

# Outline

## Abstract

The prospect of writing a cryptographic application started out simply; code something up capable of encrypting and decrypting content. The idea of encrypting messages quickly expanded into encryption of notes, or entire files. This generalization of target content led us to the idea of a platform-independent editor plugin that could handle encryption of arbitrary data. The choice of editor was clear: VIM. It runs on many platforms and can execute Python through it's vimconfig and plugin interfaces. This plugin registers file io handling functions with VIM which replace the default ones. All disk access from the editor is thus routed through this plugin, ensuring that all externally observable data is put through an encryption engine, including temporary files.

## Introduction

Any encryption plugin needs to be flexibly architected so that it can keep up with the cryptographic arms race. To support this, we identified two main areas of development. The first is the VIM interface, which describes to the editor what our library will be responsible for. The second is an extensible encryption library that can handle file IO.

- VIM's existing encryption functionality is limited in configuration options.

The encryption library is based on encryption engines, which implement the header processing and encryption/decryption APIs. Once a file is loaded and the header is processed, if that file requires vimcrypt, the necessary Engine is loaded. That engine is then handed the file handle to scan for any additional meta-data it requires. Any subsequent disk reads are done through `EncryptionEngine.decrypt` and disk writes through `EncryptionEngine.encrypt`.

## Background

Vim provides some builtin encryption functionality that can be used with the `-x` argument on the commandline and `:X` command in Vim, which both will prompt you for a key with which to encrypt the file. Vim supports 3 ciphers (Pkzip, blowfish, and blowfish2) and by default will use Pkzip which the `:help encryption` documentation in vim describes as "The algorithm used is breakable. A 4 character key in about one hour, a 6 character key in one

day (on a Pentium 133 PC)". Blowfish is also compromised but fixed in blowfish2. Blowfish2 provides strong encryption but is vulnerable to undetected modification. [4]

Vim's script repository has also published a plugin which passes through file reads and rights to the Gnu Privacy Guard suite, known as `gnupg.vim`. The plugin implements encryption by attaching commandline GPG calls to Vim's "cmd-event" triggers which allow plugins to overload filesystem operations. The dependancy on Gnu Privacy Guard being installed greatly reduces the value of the plugin compared to a stand-alone solution. [5]

We also searched <https://vimawesome.com/>, the largest directory of vim plugins on the web, for any plugins which implement encryption. The only plugin related to cryptography available at this time is for cryptographic checksums. Vim appears to lack cross platform, configurable, and secure encryption functionality. Vimcryption attempts to address this need as a self-contained python based plugin.

## Methodology

- Setup VIM disk access hooks
  - File Read
  - Buffer Read
  - File Write
  - Buffer Write
  - File Append

The encryption API was defined and encoded into an abstract base class called `EncryptionEngine`. The API requires any `EncryptionEngine` to define two methods, `encrypt(buffer, file_handle)` and `decrypt(file_handle, buffer)`. `encrypt` takes a VIM buffer and applies encryption to it before writing it to the file handle. `decrypt` takes a file handle which it reads and decrypts into a VIM buffer. We set up a unit test environment using Python 2.7 and 3.4 with `nose2` to test the `EncryptionEngines`. A virtual environment is created for both Python versions, `vimcryption` is installed in each and then the test suite is run. Each engine is tested to ensure that it's `encrypt/decrypt` functions match the algorithm they are supposed to implement. Once the disk access hooks were configured, a pass-through engine was implemented to test the connection. This `PassThroughEngine` simply writes buffer contents to the file and vice versa, but proved that the paradigm would work. With the architecture proven, we needed an engine that actually modified file contents on disk. To start, a simple keyless base64 encoding was used to scramble the contents.

With the framework set up and doing simple encryption, continued development will be focused on implementing additional testing and

encryption schemes. VIM configuration and installation unit tests are needed to ensure compatibility isn't affected by any future changes. A pure python implementation of AES128 will allow passkey based encryption in an easily delivered, cross-platform package.

## Resources

[1] Losh, Steve. "Learn Vimscript the Hard Way." Learn Vimscript the Hard Way. Accessed March 08, 2018.

<http://learnvimscriptthehardway.stevelosh.com/>.

[2] "Vim Documentation: Autocmd" vimdoc. Accessed March 08, 2018.

<http://vimdoc.sourceforge.net/html/doc/autocmd.htm>

[3] "Vim Documentation: Python Module" vimdoc. Accessed March 08, 2018.

[http://vimdoc.sourceforge.net/html/doc/if\\_pyth.htm](http://vimdoc.sourceforge.net/html/doc/if_pyth.htm)

[4] "Encryption" Vim Tips Wiki. Accessed March 08, 2018.

<http://vim.wikia.com/wiki/Encryption>

[5] Markus Braun, James McCoy. "gnupg.vim" (2012) GitHub Repository.

<https://github.com/vim-scripts/gnupg.vim>