This document describes the current stable version of Celery (4.2). For development docs, go here.

Security

Introduction

While Celery is written with security in mind, it should be treated as an unsafe component.

Depending on your Security Policy, there are various steps you can take to make your Celery installation more secure.

Areas of Concern

Broker

It's imperative that the broker is guarded from unwanted access, especially if accessible to the public. By default, workers trust that the data they get from the broker hasn't been tampered with. See Message Signing for information on how to make the broker connection more trustworthy.

The first line of defense should be to put a firewall in front of the broker, allowing only white-listed machines to access it.

Keep in mind that both firewall misconfiguration, and temporarily disabling the firewall, is common in the real world. Solid security policy includes monitoring of firewall equipment to detect if they've been disabled, be it accidentally or on purpose.

In other words, one shouldn't blindly trust the firewall either.

If your broker supports fine-grained access control, like RabbitMQ, this is something you should look at enabling. See for example http://www.rabbitmq.com/access-control.html.

If supported by your broker backend, you can enable end-to-end SSL encryption and authentication using `broker_use_ssl`.

Client

In Celery, "client" refers to anything that sends messages to the broker, for example web-servers that apply tasks.

Star 11,909

Please help support this community project with a donation:

Donate

Previous topic

Monitoring and Management Guide

Next topic

Optimizing

This Page

Show Source

Quick search

Go

Having the broker properly secured doesn't matter if arbitrary messages can be sent through a client.

[Need more text here]

Worker

The default permissions of tasks running inside a worker are the same ones as the privileges of the worker itself. This applies to resources, such as; memory, file-systems, and devices.

An exception to this rule is when using the multiprocessing based task pool, which is currently the default. In this case, the task will have access to any memory copied as a result of the **`fork()`** call, and access to memory contents written by parent tasks in the same worker child process.

Limiting access to memory contents can be done by launching every task in a subprocess (**`fork()`** + **`execve()`**).

Limiting file-system and device access can be accomplished by using chroot, jail, sandboxing, virtual machines, or other mechanisms as enabled by the platform or additional software.

Note also that any task executed in the worker will have the same network access as the machine on which it's running. If the worker is located on an internal network it's recommended to add firewall rules for outbound traffic.

Serializers

The default serializer is JSON since version 4.0, but since it has only support for a restricted set of types you may want to consider using pickle for serialization instead.

The *pickle* serializer is convenient as it can serialize almost any Python object, even functions with some work, but for the same reasons *pickle* is inherently insecure [*], and should be avoided whenever clients are untrusted or unauthenticated.

You can disable untrusted content by specifying a white-list of accepted content-types in the **`accept_content`** setting:

New in version 3.0.18.

Note:

This setting was first supported in version 3.0.18. If you're running an earlier version it will simply be ignored, so make sure you're running a version that supports it.

```
accept_content = ['json']
```

This accepts a list of serializer names and content-types, so you could also specify the content type for json:

```
accept_content = ['application/json']
```

Celery also comes with a special *auth* serializer that validates communication between Celery clients and workers, making sure that messages originates from trusted sources. Using *Public-key cryptography* the *auth* serializer can verify the authenticity of senders, to enable this read Message Signing for more information.

Message Signing

Celery can use the pyOpenSSL library to sign message using *Public-key cryptography*, where messages sent by clients are signed using a private key and then later verified by the worker using a public certificate.

Optimally certificates should be signed by an official Certificate Authority, but they can also be self-signed.

To enable this you should configure the **`task_serializer`** setting to use the *auth* serializer. Also required is configuring the paths used to locate private keys and certificates on the file-system: the **`security_key`**, **`security_certificate`**, and **`security_cert_store`** settings respectively. With these configured it's also necessary to call the **`celery.setup_security()`** function. Note that this will also disable all insecure serializers so that the worker won't accept messages with untrusted content types.

This is an example configuration using the *auth* serializer, with the private key and certificate files located in */etc/ssl*.

```
app = Celery()
app.conf.update(
    security_key='/etc/ssl/private/worker.key'
    security_certificate='/etc/ssl/certs/worker.pem'
    security_cert_store='/etc/ssl/certs/*.pem',
)
app.setup_security()
```

Note:

While relative paths aren't disallowed, using absolute paths is recommended for these files.

Also note that the *auth* serializer won't encrypt the contents of a message, so if needed this will have to be enabled separately.

Intrusion Detection

The most important part when defending your systems against intruders is being able to detect if the system has been compromised.

Logs

Logs are usually the first place to look for evidence of security breaches, but they're useless if they can be tampered with.

A good solution is to set up centralized logging with a dedicated logging server. Access to it should be restricted. In addition to having all of the logs in a single place, if configured correctly, it can make it harder for intruders to tamper with your logs.

This should be fairly easy to setup using syslog (see also syslog-ng and rsyslog). Celery uses the **`logging`** library, and already has support for using syslog.

A tip for the paranoid is to send logs using UDP and cut the transmit part of the logging server's network cable :-)

Tripwire

Tripwire is a (now commercial) data integrity tool, with several open source implementations, used to keep cryptographic hashes of files in the file-system, so that administrators can be alerted when they change. This way when the damage is done and your system has been compromised you can tell exactly what files intruders have changed (password files, logs, back-doors, root-kits, and so on). Often this is the only way you'll be able to detect an intrusion.

Some open source implementations include:

- OSSEC
- Samhain
- Open Source Tripwire
- AIDE

Also, the ZFS file-system comes with built-in integrity checks that can be used.

Footnotes

[*] https://blog.nelhage.com/2011/03/exploiting-pickle/