DcDcrypt Ransomware Decryptor

labs.k7computing.com/index.php/dcdcrypt-ransomware-decryptor/

By Gaurav Yadav September 26, 2022

We at K7 Labs came across DcDcrypt ransomware sample, that had infected a user machine and encrypted all user files. Usually the chances of getting back the encrypted files in such a scenario is 0, given the level of sophistication involved in the encryption. Also paying up the ransomware does not guarantee the files be decrypted as “promised” in the ransomware note. Usually the ransom notes and the encrypted are only left behind in a victimized system, the ransomware sample usually gets self-deleted. This case wasn’t so, the ransomware binary was available and naturally we had a closer look at it. Turns out we were able to decrypt the user files after having looked at the malware. The malware and the decryptor would be henceforth discussed.

Analysing Ransomware

This is a basic ransomware written in C#. It encrypts the user files and writes a ransom note in every directory. It does not delete backups, does not create persistence,does not even self-delete(like we mentioned earlier).

Upon execution, this ransomware first checks for a file named “**ID.cI”**. If the file is not present in the same directory, it creates the file and writes a randomly generated ID (used as Victim ID in this case) into it and also stores the ID for further use in a variable named **userID**. If the **ID.cI** file is already present in the same directory it will read and store the content (ID) into the same variable **userID** as shown in Figure 1**.**

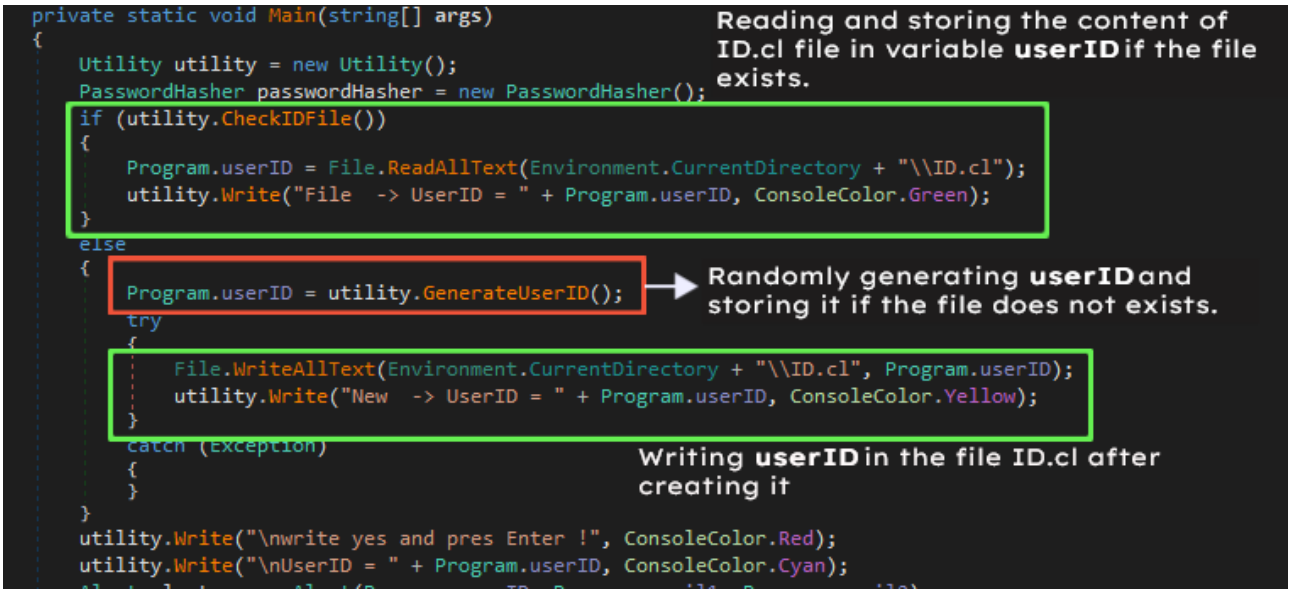


Figure 1: Creating a userID for the victim

This ransomware then asks the user to write yes and press enter to continue the encryption through the command line. Once obliged before the start of the encryption the ransomware first uses the **userID** and a hardcoded **salt** to create a password. It uses the method **GetHashCode** of **passwordHasher** class which is then stored in a variable named **password** as shown in Figure 2.

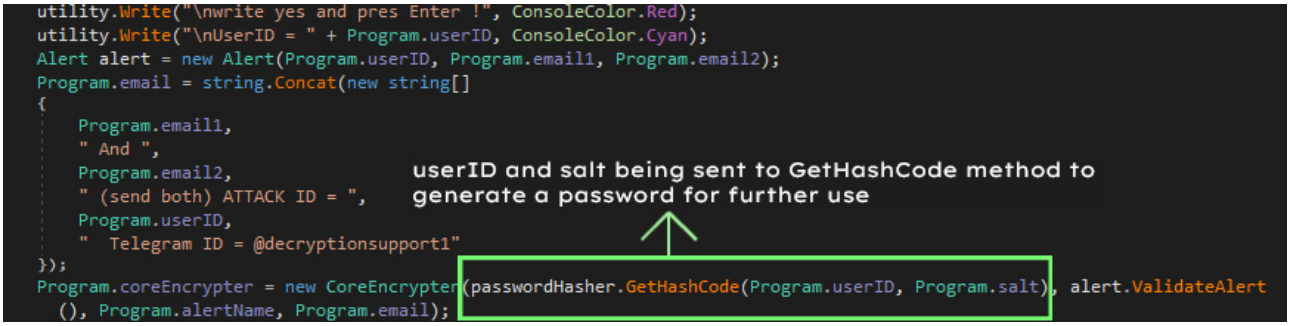


Figure 2: Generating a password

GetHashCode method just adds the userID and salt and sends it to another method called **Hasher** which will create a Sha512 hash of the added userID and salt. Then the sha512 hash is converted into base64 as shown in Figure 3. The resultant value is stored in the variable named **password** of the class **CoreEncrypter**.

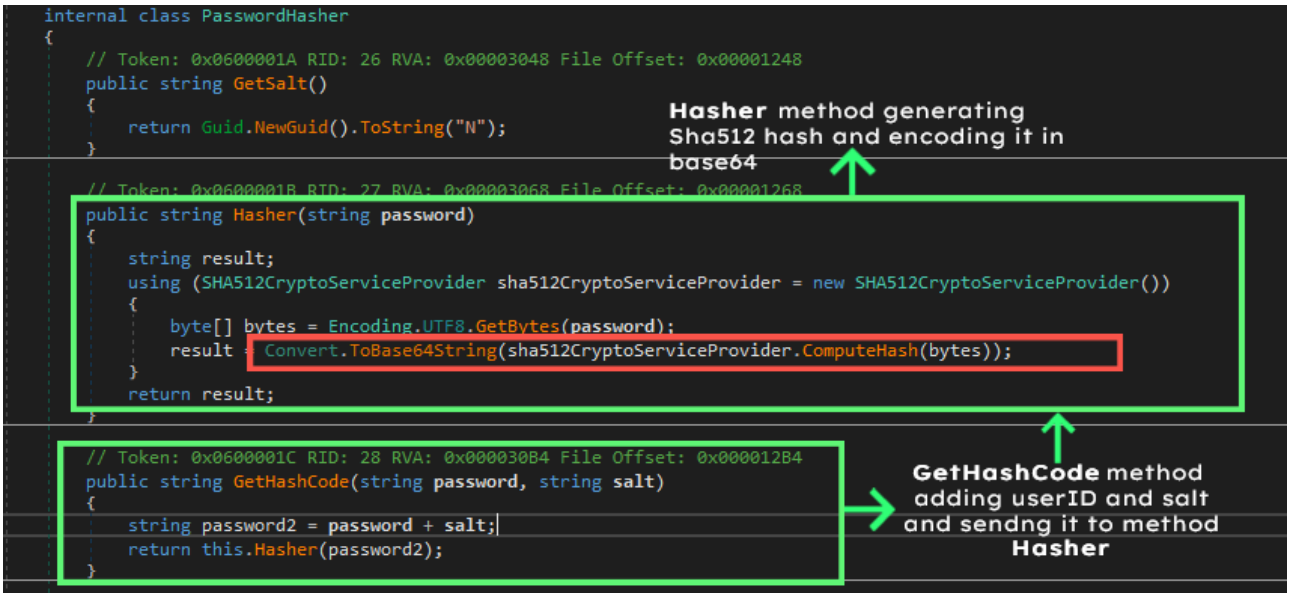


Figure 3: GetHashCode and Hasher method

After generating the password, the ransomware starts encrypting the contents of the current directory as shown in Figure 4. **Enc** method takes the current directory path as an argument and starts encrypting the files within, traversing all the folders inside the current directory. This ransomware does not encrypt anything other than the contents of the current directory it has been run from.

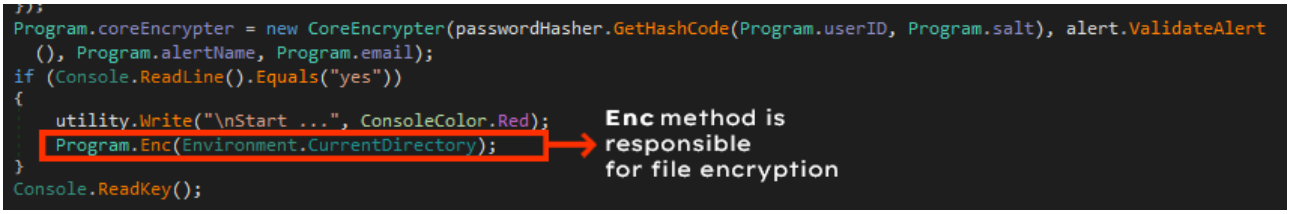


Figure 4: **Enc** method being called with the current directory path as argument

The **Enc** method contains two “for” loops, Figure 4: **Enc** method being called with the current directory path as argumentone to traverse the sub-directories within the current directory using recursive call to **Enc** method and the other one to encrypt the files of the current directory. Before encrypting the file, it first checks if the file name contains any of the following strings:

- ‘**.[Enc]**’ which is present in the extension of the encrypted file.
- ‘**.hta’** which is the extension of ransom note,’**ID.cl**’ is the userID file that ransomware creates in the beginning.
- ’**desktop.ini**’ is a configuration file.
- ‘**Encrypter.exe**’ is the name of ransomware.

If any one of these strings is present in the filename then the ransomware won’t encrypt that file. Check Figure 5 for details.

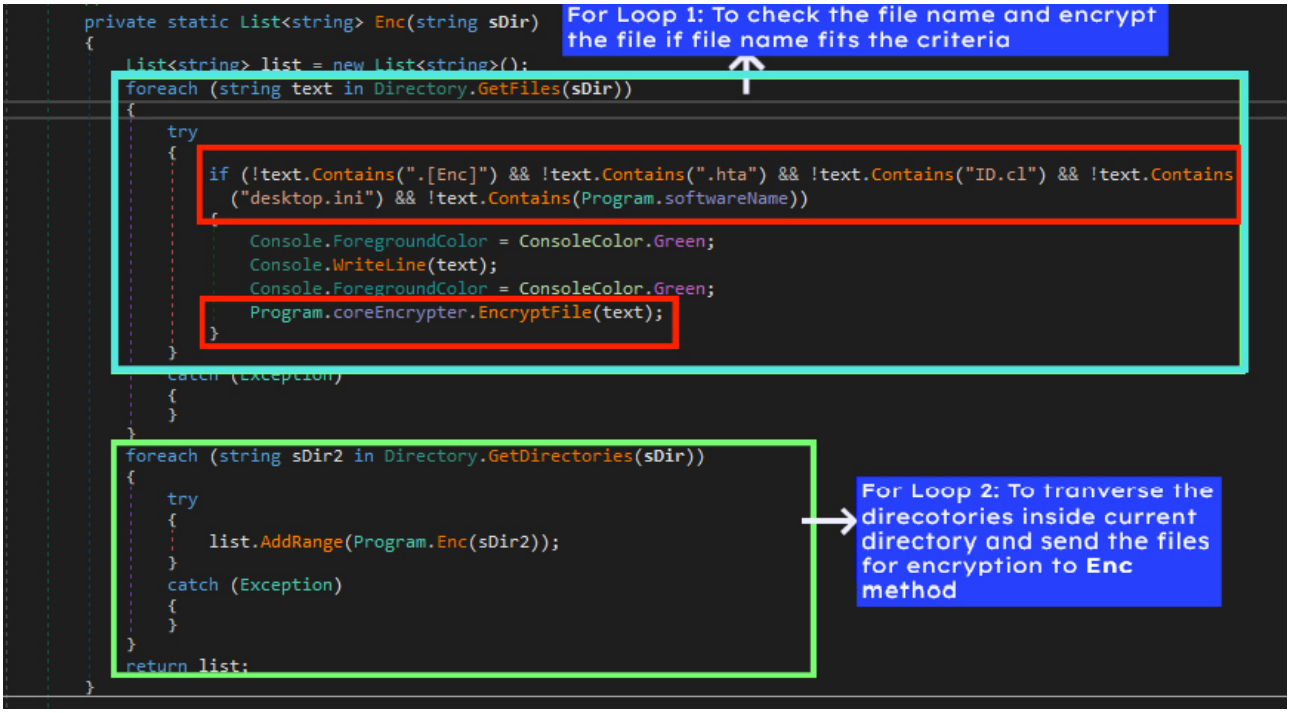


Figure 5: Implementation of **Enc** method

EncryptFile contains the encryption routine which is being called in the For loop 1 with the filename as an argument as shown in Figure 5. Let's analyse this method so that we can figure out how the encryption is done.

As we can see in Figure 6 **EncryptFile** method uses **Rfc2898DeriveBytes** to generate bytes using a password (generated previously using userID and salt) and a salt which can later be used to derive the key and IV(Initialization Vector) for the encryption algorithm. This ransomware uses **Rijndael** as encryption algorithm which is the predecessor of AES algorithm, at default block size (128 bytes) and in cbc mode it works like AES itself.

Microsoft declared this algorithm as obsolete and suggests using AES instead.

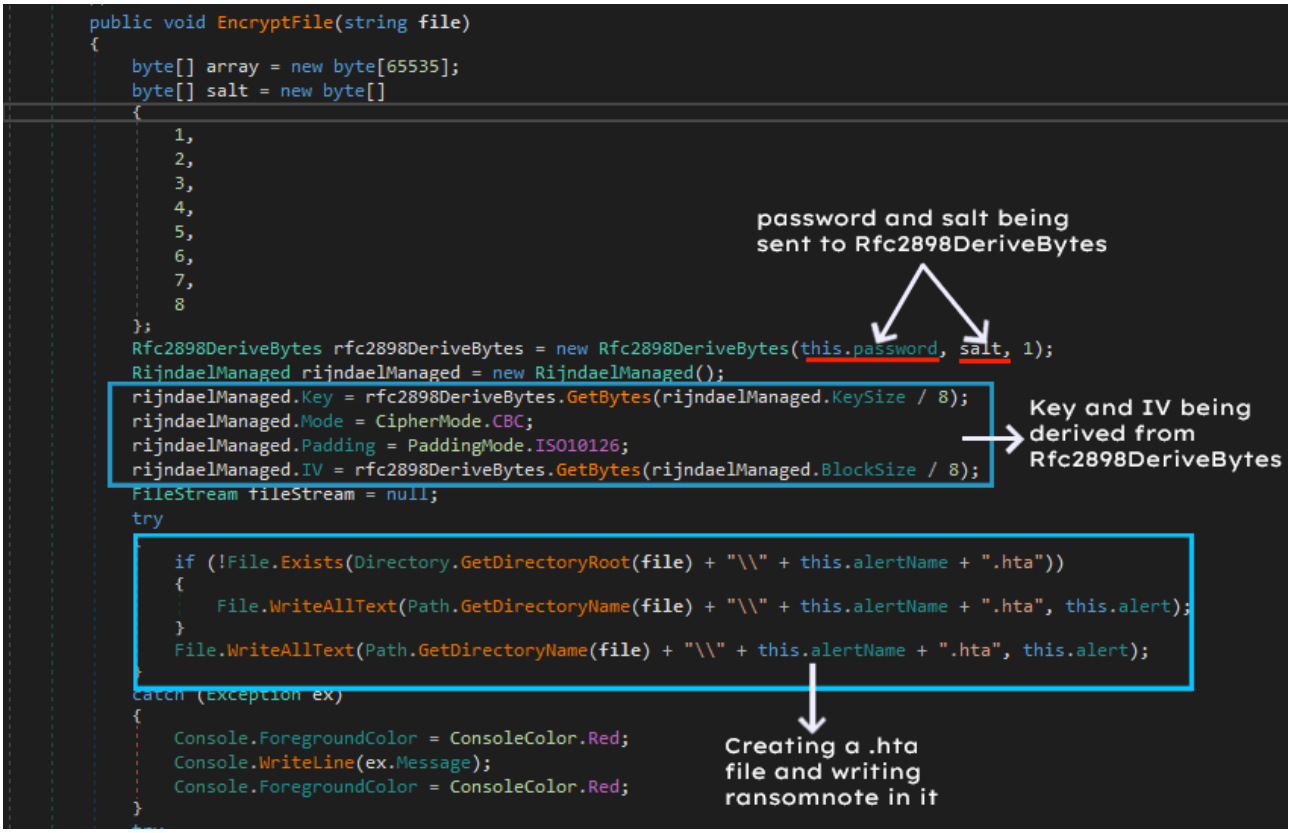


Figure 6: Encryption routine and ransom note

The password that is generated using userID and hardcoded salt remains the same for the same userID (check Figure 2 for reference) and since the password is the same, the key and IV generated using **Rfc2898DeriveBytes** also remains the same for the same userID. **Rijndael** is a symmetric algorithm so the Key-IV pair used for encryption can be used for decryption also.

After initialising all the variables related to encryption, it writes a ransom note in the current directory. Then the ransomware checks for the file size. If it is less than 1000000 bytes, it will create a new file with the same name + encryption extension and then encrypt the original file and put the encrypted content in new file and deletes the original file but if it is larger than the mentioned size it will just XOR the 1st byte of the file with 255 and renames the file with original name + encryption extension, where the encryption extension is **.[Enc] [dc.dcrypt@cyberfear.com And dc.dcrypt@mailfence.com (send both) ATTACK ID = %userID% Telegram ID = @decryptionsupport1]**.

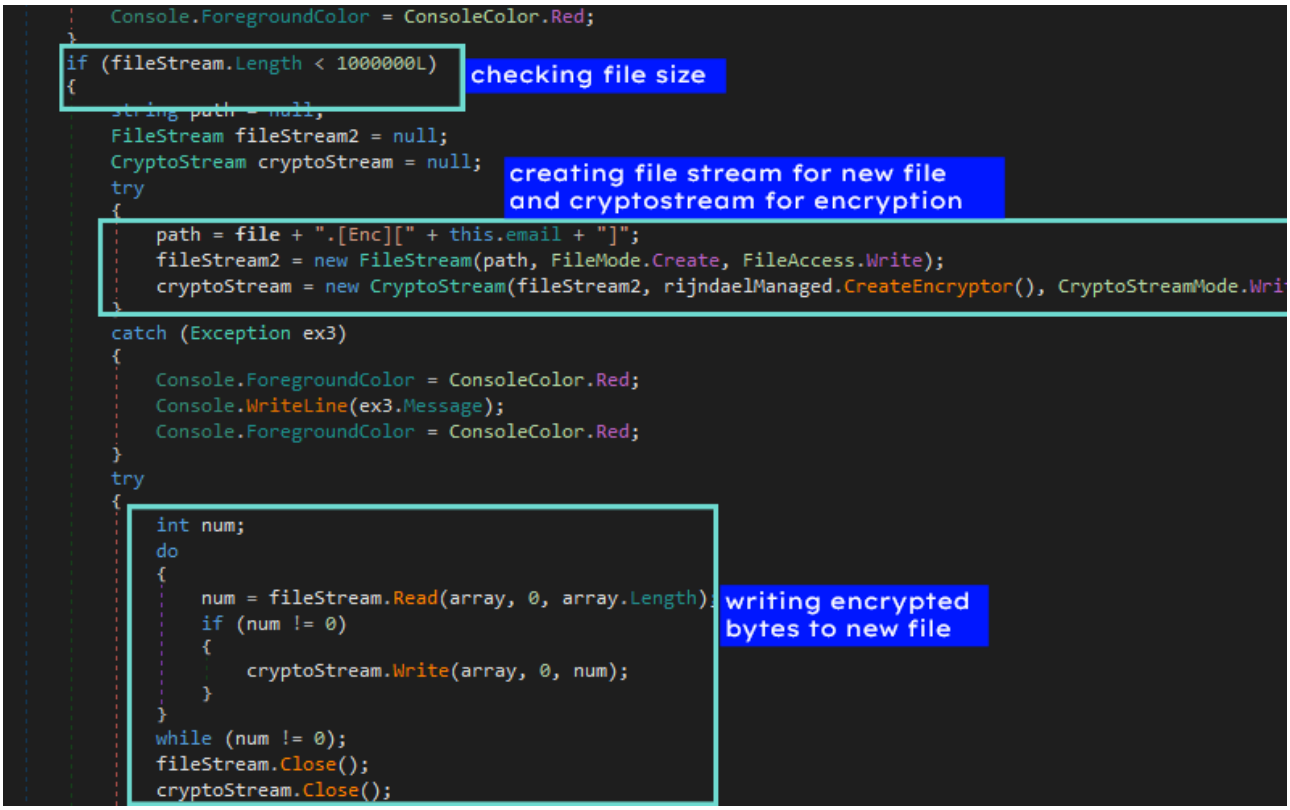


Figure 7: Checking file size encrypting file

```
string destFileName = file + ".[Enc][" + this.email + "]_";
try
{
    long position = fileStream.Position;
    int num2 = fileStream.ReadByte() ^ 255;
    fileStream.Seek(position, SeekOrigin.Begin);
    fileStream.WriteByte((byte)num2);
    fileStream.Close();
    File.Move(file, destFileName);
}
catch (Exception ex5)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex5.Message);
    Console.ForegroundColor = ConsoleColor.Red;
}
```

Figure 8: XORing 1st byte of large files

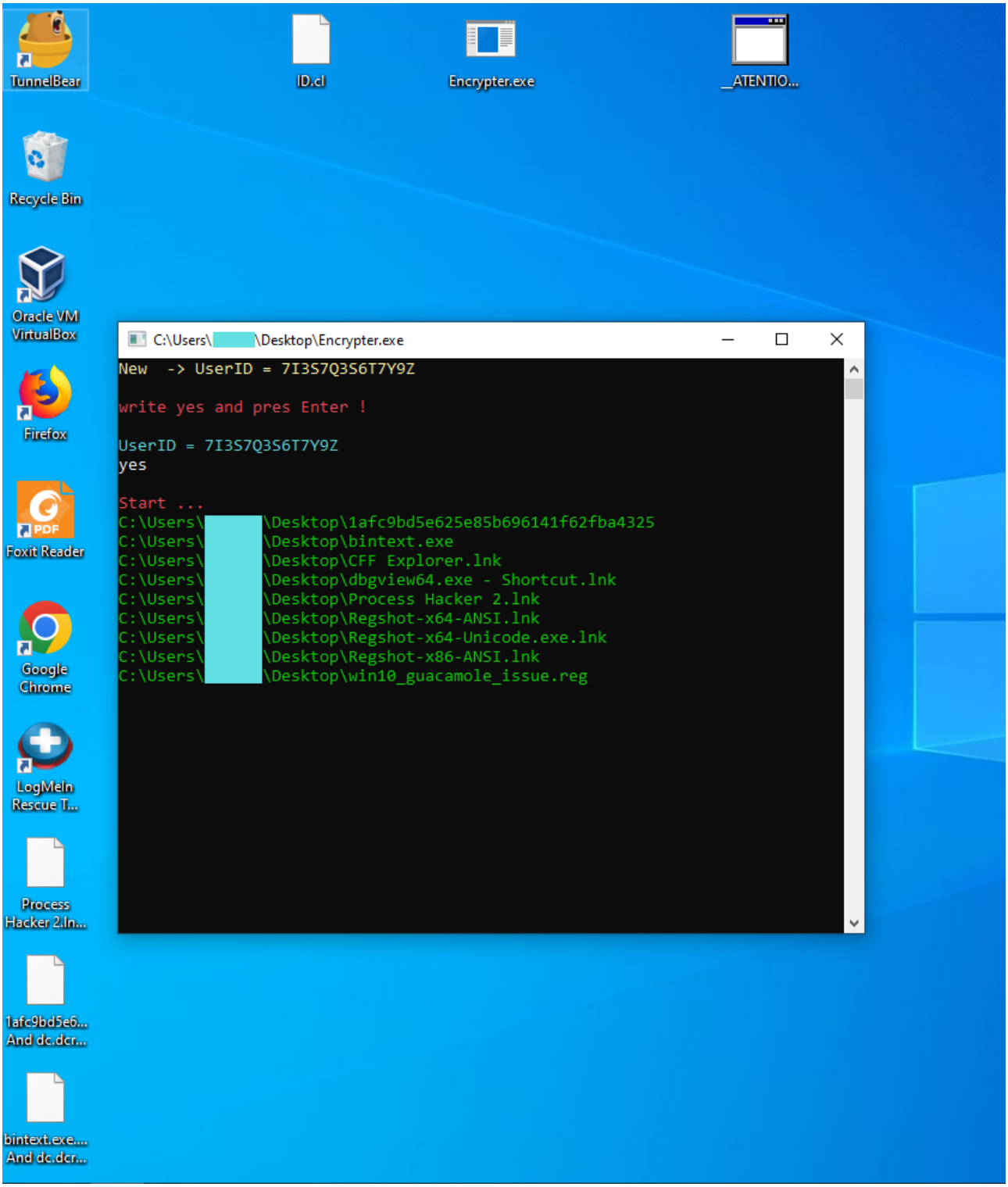


Figure 9: Ransomware execution

Figure 10: Ransom Note

Decryptor Internals

As we already know the ransomware generates a password using userID that it created randomly and a hardcoded salt, since every infected user have a unique userID and we know it will be stored in a file called **ID.cl** so in our decryptor, instead of creating the userID we would instead read ID.cl file to get the userID and use the same method for generating the password (GetHashCode in Figure 3) as ransomware did to generate the password.

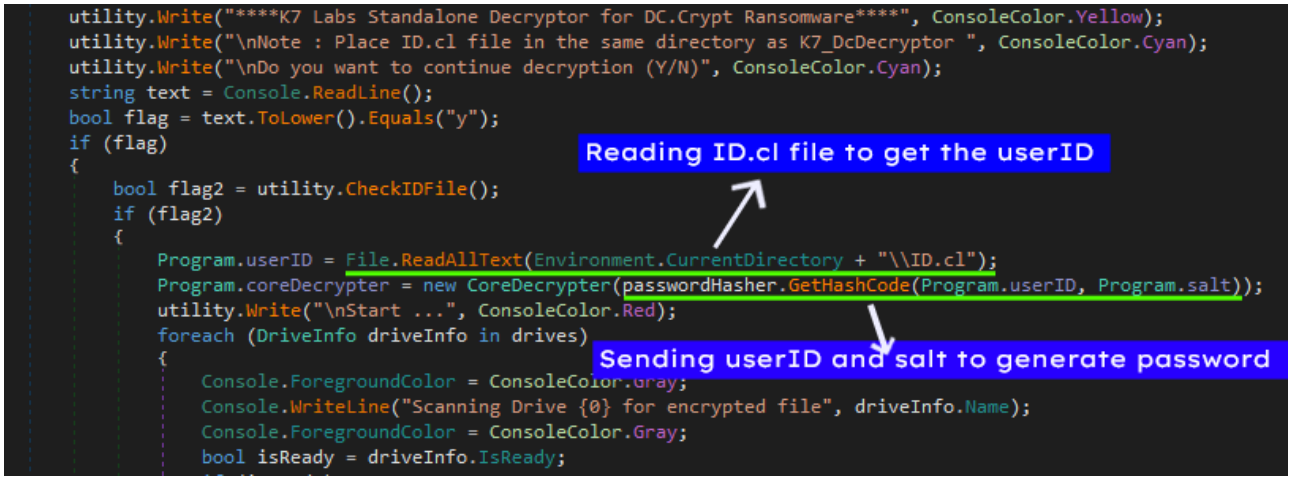
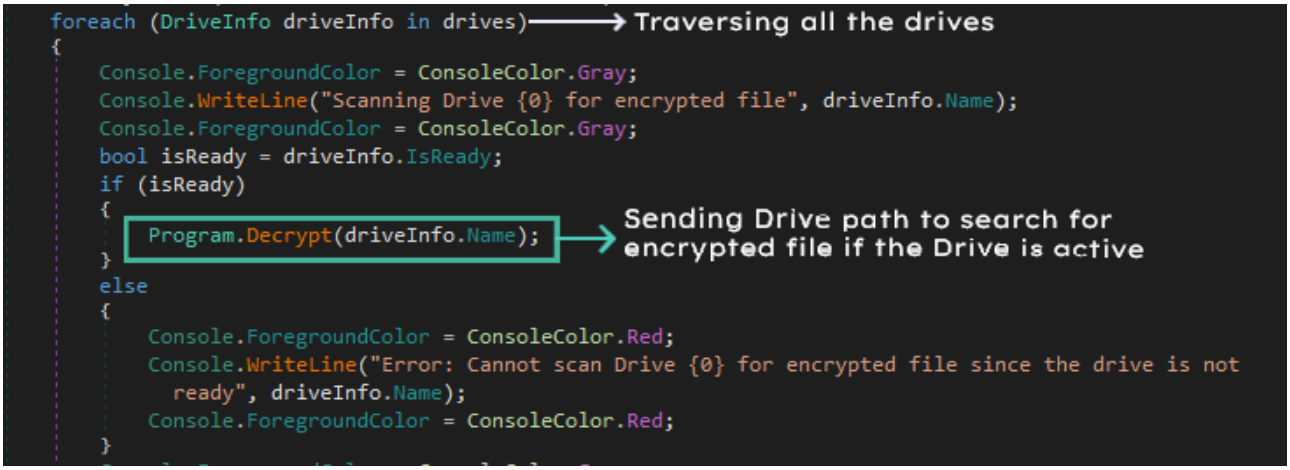


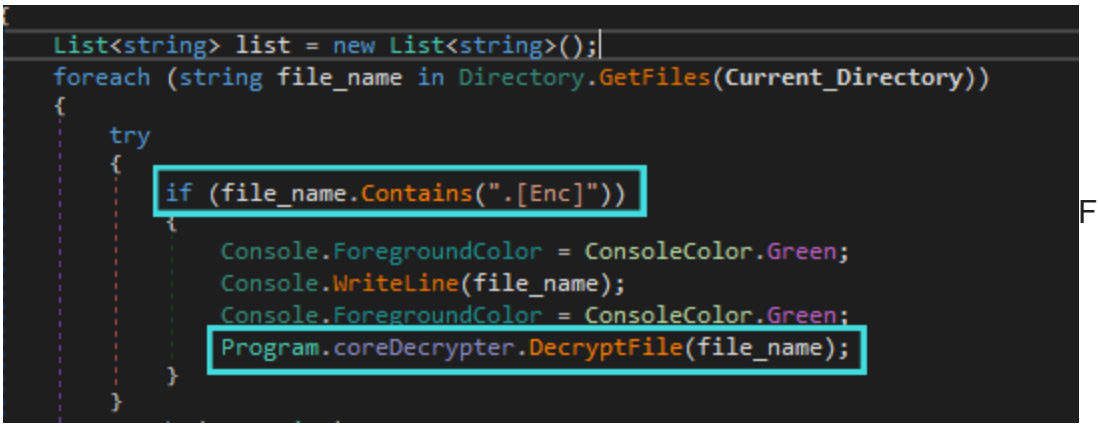
Figure 11: Generating password with userID

After that this tool would scan all the drives for any encrypted files, all the encrypted files have .[Enc] in their extension hence identifying the encrypted file wouldn't be too hard.




```
foreach (DriveInfo driveInfo in drives)
{
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.WriteLine("Scanning Drive {0} for encrypted file", driveInfo.Name);
    Console.ForegroundColor = ConsoleColor.Gray;
    bool isReady = driveInfo.IsReady;
    if (isReady)
    {
        Program.Decrypt(driveInfo.Name);
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Error: Cannot scan Drive {0} for encrypted file since the drive is not
          ready", driveInfo.Name);
        Console.ForegroundColor = ConsoleColor.Red;
    }
```

Figure 12: Scanning all drives



```
List<string> list = new List<string>();
foreach (string file_name in Directory.GetFiles(Current_Directory))
{
    try
    {
        if (file_name.Contains(".[Enc]"))
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine(file_name);
            Console.ForegroundColor = ConsoleColor.Green;
            Program.coreDecrypter.DecryptFile(file_name);
        }
    }
```

Figure 13: Checking file name for **.[Enc]** and sending it for decryption

Since the ransomware used built-in classes and methods provided by .Net for encrypting the files (RijndaelManaged) we can use the same for decrypting the files. We will generate the Key and IV same way ransomware generated with the password using **Rfc2898DeriveBytes**.

```
public void DecryptFile(string file)
{
    byte[] salt = new byte[]
    {
        1,
        2,
        3,
        4,
        5,
        6,
        7,
        8
    };
    Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(this.password, salt, 1);
    RijndaelManaged rijndaelManaged = new RijndaelManaged();
    rijndaelManaged.Key = rfc2898DeriveBytes.GetBytes(rijndaelManaged.KeySize / 8);
    rijndaelManaged.Mode = CipherMode.CBC;
    rijndaelManaged.Padding = PaddingMode.ISO10126;
    rijndaelManaged.IV = rfc2898DeriveBytes.GetBytes(rijndaelManaged.BlockSize / 8);
    FileStream fileStream = null;
    string new_filename = null;
    string destFileName = null;
    try
    {
        fileStream = new FileStream(file, FileMode.Open, FileAccess.ReadWrite);
        int startIndex = file.IndexOf(".[Enc]");
        new_filename = file.Remove(startIndex);
        destFileName = new_filename + ".[backup]";
    }
    catch (Exception ex)
    {
```

Generating Key and IV same way ransomware did

Removing encryption extension from filename

Figure 14: Initialising Key and IV for decryption

In order to decrypt the file size is checked if less than 1000000 bytes then the tool would proceed to decrypt it. Ransomware uses **rijndaelManaged.CreateEncryptor** to create encryption stream in the same manner we can use **rijndaelManaged.CreateDecryptor** to create thedecryption stream it would use the Key-IV generated before for decryption.

```
if (fileStream.Length < 1000000L)
{
    FileStream fileStream2 = null;
    CryptoStream cryptoStream = null;
    try
    {
        fileStream2 = new FileStream(new_filename, FileMode.Create, FileAccess.Write);
        cryptoStream = new CryptoStream(fileStream, rijndaelManaged.CreateDecryptor(rijndaelManaged.Key, rijndaelManaged.IV),
          CryptoStreamMode.Read);
    }
    catch (Exception ex2)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex2.Message);
        Console.ForegroundColor = ConsoleColor.Red;
    }
    try
    {
        byte[] array = new byte[(int)fileStream.Length];
        cryptoStream.Read(array, 0, (int)fileStream.Length);
        fileStream2.Write(array, 0, array.Length);
        fileStream2.Flush();
        cryptoStream.Close();
        fileStream.Close();
        fileStream2.Close();
        File.Move(file, destFileName);
        return;
    }
```

Creating filestream for new file and Cryptostream for decryption

Writing decrypted bytes to new file and renaming encrypted file as .[backup]

Figure 15: Decrypting file

After decryption, we will write the decrypted bytes into a new file and keep the encrypted file as .[backup] in case the file is not decrypted correctly. For files above 1000000 bytes we will XOR the 1st byte with 255 again to get the original byte and rename the file as the original name (without encryption extension).

```
try
{
    long position = fileStream.Position;
    int num = fileStream.ReadByte() ^ 255;
    fileStream.Seek(position, SeekOrigin.Begin);
    fileStream.WriteByte((byte)num);
    fileStream.Close();
    File.Move(file, new_filename);
}
catch (Exception ex4)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex4.Message);
    Console.ForegroundColor = ConsoleColor.Red;
}
```

Figure 16: XORing 1st byte with 255 for larger files

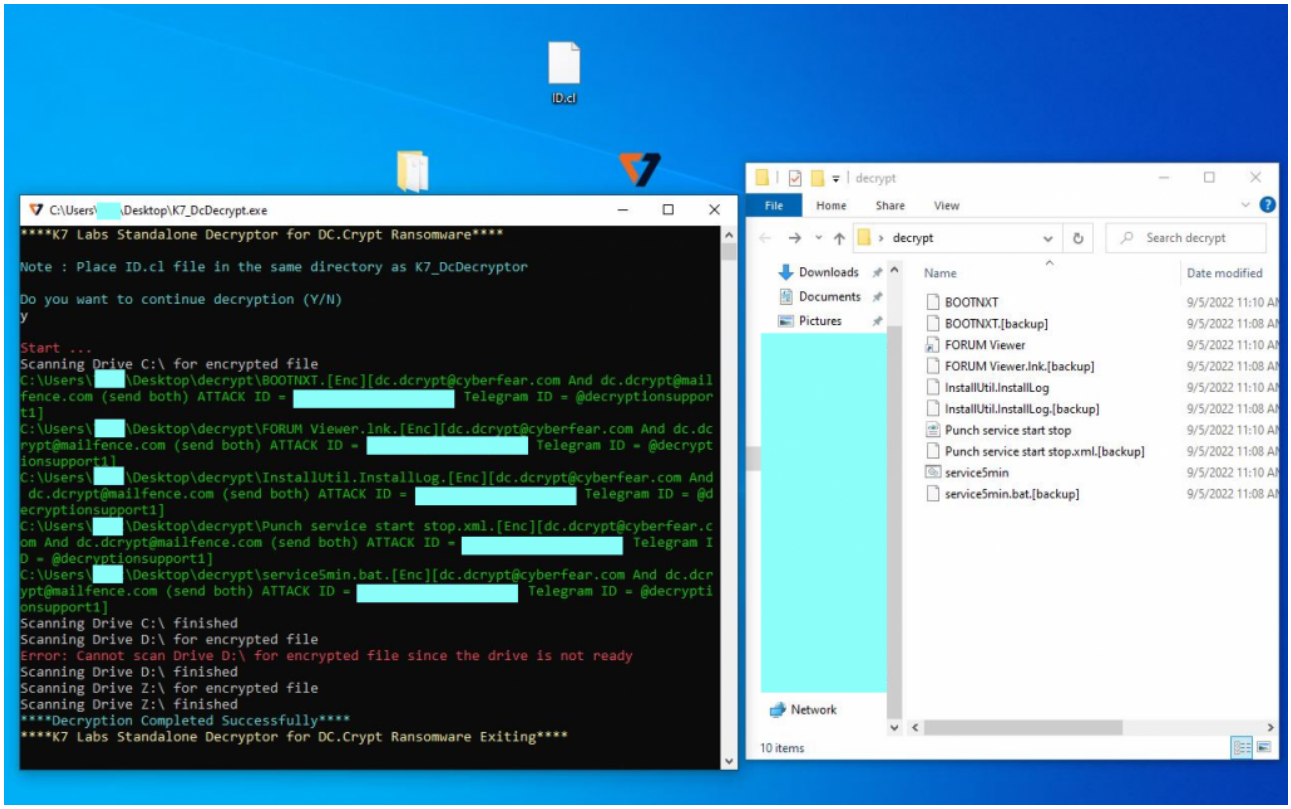


Figure 17: Decryption Tool

We at K7 Labs provide detection for the DcDcrypt ransomware and all the latest threats. Users are advised to use a reliable security product such as **“K7 Total Security” and** keep it up-to-date to safeguard their devices.

Indicators of Compromise (IOCs)

Filename	Hash	Detection Name
Encrypter.exe	1A5C50172527D4F867951FF73AB09ED5	Trojan(0001140e1)

References

https://docs.microsoft.com/en-us/archive/blogs/shawnfa/the-differences-between-rijndael-and-aes

https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rijndaelmanaged?view=net-6.0