Semiconductor Science and Information Devices
https://ojs.bilpublishing.com/index.php/ssid

ARTICLE

Extracting Cryptographic Keys from .NET Applications

Shaun Mc Brearty[1*] **William Farrelly**[2] **Kevin Curran**[3]

1. Institute of Technology, Sligo, Ireland
2. Letterkenny Institute of Technology, Letterkenny, Ireland
3. Ulster University, Derry, United Kingdom

ARTICLE INFO

Article history
Received: 10 June 2021
Accepted: 28 June 2021
Published Online: 5 July 2021

Keywords:
Cryptography
Cryptanalysis
Memory dump analysis
Memory hygiene
Key finding attack
Secure coding
.NET framework

ABSTRACT

In the absence of specialized encryption hardware, cryptographic operations must be performed in main memory. As such, it is common place for cyber criminals to examine the content of main memory with a view to retrieving high-value data in plaintext form and/or the associated decryption key. In this paper, the author presents a number of simple methods for identifying and extracting cryptographic keys from memory dumps of software applications that utilize the Microsoft .NET Framework, as well as source-code level countermeasures to protect against same. Given the EXE file of an application and a basic knowledge of the cryptographic libraries utilized in the .NET Framework, the author shows how to create a memory dump of a running application and how to extract cryptographic keys from same using WinDBG - without any prior knowledge of the cryptographic key utilized. Whilst the proof-of-concept application utilized as part of this paper uses an implementation of the DES cipher, it should be noted that the steps shown can be utilized against all three generations of symmetric and asymmetric ciphers supported within the .NET Framework.

1. Introduction

A memory dump is generated when the contents of computers' main memory and CPU registers - at a specific moment in time - are written to file.

Memory dumps are auto-generated by modern operating system whenever a fault occurs during the execution of the operating system itself or any processes it is executing. In addition, it is also possible for computer users to manually generate a memory dump for a given process while the system is running. While traditionally utilized by software developers for diagnostic purposes, memory dumps have also been utilized by cybercriminals to gain access to sensitive data that is resident in main memory [1].

A number of recent cyber-attacks have utilized memory dumps to gain access to user passwords and authentication tokens [2-5]. In addition, a number of high-profile password management applications have recently been found to contain numerous memory hygiene vulnerabilities that can easily be exploited using memory dumps to gain access to user password information [6].

In order to protect sensitive data in main memory, software developers must exercise good memory hygiene. In essence, sensitive data must only reside in main memory for the duration of time that it is required and must be expunged once no longer required for further processing [7].

In this paper, the author presents a simple approach to extract cryptographic keys from memory dumps of software

**Corresponding Author:*
Shaun Mc Brearty,
Institute of Technology, Sligo, Ireland;
Email: mcbrearty.shaun@itsligo.ie

applications that utilize the Microsoft .NET Framework [8], as well as countermeasures to protect against and prevent same. Given the EXE file of an application and a basic knowledge of the cryptographic libraries utilized in the .NET Framework [9], the author shows how to create a memory dump of the running application and how to extract cryptographic keys from same using WinDBG [10] without any prior knowledge of the cryptographic key utilized.

The attack is shown in three scenarios - one where no steps have been taken to prevent cryptographic keys from being extracted from memory dumps, and one where steps have been taken to remove cryptographic keys from memory, albeit, ineffectively. Finally, the necessary steps to defend against same are demonstrated at the source code level.

1.1 Motivation

The author was motivated to carry out this research after encountering a programming tutorial on the Microsoft MSDN website outlining how to encrypt and decrypt the contents of a TXT file using the DES cipher (demonstrated using the C# programming language) [8,9]. A unique feature of the tutorial was the inclusion of a Method which claimed to remove a cryptographic key from main memory after it was utilized by the application. The Method in question was defined as an External Method [11] named ZeroMemory() - which was configured to execute the RtlZeroMemory function contained within KERNEL32.DLL (a DLL that is external to the .NET Framework - but which is associated with the Windows Operating System).

The author is an experienced C#.NET developer and considered the use of this mechanism to be non-standard; as such, the author opted to research the effectiveness of this mechanism.

1.2 Related Work

Identifying cryptographic keys in memory dumps were first considered by Shamir and van Someren [12]. In their seminal paper, the authors outline two approaches for identifying the presence of cryptographic keys in memory dumps: one for identifying the presence of symmetric and asymmetric keys based on data entropy and another for identifying private RSA keys based on mathematical properties.

Klein [13] showed how to extract RSA private keys and SSL certificates from memory dumps of Apache web servers running on the Microsoft Windows platform (using IDA Pro). Klein identifies the presence of same in memory utilizing a pattern matching mechanism based on the file format outlined in the PKCS #8 and x509 specifications.

Taubmann et al. [14] utilizes a combination of both approaches outlined previously to identify TLS session keys in memory dumps of Android OS applications.

In addition to analyzing memory dumps produced on running systems, significant research has also focused on so called 'cold-boot-attacks' whereby a hard reset is performed on a running computer systems and the data remaining in RAM is dumped to file prior to rebooting [15-17]; evidently, this approach requires physical access to the target system in comparison to the approaches outlined previously (which only require digital access).

To the best of the author's knowledge, this paper is the first to show how to identify and extract cryptographic keys from memory dumps associated with the .NET Framework.

1.3 The .NET Framework

The .NET Framework is a software framework developed by Microsoft that offers a large collection of pre-written Classes and functionality that greatly simplifies the process of developing software applications for the Microsoft Windows operating system. Programs written for the .NET Framework execute in an application virtual machine environment known as Common Language Runtime (CLR). The CLR provides many services to applications - including automated memory management/garbage collection. The .NET Framework supports the execution of programs written in over 20 languages, the most popular being C# and VB. Since Windows XP SP1 (released in 2002), the .NET Framework comes pre-installed with all versions of Microsoft Windows [8].

1.4 Cryptography in the .NET Framework

The .NET Framework includes support for three generations of cipher implementations. Ciphers supported within the .NET Framework include DES, AES, TripleDES, RC2, Rijndael and RSA.

The first generation of ciphers is denoted by the suffix *CryptoServiceProvider* in the Class name, *e.g. DESCryptoServiceProvider*. The functionality of the first generation of ciphers is defined externally to the .NET Framework (within DLLs that are native to the Windows Operating System).

The second generation of ciphers is denoted by the suffix *Managed* in the Class name, *e.g. AESManaged.* The associated library Classes are native to the .NET Framework and have been available since version 3.5 of the.NET Framework (November 2007).

The most recent generation of ciphers is denoted by

1 Cng: Cryptography Next Generation.

the suffix Cng[1] in the Class name, *e.g. AESCng*. The associated library Classes are also native to the .NET Framework and have been available since version 4.6.2 of the.NET Framework (August 2016) [18].

Note that the proof-of-concept application utilized as part of this paper uses the *DESCryptoServiceProvider* implementation of the DES cipher; however the attacks shown are also effective against all other symmetric and asymmetric ciphers in the .NET Framework - across all three of the aforementioned generations.

1.5 Proof-of-concept Application

The proof-of-concept application utilized (see Figure 1) is derived from the previously mentioned programming tutorial on the Microsoft MSDN website [8,9]. The tutorial shows how to encipher and decipher the contents of a text file using a symmetric key that is randomly generated by the application at runtime. The tutorial also includes steps to remove the cryptographic key from main memory; however as shown in Section 3.2, these steps are ineffective as it is in fact possible to recover the cryptographic key from a memory dump produced whilst the application is running.

The proof-of-concept application utilized in this paper comprises five Methods: GenerateKey(), EncryptFile(), DecryptFile(), ZeroMemory() and Main().

• The GenerateKey() Method randomly generates a 64-bit symmetric key using the DESCryptoServiceProvider Class of the .NET Framework and returns the cryptographic key as an ASCII encoded eight-character String Object.

• The EncryptFile() Method reads the contents of a (plaintext) TXT file into main memory, encrypts the data, and then writes the encrypted data to a different TXT file. Note that the text files and cryptographic key to be utilized are specified as Method Parameters.

• The DecryptFile() Method reads the contents of a (encrypted) TXT file into main memory, decrypts the data, and then writes the decrypted data to a different TXT file. Again, the text files and cryptographic key to be utilized are specified as Method Parameters.

• The Main() Method utilizes each of the aforementioned Methods. The String value, *i.e. the symmetric key*, randomly generated by the GenerateKey() Method is assigned to a String variable and is first passed into the EncryptFile() Method to be used for encryption purposes. Following the completion of the EncryptFile() Method, the cryptographic key is then passed into the DecryptFile() Method. The EncryptFile() Method is configured to read plaintext data from 'MyData.TXT', and to write encrypted data to 'Encrypted.TXT'; whilst the DecryptFile() Method is configured to read encrypted data from 'Encrypted.TXT', and to write plaintext data to 'Decrypted.TXT'. In addition to this, a GCHandle Object is also created in the Main() Method. The GCHandle Class is used to track the location of an Object in main memory (à la Pointers in the case of programming languages which require manual memory management) - in this case, the Object being the cryptographic key generated by the GenerateKey() Method. In turn, the GCHandle Object is passed into the ZeroMemory() Method (discussed next) in an effort to remove the associated cryptographic key from main memory [19].

• The ZeroMemory() Method is notable in that it's functionality is implemented in the KERNEL32.DLL included as part of the Microsoft Windows operating system. The purpose of this Method is to replace all data located at a specific address in RAM with a sequence of zeros - in this case, the randomly generated symmetric key whose address is contained within the GCHandle Object mentioned previously. It should be noted that the name of this Method within the KERNEL32.DLL is actually RtlZeroMemory; however it has been assigned the name ZeroMemory() in the application source code. Is should also be noted that the KERNEL32.DLL is not a part of the .NET Framework [20].

Three versions of the proof-of-concept application were developed and analyzed as part of this research.

• Version A comprises the source code shown in Figure 1, with the exception of line 47 and lines 50 - 52. Version A makes no attempt to remove the cryptographic key from memory (other than relying on the automatic execution of the Garbage Collector by the .NET Framework [21]).

• Version B comprises the source code shown in Figure 1, with the exception of line 52. Version B attempts to manually remove the cryptographic key from memory through the use of the ZeroMemory() Method outlined previously.

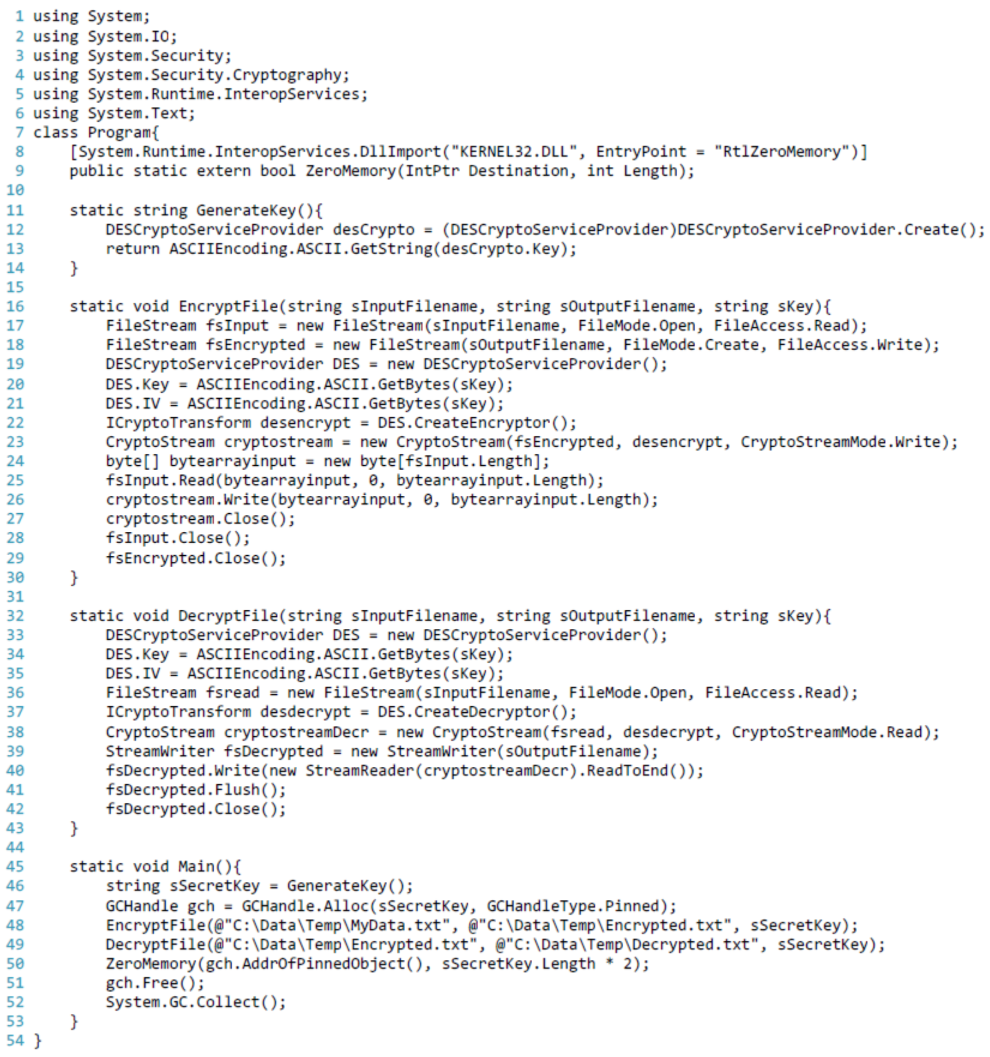
• Version C comprises the full source code shown in Figure 1. Version C successfully removes the cryptographic key from memory through the manual execution of the Garbage Collection routine. Version C was developed by the author to address the issues identified when analyzing Version A and Version B.

1.6 WinDBG

WinDBG is a multipurpose debugging tool produced by Microsoft for debugging a variety of software applications, including user applications, device drivers and the Windows operating system [10].

In relation to the goal of this paper, the major functionality of interest is the ability to view all .NET Objects contained within the portion of main memory managed by the .NET Framework [22].

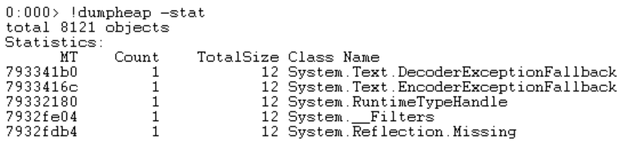
Figure 2 shows sample output of executing the !dum-



```
using System;
using System.IO;
using System.Security;
using System.Security.Cryptography;
using System.Runtime.InteropServices;
using System.Text;
class Program{
    [System.Runtime.InteropServices.DllImport("KERNEL32.DLL", EntryPoint = "RtlZeroMemory")]
    public static extern bool ZeroMemory(IntPtr Destination, int Length);

    static string GenerateKey(){
        DESCryptoServiceProvider desCrypto = (DESCryptoServiceProvider)DESCryptoServiceProvider.Create();
        return ASCIIEncoding.ASCII.GetString(desCrypto.Key);
    }

    static void EncryptFile(string sInputFilename, string sOutputFilename, string sKey){
        FileStream fsInput = new FileStream(sInputFilename, FileMode.Open, FileAccess.Read);
        FileStream fsEncrypted = new FileStream(sOutputFilename, FileMode.Create, FileAccess.Write);
        DESCryptoServiceProvider DES = new DESCryptoServiceProvider();
        DES.Key = ASCIIEncoding.ASCII.GetBytes(sKey);
        DES.IV = ASCIIEncoding.ASCII.GetBytes(sKey);
        ICryptoTransform desencrypt = DES.CreateEncryptor();
        CryptoStream cryptostream = new CryptoStream(fsEncrypted, desencrypt, CryptoStreamMode.Write);
        byte[] bytearrayinput = new byte[fsInput.Length];
        fsInput.Read(bytearrayinput, 0, bytearrayinput.Length);
        cryptostream.Write(bytearrayinput, 0, bytearrayinput.Length);
        cryptostream.Close();
        fsInput.Close();
        fsEncrypted.Close();
    }

    static void DecryptFile(string sInputFilename, string sOutputFilename, string sKey){
        DESCryptoServiceProvider DES = new DESCryptoServiceProvider();
        DES.Key = ASCIIEncoding.ASCII.GetBytes(sKey);
        DES.IV = ASCIIEncoding.ASCII.GetBytes(sKey);
        FileStream fsread = new FileStream(sInputFilename, FileMode.Open, FileAccess.Read);
        ICryptoTransform desdecrypt = DES.CreateDecryptor();
        CryptoStream cryptostreamDecr = new CryptoStream(fsread, desdecrypt, CryptoStreamMode.Read);
        StreamWriter fsDecrypted = new StreamWriter(sOutputFilename);
        fsDecrypted.Write(new StreamReader(cryptostreamDecr).ReadToEnd());
        fsDecrypted.Flush();
        fsDecrypted.Close();
    }

    static void Main(){
        string sSecretKey = GenerateKey();
        GCHandle gch = GCHandle.Alloc(sSecretKey, GCHandleType.Pinned);
        EncryptFile(@"C:\Data\Temp\MyData.txt", @"C:\Data\Temp\Encrypted.txt", sSecretKey);
        DecryptFile(@"C:\Data\Temp\Encrypted.txt", @"C:\Data\Temp\Decrypted.txt", sSecretKey);
        ZeroMemory(gch.AddrOfPinnedObject(), sSecretKey.Length * 2);
        gch.Free();
        System.GC.Collect();
    }
}
```

Figure 1. Source Code for Proof-of-Concept Application.

pheap -stat command on a memory dump in WinDBG. The command provides a summarized list of the contents of the memory dump. The list comprises the set of all Classes with instances in main memory, the number of instances of the Class, the amount of space occupied by all instances of the Class , as well as the address of the Method Table (MT) for the Class - which contains the list of all instances of the Class as well as the address of each individual instance in memory.



```
0:000> !dumpheap -stat
total 8121 objects
Statistics:
      MT    Count    TotalSize Class Name
793341b0        1           12 System.Text.DecoderExceptionFallback
7933416c        1           12 System.Text.EncoderExceptionFallback
79332180        1           12 System.RuntimeTypeHandle
7932fe04        1           12 System.__Filters
7932fdb4        1           12 System.Reflection.Missing
```

Figure 2. Sample Output of !dumpheap -stat Command in WinDBG.

It should be noted that the information contained within memory dumps cannot be traced back to the variables/references to which they are assigned at the source code level - as is the case with source-code level debugging.

1.7 Objects of Interest within Memory Dumps

Given the functionality of WinDBG outlined previously, instances of the following Classes are of interest in relation to retrieving cryptographic keys from memory captured from the proof-of-concept application:

• String

• DESCryptoServiceProvider

• Byte Arrays, *i.e. byte[]*

The inclusion of the String Class is based on the fact that cryptographic keys and Initial Vectors (IVs) are stored using String Objects in the source code of the proof-of-concept application. The inclusion of the DESCryptoServiceProvider Class is based on the use of the Class for encrypting and decrypting the content of text files whilst the inclusion of the Byte Array Class is based on the author's observation that all cryptographic libraries of the .NET Framework utilize Byte Arrays for storing cryptographic keys and IVs [9].

1.8 Attack Model

Note that in order to utilize the attacks outlined in this paper, an attacker must first gain access to a host system where the target application is running.

1.9 Software Utilized

All versions of the proof of concept application were compiled to utilize version 4.7.2 of the .NET Framework and execute on the x64 CPU architecture. All applications were developed using Microsoft Visual Studio Community Edition 2019. All experiments were performed on a Windows 10 virtual machine with 8GB RAM with access to two physical cores of an Intel® CoreTM i7-4810MQ 2.80GHz CPU (Quad Core). The virtualisation software utilized was Oracle VirtualBox. All memory dump files were created and analyzed using the 64-bit version of WinDBG.

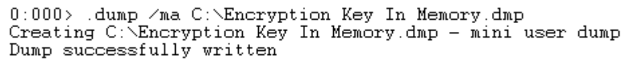
2. Method

This section presents the steps performed by the author using WinDBG when capturing memory dumps of the proof-of-concept application, as well as the steps taken by the author to extract cryptographic keys from the associated memory dumps.

2.1 Creating Memory Dumps

For the purpose of this paper, the author created the memory dumps utilized using WinDBG[2]. The authors reason for doing so is based on the fact that proof-of-concept application is a Console Application that requires no user input; as such, the application executes in a split-second, making it nigh on impossible to create a memory dump using the Task Manager (if the author were to require some form of user input and/or insert an instruction to sleep the application thread in to the source code, this approach would in fact be possible).

In order to capture a memory dump using WinDBG, File → Open Executable must be chosen and the EXE File produced when compiling the source code in Figure 1 must be selected. Following this, Debug → Go must be chosen in order to run the application. At this point, the command to create a memory dump in WinDBG must be executed (.dump - see Figure 3). Having successfully created the memory dump, Debug → Stop Debugging must then be selected.



```
0:000> .dump /ma C:\Encryption Key In Memory.dmp
Creating C:\Encryption Key In Memory.dmp - mini user dump
Dump successfully written
```

Figure 3. dump /ma Command in Use in WinDBG

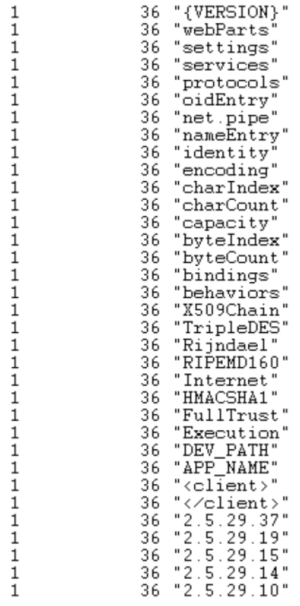
2.2 Opening Memory Dumps in WinDBG

Memory Dumps can be opened in WinDBG by selecting File → Open Crash Dump.

As the application associated with the memory dump utilizes the .NET Framework (version 4 or above), both SOS.DLL and CLR.DLL must be loaded into WinDBG in order to enable the functionality discussed previously. This is done by executing the following command: .loadby sos clr.

2.3 Attack Method #1 - Viewing String Objects

The set of all String Objects contained within the memory dump can be viewed by executing the !dumpheap -strings command in WinDBG. Where the number of String Objects in the memory dump is large, this can be filtered based on their size using the following command: !dumpheap -strings -min 35 -max 37. See Figure 4 for sample output of this command [22]. Note that the command utilized shows only those String Objects with exactly eight characters; recall that eight-character String Objects are being sought as the DES key generated by the proof-of-concept application exist in this form. Eight character String Objects occupy exactly 36 bytes of memory in the.NET Framework; hence the use of the values 35 and 37 in the command specified, *i.e. greater than 35; less than 37.*



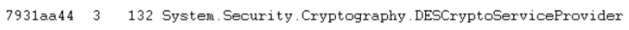
```
1          36 "{VERSION}"
1          36 "webParts"
1          36 "settings"
1          36 "services"
1          36 "protocols"
1          36 "oidEntry"
1          36 "net.pipe"
1          36 "nameEntry"
1          36 "identity"
1          36 "encoding"
1          36 "charIndex"
1          36 "charCount"
1          36 "capacity"
1          36 "byteIndex"
1          36 "byteCount"
1          36 "bindings"
1          36 "behaviors"
1          36 "X509Chain"
1          36 "TripleDES"
1          36 "Rijndael"
1          36 "RIPEMD160"
1          36 "Internet"
1          36 "HMACSHA1"
1          36 "FullTrust"
1          36 "Execution"
1          36 "DEV_PATH"
1          36 "APP_NAME"
1          36 "<client>"
1          36 "</client>"
1          36 "2.5.29.37"
1          36 "2.5.29.19"
1          36 "2.5.29.15"
1          36 "2.5.29.14"
1          36 "2.5.29.10"
```

Figure 4. Sample Output Of !dumpheap -strings -min 35 -max 37 Command in WinDBG

2 In Windows Vista - as well as in subsequent releases of Microsoft Windows - it is in fact possible to create a memory dump of any running application/service using the Task Manager (by right clicking the name of the application/service in the Task Manager and selecting Create Dump File from the resulting menu), as well as the Command Line [23].

2.4 Attack Method #2 -Viewing DESCryptoServiceProvider Objects

In order to view instances of the DESCryptoS-ervice-Provider Class in WinDBG (or any other Class instances for that matter - other than String Objects) - the Method Table address associated with the Class is required. The Method Table address can be obtained using the !dumpheap -stat command discussed previously in Section 1.6. Figure 5 shows sample output of this command for the DESCryptoServiceProvider Class - note the Method Table address for the Class in the left column.



```
7931aa44   3    132 System.Security.Cryptography.DESCryptoServiceProvider
```

Figure 5. Sample Output of !dumpheap -stat Command In WinDBG (for DESCryptoServiceProvider Object).

The command for obtaining the list of all instances of a Class is: !dumpheap -mt method-table-address. Figure 6 shows sample output for this command where the associated Objects are instances of the DESCryptoServiceProvider Class - where method-table-address has a value of 7931aa44 (as shown previously in Figure 5). Note that the left column contains the unique address of each instance of the Class.

Given the address of each instance of the DESCryptoServiceProvider Class, the internal attributes and values of each instance can be viewed using the !do object-address Command. Figure 7 shows sample command output of this command for the first DESCryptoServiceProvider instance listed previously in Figure 6 - where object-address has a value of 01312dbc).

Within the DESCryptoServiceProvider Class, the attributes of most interest are KeyValue and IVValue. As both attributes are Arrays, their contents must be viewed using the !da -details address-of-array command. Figure 8 shows sample output for the associated command where the Byte Array shown is the KeyValue Byte Array denoted previously in Figure 7 - where address-of-array has a value of 01312ef4) [22].

2.5 Attack Method #3 - Byte Arrays

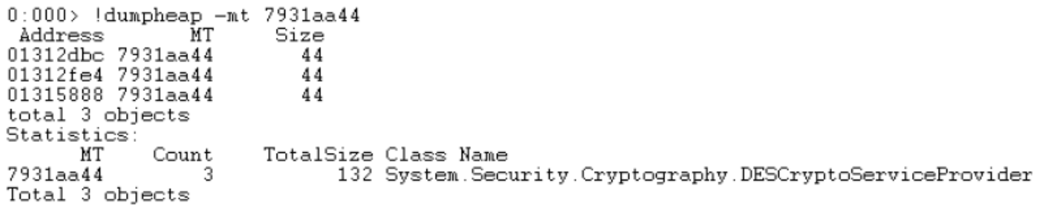
The steps involved in retrieving Byte Arrays from memory dumps are similar to those shown previously in Section 2.4, with the exception that the name of the target Class is System.Byte[]. The list of all Byte Arrays can be obtained using the !dumpheap -mt command, whilst the values contained within each Byte Array can be obtained using the !da -details command. Note that eight element Byte Array Objects occupy exactly 20 bytes of memory in the.NET Framework.

3. Results and Discussion

This section presents the results of carrying out the steps outlined in the Method section on Versions A, B and C of the proof-of-concept application.

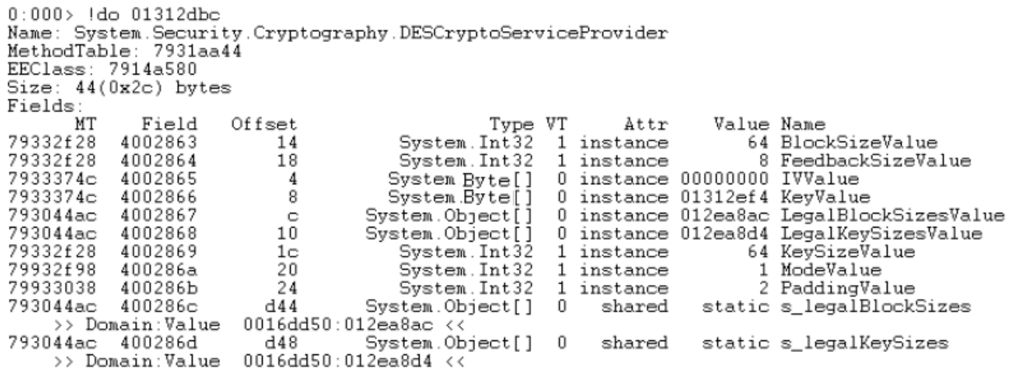
3.1 String Objects Recovered

When viewing the set of all eight-character String



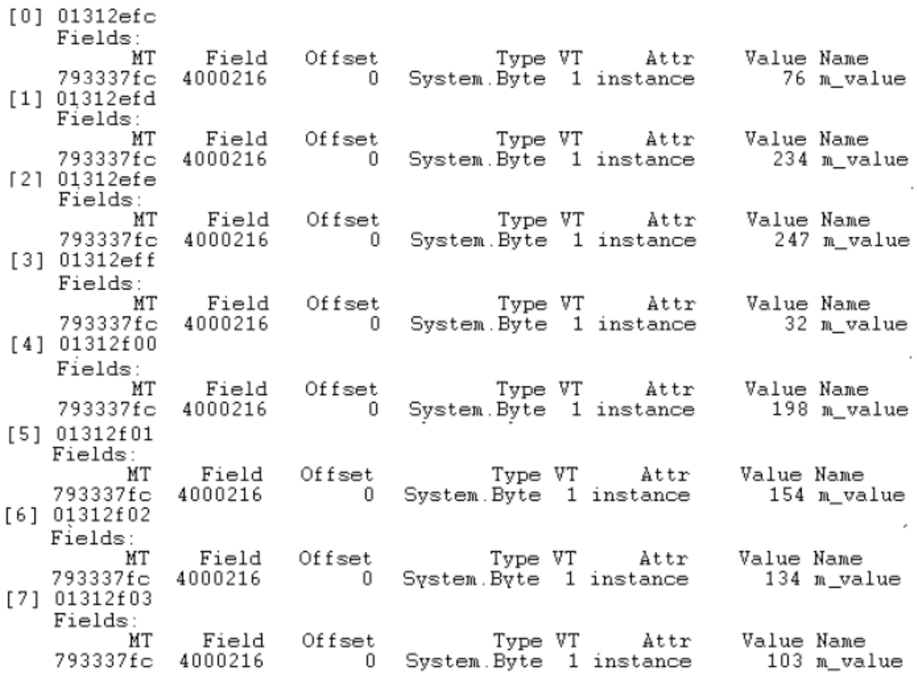
```
0:000> !dumpheap -mt 7931aa44
 Address       MT     Size
01312dbc 7931aa44       44
01312fe4 7931aa44       44
01315888 7931aa44       44
total 3 objects
Statistics:
      MT    Count    TotalSize Class Name
7931aa44        3          132 System.Security.Cryptography.DESCryptoServiceProvider
Total 3 objects
```

Figure 6. Sample Output Of !dumpheap -mt Command in WinDBG (for DESCryptoServiceProvider Objects).



```
0:000> !do 01312dbc
Name: System.Security.Cryptography.DESCryptoServiceProvider
MethodTable: 7931aa44
EEClass: 7914a580
Size: 44(0x2c) bytes
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
79332f28  4002863       14         System.Int32  1 instance       64 BlockSizeValue
79332f28  4002864       18         System.Int32  1 instance        8 FeedbackSizeValue
7933374c  4002865        4        System.Byte[]  0 instance 00000000 IVValue
7933374c  4002866        8        System.Byte[]  0 instance 01312ef4 KeyValue
793044ac  4002867        c      System.Object[]  0 instance 012ea8ac LegalBlockSizesValue
793044ac  4002868       10      System.Object[]  0 instance 012ea8d4 LegalKeySizesValue
79332f28  4002869       1c         System.Int32  1 instance       64 KeySizeValue
79932f98  400286a       20         System.Int32  1 instance        1 ModeValue
79933038  400286b       24         System.Int32  1 instance        2 PaddingValue
793044ac  400286c      d44      System.Object[]  0   shared   static s_legalBlockSizes
    >> Domain:Value  0016dd50:012ea8ac <<
793044ac  400286d      d48      System.Object[]  0   shared   static s_legalKeySizes
    >> Domain:Value  0016dd50:012ea8d4 <<
```

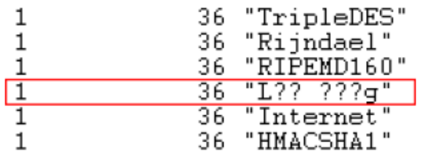
Figure 7. Sample Output Of !do Command in WinDBG (for DESCryptoServiceProvider Object)



```
[0] 01312efc
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance       76 m_value
[1] 01312efd
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      234 m_value
[2] 01312efe
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      247 m_value
[3] 01312eff
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance       32 m_value
[4] 01312f00
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      198 m_value
[5] 01312f01
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      154 m_value
[6] 01312f02
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      134 m_value
[7] 01312f03
    Fields:
          MT    Field   Offset         Type VT     Attr    Value Name
    793337fc  4000216        0  System.Byte  1 instance      103 m_value
```

Figure 8. Sample Output Of !do -details Command in WinDBG (for Byte Array Object)

Objects contained within the memory dump associated with Version A of the proof-of-concept application (66 in total), it was notable that many of the String Objects were either dictionary words or IP addresses. One String value did stand out for its apparent random value (see String value highlighted in Figure 9).



```
1          36 "TripleDES"
1          36 "Rijndael"
1          36 "RIPEMD160"
1          36 "L?? ???g"
1          36 "Internet"
1          36 "HMACSHA1"
```

Figure 9. String Value Extracted From Memory Dump for Version A (Note Pseudorandom Value Highlighted)

In relation to the memory dump associated with Version B of the proof-of-concept application, carrying out the same steps resulted in an identical set of String Objects being returned as with Version A with the exception that the apparently random String mentioned previously was now empty, *i.e.* *“”*; therefore suggesting that the ZeroMemory() Method is in fact effective at removing the cryptographic key from memory (at least in String form).

In relation to Version C of the proof-of-concept application, a total of 55 eight-character String Objects were recovered from the associated memory dump (as opposed to 66 in the case of Version A and B).

Using the apparently random String value recovered from Version A of the application, the author was able to successfully decrypt the contents of the associated ‘Encrypted.TXT’ file. It was not possible to decrypt those versions of ‘Encrypted.TXT’ associated with Version B or C using any of the String values extracted from the respective memory dumps.

3.2 DESCryptoServiceProvider Objects Recovered

When analysing the memory dump for Version A of the proof-of-concept application, three instance of the DESCryptoServiceProvider Class were contained in the memory dump.

When examining the contents of the KeyValue attribute for each DESCryptoServiceProvider instance, the values shown in Table 1 were obtained.

Table 1. Values Extracted From KeyValue Attribute of DESCryptoServiceProvider Instances In Memory Dump For Version A

Version A			
Array Index	**Instance #1**	**Instance #2**	**Instance #3**
0	76	76	76
1	234	63	63
2	247	63	63
3	32	32	32
4	198	63	63
5	154	63	63
6	134	63	63
7	103	103	103

As per Table 1, the values located at Array Index 0, 3 and 7 match in all three instances while the values at the remaining Indexes do not (instance #2 and #3 do in fact match; however instance #1 does not match either instance #2 or instance #3). This discrepancy is a result of converting the randomly generated Byte Array to a String Object within the GenerateKey() Method. This conversion is performed using the ASCIIEncoding.ASCII.GetString() Method. Byte values between 128 and 255 are rendered as a question mark in the String Objects produced by this Method - and when converted back to a Byte Array, each question mark character is rendered as 63, *i.e. the ASCII decimal value for a question mark*. As such, it would appear that instance #1 denotes the Byte Array originally generated by the GenerateKey() Method, whilst instance #2 and #3 denote the cryptographic key values utilized in the EncryptFile() and DecryptFile() Methods.

Given the String value obtained previously in Section 3.1 when analyzing Version A (see Figure 9), it is apparent that this String value was derived from the Byte Arrays shown in instance #2 and #3 in Table 1 (see ASCII Conversion table with pertinent values in Table 2).

Table 2. ASCII Conversion Table for Values Obtained in Version A

ASCII Code	Character
32	Space
63	?
76	L
103	G

When analyzing the memory dump for Version B of the proof-of-concept application, three instance of the DESCryptoServiceProvider Class were contained in the memory dump. The contents of each KeyValue Byte Array recovered can be seen in Table 3.

Table 3. Values Extracted From KeyValue Attribute of DESCryptoServiceProvider Instances In Memory Dump For Version B

Version B			
Array Index	Instance #1	Instance #2	Instance #3
0	56	56	56
1	75	75	75
2	8	8	8
3	84	84	84
4	234	63	63
5	66	66	66
6	45	45	45
7	223	63	63

Using the KeyValue Byte Array values recovered, the author was able to successfully decrypt the contents of 'Encrpyted.TXT' produced by Version A and Version B of the proof-of-concept application. In relation to Version B, it is clear that the ZeroMemory() Method is only effective in removing the cryptographic key from main memory where it exists in String form - with the cryptographic key remaining in main memory in Byte Array form.

In relation to Version C of the proof-of-concept application, no instances of the DESCryptoServiceProvider Class could be located in the associated memory dumps; as such, it was not possible to recover the cryptographic key or decrypt any data using the approach outlined in this section.

3.3 Byte Array Objects Recovered

When analyzing the memory dump for Version A of the proof-of-concept application, a total of 41 eight-element Byte Arrays were located. Of these 41 - only seven distinct values were noted:

• 30 of the Byte Arrays recovered contained the cryptographic key retrieved previously in Section 3.2; six in the form of the 'original key', *i.e. prior to being converted to a String Object in the GenerateKey() Method*, and twenty-four in the form actually utilized by the application. Note that the same value was used for both the KeyValue and IVValue attributes.

• Another three Byte Arrays contained matching values; however the values in question were not associated with either the KeyValue or IVValue attribute values utilized.

• Four of the Byte Arrays contained the value zero at every Index.

• Three of the Byte Arrays contained completely unique values that did not occur in any of the other 38 instances.

When analyzing Version B of the application, the exact same statistics outlined above for Version A were also noted.

For both Version A and B of the proof-of-concept application, the author was able to successfully decrypt the contents of 'Encrypted.TXT' using the Byte Array values recovered.

In relation to Version C, 13 instances of the Byte Array Class were recovered from the associated memory dump; however, the author was unsuccessful in decrypting the associated data using any of the Byte Array values recovered.

4. Conclusions

As the results of the paper show, the author was able to retrieve the cryptographic keys associated with Version A and Version B of the proof-of-concept application. Whilst the author expected to retrieve the cryptographic key associated with Version A of the application - given that no effort was made to manually remove the cryptographic key from memory, the same was not expected of Version B (given that steps were taken to remove same from memory - albeit ineffectively). In response to these findings, the author developed Version C of the proof-of-concept application which proved secure against all three attack Methods outlined in this paper.

The use of the ZeroMemory() Method in Version B of the proof-of-concept was shown to be ineffective given that the cryptographic key remained in memory in the form of a Byte Array - the String version was removed however. Given that the .NET Framework is a managed code environment, the author believes it should be a logical decision to utilize the standard garbage collection mechanism for managed code - as is the case with Version C - as opposed to the RtlZeroMemory function contained within the KERNEL32.DLL (Version B).

The use of pre-written Classes is commonplace amongst developers utilizing the .NET Framework. The inner workings of such classes are typically abstracted from software developers with a view to increasing developer productivity. Given this abstraction, developers should avoid manually managing the memory for such Classes. This is perhaps best demonstrated by the fact that five Byte Arrays are manually created in the source code of the proof-of-concept application; however 30 Byte Arrays were found in the memory dumps for both Version A and Version B (evidently, these other Byte Arrays were created by the pre-written Classes utilized by the proof-of-concept application). Version C's use of the standard Garbage Collection Method for the .NET Framework (System.GC.Collect()) demonstrates best practice as all cryptographic keys and IVs were successfully removed from main memory immediately after being used (doing so protects against all three attack Methods utilized). Version C also demonstrates the importance of explicitly requesting that sensitive data be removed from memory in managed code environments, *i.e. manually executing the Garbage Collection function*, as opposed to waiting/hoping for sensitive data to be cleared automatically by the underlying application virtual machine (as is the case with Version A - note that this usually only occurs whenever the amount of memory available to the application virtual machine (CLR) is deemed low) [21].

The extent to which the vulnerable source code discussed in this paper has been utilized in publically available software applications cannot be determined given the closed nature of executable files; nonetheless, the associated programming tutorial was hosted on the Microsoft MSDN website for some twelve years[3]. A search of publically available source code repositories conducted in June 2021 using searchcode.com located four projects that have copied and pasted the ZeroMemory() Method verbatim, as well as a further three projects that utilize the ASCIIEncoding.ASCII.GetString() Method to insecurely encode DES cryptographic keys from Byte Array form to String form. Access to a source code search engine with support for fuzzy search or regular expression based search would aid in the identification of further affected applications.

In terms of the attack Methods demonstrated, the author acknowledges that the reader may deem the attacks using String Objects and DESCryptoServiceProvider Objects as being specific to the proof-of-concept application - given that the author had access to the associated source code however, the use of String Objects for storing cryptographic keys is a common occurrence in the authors experience. In relation to the attack utilizing Byte Arrays, it should be noted that Byte Arrays are used extensively within the various cryptographic libraries of the .NET Framework for storing cryptographic keys and IVs [9]; as such, the author considers this attack to be the most widely applicable and easily transferable for attacking other ciphers.

While this paper focuses on the first-generation DESCryptoServiceProvider implementation (given its inclusion in the programming tutorial that motivated this paper), the author also repeated these experiments on the second and third generation cipher implementations of the .NET Framework. In the case of the second-generation of ciphers supported by the .NET Framework, *e.g. AesManaged*, it appears that steps are taken to remove cryptographic keys from memory once the encryption or decryption operation has completed - in a manner that is similar to the mechanism utilized in Version B of the proof-of-concept application. Whilst this successfully defends against the second attack method outlined in this paper, attack method three still succeeds. Curiously, the most recent generation of ciphers supported by the .NET Framework does not appear to take such steps - resulting in the associated ciphers being susceptible to attack methods two and three.

The proof-of-concept application is a text-book example of the incorrect usage of cryptography at the source code level - an all too common occurrence in modern software applications [24]. The source code shows the incorrect usage of IVs as both the IV and symmetric key are assigned the same value. In addition, the decision to convert Byte Arrays to String Objects within the GenerateKey() Method inadvertently reduces the size of the cipher keyspace and

[3] The programming tutorial upon which the proof-of-concept application was based was available on the Microsoft MSDN website between 2005 and 2017. Despite the various issues outlined in this paper (as well as in other media [22], [23]), the tutorial was reviewed and re-approved on a number of occasions.

also increases the prevalence of a specific value, *i.e. 63*, appearing in keys and IVs generated by the application[4]. Evidently, software developers must exercise caution when utilizing ciphers in order to ensure they are utilized securely with a view to preventing the vulnerabilities and attacks outlined. In addition to this, authors and publishers of programming tutorials should also take steps to ensure same.

In terms of further research, the author hopes to extend this research at a later date to target the .NET Core Framework - a version of the .NET Framework that includes cross-platform support for the Windows, Linux and macOS platforms [25] - which has been increasing in popularity in recent years.

References

[1] D. Kleiman et al., "Windows and Linux Forensics," in The Official CHFI Study Guide (Exam 312-49), Syngress, 2007, pp. 287-349.

[2] J. M. Porup, "What is Mimikatz? And how this password-stealing tool works," 2019. [Online]. Available: https://www.csoonline.com/article/3353416/what-is-mimikatz-and-how-to-defend-against-this-password-stealing-tool.html. [Accessed: 05-Jun-2021].

[3] J. Fruhlinger, "Petya ransomware and NotPetya malware: What you need to know now," 2017. [Online]. Available: https://www.csoonline.com/article/3233210/petya-ransomware-and-notpetya-malware-what-you-need-to-know-now.html. [Accessed: 09-Jun-2021].

[4] S. Ragan, "BadRabbit ransomware attacks multiple media outlets," 2017. [Online]. Available: https://www.csoonline.com/article/3234691/badrabbit-ransomware-attacks-multiple-media-outlets.html. [Accessed: 09-Jun-2021].

[5] G. Keizer, "Hackers spied on 300,000 Iranians using fake Google certificate," 2011. [Online]. Available: https://www.computerworld.com/article/2510951/hackers-spied-on-300-000-iranians-using-fake-google-certificate.html. [Accessed: 09-Jun-2021].

[6] Independent Security Evaluators (ISE), "Password Managers: Under the Hood of Secrets Management," 2019. [Online]. Available: https://www.securityevaluators.com/casestudies/password-manager-hacking/. [Accessed: 05-Jun-2021].

[7] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation," in USENIX Security Symposium, 2005.

[8] Microsoft, "Introduction to the C# Language and the .NET Framework | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework. [Accessed: 31-May-2021].

[9] Microsoft, "System.Security.Cryptography Namespace." [Online]. Available: https://msdn.microsoft.com/en-us/library/system.security.cryptography(v=vs.110).aspx. [Accessed: 31-May-2021].

[10] Microsoft, "Getting Started with WinDbg (User-Mode) | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg. [Accessed: 31-May-2021].

[11] Microsoft, "extern modifier - C# Reference," 2015. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/extern. [Accessed: 05-Jun-2021].

[12] A. Shamir and N. van Someren, "Playing 'Hide and Seek' with Stored Keys," in International Conference on Financial Cryptography, 1999, pp. 118-124.

[13] T. Klein, "All your private keys are belong to us Extracting RSA private keys and certificates out of the process memory," 2006.

[14] B. Taubmann, O. Alabduljaleel, and H. P. Reiser, "DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps," Digit. Investig., vol. 26, pp. S67-S76, Jul. 2018.

[15] J. A. Halderman et al., "Lest We Remember: Cold Boot Attacks on Encryption Keys," Commun. ACM, vol. 52, no. 5, p. 91, May 2009.

[16] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold Boot Attacks are Still Hot: Security Analysis of

[4] As outlined in Section 3.2, the conversion of Byte Arrays to String Objects in the GenerateKey() Method results in the apparent absence of Byte values above 127 in the cryptographic keys generated and utilised by the proof-of-concept application. Instead, all values in this range were rendered as 63. This is a result of using the ASCIIEncoding. ASCII.GetString() Method to create a String Object from a Byte Array. Byte values between 127 and 255 are rendered as a question mark in the String Objects produced by this Method and when converted back to a Byte Array, each question mark character is rendered as 63 in the resulting Byte, *i.e. the ASCII decimal value for a question mark*. Given that cryptographic keys are represented as Byte Arrays in the .NET Framework, this issue results in the most significant bit of each Byte being rendered as zero therefore reducing the effective keyspace of a ciphers by $2^{(\text{Keyspace Of Cipher In Bits/8})}$. Evidently, this is a serious security issue. Ironically, this issue does not affect the DES cipher as the effective keyspace of DES is 2^{56} bits (DES keys are represented as eight-element Byte Arrays in the .NET Framework with the most significant bit in each Byte being ignored, *i.e. a total of 64 Bits are utilised, but only 56 are relevant*); however other ciphers such as AES would be affected by this issue.

From a key generation perspective, the GenerateKey() Method is extremely insecure as there is a 50% chance that a randomly generated Byte will have a specific value, *i.e. 63 -therefore further reducing the effective keyspace of all ciphers*. In the case of the cryptographic key recovered for Version A of the application, the Byte value 63 occurred five times in a sequence of eight values, whilst 63 occurs twice in the cryptographic key recovered for Version B of the application.

Memory Scramblers in Modern Processors," in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 313-324.

[17] R. Zahno, "Key Recovery from Decayed Memory Images and Obfuscation of Cryptographic Algorithms," Concordia University, 2012.

[18] Microsoft, ."NET Framework Cryptography Model," 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/security/cryptography-model. [Accessed: 05-Jun-2021].

[19] Microsoft, "GCHandle Structure (System.Runtime.InteropServices)," 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.gchandle?redirectedfrom=MSDN&view=netframework-4.7.2. [Accessed: 31-May-2021].

[20] Microsoft, "RtlZeroMemory macro (wdm.h)," 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-RtlZeroMemory. [Accessed: 05-Jun-2021].

[21] Microsoft, "Fundamentals of Garbage Collection | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals. [Accessed: 31-May-2021].

[22] Microsoft, "SOS.dll (SOS Debugging Extension)," 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/tools/sos-dll-sos-debugging-extension. [Accessed: 31-May-2021].

[23] Microsoft, "How to create a user-mode process dump file in Windows," 2017. [Online]. Available: https://support.microsoft.com/en-us/help/931673/how-to-create-a-user-mode-process-dump-file-in-windows. [Accessed: 31-May-2021].

[24] OWASP, "Insecure Cryptographic Storage," 2010. [Online]. Available: https://www.owasp.org/index.php/Top_10_2010-A7-Insecure_Cryptographic_Storage. [Accessed: 05-Jun-2021].

[25] Microsoft, ."NET." [Online]. Available: https://www.microsoft.com/net. [Accessed: 31-May-2021].

[26] Ponemon Institute LLC, "HSM Global Market Study," 2014.

[27] Microsoft, "How to encrypt and decrypt a file using Visual C#," 2005. [Online]. Available: https://www.dropbox.com/s/gg2dpvkl9e00qyx/03 Application Source Code Explained.pdf?dl=0. [Accessed: 31-May-2021].

[28] Microsoft, "How to encrypt and decrypt a file using Visual C#," 2012. [Online]. Available: https://web.archive.org/web/20170113084447/https://support.microsoft.com/en-us/kb/307010. [Accessed: 31-May-2021].

[29] R. Parks, "Dear Microsoft, This is How You Encrypt a File," 2017. [Online]. Available: https://hackernoon.com/dear-microsoft-this-is-how-you-encrypt-a-file-779cc0a19bfc. [Accessed: 31-May-2021].

[30] R. Parks, "How Not to Encrypt a File — Courtesy of Microsoft," 2017. [Online]. Available: https://medium.com/@bob_parks1/how-not-to-encrypt-a-file-courtesy-of-microsoft-bfadf2b0273d. [Accessed: 31-May-2021].

[31] Microsoft, "Debugging Managed Code Using the Windows Debugger," 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-managed-code. [Accessed: 31-May-2021].