argon2$_{cffi}$Documentation

Release 15.0.0

Hynek Schlawack

Aug 30, 2018

Contents

Release v15.0.0 (*What's new?*).

Argon2 won the Password Hashing Competition and `argon2_cffi` is the simplest way to use it in Python and PyPy:

```
>>> import argon2
>>> hash = argon2.hash_password(b"secret")
>>> hash
b'$argon2i$m=512,t=2,p=2$c29tZXNhbHQ$2IdoNVglVTxb9w4YVJqW8w'
>>> argon2.verify_password(hash, b"secret")
True
>>> argon2.verify_password(hash, b"wrong")
Traceback (most recent call last):
  ...
argon2.exceptions.VerificationError: Decoding failed
```

`argon2_cffi`'s documentation lives at Read the Docs, the code on GitHub. It's rigorously tested on Python 2.6, 2.7, 3.3+, and PyPy.

CHAPTER 1

User's Guide

1.1 Argon2

Note: **TL;DR**: Use Argon2**i** to securely hash your passwords.

You do **not** need to read or understand anything below this box.

Argon2 is a secure password hashing algorithm. It is designed to have both a configurable runtime as well as memory consumption.

This means that you can decide how long it takes to hash a password and how much memory is required.

Argon2 comes in two variants:

Argon2d is faster and uses data-depending memory access, which makes it less suitable for hashing secrets and more suitable for cryptocurrencies and applications with no threats from side-channel timing attacks.

Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks.

1.1.1 Why "just use bcrypt" Is Not the Answer

There's an unfortunate meme to respond to questions of storage of secrets like passwords to "just use bcrypt". The problem is, neither bcrypt nor its closest NIST-approved competitor PBKDF2 are fit for hashing passwords in the days of ASIC password breakers. In a nutshell, password crackers are able to create highly parallelized hardware specifically tailored to crack computationally expensive password hashes.

An effective measure against extreme parallelism proved making computation of password hashes also *memory* hard. The best known implementation of that approach is to date scrypt. However according to the Argon2 paper, page 2:

> [...] the existence of a trivial time-memory tradeoff allows compact implementations with the same energy cost.

Therefore a new algorithm was needed.

1.1.2 Password Hashing Competition

The Password Hashing Competition took place between 2012 and 2015 to find a new, secure, and future-proof password hashing algorithm. Previously the NIST was in charge but after certain events and revelations their integrity has been put into question by the general public. So a group of independent cryptographers and security researchers came together.

In the end, Argon2 was announced as the winner.

1.2 Installation

Generally speaking,

```
pip install argon2_cffi
```

should be all it takes.

But since Argon2 (the C library) isn't packaged on any major distribution yet, `argon2_cffi` vendors its C code which depending on the platform can lead to complications.

The C code is known to compile and work on all common platforms (including x86, ARM, and PPC). On x86, an SSE2-optimized version is used.

If something goes wrong, please try to update your `cffi`, `pip` and `setuptools` first:

```
pip install -U cffi pip setuptools
```

1.2.1 OS X & Windows

Binary wheels are provided on PyPI. With a recent-enough `pip` and `setuptools`, they should be used automatically.

1.2.2 Linux

A working C compiler and CFFI environment is required. If you've been able to compile Python CFFI extensions before, `argon2_cffi` should install without any problems.

1.3 API Reference

`argon2_cffi` comes with hopefully reasonable defaults for Argon2 parameters that result in a verification time of between 0.5ms and 1ms on recent-ish hardware.

So unless you have any special needs, all you need to know is:

```
>>> import argon2
>>> hash = argon2.hash_password(b"s3kr3tp4ssw0rd")
>>> hash
b'$argon2i$m=512,t=2,p=2$0FFfEeL6JmUnpxwgwcSC8g$98BmZUa5A/3t5wb3ZxFLBg'
>>> argon2.verify_password(hash, b"s3kr3tp4ssw0rd")
True
>>> argon2.verify_password(hash, b"t0t411ywr0ng")
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  ...
argon2.exceptions.VerificationError: Decoding failed
```

But of course, `argon2_cffi` gives you more control should you need it:

`argon2.`**`hash_password`**(*password*, *salt=None*, *time_cost=2*, *memory_cost=512*, *parallelism=2*, *hash_len=16*, *type=<Type.I: 1>*)

Hash *password* and return an **encoded** hash.

An encoded hash can be directly passed into *verify_password()* as it contains all parameters and the salt.

Parameters

- **password** (*bytes*) – Password to hash.
- **salt** (*bytes*) – A salt. Should be random and different for each password. Will generate a random salt for you if left `None` (recommended).
- **time_cost** (*int*) – Defines the amount of computation realized and therefore the execution time, given in number of iterations.
- **memory_cost** (*int*) – Defines the memory usage, given in kibibytes.
- **parallelism** (*int*) – Defines the number of parallel threads (*changes* the resulting hash value).
- **hash_len** (*int*) – Length of the hash in bytes.
- **type** (*Type*) – Which Argon2 variant to use. In doubt use the default *Type.I* which is better suited for passwords.

Return type bytes

```
>>> argon2.hash_password(
...     b"secret", b"somesalt",
...     time_cost=1,          # number of iterations
...     memory_cost=8,        # used memory in KiB
...     parallelism=1,        # number of threads used; changes hash!
...     hash_len=64,          # length of resulting raw hash
...     type=argon2.Type.D,   # choose Argon2i or Argon2d
... )
b'$argon2d$m=8,t=1,p=1$c29tZXNhbHQ$H0oN1/
↪L3H8t8hcg47pAyJZ8toBh2UbgcMt0zRFrqt4mEJCeKSEWGxt+KpZrMwxvr7M5qktNcc/bk/hvbinueJA'
```

`argon2.`**`verify_password`**(*hash*, *password*, *type=<Type.I: 1>*)

Verify whether *password* is correct for *hash* of *type*.

Parameters

- **hash** (*bytes*) – An encoded Argon2 password hash as returned by *hash_password()*.
- **password** (*bytes*) – The password to verify whether it matches the one in *hash*.
- **type** (*Type*) – Type for *hash*.

Returns `True` on success, throw exception otherwise.

Return type bool

The raw hash can also be computed:

`argon2.`**`hash_password_raw`**(*password*, *salt=None*, *time_cost=2*, *memory_cost=512*, *parallelism=2*, *hash_len=16*, *type=<Type.I: 1>*)

Hash *password* and return a **raw** hash.

This function takes the same parameters as *hash_password()*.

```
>>> argon2.hash_password_raw(b"secret", b"somesalt")
b'\xd8\x87h5X%U<[\xf7\x0e\x18T\x9a\x96\xf3'
```

class argon2.**Type**
Enum of Argon2 variants.

D = 0
Argon2**d** is faster and uses data-depending memory access, which makes it less suitable for hashing secrets and more suitable for cryptocurrencies and applications with no threats from side-channel timing attacks.

I = 1
Argon2**i** uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks.

1.4 Choosing Parameters

Finding the right parameters for a password hashing algorithm is a daunting task. The authors of Argon2 specified a method in their paper but it should be noted that they also mention that no value for `time_cost` or `memory_cost` is actually insecure (cf. section 6.4).

1. Choose whether you want Argon2i or Argon2d (`type`). If you don't know what that means, choose Argon2i (*argon2.Type.I*).
2. Figure out how many threads can be used on each call to Argon2 (`parallelism`). They recommend twice as many as the number of cores dedicated to hashing passwords.
3. Figure out how much memory each call can afford (`memory_cost`).
4. Choose a salt length. 16 Bytes are fine.
5. Choose a hash length (`hash_len`). 16 Bytes are fine.
6. Figure out how long each call can take. One recommendation for concurent user logins is to keep it under 0.5ms.
7. Measure the time for hashing using your chosen parameters. Find a `time_cost` that is within your accounted time. If `time_cost=1` takes too long, lower `memory_cost`.

`argon2_cffi`'s *CLI* will help you with this process.

1.5 CLI

To aid you with finding the parameters, `argon2_cffi` offers a CLI interface that can be accessed using `python -m argon2`. It will benchmark Argon2's password *verification* in the current environment. You can use command line arguments to set hashing parameters:

```
$ python -m argon2 -t 1 -m 512 -p 2
Running Argon2i 100 times with:
hash_len: 16
memory_cost: 512
parallelism: 2
time_cost: 1

Measuring...
```

(continues on next page)

(continued from previous page)

```
0.418ms per password verification
```

This should make it much easier to determine the right parameters for your use case and your environment.

CHAPTER 2

Project Information

2.1 Backward Compatibility

`argon2_cffi` has a very strong backward compatibility policy. Generally speaking, you shouldn't ever be afraid of updating.

If breaking changes are needed do be done, they are:

1. ...announced in the changelog.
2. ...the old behavior raises a `DeprecationWarning` for a year.
3. ...are done with another announcement in the changelog.

What explicitly *may* change over time are the default hashing parameters.

2.2 How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `argon2_cffi` is no different.

Here are a few guidelines to get you started:

- Try to limit each pull request to one change only.
- To run the test suite, all you need is a recent tox. It will ensure the test suite runs with all dependencies against all Python versions just as it will on Travis CI. If you lack some Python version, you can can always limit the environments like `tox -e py27,py35` (in that case you may want to look into pyenv that makes it very easy to install many different Python versions in parallel).
- Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.
- If you address review feedback, make sure to bump the pull request. Maintainers don't receive notifications if you push new commits.

- If your change is noteworthy, add an entry to the changelog. Use present tense, semantic newlines, and add link to your pull request.
- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- Don't break backward compatibility.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged.
- Write good test docstrings.
- Obey PEP 8 and PEP 257.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Please report any harm to Hynek Schlawack in any way you find appropriate.

Thank you for considering to contribute!

2.3 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a project maintainer at hs@ox.cx. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

This Code of Conduct is adapted from the Contributor Covenant, version 1.3.0, available at http://contributor-covenant.org/version/1/3/0/.

2.4 Changelog

Versions are year-based with a strict backward compatibility policy. The third digit is only for regressions.

2.4.1 15.0.0 (2015-12-18)

Vendoring `argon2` @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Changes:

- `verify_password()` doesn't guess the hash type if passed `None` anymore. Supporting this resulted in measurable overhead (~ 0.6ms vs 0.8ms on my notebook) since it had to happen in Python. That means that naïve usage of the API would give attackers an edge. The new behavior is that it has the same default value as `hash_password()` such that `verify_password(hash_password(b"password"), b"password")` still works.
- Conditionally use the SSE2-optimized version of `argon2` on x86 architectures.
- More packaging fixes. Most notably compilation on Visual Studio 2010 for Python 3.3 and 3.4.
- Tweaked default parameters to more reasonable values. Verification should take between 0.5ms and 1ms on recent-ish hardware.

2.4.2 15.0.0b5 (2015-12-10)

Vendoring `argon2` @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Initial work. Previous betas were only for fixing Windows packaging. The authors of `argon2` were kind enough to help me to get it building under Visual Studio 2008 that we're forced to use for Python 2.7 on Windows.

2.5 Credits & License

`argon2_cffi` is maintained by Hynek Schlawack and released under the MIT license.

The development is kindly supported by Variomedia AG.

A full list of contributors can be found on GitHub.

2.5.1 Vendored Code

Argon2

The original Argon2 repo can be found at https://github.com/P-H-C/phc-winner-argon2/.

Except for the components listed below, the Argon2 code in this repository is copyright (c) 2015 Daniel Dinu, Dmitry Khovratovich (main authors), Jean-Philippe Aumasson and Samuel Neves, and under CC0 license.

The string encoding routines in src/encoding.c are copyright (c) 2015 Thomas Pornin, and under CC0 license.

The BLAKE2 code in `src/blake2/` is copyright (c) Samuel Neves, 2013-2015, and under CC0 license.

The authors of Argon2 also were very helpful to get the library to compile on ancient versions of Visual Studio for ancient versions of Python.

The documentation also quotes frequently from the Argon2 paper to avoid mistakes by rephrasing.

msinttypes

In order to be able to compile on Visual Studio 2008 and Visual Studio 2010 which are required for Python 2.6/2.7 and 3.3/3.4 respectively, we also ship two C headers with integer types. They are from the msinttypes project (auto-import on GitHub) and licensed under New BSD:

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

a

Index