.default` configuration to decide what to push (See git-config(1) for the meaning of `push.default`).

When neither the command-line nor the configuration specify what to push, the default behavior is used, which corresponds to the `simple` value for `push.default`: the current branch is pushed to the corresponding upstream branch, but as a safety measure, the push is aborted if the upstream branch does not have the same name as the local one.

OPTIONS

<repository>

The "remote" repository that is destination of a push 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).

<refspec>...

Specify what destination ref to update with what source object. The format of a <refspec> parameter is an optional plus `+`, followed by the source object <src>, followed by a colon `:`, followed by the destination ref <dst>.

The <src> is often the name of the branch you would want to push, but it can be any arbitrary "SHA-1 expression", such as `master~4` or `HEAD` (see gitrevisions(7)).

The <dst> tells which ref on the remote side is updated with this push. Arbitrary expressions cannot be used here, an actual ref must be named. If `git push [<repository>]` without any `<refspec>` argument is set to update some ref at the destination with `<src>` with `remote.<repository>.push` configuration variable, `:<dst>` part can be omitted—such a push will update a ref that `<src>` normally updates without any `<refspec>` on the command line. Otherwise, missing `:<dst>` means to update the same ref as the `<src>`.

The object referenced by <src> is used to update the <dst> reference on the remote side. Whether this is allowed depends on where in `refs/*` the <dst> reference lives as described in detail below, in those sections "update" means any modifications except deletes, which as noted after the next few sections are treated differently.

The `refs/heads/*` namespace will only accept commit objects, and updates only if they can be fast-forwarded.

The `refs/tags/*` namespace will accept any kind of object (as commits, trees and blobs can be tagged), and any updates to them will be rejected.

It's possible to push any type of object to any namespace outside of `refs/{tags,heads}/*`. In the case of tags and commits, these will be treated as if they were the commits inside `refs/heads/*` for the purposes of whether the update is allowed.

I.e. a fast-forward of commits and tags outside `refs/{tags,heads}/*` is allowed, even in cases where what's being fast-forwarded is not a commit, but a tag object which happens to point to a new commit which is a fast-forward of the commit the last tag (or commit) it's replacing. Replacing a tag with an entirely different tag is also allowed, if it points to the same commit, as well as pushing a peeled tag, i.e. pushing the commit that existing tag object points to, or a new tag object which an existing commit points to.

Tree and blob objects outside of `refs/{tags,heads}/*` will be treated the same way as if they were inside `refs/tags/*`, any update of them will be rejected.

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. Hooks and configuration can also override or amend these rules, see e.g. `receive.denyNonFastForwards` in git-config(1) and `pre-receive` and `update` in githooks(5).

Pushing an empty <src> allows you to delete the <dst> ref from the remote repository. Deletions are always accepted without a leading `+` in the refspec (or `--force`), except when forbidden by configuration or hooks. See `receive.denyDeletes` in git-config(1) and `pre-receive` and `update` in githooks(5).

The special refspec `:` (or `+:` to allow non-fast-forward updates) directs Git to push "matching" branches: for every branch that exists on the local side, the remote side is updated if a branch of the same name already exists on the remote side.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`.

--all

Push all branches (i.e. refs under `refs/heads/`); cannot be used with other <refspec>.

--prune

Remove remote branches that don't have a local counterpart. For example a remote branch `tmp` will be removed if a local branch with the same name doesn't exist any more. This also respects refspecs, e.g. `git push --prune remote refs/heads/*:refs/tmp/*` would make sure that remote `refs/tmp/foo` will be removed if `refs/heads/foo` doesn't exist.

--mirror

Instead of naming each ref to push, specifies that all refs under `refs/` (which includes but is not limited to `refs/heads/`, `refs/remotes/`, and `refs/tags/`) be mirrored to the remote repository. Newly created local refs will be pushed to the remote end, locally updated refs will be force updated on the remote end, and deleted refs will be removed from the remote end. This is the default if the configuration option `remote.<remote>.mirror` is set.

-n

--dry-run

Do everything except actually send the updates.

--porcelain

Produce machine-readable output. The output status line for each ref will be tab-separated and sent to stdout instead of stderr. The full symbolic names of the refs will be given.

-d

--delete

All listed refs are deleted from the remote repository. This is the same as prefixing all refs with a colon.

--tags

All refs under `refs/tags` are pushed, in addition to refspecs explicitly listed on the command line.

--follow-tags

Push all the refs that would be pushed without this option, and also push annotated tags in `refs/tags` that are missing from the remote but are pointing at commit-ish that are reachable from the refs being pushed. This can also be specified with configuration variable `push.followTags`. For more information, see `push.followTags` in git-config(1).

--[no-]signed

--signed=(true|false|if-asked)

GPG-sign the push request to update refs on the receiving side, to allow it to be checked by the hooks and/or be logged. If `false` or `--no-signed`, no signing will be attempted. If `true` or `--signed`, the push will fail if the server does not support signed pushes. If set to `if-asked`, sign if and only if the server supports signed pushes. The push will also fail if the actual call to `gpg --sign` fails. See git-receive-pack(1) for the details on the receiving end.

--[no-]atomic

Use an atomic transaction on the remote side if available. Either all refs are updated, or on error, no refs are updated. If the server does not support atomic pushes the push will fail.

-o <option>

--push-option=<option>

Transmit the given string to the server, which passes them to the pre-receive as well as the post-receive hook. The given string must not contain a NUL or LF character. When multiple `--push-option=<option>` are given, they are all sent to the other side in the order listed on the command line. When no `--push-option=<option>` is given from the command line, the values of configuration variable `push.pushOption` are used instead.

--receive-pack=<git-receive-pack>

--exec=<git-receive-pack>

Path to the *git-receive-pack* program on the remote end. Sometimes useful when pushing to a remote repository over ssh, and you do not have the program in a directory on the default $PATH.

--[no-]force-with-lease

--force-with-lease=<refname>

--force-with-lease=<refname>:<expect>

Usually, "git push" refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it.

This option overrides this restriction if the current value of the remote ref is the expected value. "git push" fails otherwise.

Imagine that you have to rebase what you have already published. You will have to bypass the "must fast-forward" rule in order to replace the history you originally published with the rebased history. If somebody else built on top of your original history while you are rebasing, the tip of the branch at the remote may advance with her commit, and blindly pushing with `--force` will lose her work.

This option allows you to say that you expect the history you are updating is what you rebased and want to replace. If the remote ref still points at the commit you specified, you can be sure that no other people did anything to the ref. It is like taking a "lease" on the ref without explicitly locking it, and the remote ref is updated only if the "lease" is still valid.

`--force-with-lease` alone, without specifying the details, will protect all remote refs that are going to be updated by requiring their current value to be the same as the remote-tracking branch we have for them.

`--force-with-lease=<refname>` , without specifying the expected value, will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the remote-tracking branch we have for it.

`--force-with-lease=<refname>:<expect>` will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the specified value `<expect>` (which is allowed to be different from the remote-tracking branch we have for the refname, or we do not even have to have such a remote-tracking branch when this form is used). If `<expect>` is the empty string, then the named ref must not already exist.

Note that all forms other than `--force-with-lease=<refname>:<expect>` that specifies the expected current value of the ref explicitly are still experimental and their semantics may change as we gain experience with this feature.

"--no-force-with-lease" will cancel all the previous --force-with-lease on the command line.

A general note on safety: supplying this option without an expected value, i.e. as `--force-with-lease` or `--force-with-lease=<refname>` interacts very badly with anything that implicitly runs `git fetch` on the remote to be pushed to in the background, e.g. `git fetch origin` on your repository in a cronjob.

The protection it offers over `--force` is ensuring that subsequent changes your work wasn't based on aren't clobbered, but this is trivially defeated if some background process is updating refs in the background. We don't have anything except the remote tracking info to go by as a heuristic for refs you're expected to have seen & are willing to clobber.

If your editor or some other system is running `git fetch` in the background for you a way to mitigate this is to simply set up another remote:

```
git remote add origin-push $(git config remote.origin.url)
git fetch origin-push
```

Now when the background process runs `git fetch origin` the references on `origin-push` won't be updated, and thus commands like:

```
git push --force-with-lease origin-push
```

Will fail unless you manually run `git fetch origin-push`. This method is of course entirely defeated by something that runs `git fetch --all`, in that case you'd need to either disable it or do something more tedious like:

```
git fetch              # update 'master' from remote
git tag base master    # mark our base point
git rebase -i master   # rewrite some commits
git push --force-with-lease=master:base master:master
```

I.e. create a `base` tag for versions of the upstream code that you've seen and are willing to overwrite, then rewrite history, and finally force push changes to `master` if the remote version is still at `base`, regardless of what your local `remotes/origin/master` has been updated to in the background.

-f

--force

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. Also, when `--force-with-lease` option is used, the command refuses to update a remote ref whose current value does not match what is expected.

This flag disables these checks, and can cause the remote repository to lose commits; use it with care.

Note that `--force` applies to all the refs that are pushed, hence using it with `push.default` set to `matching` or with multiple push destinations configured with `remote.*.push` may overwrite refs other than the current branch (including local refs that are strictly behind their remote counterpart). To force a push to only one branch, use a `+` in front of the refspec to push (e.g `git push origin +master` to force a push to the `master` branch). See the `<refspec>...` section above for details.

--repo=<**repository**>

This option is equivalent to the <repository> argument. If both are specified, the command-line argument takes precedence.

-u

--set-upstream

For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less git-pull(1) and other commands. For more information, see `branch.<name>.merge` in git-config(1).

--[no-]thin

These options are passed to git-send-pack(1). A thin transfer significantly reduces the amount of sent data when the sender and receiver share many of the same objects in common. The default is `--thin`.

-q

--quiet

Suppress all output, including the listing of updated refs, unless an error occurs. Progress is not reported to the standard error stream.

-v

--verbose

Run verbosely.

--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.

--no-recurse-submodules

--recurse-submodules=check|on-demand|only|no

May be used to make sure all submodule commits used by the revisions to be pushed are available on a remote-tracking branch. If *check* is used Git will verify that all submodule commits that changed in the revisions to be pushed are available on at least one remote of the submodule. If any commits are missing the push will be aborted and exit with non-zero status. If *on-demand* is used all submodules that changed in the revisions to be pushed will be pushed. If on-demand was not able to push all necessary revisions it will also be aborted and exit with non-zero status. If *only* is used all submodules will be recursively pushed while the superproject is left unpushed. A value of *no* or using `--no-recurse-submodules` can be used to override the push.recurseSubmodules configuration variable when no submodule recursion is required.

--[no-]verify

Toggle the pre-push hook (see githooks(5)). The default is --verify, giving the hook a chance to prevent the push. With --no-verify, the hook is bypassed completely.

-4

--ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6

--ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

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.

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(1) 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>
```

OUTPUT

The output of "git push" depends on the transport method used; this section describes the output when pushing over the Git protocol (either locally or via ssh).

The status of the push 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>)
```

If --porcelain is used, then each line of the output is of the form:

```
<flag> \t <from>:<to> \t <summary> (<reason>)
```

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

flag

A single character indicating the status of the ref:

(space)

for a successfully pushed fast-forward;

`+`

for a successful forced update;

`-`

for a successfully deleted ref;

`*`

for a successfully pushed new ref;

`!`

for a ref that was rejected or failed to push; and

`=`

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

summary

For a successfully pushed 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).

For a failed update, more details are given:

rejected

Git did not try to send the ref at all, typically because it is not a fast-forward and you did not force the update.

remote rejected

The remote end refused the update. Usually caused by a hook on the remote side, or because the remote repository has one of the following safety options in effect: `receive.denyCurrentBranch` (for pushes to the checked out branch), `receive.denyNonFastForwards` (for forced non-fast-forward updates), `receive.denyDeletes` or `receive.denyDeleteCurrent`. See git-config(1).

remote failure

The remote end did not report the successful update of the ref, perhaps because of a temporary error on the remote side, a break in the network connection, or other transient error.

from

The name of the local ref being pushed, minus its `refs/<type>/` prefix. In the case of deletion, the name of the local ref is omitted.

to

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

reason

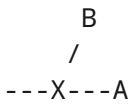
A human-readable explanation. In the case of successfully pushed refs, no explanation is needed. For a failed ref, the reason for failure is described.

NOTE ABOUT FAST-FORWARDS

When an update changes a branch (or more in general, a ref) that used to point at commit A to point at another commit B, it is called a fast-forward update if and only if B is a descendant of A.

In a fast-forward update from A to B, the set of commits that the original commit A built on top of is a subset of the commits the new commit B builds on top of. Hence, it does not lose any history.

In contrast, a non-fast-forward update will lose history. For example, suppose you and somebody else started at the same commit X, and you built a history leading to commit B while the other person built a history leading to commit A. The history looks like this:



Further suppose that the other person already pushed changes leading to A back to the original repository from which you two obtained the original commit X.

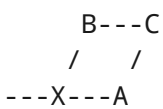
The push done by the other person updated the branch that used to point at commit X to point at commit A. It is a fast-forward.

But if you try to push, you will attempt to update the branch (that now points at A) with commit B. This does *not* fast-forward. If you did so, the changes introduced by commit A will be lost, because everybody will now start building on top of B.

The command by default does not allow an update that is not a fast-forward to prevent such loss of history.

If you do not want to lose your work (history from X to B) or the work by the other person (history from X to A), you would need to first fetch the history from the repository, create a history that contains changes done by both parties, and push the result back.

You can perform "git pull", resolve potential conflicts, and "git push" the result. A "git pull" will create a merge commit C between commits A and B.



Updating A with the resulting merge commit will fast-forward and your push will be accepted.

Alternatively, you can rebase your change between X and B on top of A, with "git pull --rebase", and push the result back. The rebase will create a new commit D that builds the change between X and B on top of A.

```
      B   D
     /   /
---X---A
```

Again, updating A with this commit will fast-forward and your push will be accepted.

There is another common situation where you may encounter non-fast-forward rejection when you try to push, and it is possible even when you are pushing into a repository nobody else pushes into. After you push commit A yourself (in the first picture in this section), replace it with "git commit --amend" to produce commit B, and you try to push it out, because forgot that you have pushed A out already. In such a case, and only if you are certain that nobody in the meantime fetched your earlier commit A (and started building on top of it), you can run "git push --force" to overwrite it. In other words, "git push --force" is a method reserved for a case where you do mean to lose history.

EXAMPLES

`git push`

Works like `git push <remote>`, where <remote> is the current branch's remote (or `origin`, if no remote is configured for the current branch).

`git push origin`

Without additional configuration, pushes the current branch to the configured upstream (`remote.origin.merge` configuration variable) if it has the same name as the current branch, and errors out without pushing otherwise.

The default behavior of this command when no <refspec> is given can be configured by setting the `push` option of the remote, or the `push.default` configuration variable.

For example, to default to pushing only the current branch to `origin` use `git config remote.origin.push HEAD`. Any valid <refspec> (like the ones in the examples below) can be configured as the default for `git push origin`.

`git push origin :`

Push "matching" branches to `origin`. See <refspec> in the OPTIONS section above for a description of "matching" branches.

`git push origin master`

Find a ref that matches `master` in the source repository (most likely, it would find `refs/heads/master`), and update the same ref (e.g. `refs/heads/master`) in `origin` repository with it. If `master` did not exist remotely, it would be created.

`git push origin HEAD`

A handy way to push the current branch to the same name on the remote.

`git push mothership master:satellite/master dev:satellite/dev`

Use the source ref that matches `master` (e.g. `refs/heads/master`) to update the ref that matches `satellite/master` (most probably `refs/remotes/satellite/master`) in the `mothership` repository; do the same for `dev` and `satellite/dev`.

This is to emulate `git fetch` run on the `mothership` using `git push` that is run in the opposite direction in order to integrate the work done on `satellite`, and is often necessary when you can only make connection in one way (i.e. satellite can ssh into mothership but mothership cannot initiate connection to satellite because the latter is behind a firewall or does not run sshd).

After running this `git push` on the `satellite` machine, you would ssh into the `mothership` and run `git merge` there to complete the emulation of `git pull` that were run on `mothership` to pull changes made on `satellite`.

`git push origin HEAD:master`

Push the current branch to the remote ref matching `master` in the `origin` repository. This form is convenient to push the current branch without thinking about its local name.

`git push origin master:refs/heads/experimental`

Create the branch `experimental` in the `origin` repository by copying the current `master` branch. This form is only needed to create a new branch or tag in the remote repository when the local name and the remote name are different; otherwise, the ref name on its own will work.

`git push origin :experimental`

Find a ref that matches `experimental` in the `origin` repository (e.g. `refs/heads/experimental`), and delete it.

`git push origin +dev:master`

Update the origin repository's master branch with the dev branch, allowing non-fast-forward updates. **This can leave unreferenced commits dangling in the origin repository.** Consider the following situation, where a fast-forward is not possible:

```
o---o---o---A---B  origin/master
     \
      X---Y---Z  dev
```

The above command would change the origin repository to

```
      A---B  (unnamed branch)
     /
o---o---o---X---Y---Z  master
```

Commits A and B would no longer belong to a branch with a symbolic name, and so would be unreachable. As such, these commits would be removed by a `git gc` command on the origin repository.

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.

GIT

Part of the git(1) suite

Last updated 2018-12-15 09:30:39 UTC