Flare-On 3: Challenge 2 Solution - DudeLocker.exe

Challenge Author: Matt Williams (@0xmwilliams)

Your task in this challenge was to reverse engineer `DudeLocker.exe` in order to decrypt the associated `BusinessPapers.doc` file.

DudeLocker Activity

`DudeLocker.exe` is a poorly implemented ransomware sample that pays homage to a popular film involving a ransom. The binary begins by checking for a folder named `Briefcase` on the current user's Desktop. If found, the current volume's serial number is compared to the value `0x7DAB1D35` ("`TDABIDES`"). If the values match, `DudeLocker.exe` decodes a string using the volume serial number as a multi-byte XOR key.

The resulting string ("`thosefilesreallytiedthefoldertogether`") is passed to a function that establishes the malware's cryptographic context. Relevant parameters passed to Windows cryptography functions are shown in Figure 1. To summarize these function calls, an AES-256 key is derived from the SHA-1 hash of the decoded string. The AES encryption mode is also set to `CBC`. This mode is actually set by default, making the `CryptSetKeyParam` call unnecessary.

Function	Relevant Parameter	Value
CryptAcquireContext	dwProvType	`PROV_RSA_AES`
CryptCreateHash	Algid	`CALG_SHA1`
CryptHashData	pbData	"`thosefilesreallytiedthefoldertogether`"
CryptDeriveKey	Algid	`CALG_AES_256`
	hBaseData	Hash object resulting from CryptHashData
CryptSetKeyParam	dwParam	`KP_MODE`
	pbData	`CRYPT_MODE_CBC`

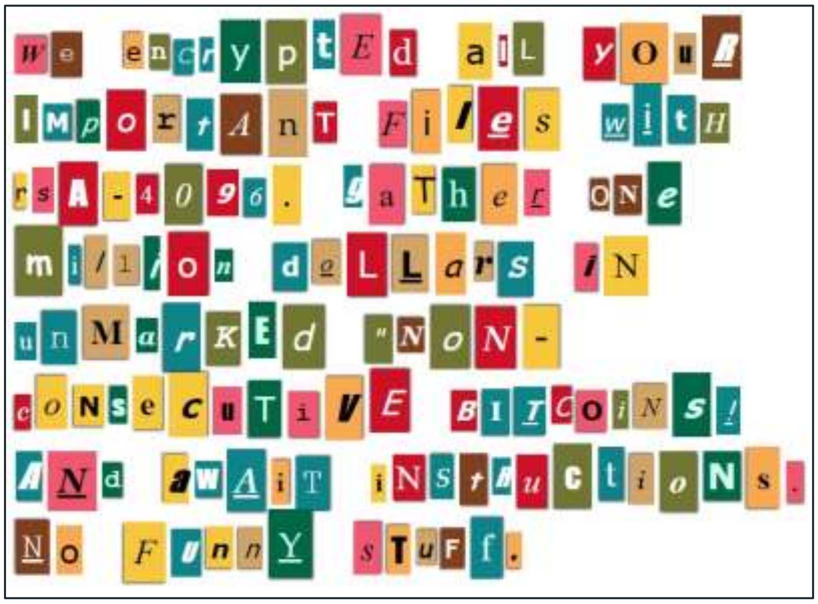
Figure 1: Deriving an AES-256 key

`DudeLocker.exe` proceeds by iterating through files found in the `Briefcase` directory and its sub-directories. When a file is found, an MD5 hash is calculated for its lowercase filename and extension. The hash is set as the AES initialization vector (IV) using the Windows cryptography functions and relevant parameters shown in Figure 2.

Function	Relevant Parameter	Value
CryptCreateHash	Algid	`CALG_MD5`
CryptHashData	pbData	Lowercase filename and extension
CryptGetHashParam	dwParam	`HP_HASHVAL`
CryptSetKeyParam	dwParam	`KP_IV`
	pbData	MD5 hash acquired from CryptGetHashParam

Figure 2: Setting a unique IV for each file

Once the IV is set, two handles to the file are obtained: one for reading and one for writing. The file's content is read, encrypted using `CryptEncrypt`, and written back to the file in 16KB blocks. After encrypting every file in the `Briefcase` directory, the binary drops an embedded resource to a file named `ve_vant_ze_money.jpg` (Figure 3).



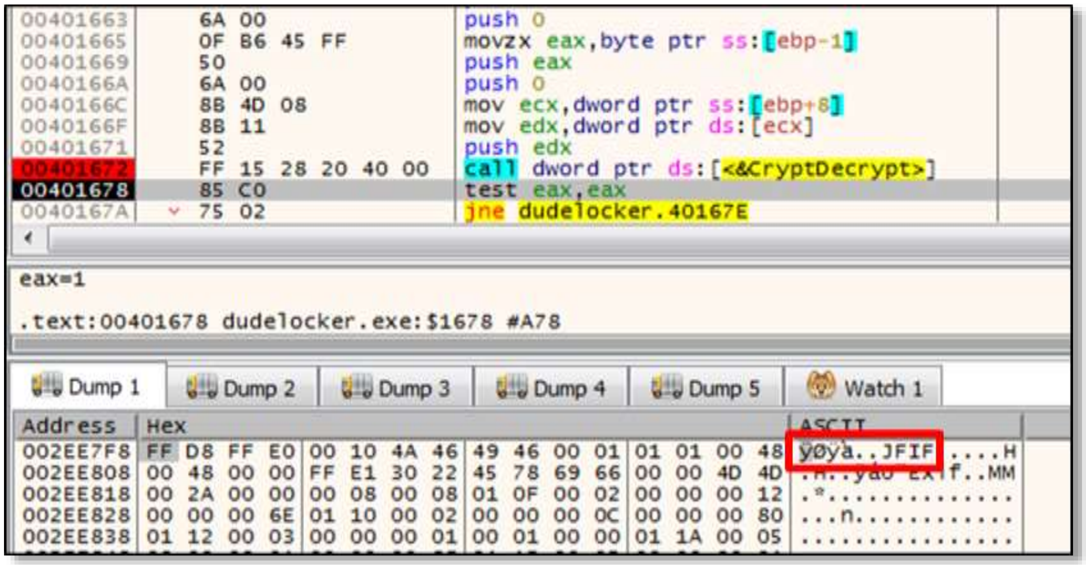
We encrypted all your important files with RSA-4096. Gather one million dollars in unmarked "non-consecutive bitcoins! And await instructions. No funny stuff.

Figure 3: Ransom note resource

Finally, `DudeLocker.exe` attempts to set the current user's desktop wallpaper to the ransom note image if the operating system version is Windows Vista+.

Decrypting BusinessPapers.doc

The decryption of `BusinessPapers.doc` can be implemented using a C/C++ program that calls the same series of cryptographic functions but instead uses CryptDecrypt. A much faster solution involves manually replacing `CryptEncrypt` with `CryptDecrypt`. Because the first six parameters of `CryptDecrypt` are identical to `CryptEncrypt`, one could simply modify the sample's import address table (IAT) statically using a PE tool or at runtime using a debugger. After performing the modification and allowing `DudeLocker.exe` to locate `BusinessPapers.doc` in the `Briefcase` folder, the initial call to `CryptDecrypt` reveals the decrypted file's signature matches a JPG file instead of a document, as shown in Figure 4 below.



```
00401663     6A 00               push 0
00401665     0F B6 45 FF         movzx eax,byte ptr ss:[ebp-1]
00401669     50                  push eax
0040166A     6A 00               push 0
0040166C     8B 4D 08            mov ecx,dword ptr ss:[ebp+8]
0040166F     8B 11               mov edx,dword ptr ds:[ecx]
00401671     52                  push edx
00401672     FF 15 28 20 40 00   call dword ptr ds:[<&CryptDecrypt>]
00401678     85 C0               test eax,eax
0040167A   v 75 02               jne dudelocker.40167E

eax=1

.text:00401678 dudelocker.exe:$1678 #A78
```

```
Address   Hex                                              ASCII
002EE7F8  FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 48  ÿØÿà..JFIF.....H
002EE808  00 48 00 00 FF E1 30 22 45 78 69 66 00 00 4D 4D  .H..ÿá0"Exif..MM
002EE818  00 2A 00 00 00 08 00 08 01 0F 00 02 00 00 00 12  .*..............
002EE828  00 00 00 6E 01 10 00 02 00 00 00 0C 00 00 00 80  ...n............
002EE838  01 12 00 03 00 00 00 01 00 01 00 00 01 1A 00 05  ................
```

Figure 4: Decrypted file's signature

Allowing the program to execute and opening the decrypted file reveals the challenge solution shown in Figure 5.

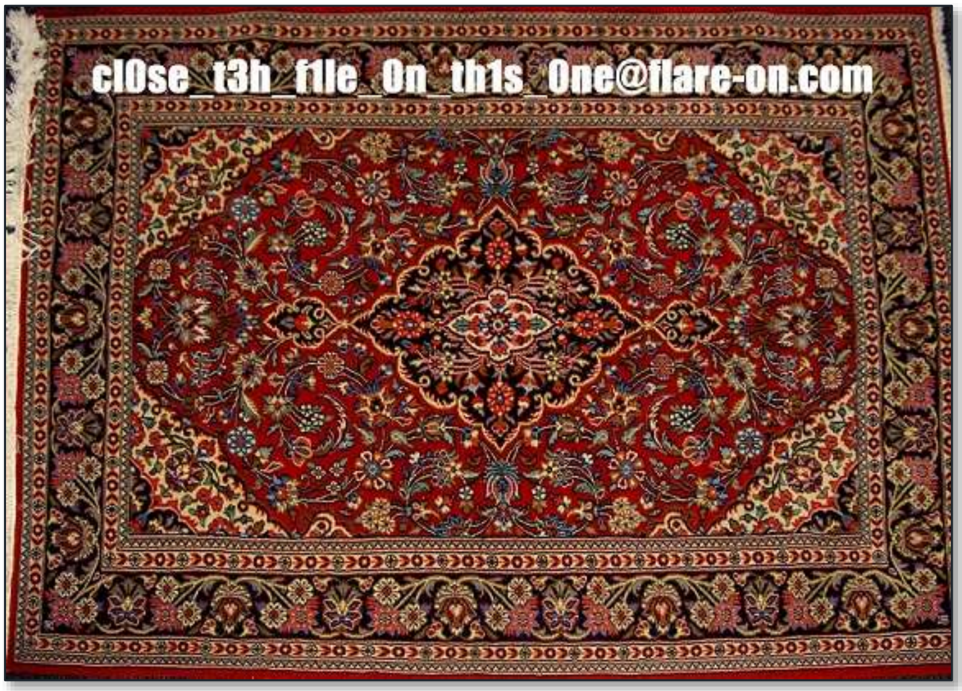


Figure 5: Final solution (`cl0se_t3h_f1le_0n_th1s_0ne@flare-on.com`)

Note that patching `CryptEncrypt` to `CryptDecrypt` will not produce a decrypted file that is identical to the original. This is a side effect of overwriting the encrypted file with the decrypted data, which in this case is 11 bytes less than the encrypted file size. Inserting a call to `SetEndOfFile` after the final `CryptDecrypt` would remove the excess bytes leftover from the encrypted file.

For those who attempted to solve the challenge using Python, a Python decryptor that does not utilize the `ctypes` module is not exactly straightforward. This is due to the `CryptDeriveKey` function, whose inner workings are described in the “Remarks” section of its MSDN page. Before we can use `PyCrypto` to decrypt the file, we must derive the AES key.

In the Python solution shown in Figure 6, the `derive_key` function is a Python implementation of the steps performed by `CryptDeriveKey` for this particular sample. After deriving the key, the Python script uses the first 32 bytes (256 bits) returned from the function as the AES key and derives the IV from the lowercase filename and extension. The file content is decrypted, unpadded, and used to overwrite the original encrypted file.

```python
import sys
import hashlib
from Crypto.Cipher import AES

def derive_key(key):
    # SHA-1 hash algorithm used
    key_sha1 = hashlib.sha1(key).digest()

    b0 = ""
    for x in key_sha1:
        b0 += chr(ord(x) ^ 0x36)

    b1 = ""
    for x in key_sha1:
        b1 += chr(ord(x) ^ 0x5c)

    # pad remaining bytes with the appropriate value
    b0 += "\x36"*(64 - len(b0))
    b1 += "\x5c"*(64 - len(b1))

    b0_sha1 = hashlib.sha1(b0).digest()
    b1_sha1 = hashlib.sha1(b1).digest()

    return b0_sha1 + b1_sha1

unpad = lambda s: s[0:-ord(s[-1])]  # remove pkcs5 padding

fname = sys.argv[1]
with open(fname, 'rb+') as f:
    encrypted_data = f.read()

    key = "thosefilesreallytiedthefoldertogether"
    # 256-bit key / 8 = 32 bytes
    aes_key = derive_key(key)[:32]

    iv_name = fname[fname.rfind('\\') + 1:]
    iv = hashlib.md5(iv_name.lower()).digest()

    decryptor = AES.new(aes_key, AES.MODE_CBC, iv)
    decrypted_data = unpad(decryptor.decrypt(encrypted_data))

    f.seek(0)
    f.write(decrypted_data)
    f.truncate(len(decrypted_data))
```

Figure 6: Python script to decrypt `BusinessPapers.doc`