Flare-On 7: Challenge 2 –garbage.exe

Challenge Author: Jon Erickson

This challenge was inspired by a real-life experience of receiving and performing analysis of corrupt packed executable.

Performing on a quick triage on this sample reveals that it is packed with UPX. Performing strings analysis reveals some partial Base64 strings but not much else. Trying to unpack the file using the standard `upx -d` command reveals an error message as show in Figure 1.

```
$ upx -d garbage-v1-prod.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95w       Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: garbage-v1-prod.exe: OverlayException: invalid overlay size; file is possibly corrupt

Unpacked 1 file: 0 ok, 1 error.
```

Figure 1 - upx -d unpacking attempt

Since we cannot unpack this file statically, we can attempt to unpack it dynamically. Attempts to open run the binary in both x32dbg and WinDbg fail with indications that the file is not a valid application as shown in Figure 2.

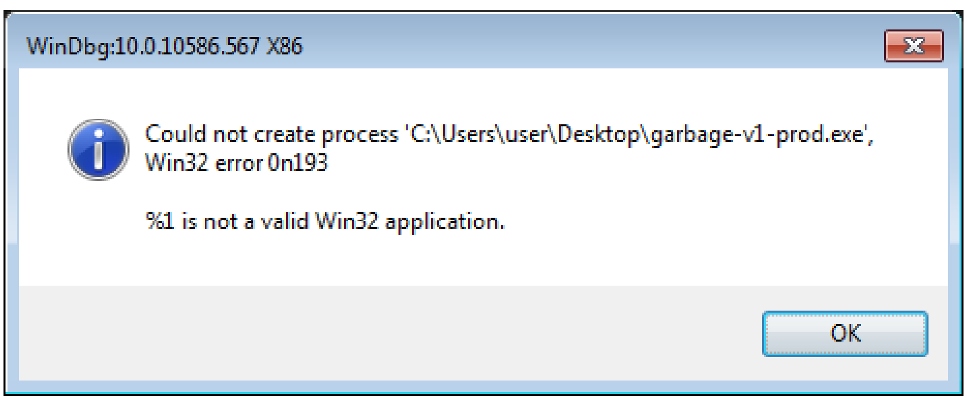


Figure 2 - invalid application error message

Looking back more closely at the strings we encountered, we noticed the following XML data at the end of the file. This data should look familiar, many applications contain an XML manifest at the end. However, this manifest is not complete, it is truncated.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>

<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>

  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">

    <securit
```

The truncated manifest is a clue that something is wrong with this file. Opening the file in a hex editor reveals that this partial XML data is at the end of the file, which means that the file itself it truncated. We can use the tool CFF explorer to examine the binary to determine what the PE file should be.

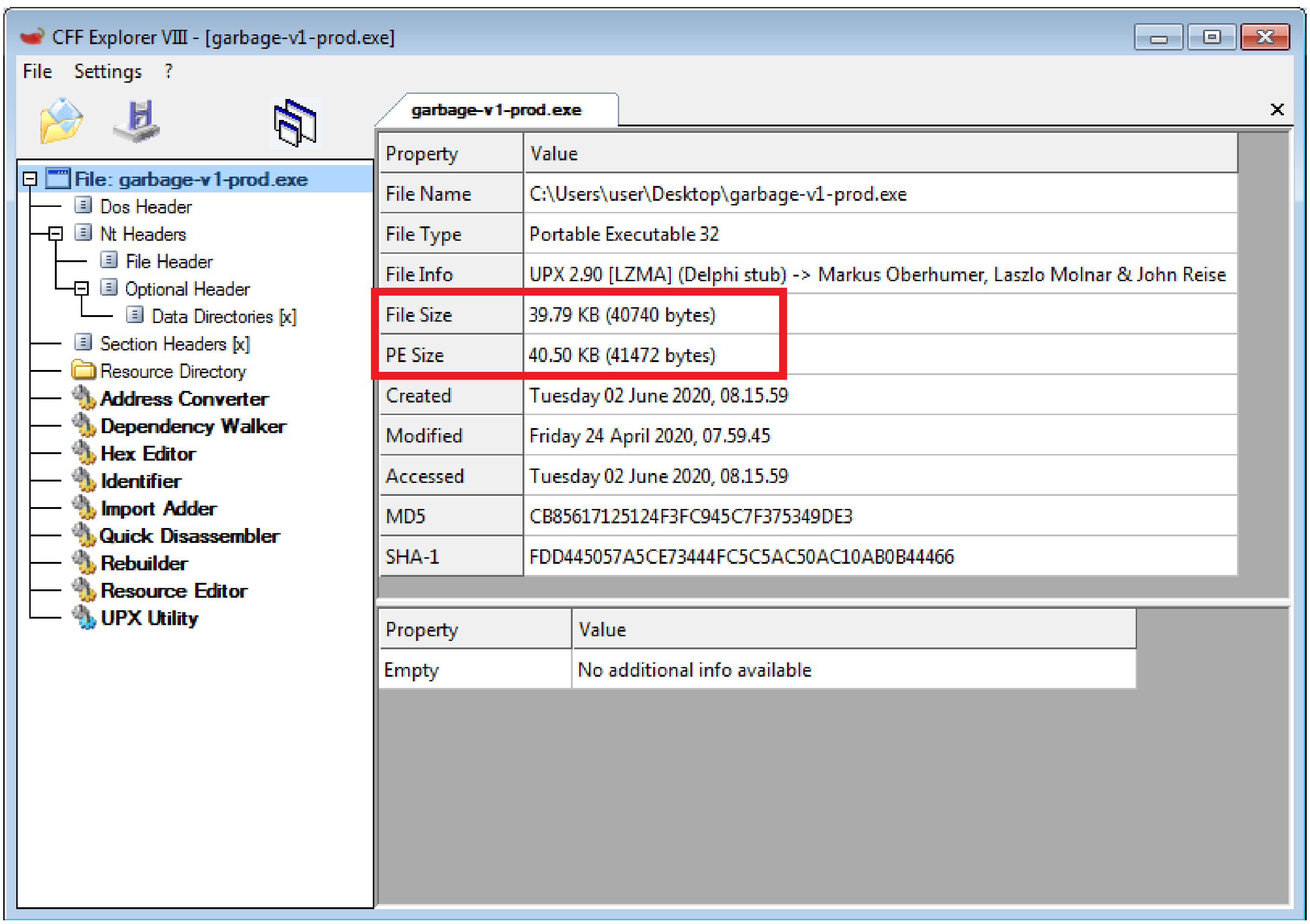


Figure 3 - CFF Explorer checking for truncation

CFF Explorer reveals that the PE size is 41,472 bytes, however the file size is only 40,740 as shown in Figure 3. Meaning the file is truncated by 732 bytes. While this is a small number of bytes missing it causes the Windows loader to fail and the UPX utility to error.

There are many solutions to this challenge. I am going to briefly walk through two solutions.

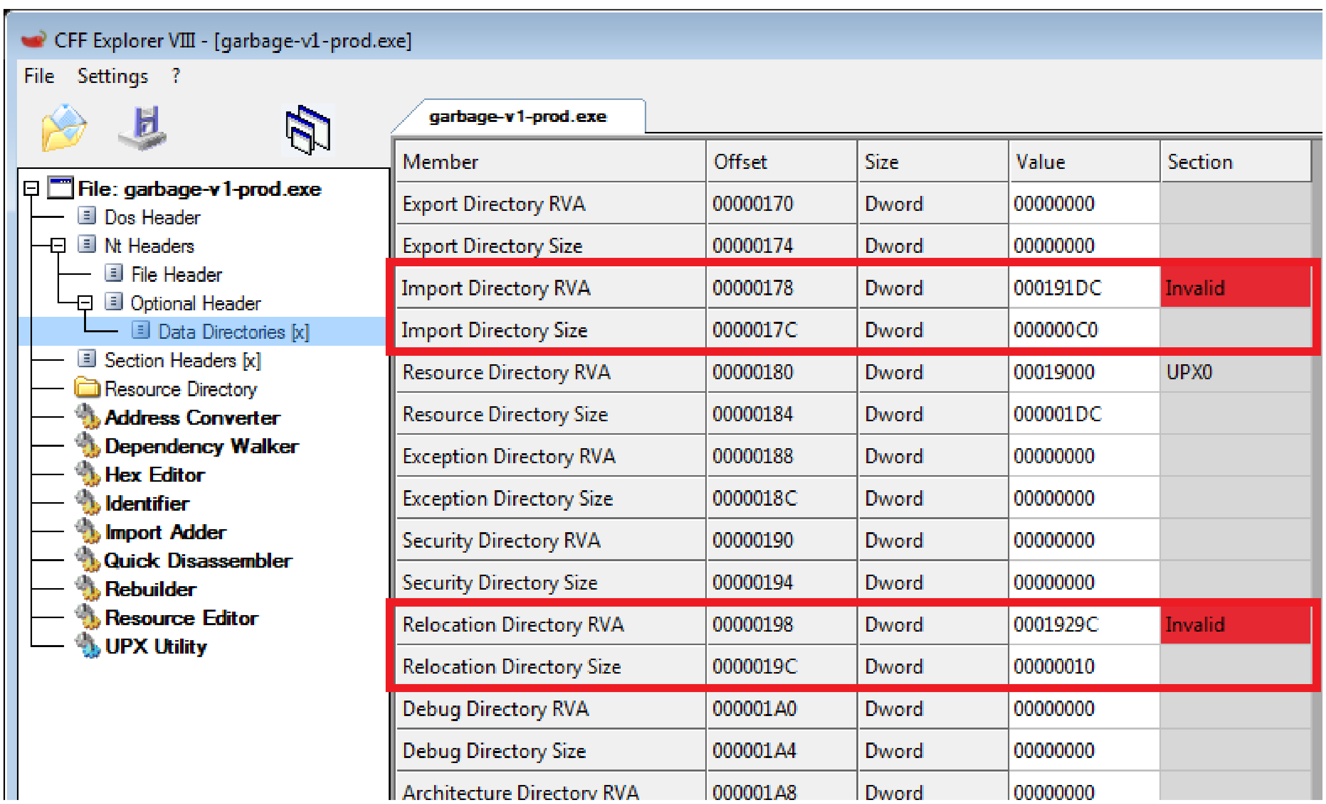
Unpacking solution #1

The first solution was pointed out to me by one of my FLARE colleagues, and is a very simple. Since the PE file has been truncated and is missing 732 bytes, we can append 732 NULL bytes to the end of the file. We can now use the standard UPX utility to unpack the padded binary. While this new unpacked file does not run, it will still allow us to perform static analysis.

Running strings on the unpacked binary does not reveal much, just two large base64 strings. Now that we have the unpacked file, we can try to determine its purpose. This will be discussed later in this paper.

Unpacking solution #2

The second solution is more compliant than the first. I am providing the details so that others can learn from it. When taking a closer look at the Data Directories of the original file in CFF Explorer we see two areas that stand out in red as shown in Figure 4.



Member	Offset	Size	Value	Section
Export Directory RVA	00000170	Dword	00000000	
Export Directory Size	00000174	Dword	00000000	
Import Directory RVA	00000178	Dword	000191DC	Invalid
Import Directory Size	0000017C	Dword	000000C0	
Resource Directory RVA	00000180	Dword	00019000	UPX0
Resource Directory Size	00000184	Dword	000001DC	
Exception Directory RVA	00000188	Dword	00000000	
Exception Directory Size	0000018C	Dword	00000000	
Security Directory RVA	00000190	Dword	00000000	
Security Directory Size	00000194	Dword	00000000	
Relocation Directory RVA	00000198	Dword	0001929C	Invalid
Relocation Directory Size	0000019C	Dword	00000010	
Debug Directory RVA	000001A0	Dword	00000000	
Debug Directory Size	000001A4	Dword	00000000	
Architecture Directory RVA	000001A8	Dword	00000000	

Figure 4 - CFF Explorer invalid directories

The reason these directories are marked in red is because the data for these directories exists within the truncated area of the file. This truncation occurs within the .rsrc section of the file. To get this file to run, we can remove the .rsrs section from the file using CFF Explorer. After doing so we also need to zero out the Import Directory, Resource Directory, and Relocation Directory from the Data Directories. Once all changes have been made, we can save our changes as a new filename.

Double clicking the resulting file results in a crash. However, it runs. At this point we can attempt to load the new file into x32dbg and examine the crash.

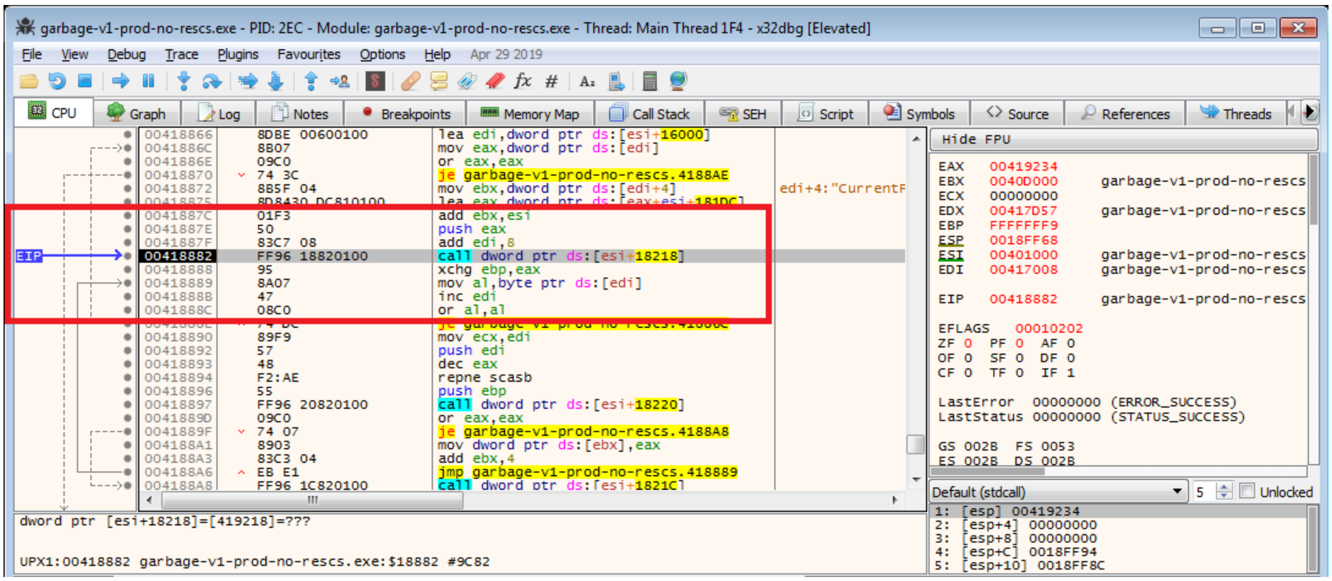


Figure 5 - x86dbg crash analysis

We can see in Figure 5 that the crash occurs when trying to call a function. Knowing that this sample was packed with UPX we can examine the source code of the UPX unpacking stub to see where this crash has occurred. (https://github.com/upx/upx/blob/d7ba31cab8ce8d95d2c10e88d2ec787ac52005ef/src/stub/src/i386-win32.pe.S#L99). Looking closely at the source reveals that the program is attempting to call LoadLibrayA. It is unfortunate that the sample crashes, but what do you expect when you remove a whole section? The great thing is that it crashes in the unpacking stub after decompression. It crashes during import resolution. Therefore, we can perform a memory dump and perform analysis on the unpacked code.

Sample analysis, the easy way

Now that we have an unpacked sample, we can start to solve this challenge. For this section we will be using the unpacked code extracted from section #1 above. The first thing to try is executing the unpacked sample. We encounter an error message te4lling us the side-by-side configuration is incorrect as shown in Figure 6 below.

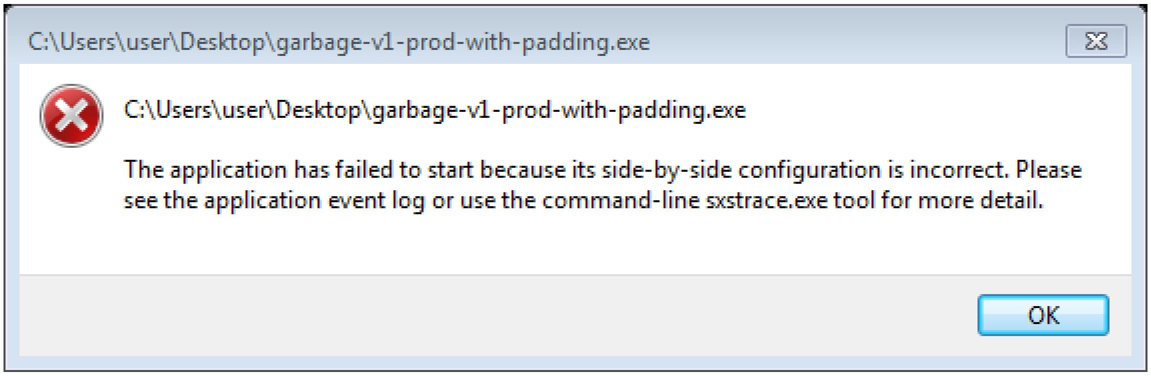


Figure 6 - Bad side-by-side configuration

This is an artifact of the truncation. Part of the side-by-side configuration is missing. To verify this, we can load the unpacked sample into CFF explorer and view its resources. Figure 7 below shows the truncated configuration. We can remove this resource by right clicking and selecting the remove resource option.

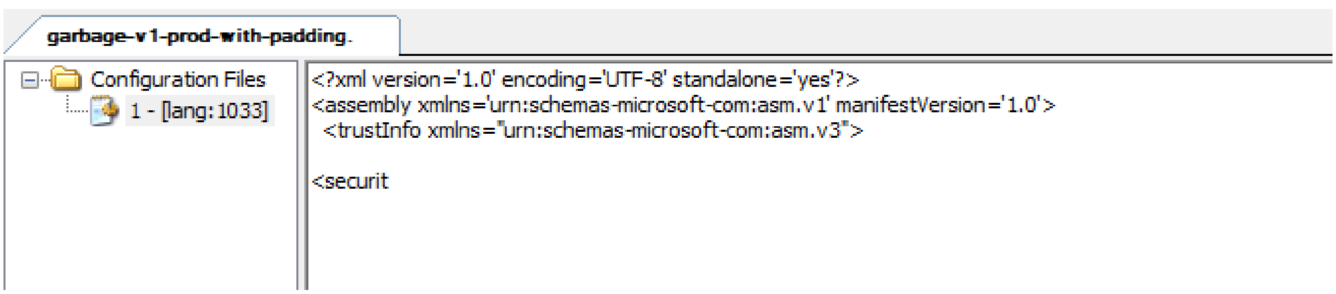


Figure 7 - Truncated resource

Saving the executable and attempting to run the sample again reveals another error message shown in Figure 8 below.

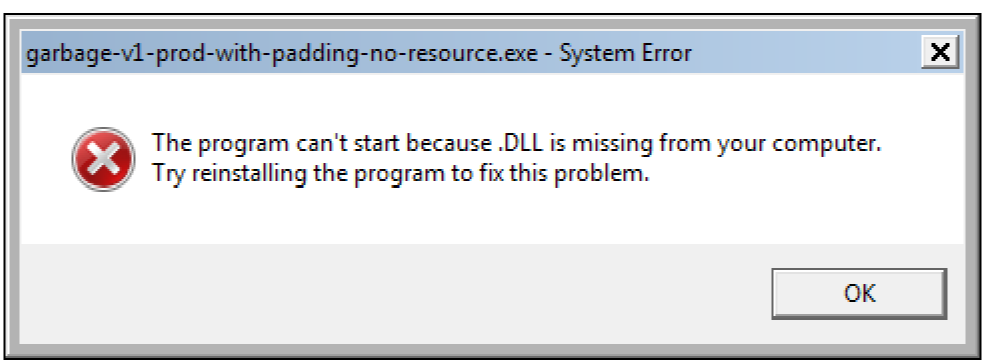


Figure 8 - Missing .DLL

Using CFF explorer again we can examine the import table. As you can see in Figure 9 below, the module names are missing. This is an artifact of the file being corrupt, even thought the upx –d command worked. We can infer the module names by looking at the functions to resolve in the lower portion of the imports view.

garbage-v1-prod-with-padding-no-resource.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
00011634	N/A	00011494	00011498	0001149C	000114A0	000114A4
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
	66	00000000	00000000	00000000	00012434	0000D000
	1	00000000	00000000	00000000	00012452	0000D10C

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
N/A	000123E4	0000	GetCurrentProcess
N/A	000123F8	0000	WriteFile
N/A	00012404	0000	TerminateProcess

Figure 9 - Missing module names

You can use CFF explorer to populate the module name fields as shown in Figure 10 below.

garbage-v1-prod-with-padding-no-resource.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
00011652	N/A	000114A8	000114AC	000114B0	000114B4	000114B8
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
kernel32.dll	66	00000000	00000000	00000000	00012434	0000D000
shell32.dll	1	00000000	00000000	00000000	00012452	0000D10C

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
N/A	00012442	0000	ShellExecuteA

Figure 10 - Fixed module names

After fixing the module names in CFF explorer and saving, we now have a working executable, which gives us the flag as shown below in Figure 11.

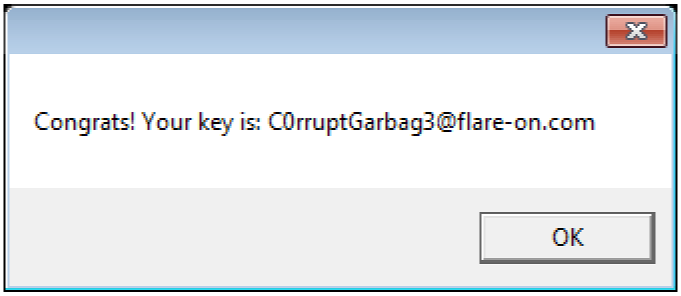


Figure 11 - The Flag

Sample analysis, the hard way

If we attempt to open the unpacked sample in IDA we quickly realize that something isn't quite right. The imports are all missing as shown below in Figure 12. The red text indicates where there would be a call to an imported function however IDA does not know which ones.

```
v12[0] = 741548835;
v12[1] = 1231306510;
strcpy(v10, "nPTnaGLkIqdcQwvieFQKGcTGOTbfMjDNmvibfBDdFBhoPaBbtfQuuGWYomtqTFqvBSKdUMmciqKSGZaosWCSoZlcIlyQpOwkcAgw ");
v12[2] = 67771914;
v12[3] = 436344355;
v12[4] = 604530244;
strcpy(v11, "KglPFOsQDxBPXmclOpmsdLDEPMRWbMDzwhDGOyqAkVMRvnBeIkpZIhFznwVylfjrkqprBPAdPuaiVoVugQAlyOQQtxBNsTdPZgDH ");
v12[5] = 745804082;
v12[6] = 255995178;
v12[7] = 224677950;
v12[8] = 387646557;
v12[9] = 84096534;
v12[10] = 134815796;
v12[11] = 237248867;
v12[12] = 1479808021;
v12[13] = 981018906;
v12[14] = 1482031104;
v13 = 84;
v14[0] = 989990456;
v14[1] = 874199833;
v14[2] = 1042484251;
v14[3] = 1108412467;
v14[4] = 1931350585;
sub_401000(v14, 20, v11, 0);
v3 = MEMORY[0x12418](v9[0], 0x40000000, 2, 0, 2, 128, 0);
sub_401045(v9);
if ( v3 != -1 )
{
  v8 = 0;
  sub_401000(v12, 61, v10, v4);
  MEMORY[0x123F8](v3, v9[0], 61, &v8, 0);
  sub_401045(v9);
  MEMORY[0x12426](v3);
  sub_401000(v14, 20, v11, v5);
  MEMORY[0x12442](0, 0, v9[0], 0, 0, 0);
  sub_401045(v9);
}
v6 = MEMORY[0x123E4](-1);
MEMORY[0x12404](v6);
```

Figure 12 - Hexrays with missing imports

At this point we could try to infer the imports based on their usage, but there is a better way. We can use the same technique as discussed in the previous section using CFF explorer to add the missing module names.

Viewing the code now in Hexrays shows a very simple program which opens and writes to a file, which it then executes using the `ShellExecute` API.

To figure out what filename and contents are used we first need to understand the decoding routine `sub_401000`. Looking at the function we can see that is performs a multi-byte xor of a buffer using a key (large Base64 strings).

Decoding the data which will be used for the `CreateFileA` API reveals the filename `sink_the_tanker.vbs`

Decoding the data which will be written to the vbs script reveals the string `MsgBox("Congrats! Your key is: C0rruptGarbag3@flare-on.com")`

This is the exact same string which was displayed in the last section after executing the sample!

Conclusion

This challenge illustrated a real issue which I have come across during my day job, analysis of a corrupt packed executable. As you can see it is still possible in certain situations to perform analysis of corrupt files. Thanks for playing!