- Blog
- Documentation

Travis Pro

Travis Pro Frequently Asked Questions

Note: These issues are related to Travis Pro, our hosted continuous integration solution for private repositories.

How can I configure Travis Pro to use private GitHub repositories as dependencies?

By default Travis CI focuses its build efforts around a single repository. We set up a private deploy key for every repository that's enabled to build on Travis Pro and assign that to the repository. That means we can't pull in dependent repositories, e.g. from your Gemfile or for your Composer setup, without some additional setup. The key can only be assigned to the single repository and not be reused throughout all of GitHub.

If you need to pull in Git submodules or dependent repositories, that's still easy to achieve, though. You can either specify the dependencies using GitHub's https URL scheme, which can include username and password, e.g. `https://user:password@github.com/organization/repo.git`. Ideally the user is a separate user in your organization that only has read access to the required repository.

As this doesn't work in all cases, e.g. with Composer, there's an alternative. It still requires setting up a separate user with pull access to the relevant repositories. But instead of specifying the username and passwords in for instance your Gemfile, you add an SSH key specifically for this user and put that key into your .travis.yml. Here are the steps to take:

- Create a new user on GitHub and add it to your organization, give it pull permission to the relevant repositories.
- Create an SSH key for the user: `ssh-keygen -f id_mustache_power`. Note: this must be a password-less key for Travis CI to be able to use it.
- Add the SSH key to the user on GitHub.
- Add a Base64-encoded version of the private key to your .travis.yml. On the Mac, you can run `base64 id_mustache_power | pbcopy` to copy the encoded version of the key to the clipboard.
- Add a new key to your main repositories' .travis.yml file: `source_key:`. The value of the key should be the encoded version of the SSH key in double quotes and on a single line.

Travis CI will automatically prefer the key specified in your .travis.yml over the deploy key for a repository, so all repositories you're pulling in for the build to succeed automatically use this key.

Note: This only works on Travis Pro, the open source version of Travis CI doesn't support private repositories.

Technicalities

These steps are necessary due to the way Git and the underlying transport SSH authenticate with a remote service. For example, when you specify more than one key that you want SSH to use, it will still use the first one that authenticates successfully. After authentication, GitHub checks if this key is authorized to access the repository requested. Given that you rely on SSH to use the second key you have configured (e.g. by adding it to an ssh-agent), it will fail because it will try the first one every time, successfully authenticate, but fail to authorize access, failing the entire git clone operation.

Can I use pull request testing on Travis Pro?

Yes, you can. It's enabled by default for all repositories set up on Travis CI. See the blog post accompanying the launch of pull requests for Travis CI.

How can I encrypt files that include sensitive data?

Some customers have files in their repositories that contain sensitive data, like passwords or API credentials, all of which are important for your build to succeed. But it's data you don't want to give everyone access to. You still want to be able to give the

source code to folks on your team without giving them access to this kind of data.

While we're working on a solution that allows you to encrypt sensitive data as environment variables, there's a simple trick you can deploy on Travis Pro in the meantime. Luke Patterson came up with this little trick, thanks for sharing it!

It utilizes the private key we generate for each repository so we can clone the code on our build machines. That key is kept on Travis CI's end only, so no external party has access to it. What you do have access to is the public key on GitHub. You can download that public key and make it part of your project to keep it around. Due to the nature of asymmetric key cryptography, though, the file needs to be encrypted with a symmetric key (e.g. using AES 256), and then the secret used to encrypt that file is encrypted using the public key.

After data is encrypted locally you can store the files in Git and run a set of commands to decrypt it on the continuous integration machine, using the SSH private key that we already store on the machine.

Below are the steps required to encrypt and decrypt data.

- Download the key from GitHub and store it in a file. E.g. `id_travis.pub`. Unfortunately, you can't get the public key from the user interface, but you can fetch it via the API: `curl -u <username> https://api.github.com/repos/<username>/<repo>/keys` Look for a key named travis-ci.com in the JSON output and copy the string that contains the public key into a file `id_travis.pub`. Here's a handy one-liner that does it for you: `curl -u <username> https://api.github.com/repos/<username>/<repo>/keys | grep -B 4 travis-ci\\.com | grep '"key":' | perl -pe 's/^[ ]+"key": //; s/^"//; s/",$//' > id_travis.pub`
- Extract a public key certificate from the public key: `ssh-keygen -e -m PKCS8 -f id_travis.pub > id_travis.pub.pem`
- Encrypt a file using a passphrase generated from a SHA hash of /dev/urandom output:

```
password=`cat /dev/urandom | head -c 10000 | openssl sha1`
openssl aes-256-cbc -k "$password" -in config.xml -out config.xml.enc -a
```

- Now you can encrypt the key, let's call it `secret`: `echo "$password" | openssl rsautl -encrypt -pubin -inkey id_travis.pub.pem -out secret`
- Add the encrypted file and the secret to your Git repository.
- For the build to decrypt the file, add a `before_script` section to your `.travis.yml` that runs the opposite command of the above:

```
before_script:
  - secret=`openssl rsautl -decrypt -inkey ~/.ssh/id_rsa -in secret`
  - openssl aes-256-cbc -k "$secret" -in config.xml.enc -d -a -out config.xml
```

It must be noted that this scenario is still not perfectly secure. While it prevents project collaborators being able to access sensitive data on a daily basis, a malicious collaborator could tamper with the build scripts to output the sensitive data to your build log. The upshot is that you'll know who's responsible for this from the commit history.

Combine Encryption and Deploy Keys For Private Dependencies for Extra Strength

You can combine the encryption outlined above with build dependencies that are private GitHub repositories. The approach outlined above, putting the deploy key into your `.travis.yml` might not be a favorable solution for everyone.

What you can do instead is generate a custom deploy key using `ssh-keygen`, assign it to a user, and then encrypt the private key using the encryption scheme outlined above. Instead of encrypting a config.xml, you encrypt a `id_private` key and store it in your repository together with the secret.

Then you replace the SSH key currently registered with the SSH agent running on the virtual machine with this new key in your .travis.yml. Below are the additional steps that need to be added to a `before_install` or `before_script` step:

```
before_install:
  - secret=`openssl rsautl -decrypt -inkey ~/.ssh/id_rsa -in secret`
  - openssl aes-256-cbc -k $secret -in id_private.enc -d -a -out id_private
  - ssh-add -D
  - chmod 600 id_private
  - ssh-add ./id_private
```

This way, the deploy key is never exposed to parties you don't want it exposed to. The same note as with the encryption scheme applies here too: when a malicious party tampers with your build script, the key could still be exposed.

It also must be noted that subsequent accesses to the original processes, should they be required by your build, won't be possible without assigning the secondary deploy key to the original repository as well.

Search

Newsletter

email address Subscribe

Travis CI for Private Repositories

Guides

- Getting started
- Validate your .travis.yml
- Services & Database setup
- Headless Testing with Browsers
- Skipping a Build
- Speeding up the build
- Caching Dependencies

Reference

- Build configuration
- Notifications
- Encrypting Sensitive Data
- Build Status Images
- The CI Environment
- The OS X CI environment
- Common Build Problems
- Travis Pro
- Command Line Client
- GitHub Permissions used by Travis CI

Services and Databases

- Using PostgreSQL
- Firefox
- Custom Host Names

Integrations

- Code Climate
- Sauce Labs
- Atom Feeds
- CCMenu / CCTray Feeds

Language-specific Guides

- C
- C++
- Clojure
- Erlang
- Go
- Groovy
- Haskell

- Java
- JavaScript (with Node.js)
- Objective-C
- Perl
- PHP
- Python
- Ruby
- Scala

Deployment

- Overview
- Appfog
- Heroku
- Modulus
- Nodejitsu
- Engine Yard
- OpenShift
- cloudControl
- RubyGems
- OpsWorks
- PyPI
- npm
- S3
- Divshot.io
- Rackspace Cloud Files
- Custom Script
- Other Providers

Developer Guides

- The API
- The Ruby Library

3rd Party Tools and Resources

- Browser Extensions
- Links & Resources
- Coverity Scan

Contact

- Twitter
- IRC
- Mailing list
- GitHub
- Blog Feed

Contributing

This documentation site is open source. Feel free to file issues about it. The README in our Git repository explains how to contribute.

Legal

- Imprint

© 2014 Travis CI GmbH,
Prinzessinnenstr. 20, 10969 Berlin, Germany

- Email
- Live Chat
- Docs
- Status