Scriptworker Documentation

Release 6.0.1

Aki Sasaki

Jan 04, 2018

Table of Contents

Scriptworker implements the TaskCluster worker model, then launches a pre-defined script.

This worker was designed for Releng processes that need specific, limited, and pre-defined capabilities.

Free software: MPL2 license

CHAPTER 1

Usage

- Create a config file. By default scriptworker will look in `./scriptworker.yaml`, but this config path can be specified as the first and only commandline argument. There is an example config file, and all config items are specified in scriptworker.constants.DEFAULT_CONFIG.

Credentials can live in `./scriptworker.yaml`, `./secrets.json`, `~/.scriptworker`, or in environment variables: `TASKCLUSTER_ACCESS_TOKEN`, `TASKCLUSTER_CLIENT_ID`, and `TASKCLUSTER_CERTIFICATE`.

- Launch: `scriptworker [config_path]`

CHAPTER 2

Testing

Note: GPG tests require gpg 2.0.x!

Without integration tests, install tox, then

`NO_TESTS_OVER_WIRE=1 tox -e py35`

With integration tests, first create a client with the `assume:project:taskcluster:worker-test-scopes` scope.

Then create a `./secrets.json` or `~/.scriptworker` that looks like:

```
{
    "integration_credentials": {
        "clientId": "...",
        "accessToken": "...",
        "certificate": "..."
    }
}
```

(`certificate` is only specified if using temp creds)

then

`tox`

It's also possible to create a `./secrets.json` as above, then:

```
cp docker/Dockerfile.test Dockerfile
docker build -t scriptworker-test . && docker run scriptworker-test tox
```

2.1 GPG Homedir testing

Sometimes it's nice to be able to test things like `rebuild_gpg_homedirs`. To do so:

```
cp docker/Dockerfile.gnupg Dockerfile
docker build -t scriptworker-gpg . && docker run -i scriptworker-gpg bash -il
# in the docker shell,
rebuild_gpg_homedirs scriptworker.yaml
```

2.1.1 Maintenance

For sheriffs, release/relops, taskcluster, or related users, this page describes maintenance for scriptworkers.

Last modified 2016.11.16.

New docker shas

For chain of trust verification, we verify the docker shas that we run in docker-worker.

For some tasks, we build the docker images in docker-image tasks, and we can verify the image's sha against docker-image task's output.

However, for decision and docker-image tasks, we download the docker image from docker hub. We allowlist the shas to make sure we are running valid images.

We specify those here. However, if we only specified them in `scriptworker.constants`, we'd have to push a new scriptworker release every time we update this allowlist. So we override this list here.

For now, we need to keep both locations updated. Puppet governs production instances, and the scriptworker repo is used for scriptworker development, and a full allowlist is required for chain of trust verification.

Chain of Trust settings

As above, other chain of trust settings live in constants.py. However, if we only specified them in `scriptworker.constants`, we'd have to push a new scriptworker release every time we update them. So we can override them here.

Ideally we keep the delta small, and remove the overrides in puppet when we release a new scriptworker version that updates these defaults. As currently written, each scriptworker instance type will need its scriptworker version bumped individually.

GPG keys

For gpg key maintenance, see the chain of trust docs

2.1.2 Scriptworker Releases

These are the considerations and steps for a new scriptworker release.

Code changes

Ideally, code changes should follow clean architecture best practices

When adding new functions, classes, or files, or when changing function arguments, please add or modify the docstrings.

Tests and test coverage

Scriptworker has 100% test coverage, and we'd like to keep it that way.

Run tests locally via tox to make sure the tests pass, and we still have 100% coverage.

Versioning

Scriptworker follows semver. Essentially, increment the

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add API functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Changelog

Update the changelog before making a new release.

Release files

If you're changing any dependencies, please update `requirements-dev.txt`, `requirements-test-dev.txt`, and `setup.py`.

If you add change the list of files that need to be packaged (either adding new files, or removing previous packaged files), modify `MANIFEST.in`.

Requirements

It's good practice to keep `requirements-prod.txt` and `requirements-test-prod.txt` up to date. To do so:

```
    # Using the local venv python>=3.5,
    pip install dephash
    dephash gen requirements-dev.txt > requirements-prod.txt
    dephash gen requirements-test-dev.txt > requirements-test-prod.txt
```

A `git diff` will then show what has changed in the scriptworker dependencies since the last time dephash was run. Assuming we only add the new dependencies when tox is green, we have a last-known-good set of dependencies. Add these to the list of changes to commit.

Versioning

Modify `scriptworker/version.py` to set the `__version__` to the appropriate tuple. This is either a 3- or 4-part tuple, e.g.

```
# 0.10.0a1
__version__ = (0, 10, 0, "alpha1")

# 1.0.0b2
__version__ = (1, 0, 0, "beta2")
```

```
# 0.9.3
__version__ = (0, 9, 3)
```

Then run `version.py`:

```
# Using the local venv python>=3.5,
python scriptworker/version.py
```

This will update `version.json`. Verify both files look correct.

Tagging

To enable gpg signing in git,

1. you need a gpg keypair
2. you need to set your `user.signingkey` in your `~/.gitconfig` or `scriptworker/.git/config`
3. If you want to specify a specific gpg executable, specify your `gpg.program` in your `~/.gitconfig` or `scriptworker/.git/config`

Tag and sign!

```
    # make sure you've committed your changes first!
    VERSION=0.9.0
    git tag -s $VERSION -m"$VERSION"
```

Push!

```
    # By default this will push the new tag to origin; make sure the tag gets pushed␣
↪to
    # mozilla-releng/scriptworker
    git push --tags
```

Pypi

Someone with access to the scriptworker package on `pypi.python.org` needs to do the following:

```
    # from https://packaging.python.org/tutorials/distributing-packages/#uploading-
↪your-project-to-pypi
    # Don't use `python setup.py register` or `python setup.py upload`; this may use
    # cleartext auth!
    # Using a python with `twine` in the virtualenv:
    VERSION=4.1.2
    # create the source tarball
    python setup.py sdist
    # sign the source tarball
    gpg --detach-sign -a dist/scriptworker-${VERSION}.tar.gz
    # upload the source tarball + signature
    twine upload dist/scriptworker-${VERSION}.tar.gz{,.asc}
```

That creates source tarball, and uploads it.

Puppet

Connect to Releng VPN.

Copy the tarball from `dist/` to `releng-puppet2.srv.releng.scl3.mozilla.com`:

```
scp dist/scriptworker-$VERSION.tar.gz releng-puppet2.srv.releng.scl3.mozilla.com:
ssh releng-puppet2.srv.releng.scl3.mozilla.com
cd /data/python/packages-3.5
sudo mv ~/scriptworker-$VERSION.tar.gz .
```

Bump the scriptworker version in the appropriate scriptworker instance puppet configs.

If desired, test in your user environment first. Otherwise, get review, land, and merge to production.

2.1.3 Adding new scriptworker instance types

This doc describes when and how to add a new scriptworker instance type, e.g. signing, pushapk, beetmover, balrog.

Last updated 2016.11.18

Is scriptworker the right tool?

Scriptworker is designed to run security-sensitive tasks with limited capabilities.

- Does this task require elevated privileges or access to sensitive secrets to run?
- Is this task sufficiently important to spin up and maintain a new pool of workers?
- Is the expected load sufficiently contained so we don't require a dynamically sizable pool of workers?

If you answered yes to the above, scriptworker may be a good option.

Chain of Trust considerations

If this is for a new task type in a product/graph that already has scriptworker tasks and chain of trust verification, then adding a new task should be an incremental change.

If this is for a new graph type or new product, and the graph doesn't look like the Firefox graph, there may be significant changes required to support the chain of trust. This is an important consideration when choosing your solution.

Creating a new scriptworker instance type

Once you've decided to use scriptworker, these are the steps to take to implement.

Write a script

This can be a script in any language that can be called from the commandline, although we prefer async python 3. This is standalone, so it's possible to develop and test this script without scriptworker.

Single purpose, but generic

The script should aim to support a single purpose, like signing or pushing updates. However, ideally it's generic, so it can sign a number of different file types or push various products to various accounts, given the right config and creds.

Commandline args

Currently, we call the script from scriptworker with the commandline

```
# e.g., ["python", "/path/to/script.py", "/path/to/script_config.json"]
[interpreter, script, config]
```

Where `interpreter` could be python3, `script` is the path to the script, and `config` is the path to the runtime configuration, which doesn't change between runs.

Config

The config could be anything you need the script to know, including paths to other config files. These config items must be specified, and must match the eventual scriptworker config:

- `work_dir`: this is an absolute path. This directory is deleted after the task and recreated before the next task. Scriptworker will place files in here for the script's consumption.
- `artifact_dir`: this is where to put the artifacts that scriptworker will upload to taskcluster. The directory layout will look like the directory layout in taskcluster, e.g. `public/build/target.apk` or `public/logs/foo.log`

Task

Scriptworker will place the task definition in `$work_dir/task.json`. The script can read this task definition and behave accordingly.

When testing locally without scriptworker, you can create your own `task.json`.

task.payload.upstreamArtifacts

If the task defines `payload.upstreamArtifacts`, these are artifacts for scriptworker to download and verify their shas against the chain of trust.

`payload.upstreamArtifacts` currently looks like:

```
[{
  "taskId": "upstream-task-id1",
  "taskType": "build",
  "paths": ["public/artifact/path1", "public/artifact/path2"],
  "formats": []
}, {
  ...
}]
```

It will download them into `$artifact_dir/public/cot/$upstream-task-id/$path`.

Scopes

Taskcluster scopes are its ACLs: restricted behavior is placed behind scopes, and only those people and processes that need access to that behavior are given those scopes. With the Chain of Trust, we can verify that restricted scopes can only be used in specific repos.

If your script is going to have different levels of access (e.g., CI- signing, nightly- signing, and release- signing), then it's best to put them each behind a different scope, and use that scope for determining which credentials to use.

Deployment considerations

You don't have to address the below during script development, but it may be helpful to know some of the considerations that will affect deployment.

Graph

We need to trace upstream tasks back to the tree. We're able to find our decision task by the `taskGroupId`, but other dependencies we need to either use *`upstreamArtifacts`* or `task.extra.chainOfTrust.inputs`, which looks like

```
"inputs": {
  "docker-image": "docker-image-taskid"
}
```

If there are upstream tasks that depend on the output of other tasks, make sure all of them are connected via at least one of these two data structures.

Puppet

There is a scriptworker module that we should use for new Firefox scriptworker instances, following the signing scriptworker example.

For other products, we can either support them within MoCo Releng, or we can spin up a new parallel set of scriptworker pools to keep the secrets and access separate. If we choose the latter, any deployment solution is acceptable: ansible, puppet, nix, ...

Hiera

For new puppet instances, we need to add new gpg keypairs in hiera, similar to when adding a new instance of an existing scriptworker type.

2.1.4 Adding new scriptworker instances of an existing type

We don't yet have a scriptworker provisioner, so spinning up new instances of a specific type is still a manual process that can definitely use improvement. Here are docs on how to spin a new instance up.

signing scriptworker

gpg keypair

If this is a chain of trust enabled scriptworker, you'll need to generate a gpg keypair. Otherwise (dev or dep scriptworker), skip to the next step.

Generate and sign a gpg keypair for cltsign@**fqdn**, per these docs.

The pubkey will need to land in the cot-gpg-keys repo, in the `scriptworker/valid` directory. The keypair will need to go into puppet hiera, as specified below.

aws

For a signing scriptworker instance, find a valid signing-range IP and add to dns, like the slave loan. These will be in similar subnets to the existing instances:

```
10.134.30.12 signing-linux-1.srv.releng.use1.mozilla.com
10.132.30.46 signing-linux-2.srv.releng.usw2.mozilla.com
10.134.30.125 signing-linux-3.srv.releng.use1.mozilla.com
10.132.30.82 signing-linux-4.srv.releng.usw2.mozilla.com
```

Go to the EC2 console, go to the appropriate region (usw2, use1).

- Instances -> Launch Instance -> My AMIs -> `centos-65-x86_64-hvm-base-2015-08-28-15-51` -> Select
- t2-micro -> configure instance details
- change the subnet to the `signing` subnet; add a public IP; specify the DNS IP at the bottom -> Add storage
- General purpose SSD -> Tag Instance
- Tag with its name, e.g. signing-linux-5 -> Configure security group
- Select an existing group; choose the `signing-worker` group; review and launch
- make sure to choose a keypair you have access to, e.g. aws-releng or generate your own keypair. Puppet will overwrite this.

puppet

If this is a chain of trust enabled scriptworker, add the gpg keypair into hiera. This will be the `scriptworker_gpg_private_keys` and `scriptworker_gpg_public_keys` dictionaries. The dictionary key is the instance fqdn; the value is the encrypted file.

ssh into the instance as root, using the ssh keypair you specified above.

Install puppet:

```
# first, edit /etc/yum-local.cfg to replace `puppetagain.*/data/` with `releng-
↪puppet2.srv.releng.scl3.mozilla.com`
yum -c /etc/yum-local.cfg install puppet
```

Then puppetize (you need the deploy pass for this):

```
# change the hostname so the cert matches
hostname FQDN
# grab puppetize.sh and run it
wget https://hg.mozilla.org/build/puppet/raw-file/tip/modules/puppet/files/
↪puppetize.sh
# if we're doing a standard puppetize
PUPPET_SERVER=releng-puppet2.srv.releng.scl3.mozilla.com sh puppetize.sh
# if we want to puppetize against an environment
PUPPET_SERVER=releng-puppet2.srv.releng.scl3.mozilla.com PUPPET_EXTRA_OPTIONS="--
↪environment=USER" sh puppetize.sh
# run puppet
puppet agent --test
```

It is probably best to reboot after puppetizing. After this point, it should Just Work.

2.1.5 Chain of Trust

Overview

Taskcluster is versatile and self-serve, and enables developers to make automation changes without being blocked on other teams. In the case of developer testing and debugging, this is very powerful and enabling. In the case of release automation, the ability to schedule arbitrary tasks with arbitrary configs can present a security concern.

The chain of trust is a second factor that isn't automatically compromised if scopes are compromised. This chain allows us to trace a task's request back to the tree.

High level view

Scopes are how Taskcluster controls access to certain features. These are granted to roles, which are granted to users or LDAP groups.

Scopes and their associated Taskcluster credentials are not leak-proof. Also, by their nature, more people will have restricted scopes than you want, given any security-sensitive scope. Without the chain of trust, someone with release-signing scopes would be able to schedule any arbitrary task to sign any arbitrary binary with the release keys, for example.

The chain of trust is a second factor. The embedded GPG keys on the workers are either the something you have or the something you are, depending on how you view the taskcluster workers.

Each chain-of-trust-enabled taskcluster worker generates and signs chain of trust artifacts, which can be used to verify each task and its artifacts, and trace a given request back to the tree.

The scriptworker nodes are the verification points. Scriptworkers run the release sensitive tasks, like signing and publishing releases. They verify their task definitions, as well as all upstream tasks that generate inputs into their task. Any broken link in the chain results in a task exception.

In conjunction with other best practices, like separation of roles, we can reduce attack vectors and make penetration attempts more visible, with task exceptions on release branches.

Chain of Trust Artifact Generation

Each chain-of-trust-enabled taskcluster worker generates and uploads a chain of trust artifact after each task. This artifact contains details about the task, worker, and artifacts, and is signed by the embedded GPG key.

Embedded GPG keys

Each supported taskcluster `workerType` has an embedded gpg keypair. These are the second factor.

`docker-worker` has the gpg privkey embedded in the AMI, inaccessible to tasks run inside the docker container. The gpg keypair is unique per AMI.

`generic-worker` can embed the gpg privkey into the AMI for EC2 instances, or into the system directories for hardware. This are permissioned so the task user doesn't have access to it.

`taskcluster-worker` will need the ability to embed a privkey when we start using them for tier1 tasks in production.

Chain-of-Trust-enabled `scriptworker` workers each have a unique gpg keypair.

For `docker-worker`, `generic-worker`, and `taskcluster-worker`, we have a set of pubkeys that are valid per worker implementation. For `scriptworker`, we have a set of trusted gpg keys; each `scriptworker` gpg pubkey is signed by a trusted gpg key.

Chain of Trust artifacts

After the task finishes, the worker creates a chain of trust json blob, gpg signs it, then uploads it as `public/chainOfTrust.json.asc`. It looks like

```
{
  "artifacts": {
    "path/to/artifact": {
      "sha256": "abcd1234"
    },
    ...
  },
  "chainOfTrustVersion": 1,
  "environment": {
    # worker-impl specific stuff, like ec2 instance id, ip
  },
  "runId": 0,
  "task": {
    # task defn
  },
  "taskId": "...",
  "workerGroup": "...",
  "workerId": "..."
}
```

- The v1 chain-of-trust json artifact schema is viewable here.
- This is a real `example artifact`.

Chain of Trust Verification

Currently, only chain-of-trust-enabled scriptworker instances verify the chain of trust. These are tasks like signing, publishing, and submitting updates to the update server. If the chain of trust is not valid, scriptworker kills the task before it performs any further actions.

The below is how this happens.

Decision Task

The decision task is a special task that generates a taskgraph, then submits it to the Taskcluster queue. This graph contains task definitions and dependencies. The decision task uploads its generated graph json as an artifact, which can be inspected during chain of trust verification.

Ideally, we would be able to verify the decision task's task definition matches the in-tree settings for its revision; that's bug 1328719. Currently we make do with task inspection and an allowlist of docker image shas that the decision task can run on.

GPG homedir management

The chain of trust artifacts are signed, but without marking the gpg public key as valid, we don't know if it's been signed by a valid worker key.

We have a github repo of pubkeys. The latest valid commit is tagged and signed with a trusted gpg key. More on this in gpg-key-management.

Each scriptworker instance

- gets the set of trusted gpg pubkeys from puppet,
- imports them into `~/.gnupg`,
- and signs them with their private gpg key, so we can validate the git commit signatures.
- we update to the latest valid-signed tag, and regenerate the worker-implementation gpg homedirs if we're on a new git revision.

Then it builds a gpg homedir per worker implementation type (`generic-worker`, `docker-worker`, `taskcluster-worker`, `scriptworker`). Each has a corresponding directory in the git repo.

Each gpg homedir is separate from the others, so malicious or outdated keys can only affect the security of that single worker implementation.

The logic for gpg homedir creation is as follows:

`flat` directories

The Taskcluster-team-maintained worker implementations use the flat directory type, to reduce maintenance overhead.

For flat directories, scriptworker imports all pubkeys from the corresponding directory, and signs them to mark the pubkeys as valid. This allows us to verify the signature on the signed chain of trust json artifacts.

`signed` directories

This is currently only for scriptworker.

- scriptworker imports all pubkeys in the `trusted/` subdirectory, signs them, and marks them as trusted.
- scriptworker imports all pubkeys in the `valid/` subdirectory. They should already be signed by one of the keys in the `trusted/` subdirectory, so scriptworker doesn't otherwise sign or mark them as valid.

Building the chain

First, scriptworker inspects the [signing/balrog/pushapk/beetmover] task that it claimed from the Taskcluster queue. It adds itself and its decision-task to the chain.

Any task that generates artifacts for the scriptworker then needs to be inspected. For scriptworker tasks, we have `task.payload.upstreamArtifacts`, which looks like

```
[{
  "taskId": "upstream-task-id",
  "taskType": "build",  # for cot verification purposes
  "paths": ["path/to/artifact1", "path/to/artifact2"],
  "formats": ["gpg", "jar"]  # This is signing-specific for now; we could make␣
↪formats optional, or use it for other task-specific info
}, {
  ...
}]
```

We add each upstream `taskId` to the chain, with corresponding `taskType` (we use this to know how to verify the task).

For each task added to the chain, we inspect the task definition, and add other upstream tasks:

- if the decision task doesn't match, add it to the chain.
- docker-worker tasks have `task.extra.chainOfTrust.inputs`, which is a dictionary like `{"docker-image": "docker-image-taskid"}`. Add the docker image `taskId` to the chain (this will likely have a different decision `taskId`, so add that to the chain).

Verifying the chain

Scriptworker:

- downloads the chain of trust artifacts for each upstream task in the chain, and verifies their signatures. This requires detecting which worker implementation each task is run on, to know which gpg homedir to use. At some point in the future, we may use `workerType` to worker implementation mappings.
- downloads each of the `upstreamArtifacts` and verify their shas against the corresponding task's chain of trust's artifact shas. the downloaded files live in `cot/TASKID/PATH` , so the script doesn't have to re-download and re-verify.
- downloads each decision task's `task-graph.json`. For every *other* task in the chain, we make sure that their task definition matches a task in their decision task's task graph. There's some fuzzy matching going on here, to allow for datestring changes, as well as retriggering, which results in a new `taskId`.
- verifies each decision task command and `workerType`, and makes sure its docker image sha is in the allowlist.
- verifies each docker-image task command and docker image sha against the allowlist, until we resolve bug 1328719. Every other docker-worker task downloads its image from a previous docker-image task, so these two allowlists help us verify every docker image used by docker-worker.
- verifies each docker-worker task's docker image sha.
- makes sure the `interactive` flag isn't on any docker-worker task.
- determines which repo we're building off of.
- matches its task's scopes against the tree; restricted scopes require specific branches.

Once all verification passes, it launches the task script. If chain of trust verification fails, it exits before launching the task script.

Chain of Trust GPG Key Management

GPG key management is a critical part of the chain of trust. There are several types of gpg keys:

- [taskcluster team] worker keys, which are unsigned pubkeys for docker- and generic- workers.
- [releng team] scriptworker keys, which are signed pubkeys for scriptworkers.
- [releng team] scriptworker trusted keys, which are the pubkeys of releng team members who are allowed to generate and sign scriptworker keys.
- [various] git commit signing keys. We keep the above pubkeys in a git repo, and we sign the commits. These are the pubkeys that are allowed to sign the git commits.

Adding new git commit signing gpg keys

To update the other pubkeys, we need to be able to add them to the git repo. We add the new pubkeys in two places: add the long keyid in-repo, and add the pubkey itself in puppet

Adding new worker gpg keys

New worker gpg keys should be committed to the repo with signed commits. Only certain people can sign the commits, as per *above*.

new docker and generic worker gpg keys

When generating a new AMI or image, the docker and generic workers generate a new gpg keypair. The Taskcluster team has the option of recording the public key and adding it to the repo.

The pubkeys for build, decision, and docker-image workerTypes should be added to the repo, with a signed tag per the readme.

new scriptworker gpg keys

First, you will need access to a trusted key (The trusted keys are in the scriptworker/trusted dir. That may mean someone else needs to generate the keys, or you may petition for access to create and sign these keys. (To do so, update the trusted keys with a new pubkey, sign that commit with a trusted git commit key, and merge. If you don't have a trusted git key, see *adding new git commit signing gpg keys*.)

Once you have access to a trusted key, generate new gpg keypairs for each host. The email address will be `username@fqdn`, e.g. `cltsign@signing-linux-1.srv.releng.use1.mozilla.com`. You can use this script, like

```
scriptworker/helper_scripts/create_gpg_keys.py -u cltsign -s host1.fqdn.com host2.
↪fqdn.com
# This will generate a gpg homedir in ./gpg
# Keys will be written to ./host{1,2}.fqdn.com.{pub,sec}
```

Next, sign the newly created gpg keys with your trusted gpg key.

1. import pubkey

```
gpg --import FQDN.pub
```

2. sign pubkey

```
gpg --list-keys EMAIL
gpg --sign-key EMAIL  # or fingerprint
```

3. export signed pubkey

```
gpg --armor --export EMAIL > USERNAME@FQDN.pub  # or fingerprint
```

The signed pubkey + private key will need to go into hiera, as described here.

The signed pubkey will need to land in scriptworker/valid with a signed tag, per the readme.

Chain of Trust Testing / debugging

The `verify_cot` entry point allows you to test chain of trust verification without running a scriptworker instance locally.

Create the virtualenv

- Install git, `python>=3.5`, and python3 virtualenv.
- Clone scriptworker and create virtualenv:

```
git clone https://github.com/mozilla-releng/scriptworker
cd scriptworker
virtualenv3 venv
. venv/bin/activate
python setup.py develop
```

Set up the test env

- Create a ~/.scriptworker or ./secrets.json with test client creds.
- Create the client at the client manager. Mine has the `assume:project:taskcluster:worker-test-scopes` scope, but I don't think that's required.
- The ~/.scriptworker or ./secrets.json file will look like this (fill in your clientId and accessToken):

```
{
  "credentials": {
    "clientId": "mozilla-ldap/asasaki@mozilla.com/signing-test",
    "accessToken": "********"
  }
}
```

Find a task to test

- Find a cot-enabled task on treeherder to test.
- Click it, click 'inspect task' in the lower left corner.
- The taskId will be in a field near the top of the page.

Run the test

- Now you should be able to test chain of trust verification!

```
verify_cot --task-type TASKTYPE TASKID  # e.g., verify_cot --task-type signing cbYd3U6dRRCKPUbKsEj1Iw
```

2.1.6 Scriptworker Readme

Scriptworker implements the TaskCluster worker model, then launches a pre-defined script.

This worker was designed for Releng processes that need specific, limited, and pre-defined capabilities.

Free software: MPL2 license

Usage

- Create a config file. By default scriptworker will look in `./scriptworker.yaml`, but this config path can be specified as the first and only commandline argument. There is an example config file, and all config items are specified in scriptworker.constants.DEFAULT_CONFIG.

Credentials can live in `./scriptworker.yaml`, `./secrets.json`, `~/.scriptworker`, or in environment variables: `TASKCLUSTER_ACCESS_TOKEN`, `TASKCLUSTER_CLIENT_ID`, and `TASKCLUSTER_CERTIFICATE`.

- Launch: `scriptworker [config_path]`

Testing

Note: GPG tests require gpg 2.0.x!

Without integration tests, install tox, then

`NO_TESTS_OVER_WIRE=1 tox -e py35`

With integration tests, first create a client with the `assume:project:taskcluster:worker-test-scopes` scope.

Then create a `./secrets.json` or `~/.scriptworker` that looks like:

```
{
    "integration_credentials": {
        "clientId": "...",
        "accessToken": "...",
        "certificate": "..."
    }
}
```

(`certificate` is only specified if using temp creds)

then

`tox`

It's also possible to create a `./secrets.json` as above, then:

```
cp docker/Dockerfile.test Dockerfile
docker build -t scriptworker-test . && docker run scriptworker-test tox
```

GPG Homedir testing

Sometimes it's nice to be able to test things like `rebuild_gpg_homedirs`. To do so:

```
cp docker/Dockerfile.gnupg Dockerfile
docker build -t scriptworker-gpg . && docker run -i scriptworker-gpg bash -il
# in the docker shell,
rebuild_gpg_homedirs scriptworker.yaml
```

2.1.7 scriptworker package

Submodules

scriptworker.client module

Scripts running in scriptworker will use functions in this file.

This module should be largely standalone. This should only depend on scriptworker.exceptions and scriptworker.constants, or other standalone modules, to avoid circular imports.

`scriptworker.client.`**`get_task`**(*config*)

Read the task.json from work_dir.

Parameters **`config`** (*`dict`*) – the running config, to find work_dir.

Returns the contents of task.json

Return type dict

Raises `ScriptWorkerTaskException` – on error.

`scriptworker.client.`**`validate_artifact_url`**(*valid_artifact_rules*, *valid_artifact_task_ids*, *url*)

Ensure a URL fits in given scheme, netloc, and path restrictions.

If we fail any checks, raise a ScriptWorkerTaskException with `malformed-payload`.

Parameters

- **`valid_artifact_rules`** (*`tuple`*) – the tests to run, with `schemas`, `netlocs`, and `path_regexes`.
- **`valid_artifact_task_ids`** (*`list`*) – the list of valid task IDs to download from.
- **`url`** (*`str`*) – the url of the artifact.

Returns the `filepath` of the path regex.

Return type str

Raises `ScriptWorkerTaskException` – on failure to validate.

`scriptworker.client.`**`validate_json_schema`**(*data*, *schema*, *name='task'*)

Given data and a jsonschema, let's validate it.

This happens for tasks and chain of trust artifacts.

Parameters

- **`data`** (*`dict`*) – the json to validate.
- **`schema`** (*`dict`*) – the jsonschema to validate against.
- **`name`** (*`str, optional`*) – the name of the json, for exception messages. Defaults to "task".

Raises `ScriptWorkerTaskException` – on failure

scriptworker.config module

Config for scriptworker.

`scriptworker.config.`**`log`**

logging.Logger – the log object for the module.

`scriptworker.config.`**`CREDS_FILES`**

tuple – an ordered list of files to look for taskcluster credentials, if they aren't in the config file or environment.

`scriptworker.config.`**`apply_product_config`**`(`*config*`)`

Apply config values that are keyed by *cot_product*.

This modifies the passed in configuration.

Parameters **`dict`** (`config`) – the config to apply cot_product keying too

Returns: dict

`scriptworker.config.`**`check_config`**`(`*config*, *path*`)`

Validate the config against DEFAULT_CONFIG.

Any unknown keys or wrong types will add error messages.

Parameters

- **`config`** (*`dict`*) – the running config.
- **`path`** (*`str`*) – the path to the config file, used in error messages.

Returns the error messages found when validating the config.

Return type list

`scriptworker.config.`**`create_config`**`(`*config_path='scriptworker.yaml'*`)`

Create a config from DEFAULT_CONFIG, arguments, and config file.

Then validate it and freeze it.

Parameters **`config_path`** (*`str, optional`*) – the path to the config file. Defaults to "scriptworker.yaml"

Returns (config frozendict, credentials dict)

Return type tuple

Raises `SystemExit` – on failure

`scriptworker.config.`**`get_context_from_cmdln`**`(`*args*, *desc='Run scriptworker'*`)`

Create a Context object from args.

This was originally part of main(), but we use it in `scriptworker.gpg.rebuild_gpg_homedirs` too.

Parameters **`args`** (*`list`*) – the commandline args. Generally sys.argv

Returns

`scriptworker.context.Context` with populated config, and credentials frozendict

Return type tuple

`scriptworker.config.`**`get_frozen_copy`**`(`*values*`)`

Convert *values*'s list values into tuples, and dicts into frozendicts.

A recursive function(bottom-up conversion)

Parameters **`values`** (*`dict/list`*) – the values/list to be modified in-place.

`scriptworker.config.`**`get_unfrozen_copy`**`(`*values*`)`

Recursively convert *value*'s tuple values into lists, and frozendicts into dicts.

Parameters **`values`** (*`frozendict/tuple`*) – the frozendict/tuple.

Returns the unfrozen copy.

Return type values (dict/list)

`scriptworker.config.`**`read_worker_creds`**(*key='credentials'*)

Get credentials from CREDS_FILES or the environment.

This looks at the CREDS_FILES in order, and falls back to the environment.

Parameters **`key`** (`str, optional`) – each CREDS_FILE is a json dict. This key's value contains the credentials. Defaults to 'credentials'.

Returns the credentials found. None if no credentials found.

Return type dict

scriptworker.context module

scriptworker context.

Most functions need access to a similar set of objects. Rather than having to pass them all around individually or create a monolithic 'self' object, let's point to them from a single context object.

`scriptworker.context.`**`log`**

logging.Logger – the log object for the module.

`class` `scriptworker.context.`**`Context`**

Bases: `object`

Basic config holding object.

Avoids putting everything in single monolithic object, but allows for passing around config and easier overriding in tests.

`config`

dict – the running config. In production this will be a FrozenDict.

`credentials_timestamp`

int – the unix timestamp when we last updated our credentials.

`proc`

asyncio.subprocess.Process – when launching the script, this is the process object.

`queue`

taskcluster.async.Queue – the taskcluster Queue object containing the scriptworker credentials.

`session`

aiohttp.ClientSession – the default aiohttp session

`task`

dict – the task definition for the current task.

`temp_queue`

taskcluster.async.Queue – the taskcluster Queue object containing the task-specific temporary credentials.

`claim_task`

dict – The current or most recent claimTask definition json from the queue.

This contains the task definition, as well as other task-specific info.

When setting `claim_task`, we also set `self.task` and `self.temp_credentials`, zero out `self.reclaim_task` and `self.proc`, then write a task.json to disk.

`config = None`

`create_queue`(*credentials*)

Create a taskcluster queue.

Parameters **credentials** (*dict*) – taskcluster credentials.

credentials

dict – The current scriptworker credentials.

These come from the config or CREDS_FILES or environment.

When setting credentials, also create a new `self.queue` and update self.credentials_timestamp.

credentials_timestamp = None

proc = None

queue = None

reclaim_task

dict – The most recent reclaimTask definition.

This contains the newest expiration time and the newest temp credentials.

When setting `reclaim_task`, we also set `self.temp_credentials`.

`reclaim_task` will be `None` if there hasn't been a claimed task yet, or if a task has been claimed more recently than the most recent reclaimTask call.

session = None

task = None

temp_credentials

dict – The latest temp credentials, or None if we haven't claimed a task yet.

When setting, create `self.temp_queue` from the temp taskcluster creds.

temp_queue = None

write_json (*path*, *contents*, *message*)

Write json to disk.

Parameters

- **path** (*str*) – the path to write to
- **contents** (*dict*) – the contents of the json blob
- **message** (*str*) – the message to log

scriptworker.exceptions module

scriptworker exceptions.

exception `scriptworker.exceptions.`**CoTError** (*msg*)

Bases: *scriptworker.exceptions.ScriptWorkerTaskException*, `KeyError`

Failure in Chain of Trust verification.

exit_code

int – this is set to 3 (malformed-payload).

exception `scriptworker.exceptions.`**ConfigError**

Bases: *scriptworker.exceptions.ScriptWorkerException*

Invalid configuration provided to scriptworker.

exit_code

int – this is set to 5 (internal-error).

exception `scriptworker.exceptions.`**`DownloadError`**(*msg*)

Bases: *`scriptworker.exceptions.ScriptWorkerTaskException`*

Failure in `scriptworker.utils.download_file`.

`exit_code`

int – this is set to 4 (resource-unavailable).

exception `scriptworker.exceptions.`**`ScriptWorkerException`**

Bases: `Exception`

The base exception in scriptworker.

When raised inside of the run_loop loop, set the taskcluster task status to at least `self.exit_code`.

`exit_code`

int – this is set to 5 (internal-error).

`exit_code = 5`

exception `scriptworker.exceptions.`**`ScriptWorkerGPGException`**

Bases: *`scriptworker.exceptions.ScriptWorkerException`*

Scriptworker GPG error.

`exit_code`

int – this is set to 5 (internal-error).

`exit_code = 5`

exception `scriptworker.exceptions.`**`ScriptWorkerRetryException`**

Bases: *`scriptworker.exceptions.ScriptWorkerException`*

Scriptworker retry error.

`exit_code`

int – this is set to 4 (resource-unavailable)

`exit_code = 4`

exception `scriptworker.exceptions.`**`ScriptWorkerTaskException`**(**args*, *exit_code=1*, ***kwargs*)

Bases: *`scriptworker.exceptions.ScriptWorkerException`*

Scriptworker task error.

To use:

```
import sys
import traceback
try:
    ...
except ScriptWorkerTaskException as exc:
    traceback.print_exc()
    sys.exit(exc.exit_code)
```

`exit_code`

int – this is 1 by default (failure)

scriptworker.gpg module

GPG support.

These currently assume gpg 2.0.x

These GPG functions expose considerable functionality over gpg key management, data signatures, and validation, but by no means are they intended to cover all gnupg functionality. They are intended for automated key management and validation for scriptworker.

`scriptworker.gpg.`**`log`**

logging.Logger – the log object for this module.

`scriptworker.gpg.`**`GPG_CONFIG_MAPPING`**

dict – This maps the scriptworker config key names to the python-gnupg names.

`scriptworker.gpg.`**`GPG`**`(`*context*, *gpg_home=None*`)`

Get a python-gnupg GPG instance based on the settings in `context`.

Parameters

- **`context`** (`scriptworker.context.Context`) – the scriptworker context.
- **`gpg_home`** (*`str, optional`*) – override `context.config['gpg_home']` if desired. Defaults to None.

Returns the GPG instance with the appropriate configs.

Return type gnupg.GPG

`scriptworker.gpg.`**`build_gpg_homedirs_from_repo`**`(`*context*, *tag*, *basedir=None*, *verify_function=<function verify_signed_tag>*, *flat_function=<function rebuild_gpg_home_flat>*, *signed_function=<function rebuild_gpg_home_signed>*`)`

Build gpg homedirs in `basedir`, from the context-defined git repo.

Parameters

- **`context`** (`scriptworker.context.Context`) – the scriptworker context.
- **`tag`** (*`str`*) – the tag name to verify
- **`basedir`** (*`str, optional`*) – the path to the base directory to create the gpg homedirs in. This directory will be wiped if it exists. If None, use `context.config['base_gpg_home_dir']`. Defaults to None.

Returns on success.

Return type str

Raises `ScriptWorkerGPGException` – on rebuild exception.

`scriptworker.gpg.`**`check_ownertrust`**`(`*context*, *gpg_home=None*`)`

In theory, this will repair a broken trustdb.

Rebuild the trustdb via –import-ownertrust if not.

Parameters

- **`context`** (`scriptworker.context.Context`) – the scriptworker context.
- **`gpg_home`** (*`str, optional`*) – override the gpg_home with a different gnupg home directory here. Defaults to None.

`scriptworker.gpg.`**`consume_valid_keys`**`(`*context*, *keydir=None*, *ignore_suffixes=()*, *gpg_home=None*`)`

Given a keydir, traverse the keydir, and import all gpg public keys.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **keydir** (*str, optional*) – the path of the directory to traverse. If None, this function is noop. Default is None.
- **ignore_suffixes** (*list, optional*) – file suffixes to ignore. Default is ().
- **gpg_home** (*str, optional*) – override the gpg_home dir. Default is None.

Returns fingerprints

Return type list

Raises `ScriptworkerGPGException` – on error.

`scriptworker.gpg.`**`create_gpg_conf`**(*gpg_home*, *keyserver=None*, *my_fingerprint=None*)

Create a gpg.conf with Mozilla infosec guidelines.

Parameters

- **gpg_home** (*str*) – the homedir for this keyring.
- **keyserver** (*str, optional*) – The gpg keyserver to specify, e.g. `hkp://gpg.mozilla.org` or `hkp://keys.gnupg.net`. If set, we also enable `auto-key-retrieve`. Defaults to None.
- **my_fingerprint** (*str, optional*) – the fingerprint of the default key. Once set, gpg will use it by default, unless a different key is specified. Defaults to None.

`scriptworker.gpg.`**`create_lockfile`**(*context*, *message='locked'*)

Create the lockfile.

Parameters **context** (scriptworker.context.Context) – the scriptworker context

`scriptworker.gpg.`**`export_key`**(*gpg*, *fingerprint*, *private=False*)

Return the ascii armored key identified by `fingerprint`.

Parameters

- **gpg** (*gnupg.GPG*) – the GPG instance.
- **fingerprint** (*str*) – the fingerprint of the key to export.
- **private** (*bool, optional*) – If True, return the private key instead of the public key. Defaults to False.

Returns the ascii armored key identified by `fingerprint`.

Return type str

Raises `ScriptworkerGPGException` – if the key isn't found.

`scriptworker.gpg.`**`fingerprint_to_keyid`**(*gpg*, *fingerprint*, *private=False*)

Return the keyid of the key that corresponds to `fingerprint`.

Keyids should default to long keyids; this will happen once `create_gpg_conf()` is called.

Parameters

- **gpg** (*gnupg.GPG*) – gpg object for the appropriate gpg_home / keyring
- **fingerpint** (*str*) – the fingerprint of the key we're searching for.
- **private** (*bool, optional*) – If True, search the private keyring instead of the public keyring. Defaults to False.

Returns the keyid of the key with fingerprint `fingerprint`

Return type keyid (str)

Raises `ScriptworkerGPGException` – if we can't find `fingerprint` in this keyring.

`scriptworker.gpg.`**`generate_key`**(*gpg*, *name*, *comment*, *email*, *key_length=4096*, *expiration=None*)

Generate a gpg keypair.

Parameters

- **`gpg`** (*`gnupg.GPG`*) – the GPG instance.
- **`name`** (*`str`*) – the name attached to the key. 1/3 of the key user id.
- **`comment`** (*`str`*) – the comment attached to the key. 1/3 of the key user id.
- **`email`** (*`str`*) – the email attached to the key. 1/3 of the key user id.
- **`key_length`** (*`int, optional`*) – the key length in bits. Defaults to 4096.
- **`expiration`** (*`str, optional`*) – The expiration of the key. This can take the forms "2009-12-31", "365d", "3m", "6w", "5y", "seconds=<epoch>", or 0 for no expiry. Defaults to None.

Returns the fingerprint of the key just generated.

Return type fingerprint (str)

`scriptworker.gpg.`**`get_body`**(*gpg*, *signed_data*, *gpg_home=None*, *verify_sig=True*, ***kwargs*)

Verify the signature, then return the unsigned data from `signed_data`.

Parameters

- **`gpg`** (*`gnupg.GPG`*) – the GPG instance.
- **`signed_data`** (*`str`*) – The ascii armored signed data.
- **`gpg_home`** (*`str, optional`*) – override the gpg_home with a different gnupg home directory here. Defaults to None.
- **`verify_sig`** (*`bool, optional`*) – verify the signature before decrypting. Defaults to True.
- **`kwargs`** (*`dict, optional`*) – These are passed directly to gpg.decrypt(). Defaults to {}. https://pythonhosted.org/python-gnupg/#decryption

Returns unsigned contents on success.

Return type str

Raises `ScriptWorkerGPGException` – on signature verification failure.

`scriptworker.gpg.`**`get_git_revision`**(*path*, *ref='HEAD'*, *exec_function=<function create_subprocess_exec>*)

Get the git revision of path.

Parameters

- **`path`** (*`str`*) – the path to run `git log -n1 --format=format:%H REF` in.
- **`ref`** (*`str, optional`*) – the ref to find the revision for. Defaults to "HEAD"

Returns the revision found.

Return type str

Raises `ScriptWorkerRetryException` – on failure.

`scriptworker.gpg.`**`get_last_good_git_revision`**(*context*)

Return the contents of the config['last_good_git_revision_file'], if it exists.

Parameters **context** (scriptworker.context.Context) – the scriptworker context.

Returns the latest good git revision, if the file exists None: if the file doesn't exist

Return type str

scriptworker.gpg.**get_latest_tag**(*path*, *exec_function=<function create_subprocess_exec>*)

Get the latest tag in path.

Parameters **path** (*str*) – the path to run `git describe --abbrev=0` in.

Returns the tag name found.

Return type str

Raises ScriptWorkerRetryException – on failure.

scriptworker.gpg.**get_list_sigs_output**(*context*, *key_fingerprint*, *gpg_home=None*, *validate=True*, *expected=None*)

Get output from gpg –list-sigs.

This will be machine parsable output, for gpg 2.0.x.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **key_fingerprint** (*str*) – the fingerprint of the key we want to get signature information about.
- **gpg_home** (*str, optional*) – override the gpg_home with a different gnupg home directory here. Defaults to None.
- **validate** (*bool, optional*) – Validate the output via parse_list_sigs_output() Defaults to True.
- **expected** (*dict, optional*) – This is passed on to parse_list_sigs_output() if validate is True. Defaults to None.

Returns the output from gpg –list-sigs, if validate is False dict: the output from parse_list_sigs_output, if validate is True

Return type str

Raises ScriptWorkerGPGException – if there is an issue with the key.

scriptworker.gpg.**get_tmp_base_gpg_home_dir**(*context*)

Return the base_gpg_home_dir with a .tmp at the end.

This function is really only here so we don't have to duplicate this logic.

Parameters **context** (scriptworker.context.Context) – the scriptworker context.

Returns the base_gpg_home_dir with .tmp at the end.

Return type str

scriptworker.gpg.**gpg_default_args**(*gpg_home*)

For commandline gpg calls, use these args by default.

Parameters **gpg_home** (*str*) – The path to the gpg homedir. gpg will look for the gpg.conf, trustdb.gpg, and keyring files in here.

Returns the list of default commandline arguments to add to the gpg call.

Return type list

`scriptworker.gpg.`**`guess_gpg_home`**(*obj*, *gpg_home=None*)

Guess gpg_home. If `gpg_home` is specified, return that.

Parameters

- **obj** (*object*) – If gpg_home is set, return that. Otherwise, if `obj` is a context object and `context.config['gpg_home']` is not None, return that. If `obj` is a GPG object and `obj.gnupghome` is not None, return that. Otherwise look in `~/.gnupg`.
- **gpg_home** (*str, optional*) – The path to the gpg homedir. gpg will look for the gpg.conf, trustdb.gpg, and keyring files in here. Defaults to None.

Returns the path to the guessed gpg homedir.

Return type str

Raises `ScriptWorkerGPGException` – if obj doesn't contain the gpg home info and os.environ['HOME'] isn't set.

`scriptworker.gpg.`**`guess_gpg_path`**(*context*)

Guess gpg_path.

Parameters **context** (`scriptworker.context.Context`) – the scriptworker context.

Returns either `context.config['gpg_path']` or 'gpg' if that's not defined.

Return type str

`scriptworker.gpg.`**`has_suffix`**(*path*, *suffixes*)

Given a list of suffixes, return True if path ends with one of them.

Parameters

- **path** (*str*) – the file path to check
- **suffixes** (*list*) – the suffixes to check for

`scriptworker.gpg.`**`import_key`**(*gpg*, *key_data*, *return_type='fingerprints'*)

Import ascii key_data.

In theory this can be multiple keys. However, jenkins is barfing on multiple key import tests, although multiple key import tests are working locally. Until we identify what the problem is (likely gpg version?) we should only import 1 key at a time.

Parameters

- **gpg** (*gnupg.GPG*) – the GPG instance.
- **key_data** (*str*) – ascii armored key data
- **return_type** (*str, optional*) – if 'fingerprints', return the fingerprints only. Otherwise return the result list.

Returns

if return_type is 'fingerprints', return the fingerprints of the imported keys. Otherwise return the results list. https://pythonhosted.org/python-gnupg/#importing-and-receiving-keys

Return type list

`scriptworker.gpg.`**`is_lockfile_present`**(*context*, *name*, *level=30*)

Check for the lockfile.

Parameters

- **context** (`scriptworker.context.Context`) – the scriptworker context

- **name** (*str*) – the name of the calling function
- **level** (*int, optional*) – the level to log to. Defaults to `logging.WARNING`

Returns "locked" on r/w lock; "ready" if ready to copy. None: if lockfile is not present

Return type str

`scriptworker.gpg.`**`keyid_to_fingerprint`**(*gpg*, *keyid*, *private=False*)

Return the fingerprint of the key that corresponds to `keyid`.

Keyids should default to long keyids; this will happen once `create_gpg_conf()` is called.

Parameters

- **gpg** (*gnupg.GPG*) – gpg object for the appropriate gpg_home / keyring
- **keyid** (*str*) – the long keyid that represents the key we're searching for.
- **private** (*bool, optional*) – If True, search the private keyring instead of the public keyring. Defaults to False.

Returns the fingerprint of the key with keyid `keyid`

Return type fingerprint (str)

Raises `ScriptworkerGPGException` – if we can't find `keyid` in this keyring.

`scriptworker.gpg.`**`overwrite_gpg_home`**(*tmp_gpg_home*, *real_gpg_home*)

Take the contents of tmp_gpg_home and copy them to real_gpg_home.

Parameters

- **tmp_gpg_home** (*str*) – path to the rebuilt gpg_home with the new keychains+ trust models
- **real_gpg_home** (*str*) – path to the old gpg_home to overwrite

`scriptworker.gpg.`**`parse_list_sigs_output`**(*output*, *desc*, *expected=None*)

Parse the output from –list-sigs; validate.

NOTE: This doesn't work with complex key/subkeys; this is only written for the keys generated through the functions in this module.

1. **Field: Type of record** pub = public key crt = X.509 certificate crs = X.509 certificate and private key available sub = subkey (secondary key) sec = secret key ssb = secret subkey (secondary key) uid = user id (only field 10 is used). uat = user attribute (same as user id except for field 10). sig = signature rev = revocation signature fpr = fingerprint: (fingerprint is in field 10) pkd = public key data (special field format, see below) grp = keygrip rvk = revocation key tru = trust database information spk = signature subpacket

There are also 'gpg' lines like

> gpg: checking the trustdb gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model gpg: depth: 0 valid: 3 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 3u

This is a description of the web of trust. I'm currently not parsing these; per [1] and [2] I would need to read the source for full parsing.

[1] http://security.stackexchange.com/a/41209

[2] http://gnupg.10057.n7.nabble.com/placing-trust-in-imported-keys-td30124.html#a30125

Parameters

- **output** (*str*) – the output from get_list_sigs_output()

- **desc** (*str*) – a description of the key being tested, for exception message purposes.
- **expected** (*dict, optional*) – expected outputs. If specified and the expected doesn't match the real, raise an exception. Expected takes `keyid`, `fingerprint`, `uid`, `sig_keyids` (list), and `sig_uids` (list), all optional. Defaults to None.

Returns

the real values from the key. This specifies `keyid`, `fingerprint`, `uid`, `sig_keyids`, and `sig_uids`.

Return type real (dict)

Raises `ScriptWorkerGPGException` – on mismatched expectations, or if we found revocation markers or the like that make for a bad key.

`scriptworker.gpg.`**`rebuild_gpg_home`**(*context*, *tmp_gpg_home*, *my_pub_key_path*, *my_priv_key_path*)

Import my key and create gpg.conf and trustdb.gpg.

Parameters

- **gpg** (*gnupg.GPG*) – the GPG instance.
- **tmp_gpg_home** (*str*) – the path to the tmp gpg_home. This should already exist.
- **my_pub_key_path** (*str*) – the ascii public key file we want to import as the primary key
- **my_priv_key_path** (*str*) – the ascii private key file we want to import as the primary key

Returns my fingerprint

Return type str

`scriptworker.gpg.`**`rebuild_gpg_home_flat`**(*context*, *real_gpg_home*, *my_pub_key_path*, *my_priv_key_path*, *consume_path*, *ignore_suffixes=()*, *consume_function=<function consume_valid_keys>*)

Rebuild `real_gpg_home` with new trustdb, pub+secrings, gpg.conf.

In this 'flat' model, import all the pubkeys in `consume_path` and sign them directly. This makes them valid but not trusted.

Parameters

- **context** (`scriptworker.context.Context`) – the scriptworker context.
- **real_gpg_home** (*str*) – the gpg_home path we want to rebuild
- **my_pub_key_path** (*str*) – the ascii public key file we want to import as the primary key
- **my_priv_key_path** (*str*) – the ascii private key file we want to import as the primary key
- **consume_path** (*str*) – the path to the directory tree to import pubkeys from
- **ignore_suffixes** (*list, optional*) – the suffixes to ignore in consume_path. Defaults to ()

scriptworker.gpg.**rebuild_gpg_home_signed**(*context*, *real_gpg_home*, *my_pub_key_path*, *my_priv_key_path*, *trusted_path*, *untrusted_path=None*, *ignore_suffixes=()*, *consume_function=<function consume_valid_keys>*)

Rebuild `real_gpg_home` with new trustdb, pub+secrings, gpg.conf.

In this 'signed' model, import all the pubkeys in `trusted_path`, sign them directly, and trust them. Then import all the pubkeys in `untrusted_path` with no signing. The intention is that one of the keys in `trusted_path` has already signed the keys in `untrusted_path`, making them valid.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **real_gpg_home** (*str*) – the gpg_home path we want to rebuild
- **my_pub_key_path** (*str*) – the ascii public key file we want to import as the primary key
- **my_priv_key_path** (*str*) – the ascii private key file we want to import as the primary key
- **trusted_path** (*str*) – the path to the directory tree to import trusted pubkeys from
- **untrusted_path** (*str, optional*) – the path to the directory tree to import untrusted but valid pubkeys from
- **ignore_suffixes** (*list, optional*) – the suffixes to ignore in consume_path. Defaults to ()

scriptworker.gpg.**rebuild_gpg_homedirs**()

Rebuild the gpg homedirs in the background.

This is an entry point, and should be called before scriptworker is run.

Raises `SystemExit` – on failure.

scriptworker.gpg.**rm_lockfile**(*context*)

Remove the lockfile.

Parameters **context** (scriptworker.context.Context) – the scriptworker context

scriptworker.gpg.**sign**(*gpg*, *data*, ***kwargs*)

Sign `data` with the key `kwargs['keyid']`, or the default key if not specified.

Parameters

- **gpg** (*gnupg.GPG*) – the GPG instance.
- **data** (*str*) – The contents to sign with the key.
- **kwargs** (*dict, optional*) – These are passed directly to gpg.sign(). Defaults to {}. https://pythonhosted.org/python-gnupg/#signing

Returns the ascii armored signed data.

Return type str

scriptworker.gpg.**sign_key**(*context*, *target_fingerprint*, *signing_key=None*, *exportable=False*, *gpg_home=None*)

Sign the `target_fingerprint` key with `signing_key` or default key.

This signs the target key with the signing key, which adds to the web of trust.

Due to pexpect async issues, this function is once more synchronous.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **target_fingerprint** (*str*) – the fingerprint of the key to sign.
- **signing_key** (*str, optional*) – the fingerprint of the signing key to sign with. If not set, this defaults to the default-key in the gpg.conf. Defaults to None.
- **exportable** (*bool, optional*) – whether the signature should be exportable. Defaults to False.
- **gpg_home** (*str, optional*) – override the gpg_home with a different gnupg home directory here. Defaults to None.

Raises ScriptWorkerGPGException – on a failed signature.

scriptworker.gpg.**update_ownertrust**(*context, my_fingerprint, trusted_fingerprints=None, gpg_home=None*)

Trust my key ultimately; trusted_fingerprints fully.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **my_fingerprint** (*str*) – the fingerprint of the key we want to specify as ultimately trusted.
- **trusted_fingerprints** (*list, optional*) – the list of fingerprints that we want to mark as fully trusted. These need to be signed by the my_fingerprint key before they are trusted.
- **gpg_home** (*str, optional*) – override the gpg_home with a different gnupg home directory here. Defaults to None.

Raises ScriptWorkerGPGException – if there is an error.

scriptworker.gpg.**update_signed_git_repo**(*context, repo='origin', ref='master', exec_function=<function create_subprocess_exec>, log_function=<function pipe_to_log>*)

Update a git repo with signed git commits, and verify the signature.

This function updates the repo.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **repo** (*str, optional*) – the repo to update from. Defaults to 'origin'.
- **ref** (*str, optional*) – the ref to update to. Defaults to 'master'.

Returns the current git revision, and the latest tag name.

Return type tuple (str, str)

Raises

- ScriptWorkerGPGException – on signature validation failure.
- ScriptWorkerRetryException – on `git pull` failure.

scriptworker.gpg.**verify_ownertrust**(*context, my_fingerprint, trusted_fingerprints=None, gpg_home=None*)

Verify the ownertrust is exactly as expected.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **my_fingerprint** (*str*) – the fingerprint of the key we specified as ultimately trusted.
- **trusted_fingerprints** (*list, optional*) – the list of fingerprints that we marked as fully trusted.
- **gpg_home** (*str, optional*) – override the gpg_home with a different gnupg home directory here. Defaults to None.

Raises ScriptWorkerGPGException – if there is an error.

scriptworker.gpg.**verify_signature**(*gpg*, *signed_data*, ***kwargs*)

Verify signed_data with the key kwargs['keyid'], or the default key if not specified.

Parameters

- **gpg** (*gnupg.GPG*) – the GPG instance.
- **signed_data** (*str*) – The ascii armored signed data.
- **kwargs** (*dict, optional*) – These are passed directly to gpg.verify(). Defaults to {}. https://pythonhosted.org/python-gnupg/#verification

Returns on success.

Return type gnupg.Verify

Raises ScriptWorkerGPGException – on failure.

scriptworker.gpg.**verify_signed_tag**(*context*, *tag*, *exec_function=<function check_call>*)

Verify git_key_repo_dir is at the valid signed tag.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **tag** (*str*) – the tag to verify.

Raises ScriptWorkerGPGException – if we're not updated to tag

scriptworker.gpg.**write_last_good_git_revision**(*context*, *revision*)

Write revision to config['last_good_git_revision_file'].

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **revision** (*str*) – the last good git revision

scriptworker.log module

Scriptworker logging.

scriptworker.log.**log**

logging.Logger – the log object for this module.

scriptworker.log.**contextual_log_handler**(*context*, *path*, *log_obj=None*, *level=10*, *formatter=None*)

Add a short-lived log with a contextmanager for cleanup.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context

- **path** (*str*) – the path to the log file to create
- **log_obj** (*logging.Logger*) – the log object to modify. If None, use `scriptworker.log.log`. Defaults to None.
- **level** (*int, optional*) – the logging level. Defaults to logging.DEBUG.
- **formatter** (*logging.Formatter, optional*) – the logging formatter. If None, defaults to `logging.Formatter(fmt=fmt)`. Default is None.

Yields *None* – but cleans up the handler afterwards.

`scriptworker.log.`**`get_log_filehandle`**(*context*)

Open the log and error filehandles.

Parameters **context** (`scriptworker.context.Context`) – the scriptworker context.

Yields log filehandle

`scriptworker.log.`**`get_log_filename`**(*context*)

Get the task log/error file paths.

Parameters **context** (`scriptworker.context.Context`) – the scriptworker context.

Returns log file path

Return type string

`scriptworker.log.`**`pipe_to_log`**(*pipe*, *filehandles=()*, *level=20*)

Log from a subprocess PIPE.

Parameters

- **pipe** (*filehandle*) – subprocess process STDOUT or STDERR
- **filehandles** (*list of filehandles, optional*) – the filehandle(s) to write to. If empty, don't write to a separate file. Defaults to ().
- **level** (*int, optional*) – the level to log to. Defaults to `logging.INFO`.

`scriptworker.log.`**`update_logging_config`**(*context*, *log_name=None*, *file_name='worker.log'*)

Update python logging settings from config.

By default, this sets the `scriptworker` log settings, but this will change if some other package calls this function or specifies the `log_name`.

- Use formatting from config settings.
- Log to screen if `verbose`
- Add a rotating logfile from config settings.

Parameters

- **context** (`scriptworker.context.Context`) – the scriptworker context.
- **log_name** (*str, optional*) – the name of the Logger to modify. If None, use the top level module ('scriptworker'). Defaults to None.

scriptworker.task module

Scriptworker task execution.

`scriptworker.task.`**`log`**

logging.Logger – the log object for the module `scriptworker.task.`**`claim_work`**(*context*)

Find and claim the next pending task in the queue, if any.

Parameters **`context`** (`scriptworker.context.Context`) – the scriptworker context.

Returns a dict containing a list of the task definitions of the tasks claimed.

Return type dict

`scriptworker.task.`**`complete_task`**(*context*, *result*)

Mark the task as completed in the queue.

Decide whether to call reportCompleted, reportFailed, or reportException based on the exit status of the script.

If the task has expired or been cancelled, we'll get a 409 status.

Parameters **`context`** (`scriptworker.context.Context`) – the scriptworker context.

Raises `taskcluster.exceptions.TaskclusterRestFailure` – on non-409 error.

`scriptworker.task.`**`get_decision_task_id`**(*task*)

Given a task dict, return the decision taskId.

By convention, the decision task of the `taskId` is the task's `taskGroupId`.

Parameters **`task`** (*`dict`*) – the task dict.

Returns the taskId of the decision task.

Return type str

`scriptworker.task.`**`get_parent_task_id`**(*task*)

Given a task dict, return the parent taskId.

The parent taskId could be a decision taskId, or an action taskId. The parent is the task that created this task; it should have a `task-graph.json` containing this task's definition as an artifact.

Parameters **`task`** (*`dict`*) – the task dict

Returns the taskId of the parent.

Return type str

`scriptworker.task.`**`get_run_id`**(*claim_task*)

Given a claim_task json dict, return the runId.

Parameters **`claim_task`** (*`dict`*) – the claim_task dict.

Returns the runId.

Return type int

`scriptworker.task.`**`get_task_id`**(*claim_task*)

Given a claim_task json dict, return the taskId.

Parameters **`claim_task`** (*`dict`*) – the claim_task dict.

Returns the taskId.

Return type str

`scriptworker.task.`**`get_worker_type`**(*task*)

Given a task dict, return the workerType.

Parameters **`task`** (*`dict`*) – the task dict.

Returns the workerType.

Return type str

`scriptworker.task.`**`kill`**(*pid*, *sleep_time=1*)

Kill `pid` with various signals.

Parameters

- **`pid`** (*`int`*) – the process id to kill.
- **`sleep_time`** (*`int, optional`*) – how long to sleep between killing the pid and checking if the pid is still running.

`scriptworker.task.`**`max_timeout`**(*context*, *proc*, *timeout*)

Make sure the proc pid's process and process group are killed.

First, kill the process group (-pid) and then the pid.

Parameters

- **`context`** (`scriptworker.context.Context`) – the scriptworker context.
- **`proc`** (*`subprocess.Process`*) – the subprocess proc. This is compared against context.proc to make sure we're killing the right pid.
- **`timeout`** (*`int`*) – Used for the log message.

`scriptworker.task.`**`prepare_to_run_task`**(*context*, *claim_task*)

Given a *claim_task* json dict, prepare the *context* and *work_dir*.

Set *context.claim_task*, and write a *work_dir/current_task_info.json*

Parameters

- **`context`** (`scriptworker.context.Context`) – the scriptworker context.
- **`claim_task`** (*`dict`*) – the claim_task dict.

Returns the contents of *current_task_info.json*

Return type dict

`scriptworker.task.`**`reclaim_task`**(*context*, *task*)

Try to reclaim a task from the queue.

This is a keepalive / heartbeat. Without it the job will expire and potentially be re-queued. Since this is run async from the task, the task may complete before we run, in which case we'll get a 409 the next time we reclaim.

Parameters **`context`** (`scriptworker.context.Context`) – the scriptworker context

Raises `taskcluster.exceptions.TaskclusterRestFailure` – on non-409 status_code from taskcluster.async.Queue.reclaimTask()

`scriptworker.task.`**`run_task`**(*context*)

Run the task, sending stdout+stderr to files.

https://github.com/python/asyncio/blob/master/examples/subprocess_shell.py

Parameters **`context`** (`scriptworker.context.Context`) – the scriptworker context.

Returns exit code

Return type int

`scriptworker.task.`**`worst_level`**(*level1*, *level2*)

Given two int levels, return the larger.

Parameters

- **`level1`** (*`int`*) – exit code 1.

- **level2** (*int*) – exit code 2.

Returns the larger of the two levels.

Return type int

scriptworker.utils module

Generic utils for scriptworker.

`scriptworker.utils.`**`log`**

logging.Logger – the log object for the module

`scriptworker.utils.`**`add_enumerable_item_to_dict`**(*dict_*, *key*, *item*)

Add an item to a list contained in a dict.

For example: If the dict is `{'some_key': ['an_item']}`, then calling this function will alter the dict to `{'some_key': ['an_item', 'another_item']}`.

If the key doesn't exist yet, the function initializes it with a list containing the item.

List-like items are allowed. In this case, the existing list will be extended.

Parameters

- **dict** (*dict*) – the dict to modify
- **key** (*str*) – the key to add the item to
- **item** (*whatever*) – The item to add to the list associated to the key

`scriptworker.utils.`**`calculate_sleep_time`**(*attempt*, *delay_factor=5.0*, *randomization_factor=0.5*, *max_delay=120*)

Calculate the sleep time between retries, in seconds.

Based off of *taskcluster.utils.calculateSleepTime*, but with kwargs instead of constant *delay_factor*/*randomization_factor*/*max_delay*. The taskcluster function generally slept for less than a second, which didn't always get past server issues.

Parameters

- **attempt** (*int*) – the retry attempt number
- **delay_factor** (*float, optional*) – a multiplier for the delay time. Defaults to 5.
- **randomization_factor** (*float, optional*) – a randomization multiplier for the delay time. Defaults to .5.
- **max_delay** (*float, optional*) – the max delay to sleep. Defaults to 120 (seconds).

Returns the time to sleep, in seconds.

Return type float

`scriptworker.utils.`**`cleanup`**(*context*)

Clean up the work_dir and artifact_dir between task runs, then recreate.

Parameters **context** (`scriptworker.context.Context`) – the scriptworker context.

`scriptworker.utils.`**`create_temp_creds`**(*client_id*, *access_token*, *start=None*, *expires=None*, *scopes=None*, *name=None*)

Request temp TC creds with our permanent creds.

Parameters

- **client_id** (*str*) – the taskcluster client_id to use

- **access_token** (*str*) – the taskcluster access_token to use
- **start** (*str, optional*) – the datetime string when the credentials will start to be valid. Defaults to 10 minutes ago, for clock skew.
- **expires** (*str, optional*) – the datetime string when the credentials will expire. Defaults to 31 days after 10 minutes ago.
- **scopes** (*list, optional*) – The list of scopes to request for the temp creds. Defaults to ['assume:project:taskcluster:worker-test-scopes',]
- **name** (*str, optional*) – the name to associate with the creds.

Returns the temporary taskcluster credentials.

Return type dict

`scriptworker.utils.`**`datestring_to_timestamp`**(*datestring*)

Create a timetamp from a taskcluster datestring.

Parameters **datestring** (*str*) – the datestring to convert. isoformat, like "2016-04-16T03:46:24.958Z"

Returns the corresponding timestamp.

Return type int

`scriptworker.utils.`**`download_file`**(*context*, *url*, *abs_filename*, *session=None*, *chunk_size=128*)

Download a file, async.

Parameters

- **context** (scriptworker.context.Context) – the scriptworker context.
- **url** (*str*) – the url to download
- **abs_filename** (*str*) – the path to download to
- **session** (*aiohttp.ClientSession, optional*) – the session to use. If None, use context.session. Defaults to None.
- **chunk_size** (*int, optional*) – the chunk size to read from the response at a time. Default is 128.

`scriptworker.utils.`**`filepaths_in_dir`**(*path*)

Find all files in a directory, and return the relative paths to those files.

Parameters **path** (*str*) – the directory path to walk

Returns

the list of relative paths to all files inside of `path` or its subdirectories.

Return type list

`scriptworker.utils.`**`format_json`**(*data*)

Format json as a sorted string (indents of 2).

Parameters **data** (*dict*) – the json to format.

Returns the formatted json.

Return type str

`scriptworker.utils.`**`get_hash`**(*path*, *hash_alg='sha256'*)

Get the hash of the file at `path`.

I'd love to make this async, but evidently file i/o is always ready

Parameters

- **path** (*str*) – the path to the file to hash.
- **hash_alg** (*str, optional*) – the algorithm to use. Defaults to 'sha256'.

Returns the hexdigest of the hash.

Return type str

scriptworker.utils.**get_results_and_future_exceptions**(*tasks*)

Given a list of futures, await them, then return results and exceptions.

This is similar to raise_future_exceptions, except that it doesn't raise any exception. They are returned instead. This allows some tasks to optionally fail. Please consider that no exception will be raised when calling this function. You must verify the content of the second item of the tuple. It contains all exceptions raised by the futures.

Parameters **tasks** (*list*) – the list of futures to await and check for exceptions.

Returns the list of results from the futures, then the list of exceptions.

Return type tuple

scriptworker.utils.**load_json**(*string*, *is_path=False*, *exception=<class 'scriptworker.exceptions.ScriptWorkerTaskException'>*, *message='Failed to load json: %(exc)s'*)

Load json from a filehandle or string, and raise a custom exception on failure.

Parameters

- **string** (*str*) – json body or a path to open
- **is_path** (*bool, optional*) – if string is a path. Defaults to False.
- **exception** (*exception, optional*) – the exception to raise on failure. If None, don't raise an exception. Defaults to ScriptWorkerTaskException.
- **message** (*str, optional*) – the message to use for the exception. Defaults to "Failed to load json: %(exc)s"

Returns the json contents

Return type dict

Raises Exception – as specified, on failure

scriptworker.utils.**makedirs**(*path*)

Equivalent to mkdir -p.

Parameters **path** (*str*) – the path to mkdir -p

Raises ScriptWorkerException – if path exists already and the realpath is not a dir.

scriptworker.utils.**match_url_regex**(*rules*, *url*, *callback*)

Given rules and a callback, find the rule that matches the url.

Rules look like:

```
(
    {
        'schemes': ['https', 'ssh'],
        'netlocs': ['hg.mozilla.org'],
        'path_regexes': [
            "^(?P<path>/mozilla-(central|unified))(/|$)",
        ]
    },
    ...
)
```

Parameters

- **rules** (*list*) – a list of dictionaries specifying lists of `schemes`, `netlocs`, and `path_regexes`.
- **url** (*str*) – the url to test
- **callback** (*function*) – a callback that takes an `re.MatchObject`. If it returns None, continue searching. Otherwise, return the value from the callback.

Returns the value from the callback, or None if no match.

Return type value

`scriptworker.utils.`**`raise_future_exceptions`**`(`*tasks*`)`

Given a list of futures, await them, then raise their exceptions if any.

Without something like this, a bare:

```
await asyncio.wait(tasks)
```

will swallow exceptions.

Parameters **tasks** (*list*) – the list of futures to await and check for exceptions.

Returns the list of results from the futures.

Return type list

Raises `Exception` – any exceptions in task.exception(), or CancelledError if the task was cancelled

`scriptworker.utils.`**`request`**`(`*context*, *url*, *timeout=60*, *method='get'*, *good=(200,)*, *retry=(500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511)*, *return_type='text'*, ***kwargs*`)`

Async aiohttp request wrapper.

Parameters

- **context** (`scriptworker.context.Context`) – the scriptworker context.
- **url** (*str*) – the url to request
- **timeout** (*int, optional*) – timeout after this many seconds. Default is 60.
- **method** (*str, optional*) – The request method to use. Default is 'get'.
- **good** (*list, optional*) – the set of good status codes. Default is (200,)
- **retry** (*list, optional*) – the set of status codes that result in a retry. Default is tuple(range(500, 512)).
- **return_type** (*str, optional*) – The type of value to return. Takes 'json' or 'text'; other values will return the response object. Default is text.
- ****kwargs** – the kwargs to send to the aiohttp request function.

Returns

the response text() if return_type is 'text'; the response json() if return_type is 'json'; the aiohttp request response object otherwise.

Return type object

Raises

- ScriptWorkerRetryException – if the status code is in the retry list.
- ScriptWorkerException – if the status code is not in the retry list or good list.

scriptworker.utils.**retry_async**(*func*, *attempts=5*, *sleeptime_callback=<function calculate_sleep_time>*, *retry_exceptions=(<class 'Exception'>,)*, *args=()*, *kwargs=None*, *sleeptime_kwargs=None*)

Retry func, where func is an awaitable.

Parameters

- **func** (*function*) – an awaitable function.
- **attempts** (*int, optional*) – the number of attempts to make. Default is 5.
- **sleeptime_callback** (*function, optional*) – the function to use to determine how long to sleep after each attempt. Defaults to calculateSleepTime.
- **retry_exceptions** (*list, optional*) – the exceptions to retry on. Defaults to (Exception,)
- **args** (*list, optional*) – the args to pass to function. Defaults to ()
- **kwargs** (*dict, optional*) – the kwargs to pass to function. Defaults to {}.
- **sleeptime_kwargs** (*dict, optional*) – the kwargs to pass to sleeptime_callback. If None, use {}. Defaults to None.

Returns the value from a successful function call

Return type object

Raises Exception – the exception from a failed function call, either outside of the retry_exceptions, or one of those if we pass the max attempts.

scriptworker.utils.**retry_request**(**args*, *retry_exceptions=(<class 'concurrent.futures._base.TimeoutError'>, <class 'scriptworker.exceptions.ScriptWorkerRetryException'>)*, *retry_async_kwargs=None*, ***kwargs*)

Retry the request function.

Parameters

- ***args** – the args to send to request() through retry_async().
- **retry_exceptions** (*list, optional*) – the exceptions to retry on. Defaults to (ScriptWorkerRetryException,).
- **retry_async_kwargs** (*dict, optional*) – the kwargs for retry_async. If None, use {}. Defaults to None.
- ****kwargs** – the kwargs to send to request() through retry_async().

Returns the value from request().

Return type object

scriptworker.utils.**rm**(*path*)

Equivalent to rm -rf.

Make sure path doesn't exist after this call. If it's a dir, shutil.rmtree(); if it's a file, os.remove(); if it doesn't exist, ignore.

Parameters **path** (*str*) – the path to nuke.

`scriptworker.utils.`**to_unicode**(*line*)

Avoid `b'line'` type messages in the logs.

Parameters **line** (*str*) – The bytecode or unicode string.

Returns

the unicode-decoded string, if line was a bytecode string. Otherwise return `line` unmodified.

Return type str

scriptworker.worker module

Scriptworker worker functions.

`scriptworker.worker.`**log**

logging.Logger – the log object for the module.

`scriptworker.worker.`**async_main**(*context*)

Run the main async loop.

http://docs.taskcluster.net/queue/worker-interaction/

This is a simple loop, mainly to keep each function more testable.

Parameters **context** (`scriptworker.context.Context`) – the scriptworker context.

`scriptworker.worker.`**main**()

Scriptworker entry point: get everything set up, then enter the main loop.

`scriptworker.worker.`**run_loop**(*context*, *creds_key='credentials'*)

Split this out of the async_main while loop for easier testing.

Parameters

- **context** (`scriptworker.context.Context`) – the scriptworker context.
- **creds_key** (*str, optional*) – when reading the creds file, this dict key corresponds to the credentials value we want to use. Defaults to "credentials".

Returns status None: if no task run.

Return type int

Module contents

Scriptworker.

2.1.8 Indices and tables

- genindex
- modindex
- search

Python Module Index

Index