INFORMATION SYSTEM APPLICATION, Vol. 01, No. 02
ISSN 2503-166X

Self-Destruct-Capable Multi-Tiered Deniable Data Encryptor

[1]Nur Hadisukmana, [2]Andika Chandra Jaya
[1,2] President University, Faculty of Computing, Jl. Ki Hajar Dewantara, Cikarang Baru – Cikarang, Bekasi 17550
[1]E-Mail: anursu2002@yahoo.com

***Abstract*— Data security and privacy often collides with government desire to protect its People. As government is responsible to create laws and execute those, government often violate innocent Citizens privacy to hunt for some lawless Person. When some of the People do not mind, many others resort to breaking some laws to protect their privacy. One controversial law is regarding encryption, some nations required the encryption system creator to deliberately leave a backdoor in which government can spy on their Citizen. This application, named New Encryptor, provides data security from unauthorized eyes, including government. Through this application, government agencies would have to spend more time in decrypting the traffic, making only warranted searches can be conducted as it should consume increasing amount of resources. This application consist of layered security, steganography, self-kill, and deniable encryption. It is capable of AES encapsulating PGP alongside with steganography to hide the password and deniable encryption that can draw random image.**

***Keywords*—encryption, decryption, layered security, steganography.**

I. INTRODUCTION

Data security is often important, even when the data in question are trivial data like downloaded images. A determined identity thief is willing to steal any data to obtain clue about someone to be sold at any market price. When information about someone is lost, so does the privacy regarding that information. Hence, privacy through data security is necessary. Problem here arises when the desire of privacy clashed with government demand of information openness under pretext of 'national security.'

Most cryptographic software requires some form of stored key. According to Kerckhoff's Principle, "The strength of any encryption delies on the strength of its keys." Therefore, the vulnerability of any encryption system is in the key. Through the stored key, it can leave the software vulnerable should it be compromised through either government intervention or sheer carelessness; and everyone who compromise that software can disguise as the actual owner. The example of the government intervention that may compromise any cryptographic system is British Regulation of Investigation Power Act of 2000. In that law, the government can legally demand the key to any encryption software [1].

In Indonesia, this kind of disclosure is achieved through Electronic Information and Transaction Law (Undang-Undang Informasi dan Transaksi Elektronik) Article 1.1 which denotes encryption system as Electronic Information and can be used as evidence in criminal case. Article 42 of the same law states that "Investigation of any crime as stated in this law, is conducted according to Code of Criminal Procedure (Kitab Undang-undang Hukum Acara Pidana) and regulations in this law." Code of Criminal Procedure of 1981, Article 175 allows convict to stay quiet if the testimony may incriminate the convict. However, the convict may be deemed as uncooperative in court to increase the verdict [2][3].

Therefore, improved encryption system should involve no internally-stored security key that can be compromised. To achieve that, there are three possible options to remove the key from the encryption; namely hiding the key in plain sight, sending the key altogether, or creating "deniable encryption."

The objectives of this research are:

1. To create application to explore the current cryptography system through layered security, steganography, and deniable encryption.
2. To exploit the limitation of current data privacy as denoted in the Fourth Amendment and Fifth Amendment to The Constitution of The United States of America.

II. SYSTEM ANALYIS

The program in this research is built based upon layering approach in which each preceding encryption algorithm encrypts whatever is in the inner layer. Additional features including steganography to hide the outermost key, self-kill capability that kills the process if there exists several failures in entering the right password, and multi-key capability that can be used to mislead the cryptanalyst as the deniable encryption.

In Figure 2.1, the arrow indicates where the data in process would go during encryption process. Notice that the PGP ciphertext and its own key are merged into one ciphertext to be encrypted deeper using AES.

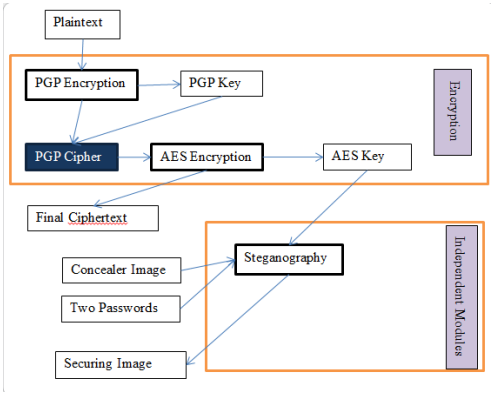

Figure 2.1 Encryption Process

In decryption, referring to Figure 2.2, the data direction also follows the arrow. Notice that the self-kill capability is closely tied with deniable encryption due to its reliance upon combination of stored passwords and input password in the decryption side.

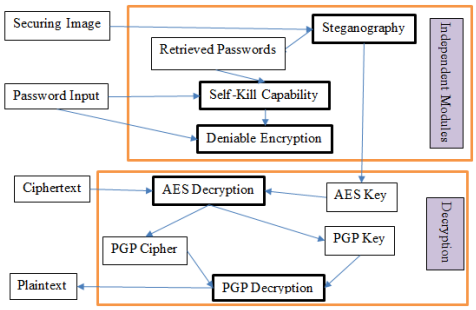

Figure 2.2 Decryption Process

The use case for the program is denoted in Figure 2.3.

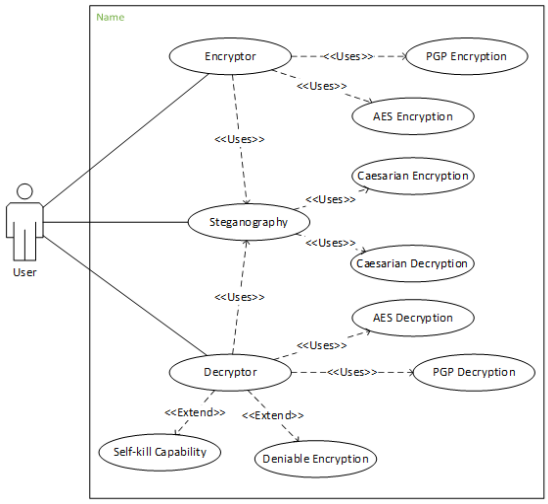

Figure 2.3 Use-Case Diagram

III. SYSEM DESIGN

3.1 GUI Design

Graphical User Interface is required for most modern application to let user intuitively guide themselves through the program without lengthy instructions.

3.1.1 Encryptor UI

Figure 3.1 is the UI for encryptor. It has straightforward look with everything is shown in one screen: input of files, passwords for further application use, preview of the concealer image, command button, and progress bar.

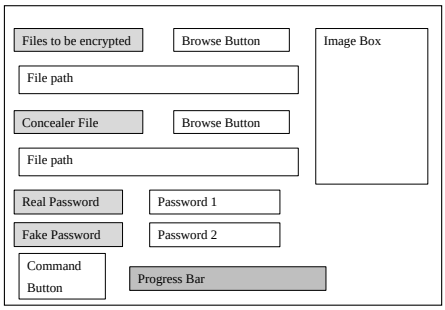

Figure 3.1 Encryptor UI

3.1.2 Decryptor UI

Shown in Figure 3.2 is the decryptor UI, similar to encryptor side, which has equally straightforward look with few noticeable differences: one password input instead of two and remaining chances instead of progress bar.

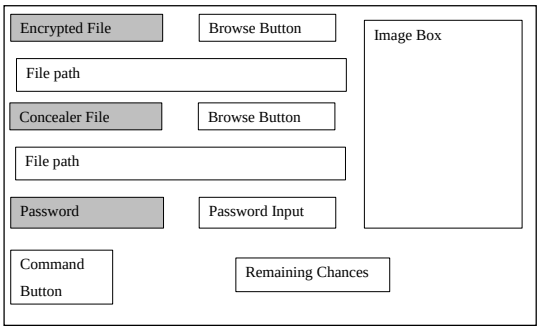

Figure 3.2 Decryptor UI

3.2 Top-Level Algorithm

The encryption is designed around Figure 3.1, it is a straightforward algorithm, user must provide two passwords, concealing image/file, and the plain file to be encrypted. Then PGP will do its job to encrypt the file and it produces its own key. After which, the PGP key will be embedded into the resulting ciphertext and denoted with new line and iterated once more with AES, also with its key embedded after new line. Both will have the steganography engaged to concealer image alongside the AES key. Finally, all files (adulterated image and ciphertext) are stored separately. In each step, the progress bar will progress alongside the steps of encryption finished.

While the decryption is designed around Figure 3.2. User must provide three items here: one password, concealer file, and the cipher file. Then self-kill system will inspect the password to check whether it is valid or not. If it failed the validity test, then the self-kill mechanism will be

triggered and allow only two more attempts which upon failure of the third attempt will kill the program.

After passing the validity check, the program will then determine whether the password is real password or faked one. If it is fake password, then the application will turn the ciphertext into image.

The real decryption starts if the password is real one. First it will retrieve AES key from the image which is used to decrypt the outer layer security. The AES plaintext contains PGP ciphertext and PGP key. Extract PGP key and the actual plaintext can be deciphered which is done immediately. Users then save the plaintext file. Then the user will be notified that the decryption is done.

IV. SYSTEM DEVELOPMENT

The application in this research is built with modular approach to ease maintenance. The application is built in three modules and one main class. The difference between module and class-object is that module does not need to be instantiated while class needs to be instantiated as object. The target is still the same, enhance modularity in which each function inside each module can be modified internally without ever having to change any calling segment.

4.1 Main Class

The Main class is used as the interface between the application and the user. The interfacing is crucial to speed up the maintenance and to improve modularity. For instance, adding or removing new modules/functions (both are considered equal in this application). Modularity in this program is greatly aided by the main class.

4.2 Deniable Encryption Capability

This application can draw an image out of the ciphertext byte as “encryption denier” scheme. As long as one can re-create the header, any random file can be generated in any format. For the matter, it does not care whether the file is valid or corrupted as long as the ciphertext byte is hidden there.

4.3 Ordinary Encryption

This side of application is used to scramble the plaintext into unintelligible (cipher) text. The sequence of the processing in this module is hash and sign, compress, symmetric encryption of the compressed plaintext, encrypt the symmetric key asymmetrically, merging (already covered in the Main Class), and finally outer layer encryption.

Most of the function in this application is API library-based. API stands for Application Programming Interface, which in turn is a set of routines to aid building software applications. APIs are typically used by developer in order to reduce their workload by applying what the IDE (Integrated Development Environment) already has in certain areas. In this application, the API used is primarily from Windows API which mostly being imported as System.Security.Cryptography. Hence, in developing the application, there is no need to obtain any library other than the ones available.

4.4 Ordinary Decryption

This side of the encryption is to retrieve the plaintext from ciphertext. Similar to the encryption side, this side also relies on library and API (all from Visual Studio). The order of decryption is the reverse order of encryption.

4.4.1 Verify Hash and Signature

To validate the hash value from the decompressed file the data is hashed and then compared with the signed hash. Also included that the function demands the RSA key to be working. Then the data is verified by comparing the signature and verifying the hash of the data.

4.4.2 Asymetric Decryption

The Asymmetric decryption is to retrieve the symmetric key required to retrieve the compressed plaintext. This code will receive the ciphertext as string directly from the decryptor near the user interface. The key for this encryption is obtained from the previous step in the whole decryption process.

It will first convert the input from base64 to byte array which will be decrypted. In Visual Basic, it is possible to have implicit return from a function as long as the function output is defined and call the function name as if it were variable.

4.4.3 Decompression

In order to decompress the decrypted plaintext, it uses MemoryStream that based on the input byte array and the stream is non-resizable. Also create buffer which has the same size as the input in order for the decompression stream to read into. In the loop to obtain the decompressed byte, another memory stream is opened which is written from the buffer. Then, the output stream is turned into byte array as output.

4.4.4 Symmetric Decryption

```
    Function symDec(ByVal input() As Byte) As Byte()
        Dim tdes As New TripleDESCryptoServiceProvider
        Dim key = TDESkey.key
        Dim iv = TDESkey.IV

        Dim memDeStream As New MemoryStream(input)
        Dim cryptStream As New CryptoStream(memDeStream,
tdes.CreateDecryptor(key, iv), CryptoStreamMode.Read)
        Dim fromCrypt(input.Length - 1) As Byte

        cryptStream.Read(fromCrypt, 0, fromCrypt.Length)

        cryptStream.Close()
        memDeStream.Close()

        Return fromCrypt
    End Function
```

Figure 4.1 Symmetric Encryption

The function in Figure 4.1 is similar to the encryption side. However, reading from stream which requires another variable as buffer is the distinct differences between the encryptor and decryptor.

The working of the function is by creating the Triple DES object, retrieve key (which has been decrypted earlier and set as module-level variable at the interface), allocate memory stream, generate decryption stream, create the

buffer, read from stream to the buffer, close everything, and finally return the buffer as output.

Cryptostream mode determines what the function running the object can do. If it is set at read, it can only read.

4.5 Additional Encryption/Decryption Modules

This Module requires three module-level variables and those will be deemed as property belong to the module. Also included five imports that enable all necessary functions in the application, particularly this module.

There are five imports (four namespaces and one class) used in this program: Security.Cryptography, IO.File, IO, IO.compression, and Text. IO namespace handles for file input/output and memory stream. IO.File class provides functions for operating with files. IO.Compression is to provide compression routines (GZip). Security.Cryptography provides all classes regarding encryption including cryptographic stream, hashing algorithm, and cryptographic service providers.

Three module-level variables that contain the key for each encryption phase. Each key is to be used either in the Main Class as part of ciphertext that will obviate the demand to store the key or in the Steganography as part of the information stored in the image.

These variables are encapsulated throughout the module, making access of these variables requires get method to read the content and set method to write to the variable from outside the module. This is required as the key is to be merged in the main class. In Visual Studio, *Structure* keyword is reserved to form the data structure in which multiple related variables can be stored as one variable.

4.6 Steganography

Steganography part in the application has all its components arranged in one module, making it entirely possible to have it reused as is. In this module, the hidden information is called plaintext or information, different from plaintext in the encryption/decryption module.

Five module-level variables, these variables are reused throughout the module. There are FileInfo, FileStream, and TSymmetricKey structure as concern. FileInfo class aids in creation of FileStream. FileStream is to expose a stream around a file. TSymmetricKey stores the symmetric encryption key to/from the encryption module.

```
Sub conceal(ByVal realPass As String, ByVal fakePass As String)
    PicBuffer = New System.IO.FileInfo(Form1.TextBox2.Text)
    PicFileStream = PicBuffer.OpenRead
    Dim PicBytes As Long = PicFileStream.Length
    Dim PicExt As String = PicBuffer.Extension

    Dim PicByteArray(PicBytes) As Byte
    PicFileStream.Read(PicByteArray, 0, PicBytes)
```

Figure 4.2 First step to conceal

The first step of steganography (Figure 4.2) concealing involves opening the image and converting it to an array of bytes. First to do is to instantiate FileInfo class to wrap the file path, denoted as string. Then set up the file stream which comes from OpenRead function to create a FileStream. Also take the length of the byte array and the image extension as the length is crucial to create the byte array and the extension aids in saving the file.

```
        'prepare the key
        rijn = Encryption.rijn

        ''sentinel string to separate the image from the hidden text
        Dim SentinelString() As Byte = {73, 116, 83, 116, 97, 114, 116, 115,
72, 101, 114, 101}
        Dim rijnkey = JulianEnc(Convert.ToBase64String(rijn.IV)) + vbCrLf +
JulianEnc(Convert.ToBase64String(rijn.key))

        'ready the string to be merged
        Dim PlainText As String = realPass + vbCrLf + fakePass + vbCrLf +
rijnkey + vbCrLf
```

Figure 4.3 Prepare the plaintext

The step denoted in Figure 4.3 is to prepare the text to be concealed inside image. The substeps are obtaining the Rijndael key to be hidden altogether from the encryption module, encrypt it slightly to deceive everyone intentionally trying to obtain the key, merge all to be concealed informations, and set Sentinel String as delimiter of the image from the hidden text.

```
        ''modify to byte array, as usual
        Dim PlainTextByteArray(PlainText.Length) As Byte
        For i As Integer = 0 To (PlainText.Length - 1)
            PlainTextByteArray(i) = CByte(AscW(PlainText.Chars(i)))
        Next
```

Figure 4.4 String to Byte Array modification

These lines in Figure 4.4 are to modify the string to byte array by converting every character inside the string into its ASCII equivalent which is then converted into byte.

This step is to conceal secret data inside the image in such a manner that it will not corrupt the data and image. The best way to do it is to place the data after the end of the file.

First to do is to create new byte array with the size of the original image plus the length of the information plus the length of sentinel string. Then, place the image bytes, sentinel string, and the information in that order to ease retrieval.

```
        'save file
        Dim format As String = "Previous File Format|" +
Form1.TextBox2.Text.Substring(Form1.TextBox2.Text.LastIndexOf("."))
        Dim save = Form1.SaveFileDialog1
        save.Filter = format
        save.ShowDialog()
        Dim newFileName As String = save.FileName
        My.Computer.FileSystem.WriteAllBytes(newFileName, PicAndText, False)
        save.Reset()
    End Sub
```

Figure 4.5 Save altered file

Figure 4.5 is the code utilized to save the altered file. Unlike the save file in the main class, this set needs to call the main form to access save file capability and its textbox.

Set the format up according to the original file format as the filter (the "Save As type" column in any save file window). Then show dialog box and accept file name from user. Last step is to write all the bytes from the conceal step as a binary file named according to the file name given by user in overwriting manner. The Reset subroutine is to reset all options inside the save file dialog.

4.7 Miscancellaneous Functions

All functions in this section are not directly involved in the related processing. However, those functions are

organized under pre-existing modules as those were conceived during development of that module.

```
Function retrieve(ByVal input As String) As String()
    Dim separated As String() = Split(input, vbCrLf, -1, 1)
    Return separated
End Function
```

Figure 4.6 Convert string to byte array

This function in Figure 4.6 is simply splitting input string based on the delimiter at every occurrence in textual basis. Textual basis means that the delimiter is compared as is instead of changed into binary form (0x0D 0x0A).

This function is simply converting a string (input) to an array of bytes (output) as is. It works by converting each character in the input into its ASCII code equivalent and convert it into byte. Remember that ASCII ranges from 0 to 255, the same range as byte, it makes them equivalent and readily converted.

4.8 Caesarian Encryption/Decryption Module

This module is only to manipulate key in such a manner that it will do deception against any person trying to intentionally open the secret file by tricking them into thinking that the key is in Base64. Through that idea, the simplest way is to encrypt it to another Base64 string, leaving modified classic cipher as the most probable choice.

Regardless of the cipher used, the pad must not be touched to retain the impression of Base64. The modification involved is expanding the character basis from whatever the original classic cipher have to 64 characters. One of well-known classic cipher is Caesar Cipher. It is the deception method run in this block.

```
Function JulianEnc(ByVal input As String) As String
    createEncode()
    Dim output As String = ""
    Dim index As Integer = 0
    While input(index) <> "="
        Dim cur = input(index)
        Dim index1 As Integer
        Dim curOut As Char

        For index1 = 0 To 63
            If cur = pattern(index1) Then
                curOut = pattern((index1 + 5) Mod 64)
            End If
        Next
        output = output + curOut
        index += 1
    End While
    For index = output.Length To input.Length - 1
        output = output + "="
    Next
    Return output
End Function
```

Figure 4.7 Modification of Caesar Cipher

First step of the function in Figure 4.7 is to generate the pattern by createEncode() subroutine. This step must be called in order to initiate the variable pattern which is used throughout the function.

Next step is to initiate a string as output that will receive encrypted character. Also initiated is the counter of the character.

Then run a loop which stops when the pad is reached. In the loop, the application will look for the place of the currently processed character in the Base64 array and do modular addition of 64 to it by any number from one to 63 (five in this application). The range was to ensure that the characters actually shift. After getting the character, append the output with the resulting character and advance the counter by one.

Last step before returning the output is to append the pad into the output string. Return the output and the function finished its job.

This function is almost the same as the encryption side. The initialization is the same: generate the pattern, declare output and counter. The working of this function is also almost identical to the encryption side: loop to change each character in the string until it hits the pad (remember that the pad must not be changed), append the output with the changed character, and pad the string.

However, there is a stark difference: the amount of the change or to which direction the change is. As long as it satisfy modular addition to base 64, either way is fine. As Visual Basic may accept negative remainder, instead of stepping five to the left (inverting the encryption); stepping 59 to the right is equivalent to stepping five to the left without the negative remainder. This is the key of decryption: how many steps to the right necessary.

V. CONCLUSIONS

New Encryptor is a new approach in security through its self-kill capability in which the application would also “kill” the entire computer should the condition be met. It may seems harsh, but it is worth the annoyance to make only infamous crime warrant the snooping; leaving ordinary Citizen out of snooping.

The program has successfully achieved its objectives, namely:

1. Layered Security in which the attempt to open the file is slowed down.
2. Self-kill capability in which the application can shut down the computer, delete all data currently in use, and exit the application.
3. Deniable Encryption in which the application can create false impression that there is nothing sensitive regarding “decrypted as corrupt image” data. Another deception in Deniable encryption scheme is that the key is shifted several times to deceive the attempt to break open the key.

The application works properly with some reservations:

1. The file should be no bigger than 450kB to achieve timely delivery of the decryption (less than three minutes) based on interpolation of decryption time in systems with medium-low strength (any laptop released around 2011). Faster systems should be able to handle bigger file for the same amount of time.
2. Authentication is somewhat weak, giving out the message that the data has been adulterated despite being encrypted and decrypted in the same system. Also it is vulnerable to Man-in-the-Middle attack in which he can forge brand new data and send it boldly as if it were the actual data.

REFERENCES

[1] Kirk, Jeremy (October 1, 2007). "Contested UK encryption disclosure law takes effect". Washington Post. PC World. Retrieved 2009-01-05.

[2] Government of Republic of Indonesia. 2006. “Undang-undang Informasi dan Transaksi Elektronik (Laws on Electronic Information and Transaction)”

[3] Government of Republic of Indonesia. 1981. “Kitab Undang-undang Hukum Acara Pidana (Code of Criminal Procedure)”