MS Crypto Derive Functions

sysopfb.github.io/malware,/reverse-engineering/2018/05/12/MS-Derivation-functions.html

Random RE May 12, 2018

May 12, 2018

Three functions come up a lot when you are trying to write scripts to decode out malware strings or configs, this is me dumping my notes on these functions.

So you have a .NET script using PasswordDeriveBytes? Or well really you have anything doing a routine on data. So what do you do? Well if the routine is semi complicated or I haven't run upon it before then I will commonly let the malware take the data and then right as it's passed off to the function in question I will force the data to be dumped out in all stages.

What does this mean? It means if I want to recreate how PasswordDeriveBytes works then I just simply need to pass the same values to it and dump the bytes out to a file on disk using a different script.

```
namespace PassDeriveTest
{
    class Program
    {
        static string bleh = "B5YDTLEDBjd+8zy5lzEfjw==";
        public static string S(string C_0)
            {
                string strPassword = "amp4Z0wpKzJ5Cg0GDT5sJD0sMw0IDAsaGQ1Afik6NwXr6rrSEQE=";
                string s = "aGQ1Afik6NampDT5sJEQE4Z0wpsMw0IDAD06rrSswXrKzJ5Cg0G=";
                string strHashName = "SHA1";
                int iterations = 2;
                int num = 256;
                string s2 = "@1B2c3D4e5F6g7H8";
                byte[] bytes = System.Text.Encoding.ASCII.GetBytes(s2);
                byte[] bytes2 = System.Text.Encoding.ASCII.GetBytes(s);
                byte[] array = System.Convert.FromBase64String(C_0);
                System.Security.Cryptography.PasswordDeriveBytes passwordDeriveBytes = new
System.Security.Cryptography.PasswordDeriveBytes(strPassword, bytes2, strHashName, iterations);
                byte[] bytes3 = passwordDeriveBytes.GetBytes(num / 8);
                File.WriteAllBytes("keydata.bin", bytes3);
                System.Security.Cryptography.ICryptoTransform transform = new
System.Security.Cryptography.RijndaelManaged
                {
                    Mode = System.Security.Cryptography.CipherMode.CBC
                }.CreateDecryptor(bytes3, bytes);
                System.IO.MemoryStream memoryStream = new System.IO.MemoryStream(array);
                System.Security.Cryptography.CryptoStream cryptoStream = new
System.Security.Cryptography.CryptoStream(memoryStream, transform,
System.Security.Cryptography.CryptoStreamMode.Read);
                byte[] array2 = new byte[array.Length];
                int count = cryptoStream.Read(array2, 0, array2.Length);
                memoryStream.Close();
                cryptoStream.Close();
                return System.Text.Encoding.UTF8.GetString(array2, 0, count);
            }
        static void Main()
        {
                string blah = S(bleh);
                Console.Write(blah);
        }
    }
}
```

So I'm writing the bytes out to a file keydata.bin while at the same time going ahead and decrypting the string, now I know exactly what the bytes used for the AES key will be which were generated from PasswordDeriveBytes. While I'm at it I can go ahead and change the iterations to 1 and output the edge case of what happens when iterations is 1 for generating the bytes, as it turns out it generates the same bytes as iterations being 2.

So now with this data we can attempt to recreate the routine in python and verify if we are in fact generating the same bytes for atleast this case. Generating the data we will need for other cases will be done in a similar manner but sometimes may involve doing things such as letting the malware call a function like CryptDeriveKey and then overwriting the bytecode in the malware itself to call an export function to dump the key into memory. Like most cyber security professions getting better at reverse engineering will usually come down to time, reading specifications and reference materials, testing or rolling your own routines/algorithms. If I want to understand how something works or how to identify it when I'm staring at disassembly then I find it much easier if I've already spent the time developing the routine or rolling my own cryto or using someones crypto library and then staring at it disassembled. It's far from glamorous but some of my most interesting research has involved staring at RFCs for months.

.NET PasswordDeriveBytes

Based on PBKDF1, however PBKDF1 doens't allow for generated bytes to exceed the hash length. Microsofts PasswordDeriveBytes however will reuse the hash before last on the iterations in order to generate more bytes in a stream.

It does this by reusing the last hash from the iteration and prepending an integer to it stringified starting at 1 and going until it has satisfied enough bytes for the requested stream.

```
import hashlib
from base64 import b64decode

def MS_PasswordDeriveBytes(pstring, salt, hashfunc, iterations, keylen):
    if iterations > 0:
        lasthash = hashlib.sha1(pstring+salt).digest()
        iterations -= 1
    else:
        print("Iterations must be > 0")
    #If iterations is 1 then basically the same thing happens as 2 based on my testing
    #if iterations == 0 and keylen > len(lasthash):
        #print("Dunno what happens here")
        #return -1
    for i in range(iterations-1):
        lasthash = hashlib.sha1(lasthash)
    bytes = hashlib.sha1(lasthash).digest()
    ctrl = 1
    while len(bytes) < keylen:
        bytes += hashlib.sha1(str(ctrl)+lasthash).digest()
        ctrl += 1
    return(bytes[:keylen])

stpass = 'amp4Z0wpKzJ5Cg0GDT5sJD0sMw0IDAsaGQ1Afik6NwXr6rrSEQE='
slt = 'aGQ1Afik6NampDT5sJEQE4Z0wpsMw0IDAD06rrSswXrKzJ5Cg0G='
initv = '@1B2c3D4e5F6g7H8'
enc_str = b64decode('B5YDTLEDBjd+8zy5lzEfjw==')
derbytes = MS_PasswordDeriveBytes(stpass, slt, hashlib.sha1, iterations=2, keylen=32)
derbytes
'4\x88m[\tz\x94\x19x\xd0\xe3\x8b\x1b\\\xa3)`tj^]d\x87\x11\xb1,g\xaa[:\x8e\xbf'
```

RFC2898DeriveBytes

This one uses PBKDF2, some simple python code for the pbkdf2 library: https://github.com/mitsuhiko/python-pbkdf2

```
passwd = "52LUPYXcB7UmIUIYCk0"
encrypted = "ETHfIKGznueFOwZzWrIPuz81fI2+EyHoMzKFUJIPBGc="
salt = "Ivan Medvedev"
blob = pbkdf2_bin(passwd, salt, keylen=32+16)
key = blob[:32]
iv = blob[32:]
from Crypto.Cipher import AES

data = "ETHfIKGznueFOwZzWrIPuz81fI2+EyHoMzKFUJIPBGc="

import base64
temp = base64.b64decode(data)
aes = AES.new(key, AES.MODE_CBC, iv)

>>> aes.decrypt(temp)

'A\x00p\x00p\x00L\x00a\x00u\x00n\x00c\x00h\x00.\x00e\x00x\x00e\x00\x06\x06\x06\x06\x06\x06'

>>> 'A\x00p\x00p\x00L\x00a\x00u\x00n\x00c\x00h\x00.\x00e\x00x\x00e\x00'.decode('utf-16')

u'AppLaunch.exe'
```

MS CryptoAPI CryptDeriveKey

As an example using RC4

If I hash the string ‘test’ using any hash but let’s say md5

And then derive an RC4 key passing 0x280011 as the flags

Then the key is bit length of the flags value 0x280011 » 16 to get the number of bytes simply divide by 8 in this case 5 bytes of the resulting md5 hash will be used as the RC4 key

```
>>> import hashlib
>>> hashlib.md5('test').digest()
"\t\x8fk\xcdF!\xd3s\xca\xdeN\x83&'\xb4\xf6"
>>> flags = 0x280011
>>> hex(flags >> 16)
'0x28'
>>> 0x28/8
5
>>> hashlib.md5('test').digest()[:5]
'\t\x8fk\xcdF'
>>> from Crypto.Cipher import ARC4
>>> rc4 = ARC4.new(hashlib.md5('test').digest()[:5])
>>> rc4.decrypt(encoded_data)
```

Hope it helps.

Thanks to @maciekkotowicz and @noottrak

References:
https://github.com/Microsoft/referencesource/blob/master/mscorlib/system/security/cryptography/passwordderivebytes.cs
https://tools.ietf.org/html/rfc2898#section-5.1