ansible-sign Documentation

Release 0.1.1

Red Hat, Inc.

Oct 31, 2022

CONTENTS

This is the documentation for the **ansible-sign** utility used for signing and verifying Ansible content.

CHAPTER

ONE

CONTENTS

1.1 License

1.2 Contributors

- Rick Elrod <relrod@redhat.com>

1.3 Changelog

1.3.1 Version 1.0.0

- Initial library and CLI for `ansible-sign`. See documentation for usage examples. Only the CLI is officially supported, and the API can change over time. We make no effort to provide backwards compatibility at the API level at this time.

1.4 Rundown of `ansible-sign` (CLI) usage

For Ansible Automation Platform content developers (project maintainers), the primary and supported way of using **ansible-sign** is through the command-line interface that comes with it.

The command-line interface aims to make it easy to use cryptographic technology like GPG to validate that specified files within a project have not been tampered with in any way.

Though in the future other means of signing and validating might be supported, GPG is the only currently supported means of signing and validation. As such, the rest of this tutorial assumes the use of GPG.

The process of creating a GPG public/private keypair for signing content is well documented online, such as in this Red Hat "Enable Sysadmin" blog post. As such, we will assume that you have a valid GPG keypair already available and in your default GnuPG keyring.

You can verify that you have a keypair with the following command:

Listing 1: Verifying that a valid secret GPG key exists for signing content

```
$ gpg --list-secret-keys
```

If the above command produces no output, or one line of output that says that a "trustdb" was created, then you do not have a secret key in your default keyring. In this case, refer to the aforementioned blog post to learn how to create a new keypair.

If it produces output other than that, then you have a valid secret key and are ready to move on to *using ansible-sign*.

1.4.1 Adding a GPG key to AWX or Ansible Automation Controller

In the command line, run the following commands:

```
$ gpg --list-keys
$ gpg --export --armour <key fingerprint> > my_public_key.asc
```

1. In AWX/Automation Controller, click "Credentials" then the "Add" button
2. Give the new credential a meaningful name (for example, "infrastructure team public GPG key")
3. For "Credential Type" select "GPG Public Key"
4. Click "Browse" to navigate to and select the file that you created earlier (`my_public_key.asc`)
5. Finally, click the "Save" button to finish

This credential can now be selected in "Project" settings. Once selected, content verification will automatically take place on future project syncs.

Vist the GnuPG documentation for more information regarding GPG keys. For more information regarding generating a GPG keypair, visit the Red Hat "Enable Sysadmin" blog post.

1.4.2 How to Access the `ansible-sign` CLI Utility

Run the following command to install `ansible-sign`:

Listing 2: Installing `ansible-sign`

```
$ pip install ansible-sign
```

Once it's installed, run:

Listing 3: Verify that `ansible-sign` was successfully installed.

```
$ ansible-sign --version
```

You should see output similar to the following (possibly with a different version number):

Listing 4: The output of `ansible-sign --version`

```
ansible-sign 0.1
```

Congratulations! You have successfully installed `ansible-sign`!

1.4.3 The Project Directory

We will start with a simple Ansible project directory. The Ansible documentation goes into more sophisticated examples of project directory structures.

In our sample project, we have a very simple structure. An `inventory` file, and two small playbooks under a `playbooks` directory.

Listing 5: Our sample project

```
$ cd sample-project/
$ tree -a .
.
├── inventory
└── playbooks
    ├── get_uptime.yml
    └── hello.yml

1 directory, 3 files
```

Note: Future commands that we run will assume that your Working Directory is the root of your project. `ansible-sign project` commands, as a rule, always take the project root directory as their last argument, thus we will simply use `.` to indicate the current Working Directory.

1.4.4 Signing Content

The way that `ansible-sign` protects content from tampering is by taking checksums (sha256) of all of the secured files in the project, compiling those into a checksum manifest file, and then finally signing that manifest file.

Thus, the first step toward signing content is to create a file that tells `ansible-sign` which files to protect. This file should be called `MANIFEST.in` and live in the project root directory.

Internally, `ansible-sign` makes use of the `distlib.manifest` module of Python's distlib library, and thus `MANIFEST.in` must follow the syntax that this library specifies. The Python Packaging User Guide has an explanation of the MANIFEST.in file directives.

For our sample project, we will include two directives. Our `MANIFEST.in` will look like this:

Listing 6: `MANIFEST.in`

```
include inventory
recursive-include playbooks *.yml
```

With this file in place, we can generate our checksum manifest file and sign it. These steps both happen in a single `ansible-sign` command.

Listing 7: Generating a checksum manifest file and signing it

```
$ ansible-sign project gpg-sign .
[OK   ] GPG signing successful!
[NOTE ] Checksum manifest: ./.ansible-sign/sha256sum.txt
[NOTE ] GPG summary: signature created
```

Congratulations, you've now signed your first project!

Notice that the `gpg-sign` subcommand lives under the `project` subcommand. For signing project content, every command will start with `ansible-sign project`. As noted above, as a rule, every `ansible-sign project` command takes the project root directory as its final argument.

Hint: As mentioned earlier, `ansible-sign` by default makes use of your default keyring and looks for the first available secret key that it can find, to sign your project. You can specify a specific secret key to use with the `--fingerprint`

option, or even a completely independent GPG home directory with the `--gnupg-home` option.

Note: If you are using a desktop environment, GnuPG will automatically pop up a dialog asking for your secret key's passphrase. If this functionality does not work, or you are working without a desktop environment (e.g., via SSH), you can use the `-p`/`--prompt-passphrase` flag after `gpg-sign` in the above command, which will cause `ansible-sign` to prompt for the password instead.

If we now look at the structure of the project directory, we'll notice that a new `.ansible-sign` directory has been created. This directory houses the checksum manifest and a detached GPG signature for it.

Listing 8: Our sample project after signing

```
$ tree -a .
.
├── .ansible-sign
│   ├── sha256sum.txt
│   └── sha256sum.txt.sig
├── inventory
├── MANIFEST.in
└── playbooks
    ├── get_uptime.yml
    └── hello.yml
```

1.4.5 Verifying Content

If you come in contact with a signed Ansible project and want to verify that it has not been altered, you can use `ansible-sign` to check both that the signature is valid and that the checksums of the files match what the checksum manifest says they should be. In particular, the `ansible-sign project gpg-verify` command can be used to automatically verify both of these conditions.

Listing 9: Verifying our sample project

```
$ ansible-sign project gpg-verify .
[OK   ] GPG signature verification succeeded.
[OK   ] Checksum validation succeeded.
```

Hint: Once again, by default `ansible-sign` makes use of your default GPG keyring to look for a matching public key. You can specify a keyring file with the `--keyring` option, or a different GPG home with the `--gnugpg-home` option.

If verification fails for any reason, some information will be printed to help you debug the cause. More verbosity can be enabled by passing the global `--debug` flag, immediately after `ansible-sign` in your commands.

1.4.6 Notes About Automation

In environments with highly-trusted CI environments, it is possible to automate the signing process. For example, one might store their GPG private key in a GitHub Actions secret, and import that into GnuPG in the CI environment. One could then run through the signing workflow above within the normal CI workflow/container/environment.

When signing a project using GPG, the environment variable `ANSIBLE_SIGN_GPG_PASSPHRASE` can be set to the passphrase of the signing key. This can be injected (and masked/secured) in a CI pipeline.

`ansible-sign` will return with a different exit-code depending on the scenario at hand, both during signing and verification. This can also be useful in the context of CI and automation, as a CI environment can act differently based on the failure (for example, sending alerts for some errors but silently failing for others).

These codes are used fairly consistently within the code, and can be considered stable:

Table 1: Status codes that `ansible-sign` can exit with

Exit code	Approximate meaning	Example scenarios
0	Success	• Signing was successful • Verification was successful
1	General failure	• The checksum manifest file contained a syntax error during verification • The signature file did not exist during verification • `MANIFEST.in` did not exist during signing
2	Checksum verification failure	• The checksum hashes calculated during verification differed from what was in the signed checksum manifest. (That is, a project file was changed but the signing process was not recompleted.)
3	Signature verification failure	• The signer's public key was not in the user's GPG keyring • The wrong GnuPG home directory or keyring file was specified • The signed checksum manifest file was modified in some way
4	Signing process failure	• The signer's private key was not found in the GPG keyring • The wrong GnuPG home directory or keyring file was specified

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search