Unpacking UPX Manually

kausrini.github.io/2021-06-20-unpacking-upx-manually/

UPX [1] is one of the most common packers used by malware authors to obfuscate their binaries. Obfuscated binaries are harder to analyze than the original binary. UPX is a packer, so it does have legitimate usage like compressing a binary for reduced file size. Not all UPX packed files are malicious but for this blogpost, we will be choosing something malicious.

UPX packed executables can be automatically unpacked by UPX tool (which available online for free). To prevent this, malware authors often tamper with the packed binary in such a way that they can't be unpacked by UPX tool but the binary unpacks itself in memory without any issues. So, learning to unpack them manually always helps. Moreover, the general principle mentioned below can be used to unpack any custom packer or obfuscation techniques used by malware authors.

I took a long time to search for a sample malicious file and in the end, chose one randomly in VirusTotal (VT) by searching for "UPX Ransomware" [2]. I used Detect It Easy (DIE) [3] tool to confirm that the binary is UPX packed.

For the purpose of this post, I'll be using packed and obfuscated interchangeably. I've renamed the downloaded binary to "sample". Using the long hash value as the binary name makes the x32dbg debug windows look cluttered as the function names are referred to as *filename.memory_address* in the debugger.

Theory Crafting

Before we proceed, we need to talk about how a packed binary is unpacked. A packed binary, when executed (starts from a point called Entry Point - EP), allocates memory space to unpack itself and then populates it with the unpacked instructions. This process of writing to the memory is unpacking.

Once unpacked, the packer will start executing instructions from the unpacked section (The starting address is called Original Entry Point - OEP). By debugging the packed executable, we can execute/debug until we can identify the OEP. Once we identify OEP, we can dump the instructions into a binary file and this is the unpacked code. This can be considered as the standard process for unpacking manually.

We already know it's packed. If we did not know that, we can use a tool like PeID [4] or Detect It Easy to check for packers. Before we start debugging, let's take a look at the packed binary in the tool - PeStudio. It gives us a quick look at the PeFile structure, strings and imported libraries. Each of these sections provides us with more context to focus on while analyzing the binary.

Pe File Structure

property	value	value	value
name	UPX0	UPX1	.rsrc
md5	n/a	24A7744756FEFA4B187566A...	820662818CDAF57E63206FC...
entropy	n/a	7.998	3.791
file-ratio (97.27%)	n/a	96.50 %	0.77 %
raw-address	0x00000400	0x00000400	0x0004EA00
raw-size (323584 bytes)	0x00000000 (0 bytes)	0x0004E600 (321024 bytes)	0x00000A00 (2560 bytes)
virtual-address	0x00401000	0x00439000	0x00488000
virtual-size (557056 bytes)	0x00038000 (229376 bytes)	0x0004F000 (323584 bytes)	0x00001000 (4096 bytes)
entry-point	-	**0x00086970**	-
characteristics	0xE0000080	0xE0000040	0xC0000040
writable	x	x	x
executable	x	x	-

Figure 1: PE File Structure

The basic unit of code within a PE file is contained within a section [5]. There are 3 sections, *UPX0*, *UPX1* and *.rsrc* in the packed binary. Sections being named as UPX is a hint to what packer might be used.

Warning: PE File section names can be anything and is not a reliable indicator of the contents within.

The section UPX0 has raw size of 0 bytes but virtual size of 0x3800 bytes. And the section UPX1 has 96.5% entropy. High entropy value indicates packed or encrypted data. In this case, the packed data in UPX1 will be unpacked into the empty space of UPX0.

Note: Sections with high entropy indicate compressed or encrypted data. Sections with 0 raw size but large virtual size might be used to write instructions dynamically and execute them during runtime.

Strings

It shows 3935 strings, but majority of it is unreadable/gibberish. The small percentage of readable strings also indicates that binary might be obfuscated. There is not much more to do here, let's move on.

Imports

PeStudio shows only 15 functions imported. The small number of library imports is another indicator of packed or obfuscated content. Malicious files might often contain obfuscate module and library names (won't show up in simple string analysis). These obfuscated libraries names, can then be deobfuscated during runtime and then loaded using LoadLibraryA api call followed by GetProcAddress to obtain the address to specific module/function within that library.

This binary has the following interesting imports

- VirtualProtect
- VirtualAlloc
- ShellExecuteW
- LoadLibraryA
- GetProcAddress

These are sufficient for the binary to unpack itself in memory and run the deobfuscated code. For the sake of this blogpost size and your time, I'll not go into details of how they are used. We can discuss them in future posts.

Identifying OEP

Let's load the binary into x32dbg. As soon as it is loaded, the execution is paused at the very beginning. As stated above, the next set of instructions are meant to unpack the original binary instructions and execute them. So, we are looking for an unconditional jump or a call instruction to a specific memory location.

You can verify that you are still in packed section of instructions by searching for the “Intermodular calls” and “String references” in the “current region”. Both these will open a new window displaying limited data. This is an indicator that you are still in packed executable region.

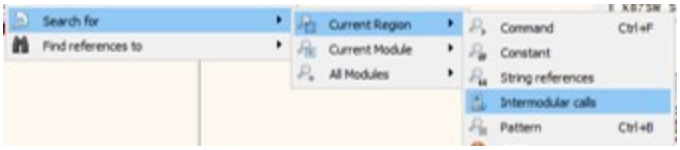


Figure 2: Search for Intermodular Calls

Address	Disassembly	Destination
77222726	mov dword ptr ds:[70020006],eax	kernelbase.75FF5656
77240E93	call ws2_32.7710A31C	ws2_32.7710A31C
772464CE	mov dword ptr ds:[77345D64],eax	apphelp.75030000
77284C96	mov esi,dword ptr ds:[773465E4]	kernel32.7561F0E0
77287A81	mov esi,dword ptr ds:[<&BaseThreadInitThunk>]	<kernel32.BaseThreadInitThunk>
7728CCD8	mov dword ptr ds:[773465E4],eax	kernel32.7561F0E0
7728CCE0	mov dword ptr ds:[773465E0],eax	kernel32.7563A680
7728CCE8	mov dword ptr ds:[<&ReleaseActCtx>],eax	<kernelbase.ReleaseActCtx>
772A999A	mov dword ptr ds:[77345D64],ebx	apphelp.75030000
772B2378	mov esi,dword ptr ds:[773465E0]	kernel32.7563A680
772B247B	mov edi,dword ptr ds:[<&ReleaseActCtx>]	<kernelbase.ReleaseActCtx>
772C44AC	mov esi,dword ptr ds:[<&BaseQueryModuleData>]	<kernel32.BaseQueryModuleData>
772CA67C	mov dword ptr ds:[77345D0C],edi	sample.004000F0
772CBC90	mov dword ptr ds:[<&BaseQueryModuleData>],ebx	<kernel32.BaseQueryModuleData>
772CC153	mov eax,dword ptr ds:[<&BaseThreadInitThunk>]	<kernel32.BaseThreadInitThunk>
772CC314	mov dword ptr ds:[77345D64],ebx	apphelp.75030000
772D1505	mov dword ptr ds:[77345D64],esi	apphelp.75030000
772D1D05	mov dword ptr ds:[77345D64],eax	apphelp.75030000
772D3E31	mov eax,dword ptr ds:[77345D0C]	sample.004000F0

*Figure 3: Intermodular calls **before** the code is unpacked*

Continue to “step over” the instructions to avoid jumping into function calls. As you continue, keep an eye on the title of the debugger. If it has ntdll.dll or some other system library, it means you are in the library code and that does not interest you (usually) as a malware analyst. You can select “Run to user code” to get back to your binary code.

After a few step instructions, you’ll notice that you are now at the very end of the binary. If you scroll further down, you’ll notice an unconditional jump right before a series of opcodes **0000** signaling the end of the binary.

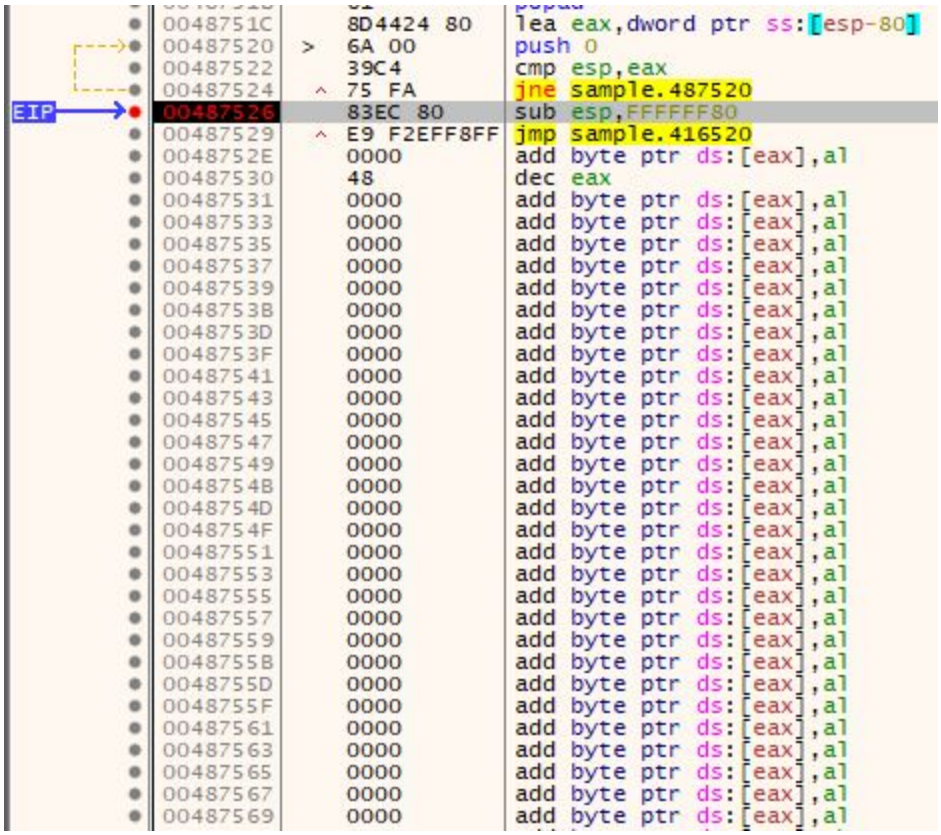
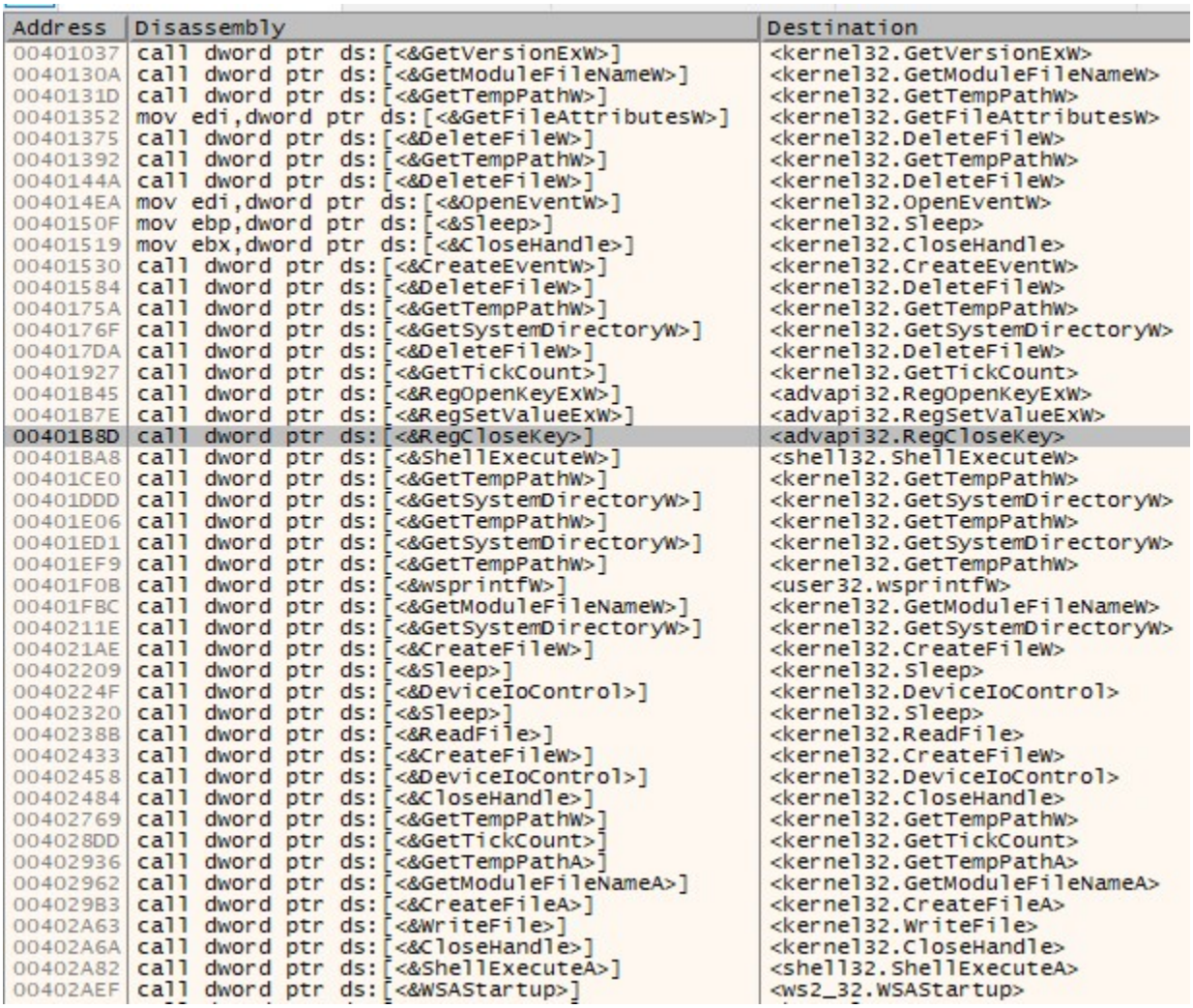


Figure 4: Likely End of the Packed Section

Set a breakpoint right before the `jmp sample.416520` instruction and check for intermodular calls again. You will still see limited number of calls indicating packed data. Now, step over this instruction, which will jump or change the instruction pointer to a new location.

This new location is the beginning of the the unpacked called or also called as OEP. the address location **0x416520 is the OEP** where the unpacked code (instructions) resides. You can (and need to) verify this by checking for intermodular calls which will show a larger number of function calls, indicating that the binary has been packed.



Address	Disassembly	Destination
00401037	call dword ptr ds:[<&GetVersionExW>]	<kernel32.GetVersionExW>
0040130A	call dword ptr ds:[<&GetModuleFileNameW>]	<kernel32.GetModuleFileNameW>
0040131D	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
00401352	mov edi,dword ptr ds:[<&GetFileAttributesW>]	<kernel32.GetFileAttributesW>
00401375	call dword ptr ds:[<&DeleteFileW>]	<kernel32.DeleteFileW>
00401392	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
0040144A	call dword ptr ds:[<&DeleteFileW>]	<kernel32.DeleteFileW>
004014EA	mov edi,dword ptr ds:[<&OpenEventW>]	<kernel32.OpenEventW>
0040150F	mov ebp,dword ptr ds:[<&Sleep>]	<kernel32.Sleep>
00401519	mov ebx,dword ptr ds:[<&CloseHandle>]	<kernel32.CloseHandle>
00401530	call dword ptr ds:[<&CreateEventW>]	<kernel32.CreateEventW>
00401584	call dword ptr ds:[<&DeleteFileW>]	<kernel32.DeleteFileW>
0040175A	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
0040176F	call dword ptr ds:[<&GetSystemDirectoryW>]	<kernel32.GetSystemDirectoryW>
004017DA	call dword ptr ds:[<&DeleteFileW>]	<kernel32.DeleteFileW>
00401927	call dword ptr ds:[<&GetTickCount>]	<kernel32.GetTickCount>
00401B45	call dword ptr ds:[<&RegOpenKeyExW>]	<advapi32.RegOpenKeyExW>
00401B7E	call dword ptr ds:[<&RegSetValueExW>]	<advapi32.RegSetValueExW>
00401B8D	call dword ptr ds:[<&RegCloseKey>]	<advapi32.RegCloseKey>
00401BA8	call dword ptr ds:[<&ShellExecuteW>]	<shell32.ShellExecuteW>
00401CE0	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
00401DDD	call dword ptr ds:[<&GetSystemDirectoryW>]	<kernel32.GetSystemDirectoryW>
00401E06	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
00401ED1	call dword ptr ds:[<&GetSystemDirectoryW>]	<kernel32.GetSystemDirectoryW>
00401EF9	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
00401F0B	call dword ptr ds:[<&wsprintfW>]	<user32.wsprintfW>
00401FBC	call dword ptr ds:[<&GetModuleFileNameW>]	<kernel32.GetModuleFileNameW>
0040211E	call dword ptr ds:[<&GetSystemDirectoryW>]	<kernel32.GetSystemDirectoryW>
004021AE	call dword ptr ds:[<&CreateFileW>]	<kernel32.CreateFileW>
00402209	call dword ptr ds:[<&Sleep>]	<kernel32.Sleep>
0040224F	call dword ptr ds:[<&DeviceIoControl>]	<kernel32.DeviceIoControl>
00402320	call dword ptr ds:[<&Sleep>]	<kernel32.Sleep>
0040238B	call dword ptr ds:[<&ReadFile>]	<kernel32.ReadFile>
00402433	call dword ptr ds:[<&CreateFileW>]	<kernel32.CreateFileW>
00402458	call dword ptr ds:[<&DeviceIoControl>]	<kernel32.DeviceIoControl>
00402484	call dword ptr ds:[<&CloseHandle>]	<kernel32.CloseHandle>
00402769	call dword ptr ds:[<&GetTempPathW>]	<kernel32.GetTempPathW>
004028DD	call dword ptr ds:[<&GetTickCount>]	<kernel32.GetTickCount>
00402936	call dword ptr ds:[<&GetTempPathA>]	<kernel32.GetTempPathA>
00402962	call dword ptr ds:[<&GetModuleFileNameA>]	<kernel32.GetModuleFileNameA>
004029B3	call dword ptr ds:[<&CreateFileA>]	<kernel32.CreateFileA>
00402A63	call dword ptr ds:[<&WriteFile>]	<kernel32.WriteFile>
00402A6A	call dword ptr ds:[<&CloseHandle>]	<kernel32.CloseHandle>
00402A82	call dword ptr ds:[<&ShellExecuteA>]	<shell32.ShellExecuteA>
00402AEF	call dword ptr ds:[<&WSAStartup>]	<ws2_32.WSAStartup>

*Figure 5: Truncated Image of Intermodular calls **after** the code is unpacked*

Note: Figure 5 has significantly greater number of function calls than Figure 3. Furthermore, the function names are indicated clearly and not obfuscated. This indicates that we have successfully unpacked the binary.

Before we proceed, make sure your Instruction Pointer is pointing to the OEP we have identified above.

Dump Unpacked Binary From Memory

Once we have the unpacked binary instructions and the instruction pointer is pointing to the OEP, we can use the “OllyDumpEx” plugin [6] for x32dbg to dump the process to a file. Make sure that the OEP is pointing to the very first instruction after the Jump instruction we previously identified. This makes sure that we are dumping only the unpacked code in memory to a file. This plugin takes care of building the PE file structure around the dumped file. There is no need to change any parameters in the OllyDumpEx window. Select Dump and you will have the unpacked binary.

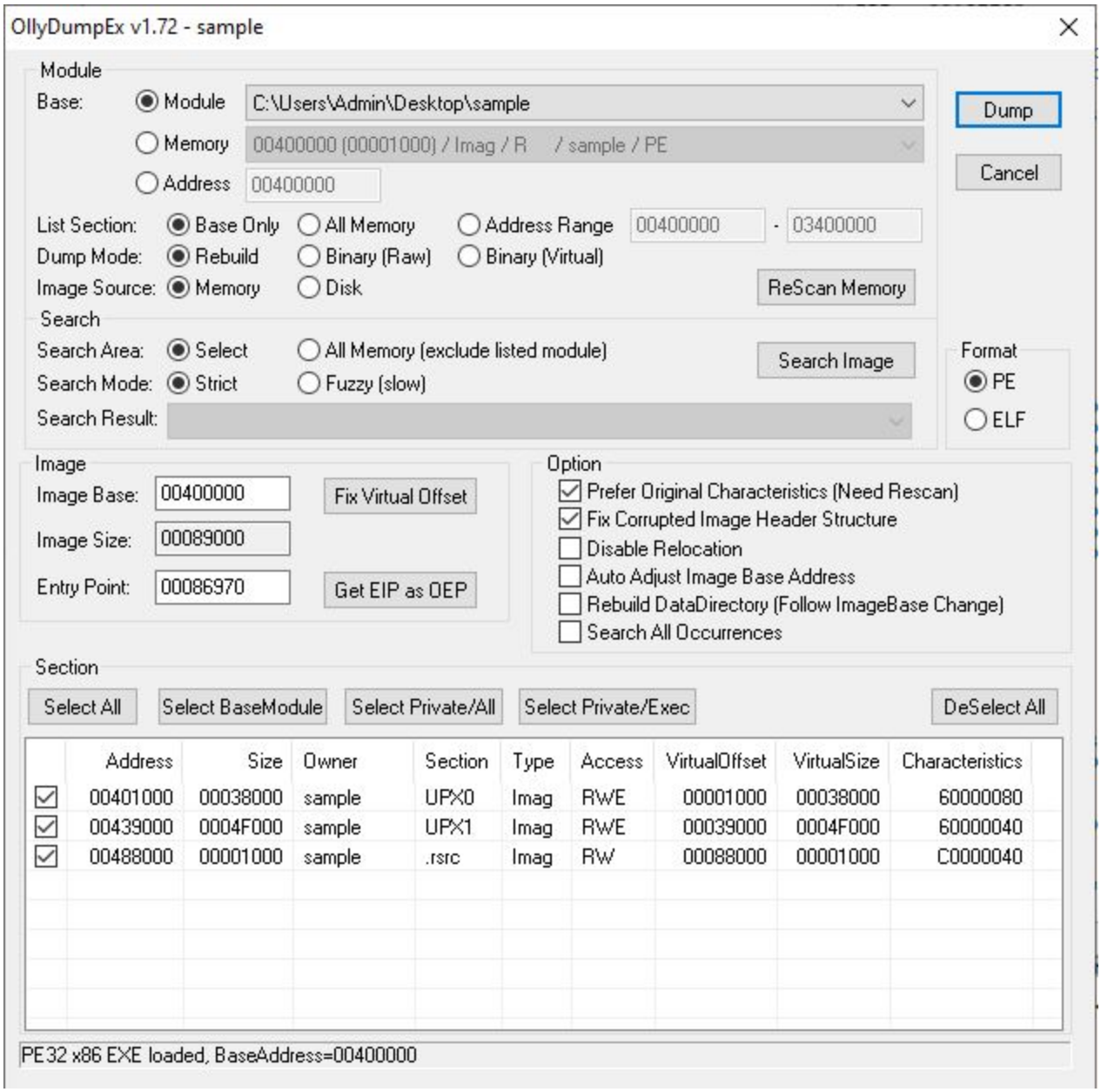


Figure 6: OllyDumpEx Plugin

The dumped file is named as “sample_dump.exe”. Opening this file in PeStudio, will show you 15 imports with no name values. This is because the OllyDumpEx does not take care of rebuilding the imports while dumping it from memory. Imports on a file and imports in memory have different address offsets and this causes the dumped binary to be ‘incomplete’ in some ways as shown in the image below.

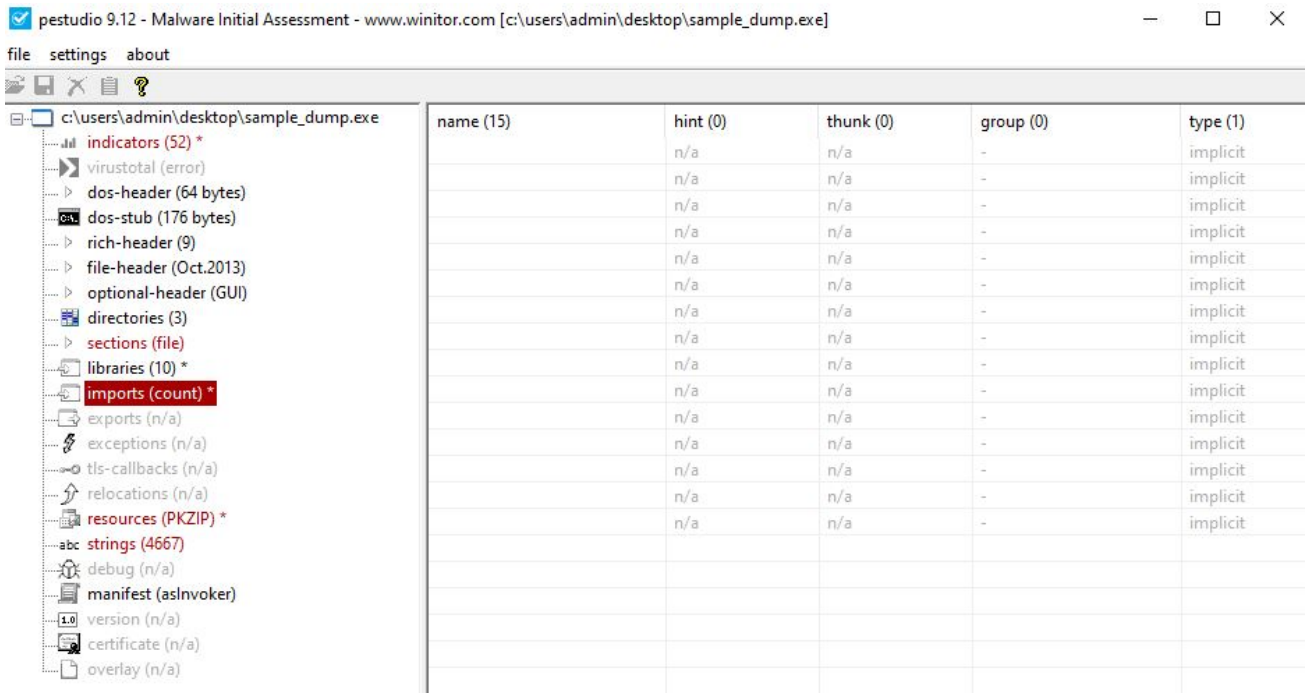


Figure 7: PeStudio Output For The Dumped File

To rebuild the import table, we will use a different plugin - Scylla x86 [7] which searches for the Import Address Table in the packed binary and obtains the imports from it. Select “IAT Autosearch” and when prompted for advanced search choose no. The plugin will return with the starting address of the IAT. Now select “Get Imports” to obtain the list of imports for the binary.

In this case, it will return with 247 valid APIs and missing 4 APIs as shown below. If there were large number of missing APIs, try the advanced search mentioned earlier and check if it returns better results.

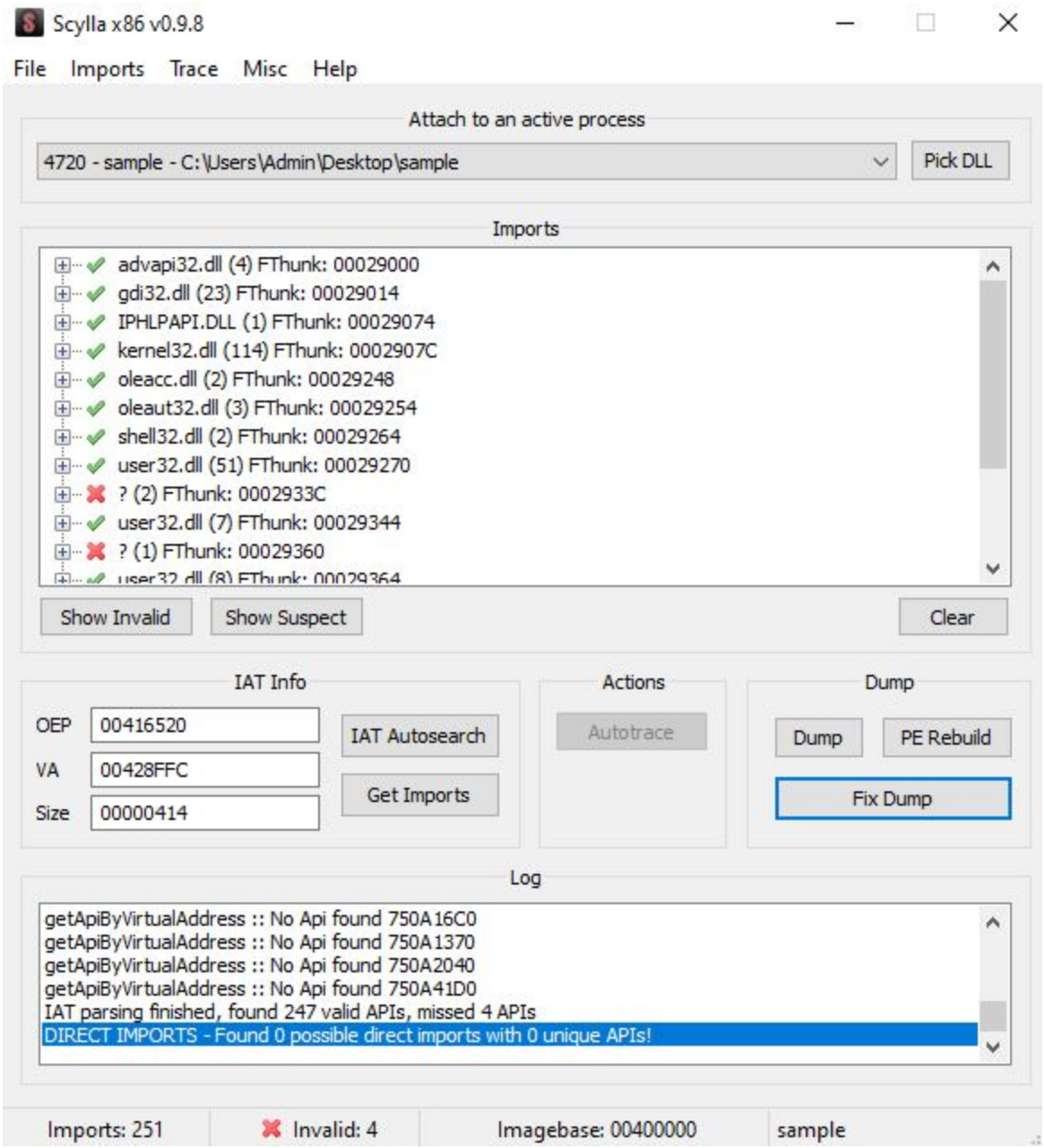


Figure 8: Scyllax86 Plugin

Once Scylla returns the above response, select “Fix Dump”, and this will generate a new file called “sample_dump_scy.exe”. As shown in the image below, opening “sample_dump_scy.exe” in the PeStudio shows that the imports are now populated.

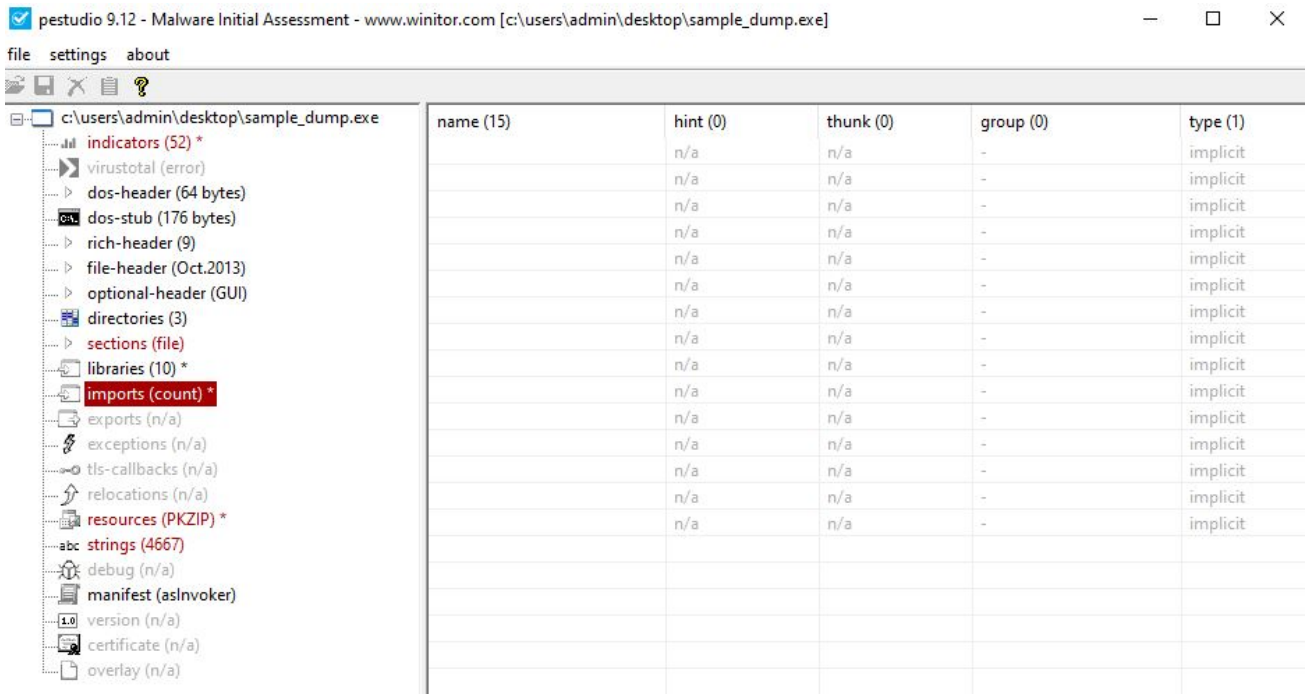


Figure 9: PeStudio Output For The Dumped File

Compare Figure 7 with Figure 9 to see how Scyllax86 helps with populating the imports. **We have successfully extracted the UPX packed binary into a file "sample_dump_scy.exe"**. We can now continue analayzing this sample for identifying its functionality and purpose. Subsequent posts will be focused on this.

References

Tags: malware analysis upx unpacking

Next Post →