Agile Crypto Speclet

Michael Howard ([mikehow@microsoft.com](mailto:mikehow@microsoft.com))

1/11/2018

The provided code is a simple proof concept that demonstrates how to be more agile when building applications that use crypto. In this proof of concept, three aspects are demonstrated:

* Agile password-based key derivation
* Agile symmetric encryption
* Agile message authentication codes

Again, remember, this is a proof of concept only to get people thinking about the issue, any suggestions or improvements are welcome!

If you are not aware of what agile crypto is, please read Bryan Sullivan’s intro to the topic at <http://msdn.microsoft.com/en-us/magazine/ee321570.aspx>.

# High level operation

The code can either encrypt & MAC some plaintext or MAC & decrypt some ciphertext. When encrypting, the proof of concept code takes a small config file containing crypto policy (named crypto.config) and uses this to encrypt and MAC a string of data. The results of the crypto operations are then displayed on the screen and can be redirected to a file if needed.

If the MAC over the ciphertext check fails, the data is not decrypted and an exception is raised.

# Assumptions & Limitations

* A basic understanding of crypto is assumed.
* The code uses .NET 4.x and the appropriate libraries are already installed on the user’s machine.
* The code has not been tested with every possible combination of symmetric cipher x block mode x key size x padding mode.
* The class names are .NET specific – this could not allow interop with Java or Python.[[1]](#footnote-1)
* The code is not particularly robust, it’s a PoC!
* The code assumes there’s a class factory that allows crypto objects to be created at runtime based on their names.
* The code only works with symmetric algs.
* The code is designed to protect small amounts of data.

# Crypto Config file

The format of the config file is very simple, it’s three lines of text (not XML,) the lines are:

* PBKDF class settings
* Symmetric key algorithm settings
* HMAC algorithm settings

A sample config file looks like this:

System.Security.Cryptography.Rfc2898DeriveBytes|{"IterationCount":10000}

System.Security.Cryptography.AesCryptoServiceProvider|{"KeySize":128,"BlockSize":128,"FeedbackSize":8,"Mode":1,"Padding":2}

System.Security.Cryptography.HMACSHA256|{"HashName":"SHA256","HashSize":256}

Note the first part of each line (before the ‘|’) is the name of the class to create at runtime, and the second half are JSON representations of class members. Some class members are missing as they are populated at runtime when the class is instantiated.

# Demo

The following is a series of steps to demo the functionality.

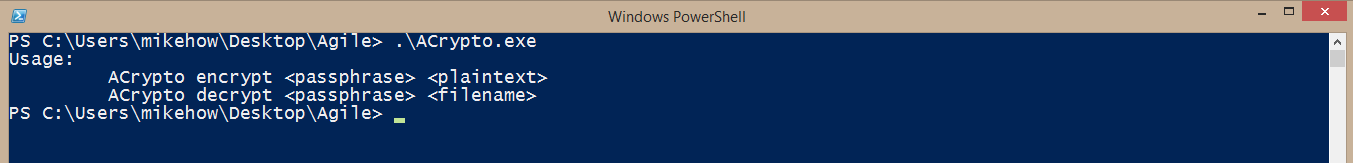
Open the SLN file and rebuild the solution.

Navigate to the

Type the following:

acrypto

And you should see usage requirements:

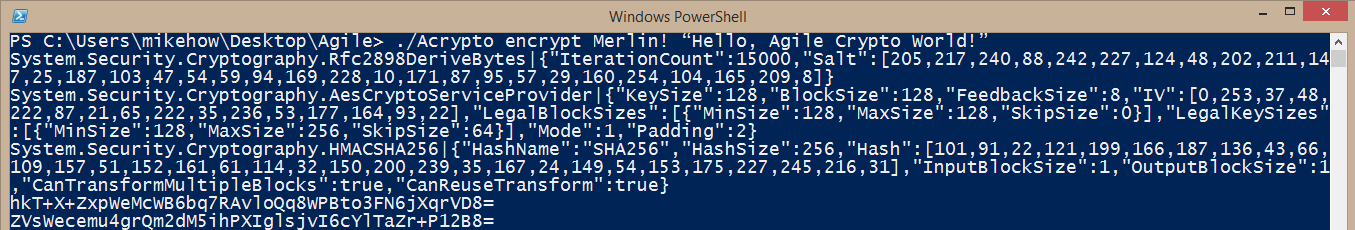


If you see errors about missing assemblies you should probably install VS 2013.

Now, let’s encrypt some data.

acrypto encrypt Merlin! “Hello, Agile Crypto World!”

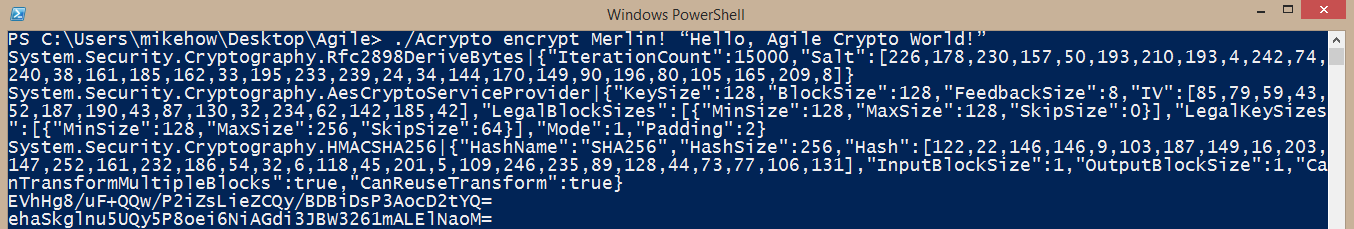
You will see output like this:



The first set of lines are the JSON metadata for each of the three classes (PBKDF, symmetric cipher and HMAC.) The two lines at the bottom are Base64 ciphertext and HMAC respectively.

Run the same command again.

acrypto encrypt Merlin! “Hello, Agile Crypto World!”



Note the ciphertext and HMAC are different, this is because the salt for the PBKDF and the IV for the block cipher are generated by the code automatically each time it’s run. So even in the face of the same password being used, the output is different.

You can see the IV and salt in the output.

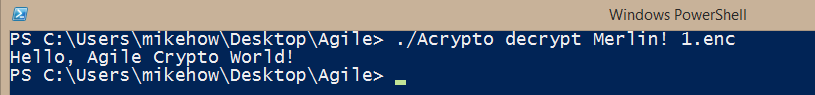
Run the command again, but let’s output to a file.

acrypto encrypt Merlin! “Hello, Agile Crypto World!” > 1.enc

To decrypt (and integrity check) the blob, type the following:

acrypto decrypt Merlin! 1.enc

and you will see the plaintext:



Now let’s rev this up a little. Open the crypto.config file (note, it’s fragile, so keep a backup copy!) and change this:

System.Security.Cryptography.Rfc2898DeriveBytes|{"IterationCount":15000}  
System.Security.Cryptography.AesCryptoServiceProvider|{"KeySize":128,"BlockSize":128,"FeedbackSize":8,"Mode":1,"Padding":2}  
System.Security.Cryptography.HMACSHA256|{"HashName":"SHA256","HashSize":256}

To this

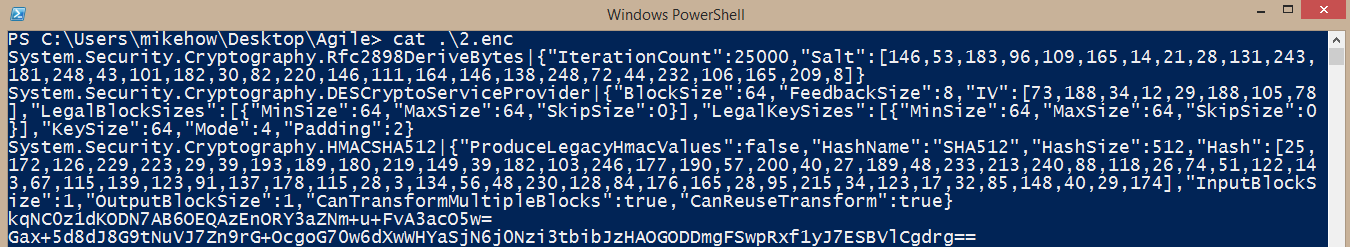
System.Security.Cryptography.Rfc2898DeriveBytes|{"IterationCount":25000}  
System.Security.Cryptography.DESCryptoServiceProvider|{"KeySize":64,"BlockSize":64,"FeedbackSize":8,"Mode":4,"Padding":2}  
System.Security.Cryptography.HMACSHA512|{"HashName":"SHA512","HashSize":512}

We have changed a lot; we changed the cipher (AES 🡪 DES), the PBKDF iteration count (15000 🡪 25000), and block mode (CBC (1) 🡪 ECB (4)) and finally, the HMAC (SHA256 🡪 SHA512.)

Now re-run the tool to encrypt the data ***(redirect to 2.enc, not 1.enc)***:

acrypto encrypt Merlin! “Hello, Agile Crypto World!” > 2.enc

If you look at the resulting file, you’ll see that it looks quite different from 1.enc because we changed so much. Note espically that the MAC is huge (SHA512) compared to the previous example (SHA256.)

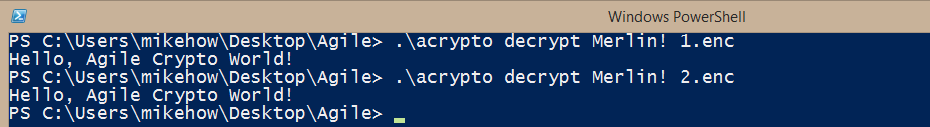


Finally, the ultimate test – decrypting both blobs with the same code and no code changes needed:

Simply type the following, one after the other and witness the result:

acrypto decrypt Merlin! 1.enc

acrypto decrypt Merlin! 2.enc



Voila!

Feedback Welcome!

<SNIP – IGNORE – NOT COMPLETE OR ACCURATE>

# Encrypt operation

1. Read crypto config
   1. Get PBKDF class data (JSON)
      1. alg
      2. iteration count
   2. Get Cipher class data (JSON)
      1. alg
      2. Get block mode
      3. Get padding mode
      4. Get block size
      5. Get key size
   3. Get HMAC class data (JSON)
2. Get passphrase from arg[1]
3. Get plaintext from arg[2]
4. Generate random salt (RNG || tick count)
5. Derive encryption key from PBKDF(passphrase,salt,iter)
6. Derive HMAC key from PBKDF(passphrase,salt,iter)

Pwd 🡨 User info

1. It would be awesome to solve this problem! [↑](#footnote-ref-1)