pwhash Documentation

Release 0.1-dev

Daniel Neuhäuser

April 03, 2013

CONTENTS

pwhash makes password hashing simple and safe. Instead of staying up at night wondering whether your hash function is compromised, pwhash does that for you, upgrading your hashes to the latest technology available on the fly:

```
from pwhash import PasswordHasher

pwhasher = PasswordHasher.from_config_file("pwhashc.json")
hash = pwhasher.create(u"password")
verified, new_hash = pwhasher.verify_and_upgrade(u"password", hash)
if verified:
    print(u"Valid Password")
    if new_hash is not None:
        save_new_safer_password_hash(hash)
```

CHAPTER

ONE

GET TO KNOW PWHASH

1.1 Getting Started

In order to use pwhash you first have to install it, you can do that with pip:

```
$ pip install pwhash
```

The next step is to generate the necessary configuration, which depends on your application and the machine it is deployed on, so it has to be created for every deployment. How exactly we can create a new configuration on every deployment conveniently will be covered later, for now let us just create the configuration on the current machine, to do this simply run:

```
$ pwhash-config create
```

This will ask you some questions, unless you have a good reason to, you should go with the defaults whenever those are provided. Once you are finished you will find a `pwhash.json` file, containing the configuration, in your current working directory. The next step you have to take is compiling this application configuration into the deployment configuration:

```
$ pwhash-config compile pwhash.json
```

This creates another file *pwhashc.json*, which is the deployment configuration. The deployment configuration has to be re-created for every machine on which your application is deployed. It contains information derived from the information about your application as well as machine specific information determined during compilation. This allows pwhash to adopt algorithmic cost to the machine you are deploying on, making it as difficult as difficult as possible for an attacker to get the hashed passwords.

Using that configuration we can now use it to create a `PasswordHasher`:

```
from pwhash import PasswordHasher

pwhasher = PasswordHasher.from_config_file("pwhashc.json")
```

You can use the *pwhasher* object to hash, verify and upgrade (we will learn later what that is) passwords. You can create a hash using `create()`:

```
hash = pwhasher.create(u"password")
```

In order to verify a password against that hash use:

```
pwhasher.verify_and_upgrade(u"password", hash)
```

This will return a tuple `(is_correct_password, upgraded_hash)`, *is_correct_password* is *True* if `u"password"` is actually the correct password and *False* otherwise. *upgraded_hash* will be *None* or a new more secure hash you should replace the old one with.

pwhash will upgrade a hash and provide you with an *upgraded_hash* if a new more secure hashing function is available or if the parameters for the hash function changed, which happens if you upgrade machines and re-compile the configuration.

1.2 Deploying Applications

Running `pwhash-config compile` everytime your application is deployed, is inconvenient and it is not something you can expect non-developers to do. Instead this is something that should happen if your application is deployed be it by installing using `python setup.py install` or `pip install your-app`.

1.2.1 setup.py and pip

This can be achieved by using `pwhash.packaging.Install`, this is an implementation of the command that is invoked when you execute `python setup.py install` (which is also executed by *pip*). `Install` automatically compiles all *pwhash.json* files you have declared as package data and are found within a python package.

In order to use this you have to first make `setuptools.setup()` use `pwhash.packaging.Install`, this is done by overriding the default *install* implementation using the *cmdclass* keyword argument:

```
from pwhash.packaging import Install

setup(
    ...
    cmdclass={"install": Install}
)
```

Once you have done that you to make `setuptools.setup()` include package data, which you can do using the *include_package_data* keyword argument:

```
setup(
    ...
    include_package_data=True
)
```

The next step is to teach `setuptools.setup()` which files are package data, which you can do using a *MANIFEST.in* file. You can simply include all *pwhash.json* files using the following command:

```
global-include pwhash.json
```

If you want to be more restrictive take a look at the *distutils documentation* which describes the format of *MANIFEST.in* files in more detail.

1.2.2 Other Deployments

If the above is not a solution in your case, you can roll your own. The `pwhash.config` module provides the `load()`, `compile()` and `dump()` functions which can be used to compile application configuration. If you feel your solution warrants inclusion into pwhash or if you feel you have a problem that is in need of a solution to warrants inclusion, please create an issue in the tracker.

1.3 Storing Hashes

When storing any kind of data you often need to know the size of the data, at least to do storage efficiently. If you use a hash function like bcrypt directly that's not a problem because you know the length of the hash in that case and don't have to worry about the length changing.

If you are using pwhash that is not the case. Everytime you update your pwhash version or your configuration the hash functions that are used or the parameters to these functions may change causing the password hash itself to change in length.

In order to solve this issue pwhash exposes the `max_hash_length` and `min_hash_length`. You can use these to either get an upper bound directly or calculate it yourself if you use hashers whose hash length depends on application specific factors like the password length.

In practice this means that everytime pwhash is updated or you update your configuration you possibly need to run database migrations.

CHAPTER

TWO

API

CHAPTER

THREE

ADDITIONAL NOTES

3.1 Security Considerations

Hashing and verifying passwords is one of the most sensitive parts of any application. Why then should you trust pwhash to do the right thing?

The only honest answer I can give you is that you shouldn't, nevertheless there are several measures I have taken to ensure that I don't fuck up.

3.1.1 Not implementing hashing functions

The first measure of defense is that pwhash itself contains no code that implements cryptographic hash functions. Instead pwhash uses libraries and bindings to libraries that implement those hash functions that are considered to be trustworthy and secure.

Specifically pwhash uses CommonCrypto (on OS X) and OpenSSL (everywhere else) for PBKDF2, py-bcrypt for BCrypt and the standard library modules `hmac` and `hashlib` for anything else.

3.1.2 Using timing-safe comparision functions

Unfortunately none of the implementations mentioned above provide a way to verify hashes. This is a problem because the naive solution of using `a == b` may be algorithmically correct but leaks timing information due to == being lazy. This makes it possible to determine the secret, byte by byte using a fairly low amount of attempts in a non-negligible amount of time.

So to verify passwords pwhash uses the *timingsafe_bcmp* function OpenBSD uses, to compare hashes in a manner that exposes only timing information about the total length of the compared secret hash. This should prevent any timing attacks an attacker might launch on the off chance she finds a way to control the hashes compared in a way that is useful to her.

As the output of a cryptographic hash function is supposed to be unpredicatable and we only compare hashes this is unlikely.

3.2 Contributing

Contributions in any form are always welcome. If you think you have found a bug, you have an idea you think might improve things or even better you already have code and can make a pull request, do not hesistate to create an issue or make that pull request.

Nevertheless there are a couple of things to be aware of:

1. Be nice. We have a *Code of Conduct*, please read and follow it.
2. If you have found a bug or have an idea on how to improve things, chances are someone else did as well; so please run a search on the issue tracker if someone else had the same idea already, contribute to the discussion if that makes sense, if you are the first create an issue.
3. If you are contributing code, try to be consistent with the code you are modifying, if in doubt follow **PEP 8**.
4. If you are thinking of contributing a lot of code or making big changes, get in contact before writing it. The more code you write and the more changes you do, the bigger the chance of disagreement and it is very disappointing for everyone if the code will not be merged. Let us prevent this early through communication.

3.2.1 Setting up a development environment

Setting up a development environment to work on pwhash is fairly trivial. Nevertheless this is how you get started:

The first step is forking the repository on GitHub. Once you have done that you can clone your fork to create a local repository using:

```
$ git clone git@github.com:YourGitHubName/pwhash.git
```

Having done that you should probably create a virtual environment using virtualenv and activate it. If you don't know what virtualenv is, you should learn about it before continuing, it is a very useful tool that you should be familiar with if you are doing anything serious in Python.

So once you have created your virtual environment and activated it, it is time to install all development tools and pwhash dependencies. You can do that by running:

```
$ make dev
```

This may take a while, once it is done you should have setup the development environment successfully. You can test that everything works by running:

```
$ py.test --fast
```

This will run all tests using your default python interpreter, all of which should pass.

You have no idea what to do now? Take a look at the open issues may be you will find an interesting challenge there.

3.2.2 Running tests

We use pytest for testing, you can execute the tests with it using:

```
$ py.test
```

This can take quite a while, so to speed up testing you can use the *–fast* option to skip a couple of tests that take particularily long:

```
$ py.test --fast
```

To test that everything works on all supported interpreters, with or without optional dependencies and that there are no issues with the documentation we use tox, which you can use like so:

```
$ tox
```

As the tests themselves can take quite a lot of time already, running tox takes even longer. This can be solved somewhat by using detox, which works just like tox but executes all test environments in parallel. Detox can be invoked like tox, it takes all the same arguments etc. it is just called differently:

```
$ detox
```

3.2.3 Code of Conduct

Like the technical community as a whole, the pwhash team and community is made up of a mixture of professionals and volunteers from all over the world, working on every aspect of the missing - including mentorship, teaching and connecting people.

Diversity is one of our huge strengths, but it can also lead to communication issues and unhappiness. To that end, we have a few ground rules that we ask people to adhere to when they're participating within this community and project. These rules apply equally to contributors and those seeking help.

This isn't an exhaustive list of things you can't do. Rather, take it in the spirit in which it's intended - a guide to make it easier to enrich all of us and the technical communities in which we participate.

This code of conduct applies to all communication.

- Be welcoming, friendly, and patient.
- Be considerate. Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions.
- Be respectful. Not all of us will agree all the time, but disagreement is no excuse for poor behaviour and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. Members of the pwhash community should be respectful when dealing with other members as well as with people outside the pwhash community.
- Be careful in the words that you choose. Remember that sexist, racist, and other exclusionary jokes can be offensive to those around you. Be kind to others. Do not insult or put down other participants. Behave professionally. Remember that harassment and sexist, racist, or exclusionary jokes are not appropriate for the community.

When we disagree, we try to understand why. Disagreements, both social and technical, happen all the time and pwhash is no exception. It is important that we resolve disagreements and differing views constructively. Remember that we're different. The strength of pwhash comes from its varied community, people from a wide range of backgrounds. Different people have different perspective on issues. Being unable to understand why someone holds a viewpoint doesn't mean that they're wrong. Don't forget that is is human to err and blaming each other doesn't get us anywhere, rather offer to help resolving issues and to help learn from mistakes.

This text is a slightly modified version of the Speak Up! Code of Conduct which is available under a CC BY 3.0 license and was inspired by the Fedora Project and the Python Mentorship Project.

3.3 Changelog

3.3.1 Version 0.0.1

In development

Initial release.

3.4 Licensing

3.4.1 pwhash License Text

3.4.2 Third Party Licenses

timingsafe_bcmp

PYTHON MODULE INDEX

p