8.4 Customizing Git - An Example Git-Enforced Policy

An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we'll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can't necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

Server-Side Hook

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

You also have access to the user doing the pushing if the push is being run over SSH. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we'll assume the connecting user is in the `$USER` environment variable, so your update script begins by gathering all the information you need:

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

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information.

So, to get a list of all the commit SHA-1s introduced between one commit SHA-1 and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. We'll go over all these plumbing commands in detail in Git Internals; but for now, here's what that command gives you:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

Change the version number
```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
Change the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Putting that in your `update` script will reject updates that contain commits that have messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the `update` hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (`|`) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc`directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Git Basics):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
```

```
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Testing It Out

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see:

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'Test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'Test [ref: 132]'
[master e05c914] Test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

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

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. You have to change the path to the ACL file from this:

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

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

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

This script uses a syntax that wasn't covered in Revision Selection. You get a list of commits that have already been pushed up by running this:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

The `SHA^@` syntax resolves to all the parents of that commit. You're looking for any commit that is reachable from the last commit on the remote and that isn't reachable from any parent of any of the SHA-1s you're trying to push up – meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don't try to force the push with `-f`, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

prev | next