jak

Release Jak v0.14.3 (Troubled Toddler)

Mar 30, 2017

Contents:

jak is a tool for encrypting files. jak works extra well if used in a git repository.

> **Warning:** jak is not ready yet. You are welcome to try jak, but we highly recommend against using it on any actual secrets yet!

CHAPTER 1

Install

```
pip install jak
```

CHAPTER 2

Quickstart

```
cd ~/folderThatMayOrMayNotBeARepo
jak start

edit jakfile # add which files you want to have be encrypted.

jak encrypt all

# All files listen in the jakfile should now be encrypted.
```

We also made this video that might be helpful for running through a basic setup in a git repository.

CHAPTER 3

Stewardship

Dispel is the main steward of jaks development. But all contributions are encouraged and welcome. Please read the *contribution guide* for more information on contributing.

CHAPTER 4

Table of contents

Basic usage

Installation

Assuming you *fullfill the basic support requirements* all you need to do is `pip install jak`.

Getting started

jak is intended for developers who want to protect secret files (such as .env files) in their shared git repositories. However there is nothing stopping jak from being instantiated into any folder nor encrypting any text file.

Let's say we have the project `flowers` which has two secret files we want to protect, `/flowers/.env` and `/flowers/settings/keys`.

```
$> cd /path/to/flowers

$> jak start

# edit the jakfile to look like this
# {
#     "files_to_encrypt": [".env", "settings/keys"],
#     "keyfile": ".jak/keyfile"
# }

$> jak encrypt all

# you can also encrypt/decrypt specific files
$> jak decrypt .env
```

Easy peasy lemon squeeze! *Read more about initializing jak here.*

Using jak without a jakfile

Heres a video that explains:

- Using jak without setup (which is fine, but not recommended for teams).
- Generating a secure key.
- Using the key to encrypt/decrypt a file via the CLI.
- Creating your own keyfile.
- One thing I do want to highlight is that the key will be stored in your CLI history, so this is not inherently more secure than keeping the key in a keyfile.

Which jak files should be committed?

commit: jakfile

ignore: .jak folder (which by default includes the keyfile)

NEVER EVER COMMIT YOUR KEYFILE! IT IS WHAT ENCRYPTS/DECRYPTS YOUR SECRETS!

keyfile

The keyfile is optional, as you can always pass through a key if you wish. This means you can store the key somewhere else if you are worried about having it in plaintext, in a file, on your computer. Which is a really bad idea if someone else has access to your computer, or you suspect your computer has been in some other way compromised. However, since you do need to use the key in some fashion to decrypt/encrypt files with jak an argument can definitely be made that having it in a file as opposed to having it in your command history (`$> history`) is about the same level of security. Passing keys to jak in a more secure way is something we are actively thinking about, and if you have opinions you should get in touch.

A Keyfile can be referenced from the jakfile (see below) or directly `jak encrypt --keyfile /path/to/keyfile`.

The keyfile should have NO INFORMATION other than a *secure key*.

key

Generate a new key by issuing the *jak keygen* command.

Since jak generates a key 32 byte key (64 characters, which jak generates as Nibbles (4bit) to keep things easy to read. If you really know what you are doing there is nothing stopping you from feeding jak 64 characters where each is a full byte though, so you could theoretically go for AES512 under this scheme.

jakfile

A jakfile holds the common settings when issuing jak commands from the current working directory that has the jakfile in it.

```
{
   "files_to_encrypt": ["file1", "dir/file2"],
   "keyfile": "/path/to/keyfile"
}
```

A jakfile has two values in it: "`files_to_encrypt`" and "`keyfile`".

The `keyfile` value is optional as you can supply a key or a different keyfile manually as an optional argument. It should point to where your keyfile is located either absolutely or relatively to the location of the jakfile. We recommend using the `keyfile` value in the jakfile due to it (1) being easier and (2) not being less secure than supplying it as a command.

You should switch your key and cycle all of your secrets if you computer is compromised.

The `files_to_encrypt` value is a list specifying the files you wish to encrypt. This serves two purposes:

1. If you are in a git repository and have added the *pre-commit hook* the hook will check against this list to identify whether you are adding a secret file in its decrypted state, and if so encrypt it for you.
2. It allows you to use the `jak stomp/shave` commands for encrypting and decrypting all of the files in the list really easily.

Diffing

Reference on the diff command.

The file being diffed should have a conflict looking something like this:

```
<<<<<<< HEAD
ZDRiM2Q0Yjg0ZTFkNDg3NzRhOTljOWVmYjAxOTE4NmI4Y2UzMTkwNTM5N2Nj
YjdiYmQyZDU3MjI1MDkwY2ExYmU0NTMzOGYxYTViY2I0YWNlYzdmOWM2OTgz
NmI5ODkxOWNhNjc5YjdiNGQ5ZDJiMTYyNDFhMzcwMWYxNDVmMWO8ttnsUSsa
iDNgzDF18NB5RMHOOxjt13wRdV_RHxtZgw==
=======
MGUwMWJhYjgxNDcyMjY2MjhmMzMzNWFlYTMwZDYzYzc5ZDc0NzVhMDc0M2Ji
ZWUyMDc2NTAyZWM5MTRkMzQ5MmU4NTBlYzY1YjlmYTUwYTdlN2M2MDg3ZTI4
NGMxNDZjYzJiZDczNGE1ZDEzYmRkZDMyY2IwMDI5Mjc3MWJmOWNXRvFeiNn8
b6JFJwpATrZOE2srs1sc3p2TM529sw-11Q==
>>>>>>> f8eb651525b7403aa5ed93c251374ddef8796dee
```

Here is a video for your viewing pleasure.

Advanced usage

Here we answer questions not many people will have.

Encryption & Security

If provided a jak generated key (complexity of the key is what determines the "number" for AES), jak will encrypt the files using AES256 which is secure. Seriously, I cannot stress this enough, using a poor password will result in poor encryption, don't do it! *Jak will generate the password for you if you ask nicely.*

What is in the hidden .jak folder?

Basically, it holds the things jak doesn't feel like you should need to be looking at.

One of the more important things jak recommends it holds is the *keyfile* which holds the auto generated key that jak uses.

It also holds the backups used to *maintain state* of the encrypted files.

How does jak maintain state?

Or stated a different (more verbose) way: **How come the encrypted content of a file doesn't change unless the files content changes?**

jak saves a copy of the encrypted files in the *.jak folder* on decryption. On re-encryption it checks whether encrypting the contents with the backups IV creates the same encrypted content. If it is the same it simply reverts to using the previously encrypted content. This method has 2 main benefits: first, it is very simple. If given the option between a simple solution and an advanced one, you should pick the easy one. Two, nothing unencrypted is stored anywhere, the backup is of the encrypted content. The only issue (as described below) is that on encryption you end up performing 2 encryptions instead of 1 for content that has changed, which for very large files (hasn't been measured yet) may incur a time cost that is deemed unacceptable.

The dev team has discussed switching to a slightly more "stupid" way of dealing with this, namely to save the modified time and simply compare that. However that would fail if the file was modified and then the changes were discarded or otherwise changed back. However that would be a constant time lookup, so it may be preferable if we find jak is used for very large files (where performing the encryption twice might become an issue).

How does jak perform the diffing at a merge conflict?

Basically jak extracts the LOCAL and REMOTE parts of the merge conflict and decrypts them back into the same file. It then provides some options for a merge tool (or plain for decrypting and then leaving it alone) to merge with.

Example conflict can looks something like this: Where the LOCAL is the top and the REMOTE is the bottom.

```
<<<<<<< SOMETHING (usually HEAD)
<some local like: asfs6e024f69113940ead0
19e7dc63e7eee99fb5db2ae37352c1d5de8643a3
f78ae736ae4027fae2acc1530a356dc6d1e360ca
cyz>
=======
<some remote like: asf6e024f69113940ead0
ff9790b8cccd50e1276c4b9ac18475d4e048f2e0
4e0034e782b64b1c9e1ac8c1cb81c3b4e43cb93f
cyz>
>>>>>>> SOME OTHER HASH
```

If you are a developer the code for diffing is right here.

Here is information on how to perform a diff.

How does the pre-commit hook work?

First and foremost, we recommend against trusting that the pre-commit hook will work, this is a failsafe and should be treated as such.

The pre-commit hook embeds logic into the regular old `.git/hooks/pre-commit` hook that exists in all git repositories. It's functionality in pseudocode is roughly this:

1. Read the jakfile for the list of files that should be encrypted and retreive the key.
2. If it can't get a list of files or there isn't a key, do nothing.
3. If there is a list of files, compare it to the files that are currently staged.
4. If a file is in both the list and in staging, encrypt it.
5. Profit.

To view the actual code you can see it in the outputs.py file. It is the `PRE_COMMIT_ENCRYPT` variable.

Command reference

Commands are given in the format `jak <COMMAND> <ARGUMENTS> <OPTIONS>`. Some example commands:

```
jak --help
jak start
jak keygen
jak keygen -m
jak encrypt file
jak encrypt file --key␣
↪64af685c12bf9f2245b851c528bdd6f41e351c8dbe614db4ea81d3486fc0ee5c
jak decrypt file --keyfile secrets/jak/keyfile
jak encrypt all
jak stomp
jak decrypt all
jak shave
jak diff
```

–help

`jak <OPTIONAL COMMAND> --help`

More information about jak or a jak command.

`jak == jak -h == jak --help`

`jak <COMMAND> -h == jak <COMMAND> --help`

–version, -v

`jak -v`

Prints out the version.

start

`jak start`

Initializes jak into your current working directory.

We highly recommend running this in the root of a git repository.

Specifically it will:

- Add a hidden `.jak` directory
- Add a *jakfile*.
- Add a *keyfile* (with a generated random 32 byte password in it) inside the `.jak` directory.
- Check if it is being run in a in a git repository
  - IF GIT: it will ask if you want to add a pre-commit hook for auto encrypting files which are specified in the `"file_to_encrypt"` value in the *jakfile* IF you should accidentally try to commit them.

  - IF GIT: It will add the `.jak` folder to the `.gitignore`.

It should give you very detailed output about what is happening.

The start command is idempotent, so you can run it many times if you (for example) on second thought would like to add the git pre-commit hook.

encrypt

`jak encrypt <FILE> <OPTIONS>` or `jak encrypt all`.

The `all` command requires a *jakfile* to exist, and will encrypt all files that are designated in the `"files_to_encrypt"` value.

optional arguments:

```
-k, --key
   jak encrypt <FILE> -k <KEY>

-kf, --keyfile
   jak encrypt <FILE> -kf <PATH TO KEYFILE THAT MUST HAVE A 32 BYTE KEY IN IT>
```

decrypt

`jak decrypt <FILE> <OPTIONS>` or `jak decrypt all`.

The `all` command requires a *jakfile* to exist, and will decrypt all files that are designated in the `"files_to_encrypt"` value.

optional arguments:

```
-k, --key
   jak decrypt <FILE> -k <KEY>

-kf, --keyfile
   jak decrypt <FILE> -kf <PATH TO KEYFILE THAT MUST HAVE A 32 BYTE KEY IN IT>
```

keygen

Generate a 32byte key that jak will accept. Returns it to the command line.

optional arguments:

```
-m, --minimal
   Makes the command only return the key with no comments
```

diff

`jak diff <FILE> <OPTIONS>`

This command will decrypt the LOCAL and REMOTE parts of a merge conflict.

It will then prompt you for if you want to open the conflict in a merge tool such as vimdiff or opendiff (default on macOS) or if you simply want the decrypted content written back into the file so you can solve it yourself using your favorite text editor.

optional arguments:

```
-k, --key
   jak encrypt <FILE> -k <KEY>

-kf, --keyfile
   jak encrypt <FILE> -kf <PATH TO KEYFILE THAT MUST HAVE A 32 BYTE KEY IN IT>
```

Read more here.

stomp

`jak stomp`

Alias for `jak encrypt all`.

Has the same options as the encrypt/decrypt commands.

shave

`jak shave`

Alias for `jak decrypt all`.

Has the same options as the encrypt/decrypt commands.

Contributor information

Just like with jaks functionality we've worked hard to make developer installation consistent and easy. It should always be quick for people to contribute and the tests should help people know whether their changes work or not BEFORE they open a PR.

We aim to be friendly to rookie devs, so if you are in doubt about proper operating procedure don't hesitate to reach out by creating an issue, we are super friendly =).

Developer machine setup

1. Clone this repo
2. Install the excellent [vagrant](https://www.vagrantup.com/)
3. See below.

```
# Boot up the vagrant machine
# This will run for quite some time, I HIGHLY recommend looking at the Vagrantfile
# for a description of what is happening.
vagrant up

# Enter sandman
vagrant ssh

# This is where the project files are mirrorer on the virtual machine
cd /vagrant

# Choose a virtualenv to work on (see virtualenvwrapper docs)
```

```
# It is recommended you use the py27 environment for development and
# then switching to py35 when you have an issue in Python 3.
workon py27

# Run tests for multiple Python versions (see tox.ini)
tox

# Or run tests in just the current environment
pytest
```

Notes of import

To edit which environments the tests should be run as see the *tox.ini* file. We would prefer to be developing against Python 3 but the reality is that it is easier to dev against Python 2 and continually make sure it also works for 3.

Updating documentation

Documentation is important because it helps people understand jak. We welcome spelling, grammar, and generally any improvements to the documentation that help people understand jak and stay secure.

jak uses sphinx to generate documentation.

To update the docs edit the `*.rst` file that has the information you want to improve upon and then:

```
# from the root of jak, probably /vagrant
cd docs

# clean out previous version and remake the html
rm -rf _build && make html
```

Inspect the new docs by simply double clicking `docs/_build/html/index.html` to open it in your browser.

Pull requests

Once your branch or fork looks good make a PR and one of the stewards will take a look at it and give you a review (and hopefully merge it!).

Alerts

It currently (2017-01-25) appears that tox will NOT run from Python 3.6 due to urrlib3 not existing. So run your tox from a different base Python environment for now.

Also PyPy tests don't seem to run in tox either, I recommend checking out the virtualenv (`workon pypy`) and running them straight with `pytest`. If someone could fix this, it would be much appreciated.

Future versions

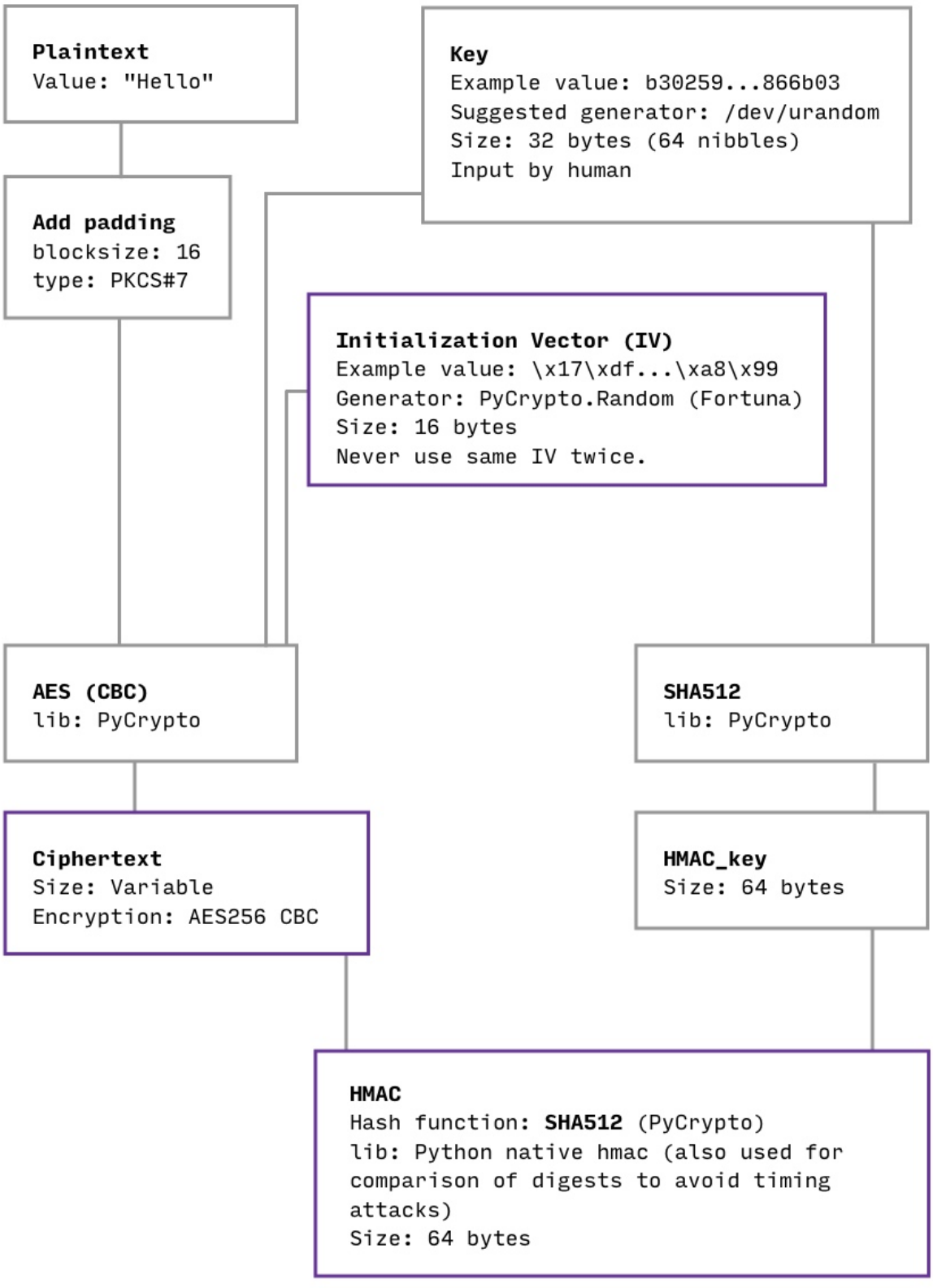
0 - 10 are the formative years. If we get past them (which seems frankly highly unlikely) we will start a new naming scheme. We use semantic versioning so the only time we shift the first number would be if we make backwards incompatible changes. The exception to this is 1.0 which will be assigned when Chris DiLorenzo thinks jak is (1) verified to be secure and (2) have no known bugs and (3) have decent tests for it's core functionality.

```
0.X Troubled Toddler        <-- CURRENT
1.X Young Whippersnapper
2.X Teenage Wasteland
3.X Highschool Sweetheart
4.X Wannabee Scientist
5.X Jaded Hipster
6.X Midlife Maniac
7.X Dorky Parent
8.X Rattled Retiree
9.X Cranky Old Seafarer
10.X Wizened Witch
```

Security

jak aims to use well-tested computer security methods, including cryptography, to protect your information. What follows is a description of how jak uses cryptographic primitives to achieve this goal. Please report any security issues related to the architecture, design, or implementation you find as a github issue or via email to cdilorenzo@dispel.io

Hopefully this image will be helpful in having you understand how the encryption and authentication works.


Plaintext
Value: "Hello"
Key
Example value: b30259...866b03
Suggested generator: /dev/urandom
Size: 32 bytes (64 nibbles)
Input by human
Add padding
blocksize: 16
type: PKCS#7
Initialization Vector (IV)
Example value: \x17\xdf...\xa8\x99
Generator: PyCrypto.Random (Fortuna)
Size: 16 bytes
Never use same IV twice.
AES (CBC)
lib: PyCrypto
SHA512
lib: PyCrypto
Ciphertext
Size: Variable
Encryption: AES256 CBC
HMAC_key
Size: 64 bytes
HMAC
Hash function: **SHA512** (PyCrypto)
lib: Python native hmac (also used for comparison of digests to avoid timing attacks)
Size: 64 bytes
Cipher version tag
Length: 7 characters (JAK-001 to JAK-999)
Static tag used to identify which version of jaks encryption implementation performed the encryption.

Encryption

jak uses the **PyCrypto** implementation of **AES256** running in **CBC-MODE** for its encryption. What makes AES be 256 is the key space of the key you use. For 256-bit you should have a 32 byte key that is as random as possible. 1 byte is 8 bits so 256 / 8 = 32. This gives you a key space of 2^32.

jak requires a 64 character hexadecimal key. It can *generate it for you.* It should look something like this `b30259425d7e5a8b4858f72948d7a232142c292997d6431efaa6a02d7a866b03`. To keep it readable we are actually representing the bytes as hexdigits, 2 hex digits are 1 byte of complexity. `b3 02 59 42` is 4 bytes. Therefore the 64 character is key 32 bytes. jak generates this key from **/dev/urandom** (`binascii.hexlify(os.urandom(32))`).

CBC-MODE requires padding. jak uses **PKCS#7** padding. In plain English that means that jak pads the plaintext secret to be a multiple of the block size (defaults to 16) by adding padding where each character is a number equal to the amount of padding. The previous sentence might be tricky, so here is an example to clarify: `pad('aaaaaaaaaaaa') returns 'aaaaaaaaaaaa\x03\x03\x03'`.

CBC-MODE also requires an **Initialization Vector (IV)**. jak generates it using the **Fortuna (PRNG)** as implemented by **PyCrypto**.

Further reading:

- https://www.pycrypto.org
- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_.28CBC.29
- https://en.wikipedia.org/wiki/Key_space_(cryptography)
- https://en.wikipedia.org/wiki/Initialization_vector
- https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7
- https://en.wikipedia.org/wiki/Fortuna_(PRNG)

HMAC

jak uses Encrypt-then-MAC (EtM) for authentication. The hash function is **SHA512**.

The key for the HMAC is simply passed through SHA512, which is questionably necessary. The argument for passing it through SHA512 is basically that it "can't hurt" and "better safe than sorry". We would love to hear your opinion on this. Read more about our reasoning here.

Further reading:

- https://moxie.org/blog/the-cryptographic-doom-principle/
- https://en.wikipedia.org/wiki/Authenticated_encryption
- https://en.wikipedia.org/wiki/SHA-2
- http://crypto.stackexchange.com/a/8086

Obtaining randomness

The random values jak generates are the **key** and the **IV**. Measuring randomness is hard if not impossible and there seems to be a great deal of differing opinions about what is a good source. The TL;DR seems to be that /dev/urandom and Fortuna are sufficiently random. But please educate yourself, it's a really interesting subject. Here are some good links to get you started.

- https://docs.python.org/3.5/library/os.html#os.urandom
- https://docs.python.org/2.7/library/os.html#os.urandom
- https://sockpuppet.org/blog/2014/02/25/safely-generate-random-numbers/
- http://www.2uo.de/myths-about-urandom/
- https://github.com/dlitz/pycrypto/blob/master/lib/Crypto/Random/__init__.py

Final thoughts

Implementing good cryptography has many not-so-subtle opportunities for an implementer, or library, to make a mistake. This situation is not helped by the fact that the types of attacks that are available is a continually changing landscape. That is why we encourage as much openness about how jak is implemented as possible so that possible issues can be caught early on.

Support

Python Support

jak is explicitly tested on Pythons:

- 2.7.7 (It is probably safe to assume jak works for 2.7.7 - 2.7.13)
- 3.3
- 3.4
- 3.5
- 3.6
- PyPy

Planned but not tested yet, but hopefully work:

- PyPy3

jak follows the Python end of support dates, which in practice means that support ends on the following dates:

- 3.3 (PEP 398) support ends 2017-09-29
- 3.4 (PEP 429) support ends 2019-03-16
- 2.7 (PEP 373) support ends 2020-01-01
- 3.5 (PEP 478) support ends 2020-09-13
- 3.6 (PEP 494) support ends 2021-12-23

For all you Python 2.7 lunatics out there that means when this clock reaches zero we drop 2.7 in the name of courage, progress and maintaining a clean codebase. It is my understanding that dropping 2.7 may implicitly mean dropping PyPy as well, which may sway this decision, since jak is a sucker for scrappy whippersnappers.

It is however likely that even without explicitly testing for it the 3.X versions will continue to work just fine even after we officially stop supporting them.

OS Support

We believe jak should work well on most *nix systems. But is mainly developed on Ubuntu and tested on Ubuntu and macOS.

Changelog

1.0 (Young Whippersnapper)

Lifecycle: Not released yet.

1.0.0 Will be assigned when we have verified that the encryption is absolutely stable AND we believe the risk of us accidentally deleting peoples secrets is < 0.0001%. In practice this means better unit testing and talking to 2-3 more cryptography experts (especially outside of Dispel). Are you such an expert? Get in touch! cdilorenzo@dispel.io.

0.14.X

Lifecycle: UNRELEASED

- **[0.14.2]** BUG: Files with the same name now support the backup feature (maintain their encrypted state if their unencrypted state is not edited on re-encryption) if they are in different folders. (PR#40)

0.14.1

Lifecycle: 2016-02-01 - current

- **[0.14.1]** HOTFIX: Import of bytestring compatibility function was removed during a merge, and it happened unnoticed. (commit)
- **Other:**
  - DOCS: fixed static links for the terminal examples (I guess readthedocs changed something?).

0.14

Lifecycle: 2016-02-01 - 2016-02-06

- **[0.14.0]** FEATURE: jak encrypt/decrypt commands can now accept a list of files (jak encrypt file1 ... fileN -k <key>). (PR#34)
- **[0.13.0]** FEATURE: jak works for all type of files, not just text files. (PR#33)
- **[0.12.0]** FEATURE: Add encryption versioning. This allows us to upgrade/edit the cipher and still decrypt previous ciphertexts (so they don't become undecryptable) (PR#31)

0.11

Lifecycle: 2017-01-23 - 2017-02-01

- **[0.11.0]** FEATURE: Properly use HMAC to make sure the ciphertext has not been tampered with. (PR#28)
- **Other:**
  - Upgraded the dev environment (PR#29)

  - *Added security section to the documentation*

Acknowledgements:

- Huge thank you to @obscurerichard (Richard Bullington-McGuire <richard@moduscreate.com> / @obscurerichard on GitHub & Twitter) for figuring out that jaks authentication could be improved.

0.10

Lifecycle: ~2017-01.

- **[0.10.0]** FEATURE: Switched to CBC mode for AES from CFB. (PR#14)
- **[0.10.1]** CLEANUP: Encrypt/Decrypt file services were a mess.. (PR#15)
- **[0.10.2]** ENHANCEMENT: Make keyfile location in jakfile relative instead of absolute. (PR#22)
- **[0.10.3]** BUG: Wrong key should print filepath. (PR#21)
- **[0.10.4]** ENHANCEMENT: Made sure jak worked well in Python 3, 3.3, 3.4 and PyPy. (PR#19)
- **Other:**
  - DOCS: Add videos of terminal usage, a ton of text content and this changelog. (PR#27)

0.0 - 0.9 (Troubled Toddler)

Lifecycle: ~2016-11 - ~2016-12

Birth.

CHAPTER 5

Support

jak works if you have a modern Python (2.7-3.5) installed on a *nix system.

You can read about it in excrutiating detail here.

CHAPTER 6

Proposed future features and enhancements

- Make maintaining encrypted state of files optional
- Avoid polluting filesystems with .jak folders?
- Windows support
- Easier key rotation

CHAPTER 7

License