Project Final for Vimcryption

Authors: Tom Manner, Miguel Nistal MSCS630 Spring 2018 https://github.com/Vipyr/vimcryption

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, despite being one of the leading text editors in system administration and development, notably lacks extensible cross platform encyrption functionality. Builtin solutions use weak ciphers and developer provided recepies are based on a Gnu specific dependancy. Vimcryption addresses this need by providing a Vimscript to Python API to load and select self-contained ciphers at runtime.

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 vimcryption, the necessary Engine is loaded. That engine is then handed the file handle to scan for any additional meta-data it requires. Any sybsequent 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 vunerable to undetected modification. [4]

Vim's script repository has also published a plugin which passes through file reads and writes to the

Gnu Privacy Guard suite, known as `gnupg.vim`. The plugin implements encryption by attaching commandline GPG calls to Vim's "Command-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 cross platform 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

In it's role as a text editor, Vim is directly responsible for all the file operations a user needs in order to interact with the filesystem. Additionally, Vim provides quality of life features to the user such as swap and backup files in case the session is interrupted or corrupted, undo files so the user can have a persistant undo stack, and logs command history so the user can repeat commands used earlier. To implement a secure encryption extension to Vim, we need to take into account the entire dataflow to ensure that plaintext is not visible outside of the active memory of the editor's process.

Securing the unintential leak of plaintext data via temporary files is relatively simple in Vim. As discussed in the Vim Tips Wiki, we can disable the creation of temporary files by using vim-script in our plugin load [4]:

```
setl noswapfile
setl noundofile
setl nobackup
set viminfo=
```

It's important to note that the above commands use the `setl` syntax which means "set local to the buffer". Since we want users to be able to work on encrypted and unencryted files simultaneously, we need to ensure that any plugin configurations don't effect other active files.

Intercepting the actual reading and writing of the buffer is directly supported by Vim through through the use of Command-Events, which are specialized Auto-Commands that specifically allow the overloading of filesystem triggers. [2] We can then tie these file system triggers to event handlers in Python and write/read the file system and buffer directly through a file handler there.

```
au BufReadCmd    *    py VCF.BufRead()
au FileReadCmd   *    py VCF.FileRead()
au BufWriteCmd   *    py VCF.BufWrite()
au FileWriteCmd  *    py VCF.FileWrite()
au FileAppendCmd *    py VCF.FileAppend()
```

The `py` command used in the snippet above is the result of builtin Python plugin support provided by Vim. When Vim loads, it instantiates an interpreter process which is active throughout the lifetime of the application. This behavior allows our plugin libraries to maintain state in between calls, so we can read user configurations at plugin load time for defaults and file meta-data during `FileRead`/`BufRead`. Based on the settings and meta-data, the File Handler will instantiate an engine from the encryption API to process the text.

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`[6] to test the `EncryptionEngine`s. 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[7] 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.

Conclusion

This methodology proved very effective, especially when supported by some key tools and language features. Python's `generator` and `iterator` patterns made it very easy to write code that performs well and is easy to extend without making the functional intent unclear. Combining these allowed us to write generic code that is able to handle arbitrarily large files or IO streams. An iterator for block ciphers is concisely defined as a generator function, yielding one block at a time from the stream. The function reads like a simple for loop, but effectively pauses each time an element is yielded. Definition of a block cipher engine requires only that the block size and encrypt/decrypt functions are defined.

The VIM API for Python proved to be easy to set up, mostly thanks to very thorough documentation. It helped especially with some of the non-obvious steps associated with intercepting file access safely. There turns out to be a concise framework for file handling without affecting other plugins or VIM itself. The Vimscript API and ability to invoke VIM also allowed us to write integration tests that automated regression testing our plugin in situ.

To support smooth development, we set up a full agile development environment and workflow. This includes GitHub forks of the code repository for each of us, issue tracking with an agile board, and continuous integration using TravisCI. Unit tests, coverage, and static analysis tools were invoked by TravisCI on each repository update and pull request. By tuning the static analysis tools and ensuring full test coverage, we were able to nearly guarantee that code written by any developer on any fork could not cause breakage anywhere else. In the case that one fork did contain bad code, no disruption occurred because the forks fully isolated our development environments. Combined with judicious use of the agile board, independent development of work items was incredibly smooth.

Future Development Plans

The final project might be done, but development on Vimcryption is far from over! We've been looking for a fun and interesting open source project to work on for a long time and working with cryptography in Python has been a great experience. Based on what we've done so far, the path forward has been pretty clear and we've continued to document our roadmap using ZenHub and Issues on the new Vimcryption repository. The next important steps are to address security concerns in the library by updating the AES algorithm to operate in CBC mode, and to fix the salting of the password expansion to use different salt values for each password hash to defend against rainbow attacks on the header. We'll also be implementing AES192 and AES256 to improve the security of the ciphers available to the users.

Even with a variety of ciphers with secure implementations, Vimcryption still lacks the most important tool in any software toolbox - Extensibility. We've implemented a hook in the factory that allows users to dynamically import their own libraries, but this still lacks any Vimscript API or meaningful examples of how to use it. We plan on addressing this by extending the factory hook into the Vimscript to allow users to add to their cipher options directly from their `.vimrc` file. To illustrate the usage, and to begin laying the groundwork for external contributors, we'll also implement the Ceasar Cipher in such a way that we can make it available as a tutorial. The Ceasar Ciher makes a great candidate for this purpose due to it's extreme simplicity, so that users can focus on the API and Engine implementation rather than cipher details.

Finally, after all the development effort in both Vim and the Cipher engines, it's become clear that the EncryptionEngine package could easily stand on it's own as a library. Python currently lacks a standard library solution for encryption (despite cryptographic hash support via `hashlib`). Using generators to process blocks of text has become a cornerstone in efficient Python development, and being able to do so with cryptography feels like a natural extension of the language. When stepping back and looking at the EncryptionEngine package on it's own, FileIO becomes a clear next step. We plan on developing a context manager for File IO that would allow users to process encrypted files in exactly the same way as they would regular files through the standard `open()` and file handler interface. Competing libraries in the space such as `pyaes` currently don't support modern Python constructs such as context managers and generators or familiar syntax from the standard library. We see a great oppurtunity to contribute to the cryptographic python community and hope we might even get to publish a PEP!

Resources

[1] Losh, Steve. "Learn Vimscript the Hard Way." Learn Vimscript the Hard Way. http://learnvimscriptthehardway.stevelosh.com/.

[2] "Vim Documentation: Autocmd" vimdoc. http://vimdoc.sourceforge.net/htmldoc/autocmd.htm

[3] "Vim Documentation: Python Module" vimdoc. http://vimdoc.sourceforge.net/htmldoc/if_pyth.htm

[4] "Encryption" Vim Tips Wiki. http://vim.wikia.com/wiki/Encryption

[5] Markus Braun, James McCoy. "gnupg.vim" (2012) GitHub Repository. https://github.com/vim-scripts/gnupg.vim

[6] Pellerin et al. "nose2" (17 Feb 2018) GitHub Repository. https://github.com/nose-devs/nose2

[7] "Base64" Wikipedia. https://en.wikipedia.org/wiki/Base64