ENDGAME.

Dude! Where's My Ransomware?: *A Flare-On Challenge*

SERIES ONE, VOLUME ONE

BLAINE STANCILL + JOSHUA WANG

There are many tricks to the tradecraft when analyzing unknown binaries, and it requires constant honing of skills to stay on top of the latest malware and campaigns. Solving reverse-engineering challenges is one way to keep your skills sharp. In our previous post, we discussed some tips from the Flare-On Challenge. Now we'll take a deeper dive into one of the specific challenges (Challenge #2), walking through the "DudeLocker.exe" binary and a file called "BusinessPapers.doc". This is a great use case for seeing how ransomware works on a basic level, and will provide some useful tips that you can implement next time you're investigating an unknown binary.

Getting Started

When analyzing an unknown binary, you'll usually be performing a dance between static and dynamic analysis. As a rule of thumb, always use separate virtual machines (VM) when doing these forms of analysis. Here's an example of why it's a bad idea to do both on the same VM. Imagine you have performed hours of static analysis and decided to run the binary. Oh no! It took a code path you didn't account for and has now encrypted all the files on your box, including your static analysis notes/files. That definitely is not an ideal scenario!

Due to the nature of unknown binaries, only execute/run them in a dynamic analysis VM to prevent interference with your static analysis. This will additionally give you the freedom to run the binary, debug it, and revert to prior VM snapshots whenever needed. For instance, we reverted to prior snapshots after executing/running "DudeLocker.exe" a few times to get a feel for what its behavior was and to start with a clean system each time.

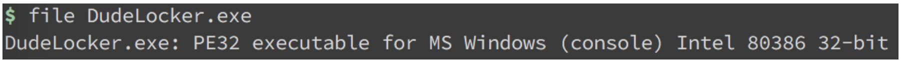
Pro Tip: Take advantage of how Windows recognizes file types and remove the file extension prior to transferring the binary into the static analysis VM. This will prevent Windows from executing it even if double-clicked.

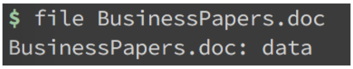
With the infrastructure established, the initial step towards analyzing an unknown binary is to determine the file type or in our case what kind of files “DudeLocker.exe” and “BusinessPapers.doc” are. On Mac/Linux the ‘file’ command comes in handy *(see below).*

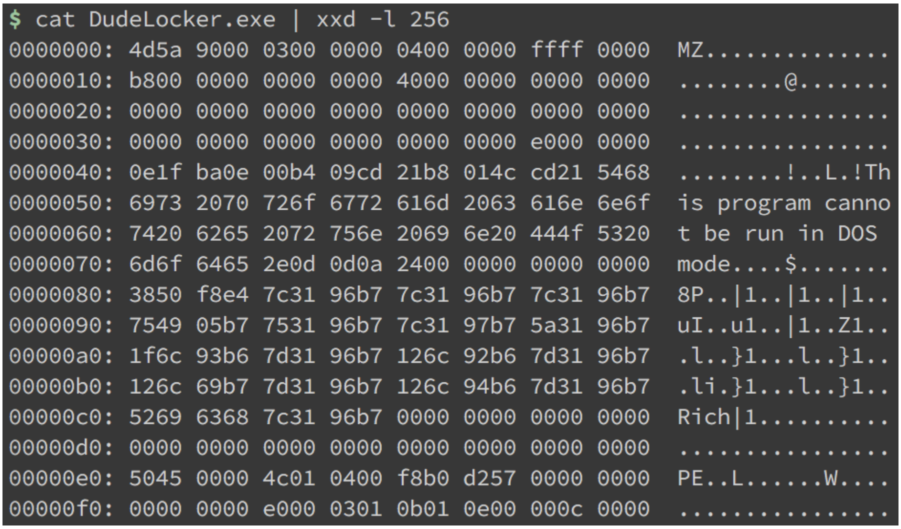
The ‘file’ command determines the file type by looking for “magic bytes” within a file. These “magic bytes”, which are described in more detail elsewhere, are frequently unique to a specific file type and can usually be found at or near the beginning of a file.

The file “BusinessPapers.doc” is not recognized as any particular file type even though its extension is “.doc”; rather, it is shown to be “data”. However, “DudeLocker.exe” is recognized as a 32-bit PE executable. If you were to take a slightly closer look at the beginning bytes using the ‘xxd’ command, which displays a hex dump of a file, you can easily see the “magic bytes” are “**MZ**” or **4D 5A**.

To understand how the ‘file’ command knows other attributes about the file such as its architecture type (32-bit), you’ll need to understand the file format in more detail. In the reverse-engineering world, you likely need to become very familiar with files that execute. On Windows these are files that follow the PE format. On Linux they are files that follow the ELF format and on Mac, Mach-O. To get familiar with the PE format we highly recommend Iczelion’s PE tutorials as well as this Microsoft article.


```
$ file DudeLocker.exe
DudeLocker.exe: PE32 executable for MS Windows (console) Intel 80386 32-bit
```


```
$ file BusinessPapers.doc
BusinessPapers.doc: data
```


```
$ cat DudeLocker.exe | xxd -l 256
0000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
0000010: b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 e000 0000  ................
0000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
0000050: 6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
0000060: 7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
0000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
0000080: 3850 f8e4 7c31 96b7 7c31 96b7 7c31 96b7  8P..|1..|1..|1..
0000090: 7549 05b7 7531 96b7 7c31 97b7 5a31 96b7  uI..u1..|1..Z1..
00000a0: 1f6c 93b6 7d31 96b7 126c 92b6 7d31 96b7  .l..}1...l..}1..
00000b0: 126c 69b7 7d31 96b7 126c 94b6 7d31 96b7  .li.}1...l..}1..
00000c0: 5269 6368 7c31 96b7 0000 0000 0000 0000  Rich|1..........
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 5045 0000 4c01 0400 f8b0 d257 0000 0000  PE..L......W....
00000f0: 0000 0000 e000 0301 0b01 0e00 000c 0000  ................
```

Light Static Analysis: Quick Look

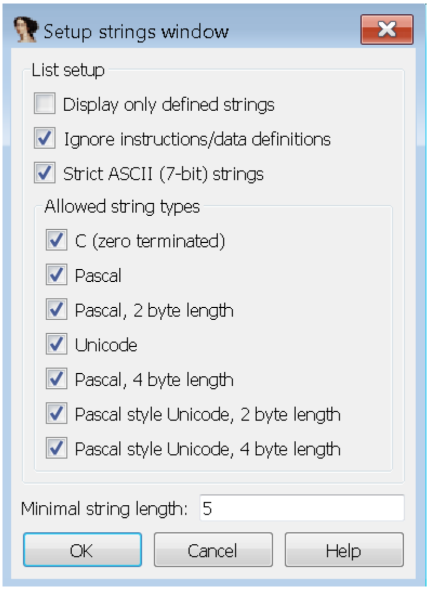
"DudeLocker.exe"

Generally, once you've established the type of file you're working with, you can perform some light static analysis and look for interesting nuggets of information prior to running it dynamically. We know "DudeLocker.exe" is a Windows executable. Some tools to use for static analysis of Windows executables are CFF Explorer, Detect It Easy, Resource Hacker, and IDA Pro/Free.

A great way to start static analysis on any binary is the tried and true *strings* utility. There are many programs that can extract strings from a binary -- in Linux/Mac there's even a 'strings' command. Strings found in a binary can sometimes leak precious information such as URLs, email addresses, passwords, packer names, registry key names, etc… and are worth exploring. We'll use IDA Pro for the moment on "DudeLocker.exe" to view its strings.

Pro-Tip: When extracting strings from a binary, there will usually be a minimum length setting. Five

characters are typically enough. Also be sure to specify all possible string types depending on what your strings utility is capable of extracting. In IDA Pro you can find: Ascii/C type, Unicode, and Pascal type strings.


Setup strings window

List setup
- [ ] Display only defined strings
- [✓] Ignore instructions/data definitions
- [✓] Strict ASCII (7-bit) strings

Allowed string types
- [✓] C (zero terminated)
- [✓] Pascal
- [✓] Pascal, 2 byte length
- [✓] Unicode
- [✓] Pascal, 4 byte length
- [✓] Pascal style Unicode, 2 byte length
- [✓] Pascal style Unicode, 4 byte length

Minimal string length: 5

Interestingly we see mentions of a "Briefcase" and an image name "\\ve_vant_ze_money.jpg". Speculating early on, we know briefcases generally contain documents used for business purposes and "ve_vant_ze_money.jpg" sounds like some kind of ransom note as it resembles "we want the money". We'll see later on exactly how these come into play. After viewing the strings of an executable, it's a good idea to view its imports (part of the PE format). Imports are functions imported from external libraries that provide functionality to an executable and can shed some light on the binary's potential behavior or capabilities. Let's look at the imports for "DudeLocker.exe", again using IDA Pro for the moment.
From these imports we can get a feeling for some of the binary's potential capabilities, including:

- **File enumeration**
    - FindFirstFileW
    - FindNextFile
- **System fingerprinting**
    - GetVolumeInformationA
    - GetVersionExW
- **Resource access**
    - FindResourceW
    - LoadResource
- **Read/Write files**
    - ReadFile
    - WriteFile
- **Modify system parameters**
    - SystemParametersInfoW
- **Output debug messages**

  - OutputDebugStringW
- **Access specific folders via CSIDL**
  - SHGetFolderPathW
- **Use of encryption**
  - CryptDeriveKey
  - CryptEncrypt
- **Perform hashing**
  - CryptCreateHash
  - CryptHashData
  - CryptGetHashParam

Pro-Tip: The 'A' vs 'W' at the end of import functions denotes whether the function uses 'ANSI' (typically ASCII) or 'wide' (Unicode) characters.

To the untrained eye these imports may seem innocuous, but if you look at the potential capabilities as a whole you can start to see how they might be utilized by "DudeLocker.exe" in general. For instance, what type of behavior would you suspect the following capabilities to express?

- **File enumeration**
- **Read/Write files**
- **Use of encryption**

These three capabilities are the minimum required to express some sort of ransomware behavior. While imports are not a guaranteed way of deriving behavior, as we've yet to execute "DudeLocker.exe", it's always good to think in terms of how the imports could be utilized, especially if used maliciously.

We could also have used a tool called CFF Explorer to view the imports of "DudeLocker.exe". CFF Explorer is great for viewing an executable's PE header information. Let's use it to explore "DudeLocker.exe". If we navigate to the "Section Headers" tab has a ".rsrc" section, which typically contains embedded resources. These resources can be anything from icons to images and in some cases even malicious files.

Whenever you want to know more about an executable's resources, you can always turn to a tool called Resource Hacker. This tool is the one-stop shop for compiling, viewing, decompiling and recompiling resources for both 32-bit and 64-bit Windows executables. Let's open "DudeLocker.exe" with Resource Hacker. We see there is an "RCData", or raw resource, with an ID of "101". This might be the

"\\ve_vant_ze_money.jpg" image we were hinted at in the strings. If we factor in the potential import capabilities - file enumeration and encryption functions - "DudeLocker.exe" is starting to resemble some sort of ransomware. This would make sense, especially given the name of the executable which resembles "CryptoLocker", a famous ransomware trojan introduced to the world in 2013 which was responsible for extorting millions of dollars from its victims.

"BusinessPapers.doc"

In the case you can't determine a file's type, it still has some useful properties you can observe. For instance, checking for the presence of strings, as we did for "DudeLocker.exe", and the file's entropy characteristics are good starters. High entropy (close to a value of 8 on a scale of 0-8) can be a sign of a packed/compressed/encrypted file. A great tool for checking both strings and file/section entropy is Detect-It-Easy (also via github). It's also useful for detecting file types, compilers used, potential packers, cryptographic functionality, and a lot more. Let's see if it can recognize the file type for "BusinessPapers.doc" and view its strings as well as entropy.

According to Detect-It-Easy, "BusinessPapers.doc" has no recognizable file type, the strings appear to be gibberish, and it has high entropy. All of these are tell-tale signs of either packing/compression or encryption. Remember our initial observation of "DudeLocker.exe" being potential ransomware? Maybe this file was encrypted by it.

Light Dynamic Analysis: Run it and Observe

Only so much information can be gleaned from light static analysis. To gain more insight into a binary's behavior, it's easiest to run it and observe what it does -- in a dynamic analysis VM, of course. In our case, we know that "DudeLocker.exe" is a 32-bit Windows executable and has some potential ransomware capabilities, so let's run it and see if our current observations hold. Some great tools to use when doing dynamic analysis are Process Monitor (ProcMon) and Process Explorer (ProcExp), both of which are included in the SysInternal Suite provided by Microsoft. Process Monitor provides a detailed event log of actions taken by the binary and Process Explorer provides a TaskManager type view of all currently running processes, making it easy to see if processes spawn or exit.

Additionally, since we saw the import “OutputDebugStringW” in “DudeLocker.exe”, we’ll also be using a tool called DebugView which displays any debug statements and is also included in Microsoft’s SysInternal Suite. Let’s begin the light dynamic analysis by using Process Monitor and DebugView to monitor the execution of “DudeLocker.exe”

Pro-Tip: In ProcMon you can set up specific filters if you want to look for more details than are set by default. If you need even more detail about the execution flow of a binary, try API Monitor. It can get information regarding thousands of Windows APIs potentially called by an executable.

From ProcMon output it appears “DudeLocker.exe” accesses the Desktop, attempts to access a directory called “Briefcase”, and then exits. From DebugView output it seems we’re no longer reverse engineers, but more like software engineers taking a debug print statement approach to debugging. The attempt to open the directory “Briefcase” on the Desktop failed, returning a “NAME NOT FOUND” error. Due to the almost immediate exit of the binary, we can assume that the binary requires a “Briefcase” directory on the Desktop to continue its execution. So let’s create an empty folder called “Briefcase” on the Desktop and re-run it.

Interesting. Now that we have the “Briefcase” folder on the desktop, “DudeLocker.exe” does something extra prior to exiting, it queries for system specific information (a technique usually called fingerprinting) as seen by the

"QueryInformationVolume" event. In addition, the debug output says "I'm out of my element". At this point we have no further clues as to what "DudeLocker.exe" requires in order to continue its execution.

Deep Static Analysis (Disassembler)

Once you've seen enough general behavior from an unknown binary or reach a dead end during light dynamic analysis you'll typically want to take a deeper look into the binary statically. Let's start our deep dive into static analysis of "DudeLocker.exe" via our disassembler of choice, IDA Pro.

Once in the disassembler you can choose to start your analysis generally one of two ways:

1. Start from the main/start function and work your way through the binary. This a top-down approach.
2. Start at an interesting string, code block, import function, or function and work your way backwards. This is a bottom-up approach.

Since we hit a roadblock after adding the "Briefcase" folder to the desktop, let's find the string "I'm out of my element" seen from the DebugView output and work our way backwards through IDA.

Once you've found the string in IDA's string view (shift + F12) you can double-click it to reach the string in the ".data" section. From here we can utilize the power of cross-referencing by pressing 'x' to see where this string is referenced in the binary. Following this backwards we can see how we would arrive at this execution path.

When doing a deep dive static analysis, it helps if you label functions and data as you go. In IDA this is done by pressing 'n' on a selected function or data name. Even if you don't know what the function really does, a best guess label still helps as you can always re-label it with something more descriptive once you understand it better.

Pro-Tip: If you can't understand the functionality of a function because there are other functions and data being referenced within it, it can help if you work your way bottom-up. This way, once you've labeled the lowest and simplest functions you'll have a better understanding of what the functions above it do.

Going back to the observed execution path in "DudeLocker.exe", we see there's a function we've labeled "GetVolumeSerialNumber" that decides the fate of execution. Let's see what the function "GetVolumeSerialNumber" actually does.

The function compares our VM's File System volume serial number against 0x7DAB1D35. If our VM's serial doesn't match this expected value, the function returns 0 in EAX, causing the execution flow to go down the path that prints out the debug string "I'm out of my element" and exit. To make the execution continue, we'll need some way to modify our VM's volume serial number.

While we could jump over to our dynamic analysis VM and try to figure out what would happen if we forced our VM's volume serial number to match, let's stay in static analysis mode and try to figure it out ourselves.

If we follow the path that would be taken if the serial numbers matched, we notice a reference to some data blob and a function that has an XOR in it. Not always, but when you encounter a data blob and an XOR, there's a good chance some XOR en/decryption is going on. This function, which we've labeled "DecryptSeedValue", takes as parameters: a data blob, an output buffer, a volume serial number, and the length of the output buffer.

The data blob is XOR decrypted using the volume serial number and stored in the output buffer. You could decrypt the data blob yourself now that you know the general algorithm and values being used via a script, or you could use dynamic analysis to see the decrypted blob -- we'll use dynamic analysis later for this.

Diving into these could require a blog post in itself. We'll leave it as an exercise for the reader to familiarize themselves with the Windows cryptography API, specifically how to use AES encryption in CBC mode. See this tutorial for some good code examples. An overview of these functions and this portion of program execution is as follows:

1. SHA-1 hash the decrypted data blob
2. Use the SHA-1 hash to derive an AES-256 key
3. Recursively encrypt each file (and each file in subdirectories) in the "Briefcase" directory:

   A. Derive a unique encryption key for the file:
   1. Get the file's name (including extension) and convert to lowercase
   2. MD5 hash the lowercase filename
   3. Use MD5 hash as initialization vector (IV) for AES-256 encryption, thus creating a "unique key" per file

   B. Encrypt the file:
   1. Open the file and read it in 16KB chunks
   2. Use the unique encryption key based on filename to encrypt each 16KB chunk
   3. Write the encrypted chunk of data back to the file

After all files have been encrypted in the Briefcase directory, "DudeLocker.exe" will look up a resource.

Notice that the function FindResourceW is looking for a RT_RCDATA, or raw, resource with an ID of 101. Looking back at the Resource Hacker output, we can see the resource with ID 101 is the ransom note image we saw earlier! "DudeLocker.exe" will write this image to disk in the "Briefcase" directory and name it "ve_vant_ze_money.jpg".

Then "DudeLocker.exe" will check if the operating system (OS) is Vista or higher by calling GetVersionEx and comparing the returned OSVERSIONINFOEX structure's dwMajorVersion value against the value 5. A dwMajorVersion of 6+ indicates Vista+. Thus, if the current version is 5 or below, we're not on Vista+. See this article for information about the OSVERSIONINFOEX structure and the differences in major vs minor version number. If the OS is Vista+, "DudeLocker.exe" will set the ransom note image as the desktop background and exit.

Recap

Let's briefly recap what we know about "DudeLocker.exe":

• The binary needs a directory on the Desktop named "Briefcase"

• The binary needs a volume serial number of 0x7DAB1D35 to continue execution and derive an AES-256 encryption key

• The binary will encrypt all files in the "Briefcase" directory using their filename as the IV for the AES-256 encryption algorithm

• The binary will look up the resource corresponding to ID 101 and write this to disk as "ve_vant_ze_money.jpg" in the "Briefcase" directory

• If on Vista+ it will set the ransom note image "ve_vant_ze_money.jpg" as the desktop background

The question now is what to do with the "BusinessPapers.doc" file. We suspect it's been previously encrypted by this variant of ransomware, but how can we decrypt it?

Hopefully you've familiarized yourself with AES encryption and had the "aha moment"! AES is a symmetric encryption algorithm meaning that as long as we can derive the same key used to encrypt the file we can decrypt the file. Well, we know its filename and we know the expected volume serial number, thus we can derive the same key!

Additionally, if you also familiarized yourself with the Windows cryptography API, you'll notice that both CryptEncrypt and CryptDecrypt share the same first 6 parameters.

The significance of these functions sharing the same 6 first parameters is that we can use CryptDecrypt in place for CryptEncrypt. Now we have a way of decrypting “BusinessPapers.doc”! To actually do this we’ll patch/modify the Import Address Table (IAT) of “DudeLocker.exe” by changing CryptEncrypt to CryptDecrypt.To patch a binary means to directly modify its contents either when it’s loaded in memory or on disk. This is easily accomplished using CFF Explorer and saving it as another executable.

Deep Dynamic Analysis (Debugger)

Once you’ve accomplished a deep dive in static analysis you’ll generally have a better understanding of the unknown binary. By performing dynamic analysis, you can validate your understanding of the binary. However, in some cases your static analysis might be too complicated to determine what a particular function does. Dynamic analysis can also be utilized in these situations to experiment with the difficult function by providing it various inputs and monitoring what output it generates.

We have a pretty solid idea of what “DudeLocker.exe” does and have patched the binary’s IAT to decrypt files instead of encrypt them. We could essentially run the binary and have it decrypt “BusinessPapers.doc” for us (making sure this file is in the Briefcase directory). However, there is still one last hurdle we must conquer -- forcing the volume serial number to be 0x7DAB1D35. To do this let’s open the patched binary

```
BOOL WINAPI CryptEncrypt(
  _In_     HCRYPTKEY  hKey,
  _In_     HCRYPTHASH hHash,
  _In_     BOOL       Final,
  _In_     DWORD      dwFlags,
  _Inout_  BYTE       *pbData,
  _Inout_  DWORD      *pdwDataLen,
  _In_     DWORD      dwBufLen
);
```

```
BOOL WINAPI CryptDecrypt(
  _In_     HCRYPTKEY  hKey,
  _In_     HCRYPTHASH hHash,
  _In_     BOOL       Final,
  _In_     DWORD      dwFlags,
  _Inout_  BYTE       *pbData,
  _Inout_  DWORD      *pdwDataLen
);
```

with a debugger. We'll be using x64dbg, which is an open source x64/x32 debugger for Windows.

Once opened, depending on your debugger and its preferences, it should hit a breakpoint either at the first system function that initialized the patched binary or at the entry point of the patched binary. If it doesn't hit a breakpoint and continues to run, check your debugger's preferences and make sure "Entry Breakpoint" is set and re-run it with the debugger.

The goal of opening the file in a debugger is so we can hit a breakpoint at the spot where the patched binary will compare the system returned volume serial number to the desired value 0x7DAB1D35 and dynamically modify the returned value to match.

However, before we start setting breakpoints it's always good to know if the binary in question uses address space layout randomization (ASLR). If it does you might have to re-set your breakpoints each time you re-run the binary in the debugger and you might also have to rebase the binary in your disassembler to match the new image base address so your addresses match up in both the debugger and disassembler. An example of how to rebase your binary in a IDA Pro can be found here.

Pro Tip: If not using ASLR, the default base address for an .exe file is 0x400000 for 32-bit images or 0x140000000 for 64-bit images. For a DLL, the default base address is 0x10000000 for 32-bit images or 0x180000000 for 64-bit images.

Furthermore, some debuggers, like WinDbg, will allow you to set unresolved breakpoints. These are breakpoints that are defined by specifying an offset from the start of a function to break on. This bypasses ASLR because when a binary that uses ASLR is loaded into memory, even if the binary file is rebased, each instruction within the executable remains at the same offset from the base address.
The only thing that changes is the base address the binary file is

loaded at. To see if the binary uses ASLR check in the PE header:

• FileHeader -> OptionalHeader -> DllCharacteristics
  And see if the following flag is set:
• IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE = 0x0040; //
  The DLL can be relocated at load time. We can see in CFF Explorer that "DudeLocker.exe" does not use ASLR as "Dll can move" is not checked:
  Since it doesn't use ASLR we're free to set breakpoints and rely on them. Let's start by setting the following breakpoints *(see below).*

Pro Tip: Most debuggers let you set breakpoints via a breakpoint menu tab or by navigating to that address and manually setting it. X64dbg and Ollydbg both let you directly navigate to an address by pressing CTRL+g and manually setting a breakpoint by pressing the F2 key.

Once set, continue execution (typically F9 key) until a breakpoint is hit -- it should pause execution at 0x00401063. From here we can right-click and modify the topmost stack value (this corresponds to EBP-4 or the volume serial number returned by the system) to 0x7DAB1D35.

If you single-step (typically F8 key), the following jump instruction should not be taken and EAX should be set to the value 0x7DAB1D35. Now if we continue execution (typically F9 key) until a breakpoint is hit we should pause execution at 0x00401AD1. We're breakpointing here to point out the decoded data blob that is hashed and used as the AES-256 key. Specifically a pointer to this decoded data blob is in ECX at this point. If we right-click the ECX register and select "Follow in Dump", we should see the value "thosefilesreallytiedthefoldertogether". Pretty neat!

Breakpoint Address	Description
0x00401063	Address following the call to GetVolumeInformationA
0x00401AD1	Address to view the decoded seed value

Let's go ahead and let this binary run to completion by continuing execution (typically F9 key). The background should be set to the ransom note image we saw in Resource Hacker previously (assuming you're using a Vista+ OS) and if we extract the "BusinessPapers.doc" file and run the 'file' command on it again we should see it is no longer classified as "data".

That's right, it's a JPEG! Using command 'xxd' we can see it has the magic byte signature: FF D8 FF E0 nn nn 4A 46 49 46 00 01 (where "nn" can be any byte). We then rename the file to "BusinessPapers.jpg."

Awesome! We've been able to recover the previously encrypted file and found the flag for this challenge: cl0se_t3h_f1le_0n_th1s_0ne@flare-on.com.

Conclusion

This blog post walked through the steps and mindset required to solve just one challenge within the 10-part 2016 Flare-On Challenge. Keeping in mind that this is only level 2, the later levels become much more difficult and present new challenges. By participating in reverse engineering and malware focused CTFs such as the Flare-On Challenge, you can quickly gain skills and expose yourself to different problem sets that you would otherwise only encounter in the real world by analyzing malicious binaries of the same complexity.

We covered the basic fundamentals that will serve as a useful starting point for anyone interested in developing or honing their reverse engineering skills. Some of the important concepts we discussed include learning how to differentiate between different file types, performing a combination of light and deep static and dynamic analysis, understanding how to use different tools to aid in each of these types of techniques, and using different analysis strategies (bottom-up vs top-down).

Reverse engineering is an art form. Even when analyzing more complex binaries, these core concepts are still applicable. The only thing that changes is the speed at which we are able to perform our analysis and understand what the binary does. As a reverse engineer gains more experience they also learn shortcuts and pick up additional tools and techniques that allow them to reverse faster and more efficiently. We will cover some of these additional tools and techniques in our next blog post.

Lastly, we would like to thank FireEye's Flare Team again for putting together another solid set of challenges this year! There is great learning value in participating in CTFs like the Flare-On Challenge. If you have never participated in a CTF or related challenges, we highly recommend giving it a try.

ENDGAME.