

REVERSING WITH IDA PRO FROM SCRATCH

PART 16

Before continuing with more reversing, we'll do one more unpacking exercise with another target. In this case, UnPackMe_ASPack 2.2.

It belongs to the easy category. More ahead in the course, we'll continue with advanced unpacking after studying other topics.

```
0046B001
0046B001
0046B001
0046B001 public start
0046B001 start proc near
0046B001
0046B001 ; FUNCTION CHUNK AT 0046B014 SIZE 00000050 BYTES
0046B001 ; FUNCTION CHUNK AT 0046B3F5 SIZE 00000026 BYTES
0046B001
0046B001 pusha
0046B002 call    loc_46B00A

0046B00A
0046B00A loc_46B00A:
0046B00A pop    ebp
0046B00B inc    ebp
0046B00C push   ebp
0046B00D retn

-37) (191,63) 00030201 0046B001: start (Synchronized with Hex View-1)
```

There, we see the packed file EntryPoint. It starts with the PUSHAD instruction that we hadn't seen among the most used, but it just pushes every register into the stack. PUSHAD = To Push or save all registers into the stack in the following order.

60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI
----	--------	--

POPAD is the inverse operation. It pops the stack content saving it in the registers in the following order. (Except ESP that is not modified by it)

61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX
----	-------	---

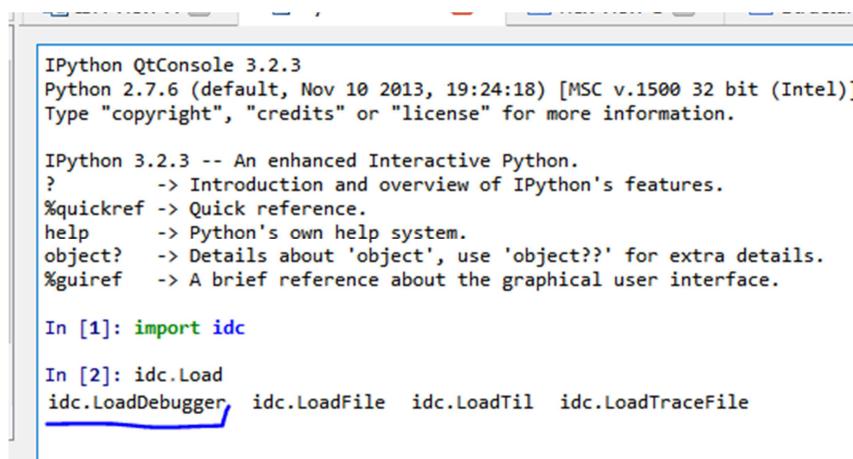
In the easy packers, most of them used PUSHAD to save the original state of the registers and POPAD to restore them before jumping to the OEP to execute the program unpacked in memory.

Thanks to this, we could get the OEP easily using the PUSHAD-POPAD method. Obviously, in modern packers, the creators knew about this and avoid using these instructions.

What is the PUSHAD-POPAD method?

First of all, we have to choose the debugger and run it. We already know how to do that in DEBUGGER - SELECT DEBUGGER and we select LOCAL WIN32 DEBUGGER.

Now, to practice, we run it through Python. You can type the instructions one at a time in the Python bar or use the IpyIDA plugin we installed and that is more comfortable. I'll do it like this.



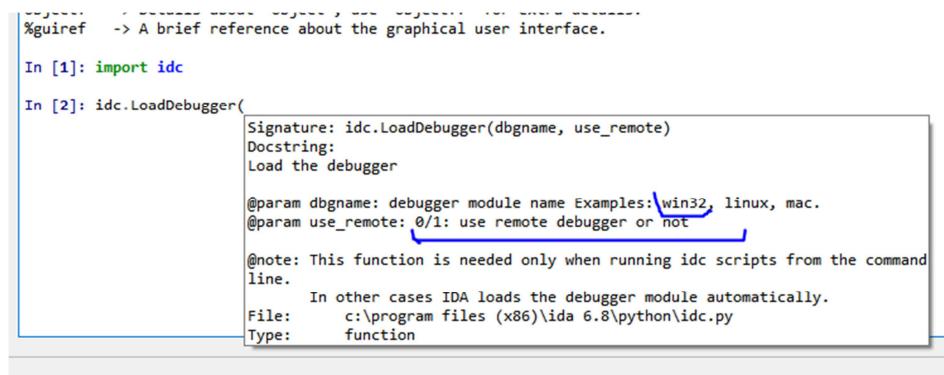
```
IPython QtConsole 3.2.3
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 3.2.3 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: import idc

In [2]: idc.Load
idc.LoadDebugger  idc.LoadFile  idc.LoadTil  idc.LoadTraceFile
```

When typing idc.Load and pressing TAB, it tells me that idc.Load exists.



```
%guiref -> A brief reference about the graphical user interface.

In [1]: import idc

In [2]: idc.LoadDebugger
Signature: idc.LoadDebugger(dbgname, use_remote)
Docstring:
Load the debugger

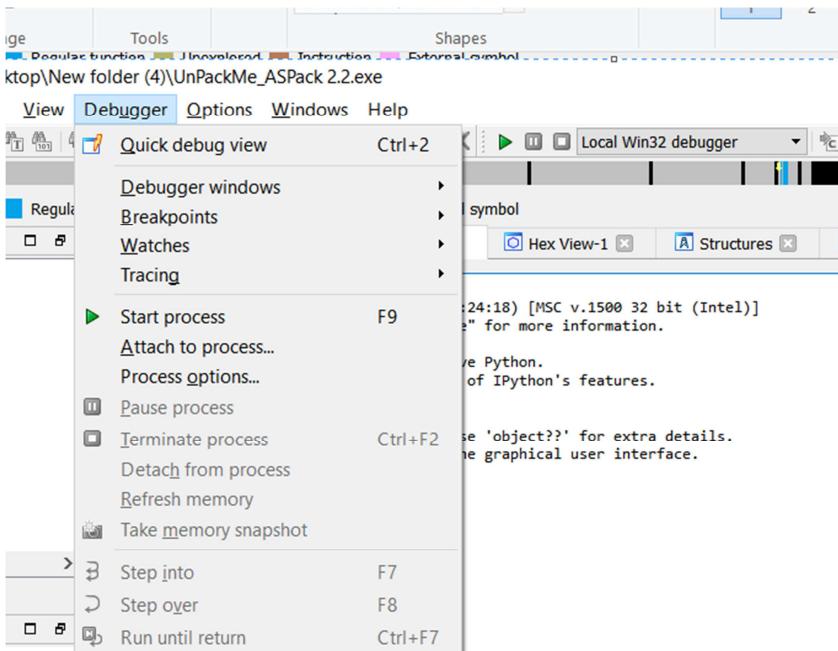
@param dbgname: debugger module name Examples: win32, linux, mac.
@param use_remote: 0/1: use remote debugger or not
@note: This function is needed only when running idc scripts from the command line.
In other cases IDA loads the debugger module automatically.
File:      c:\program files (x86)\ida 6.8\python\idc.py
Type:     function
```

In our case, we have to select win32 and 0 for local (1 is for remote debugger)

Let's try it.

```
In [1]: import idc  
In [2]: idc.LoadDebugger("win32",0)  
Out[2]: True  
In [3]: |
```

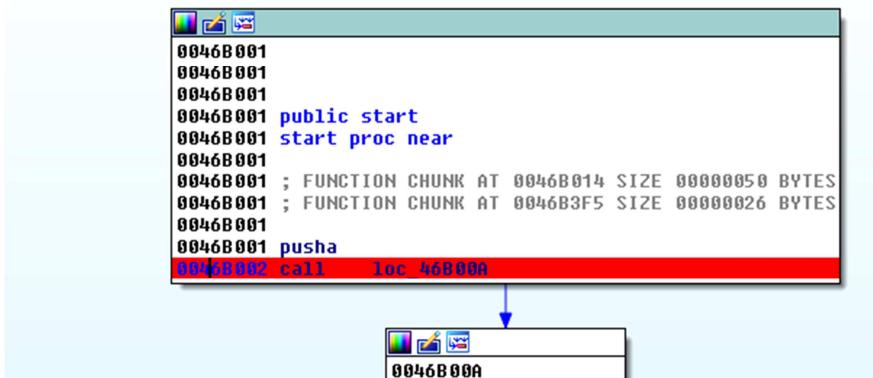
It seems it liked it. It returned TRUE.



It is already selected. If we repeat that command, it returns FALSE because is already running.

The PUSHAD method is based on executing the PUSHAD and in the next instruction, we find the registers it saved in the stack and set a breakpoint to stop the execution when it tries to recover them with POPAD just before jumping to the OEP after unpacking the original code.

Setting a breakpoint with F2 after the PUSHAD, we should stop after executing it. (PUSHA is similar to PUSHAD).



If you want to do it using Python, you can type:

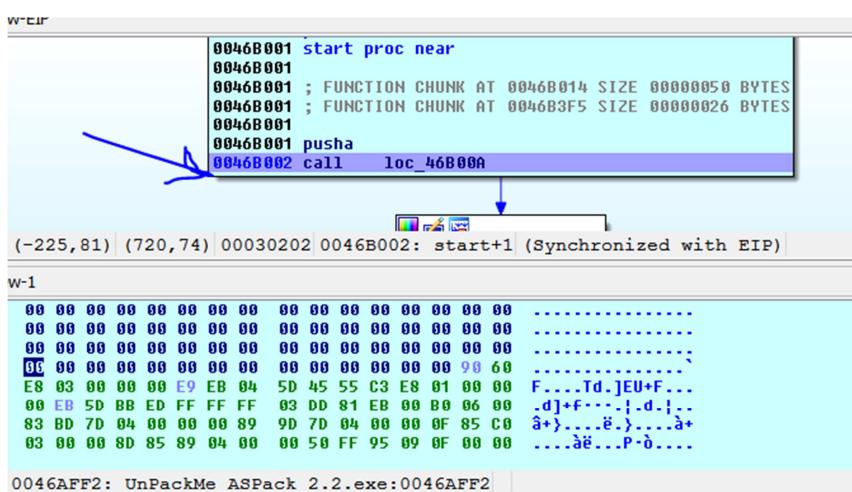
```
idaapi.add_bpt(0x46b002, 0, BPT_SOFT)
```

With that, you set a breakpoint from Python. The first argument is the address, the second one is the breakpoint size and the third one is the type. In this case, the normal breakpoint by software BPT_SOFT or 0.

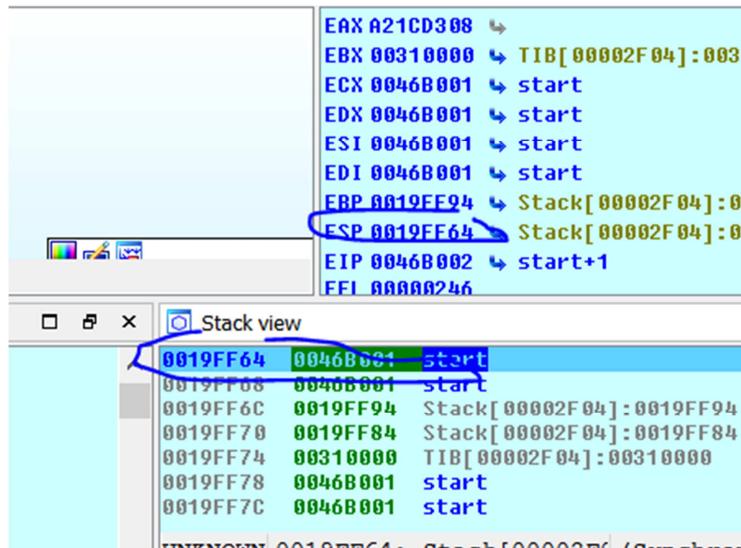
We already selected the debugger. We set the first breakpoint. Now, we have to run the debugger to stop at the breakpoint. That is easy. With F9 or from Python.

```
StartDebugger("", "", "");
```

With that command, the selected debugger will run if everything is OK and in this case, it will stop at the breakpoint we set at 0x46B002.

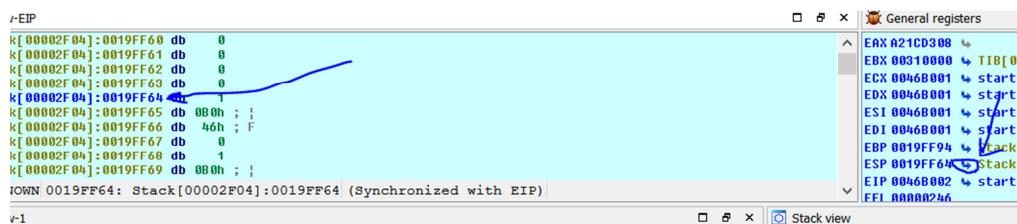


We have to see the stack here and set a breakpoint at the first line to stop there because it is the place where the register values saved by a PUSHAD are and they are recovered later with a POPAD.



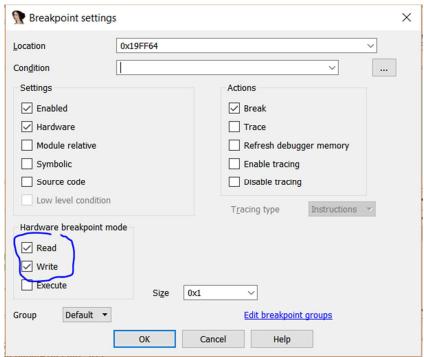
We can set a breakpoint at 0x19FF64 in my case, in yours, it is your first stack address pointed by ESP.

Now, I place the cursor on the disassembly list and press the little arrow next to ESP.



Pressing the arrow next to a register, it will try to show that address (on that window) if it exists. So we can set a breakpoint there with F2, but we have to edit it to be On Read and Write not On execution because it will stop when it recovers or reads. It won't execute any code there.

When pressing F2 it knows that and opens the breakpoint settings window.



If it doesn't appear, we should go to DEBUGGER-BREAKPOINTS-BREAKPOINT LIST.

Debug View		Structures	Enums	
Type	Location	Pass count	Hardware	Condition
Abs	0x19FF64		RW (1 byte)	
Abs	0x401000			
Abs	0x46B002			

And let's right click-EDIT to change the settings we want.

Can we set a breakpoint with Python?

```
bpttype_t type;           // type of the breakpoint:  
// Taken from the bpttype_t const definition in idd.hpp:  
// BPT_EXEC = 0,           // Execute instruction  
// BPT_WRITE = 1,          // Write access  
// BPT_RDWR = 3,           // Read/write access  
// BPT_SOFT = 4;           // Software breakpoint  
// modifiable characteristics (use update_bpt() to modify):  
int pass_count;           // how many times does the execution reach
```

idaapi.add_bpt(0x019FF64, 1, 3)

The 1 argument is the breakpoint size and 3 is the type, in this case, READ-WRITE ACCESS as we see in the table. If I type it, the same breakpoint appears we set manually.

Let's disable the previous BP in the BP list with right click-DISABLE or with Python.

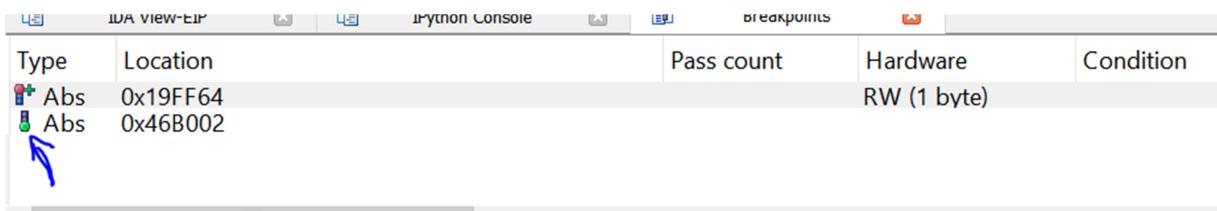
EnableBpt(0x46b002, 0)

With the second argument = 1 you enable it and with 0, you disable it.

```
In [19]: EnableBpt(0x46b002, 0)
Out[19]: True

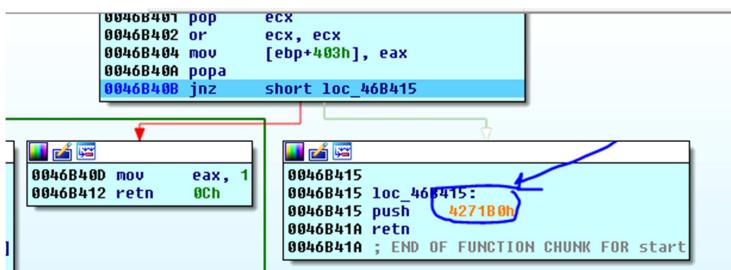
In [20]:
```

There, it is in green or disable and, of course, the one belonging to the stack is in red.

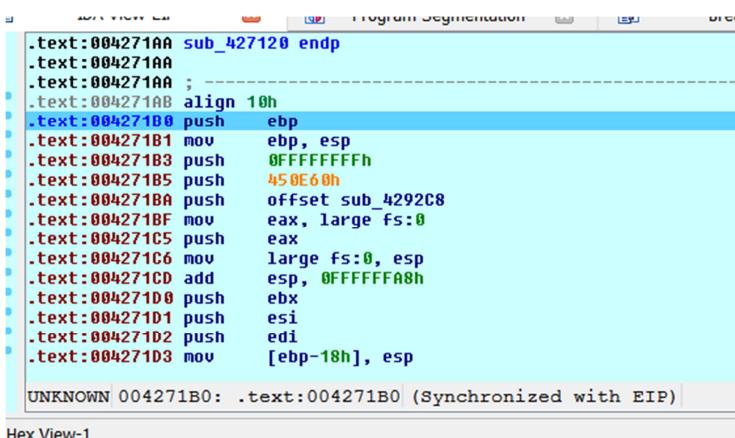


Let's continue with F9 or type in Python.

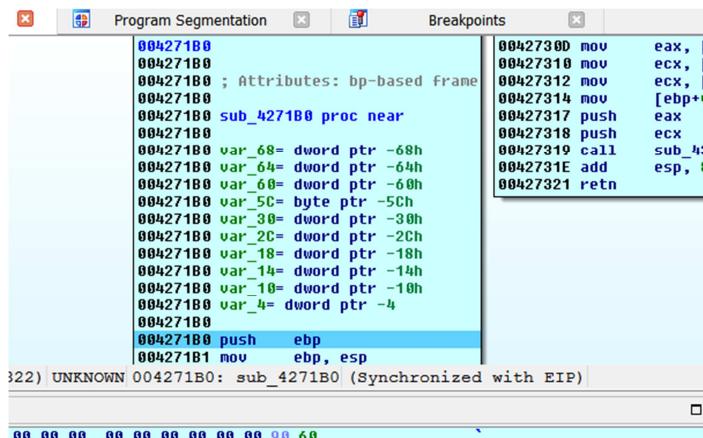
idaapi.continue_process()



It stopped just after the POPAD when it recovers the registers and from the stub, it is going to jump to the OEP at 0x4271B0 because a PUSH RET is similar to a JMP. So, let's trace a little until getting the OEP.



Now, let's reanalyze the executable, as we did in the previous case and we created the function. If we wanted just to do a snapshot of the memory to a database to study it, this would be the time, we won't repeat it here.



The next step is dumping. So, we need to find the ImageBase and the final address in the last executable segment.

In SEGMENTS, we see that the ImageBase is 0x400000 and it ends at 0x46E000.

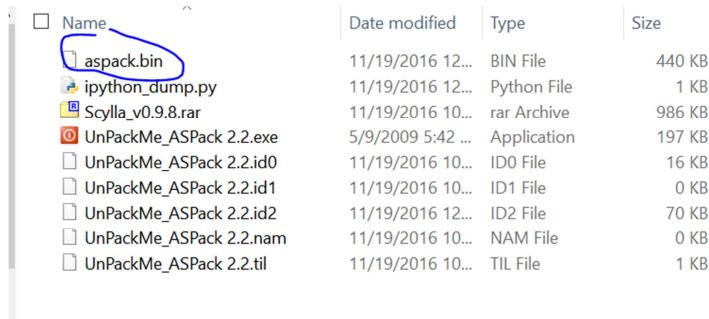
Name	Start	End	R	W	X	D	L	Align	Base	T:	
										EAX	
										EBX	
UnPackMe ASPack 2.2.exe	00400000	00401000	R	.	.	D	.	byte	0000	pi	
.text	00401000	0044B000	R	W	X	.	L	para	0001	pi	ECX
.rdata	0044B000	00457000	R	W	X	.	L	para	0002	pi	EDX
.data	00457000	00460000	R	W	X	.	L	para	0003	pi	ESI
.idata	00460000	00463000	R	W	X	.	L	para	0004	pi	EDI
UnPackMe ASPack 2.2.exe	00463000	0046B000	R	.	.	D	.	byte	0000	pi	EIP
.aspack	0046B000	0046D000	R	W	X	.	L	para	0005	pi	EBP
.adata	0046D000	0046E000	R	W	X	.	L	para	0006	pi	ESP
debug015	004A5000	004A8000	R	W	.	D	.	byte	0000	pi	EFL
debug016	004A8000	004B0000	R	W	.	D	.	byte	0000	pi	
	004B0000	004C5000	R	W	X	.	L	para	0000	pi	

Instead of using the script we used in the 15th part, we will use the version of itself in Python.

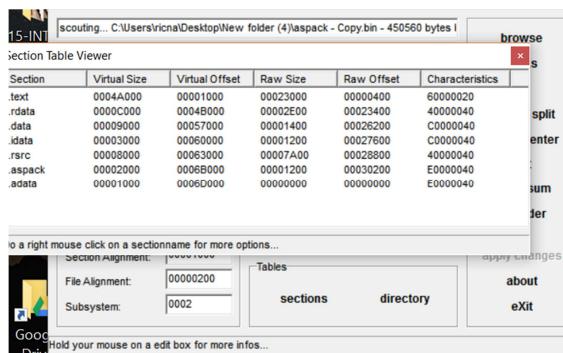
```
1 import idaapi
2 import idc
3 import struct
4
5 bin=""
6
7 file=open("aspack.bin", "wb")
8
9 for ea in range (0x400000,0x46e000,4):
10     bin+=struct.pack("<L",idc.Dword(ea))
11
12
13 file.write(bin)
14 file.close()
```

As this has many lines, I do it in a text editor and save it as **ipython_dump.py** that is included in this tutorial.

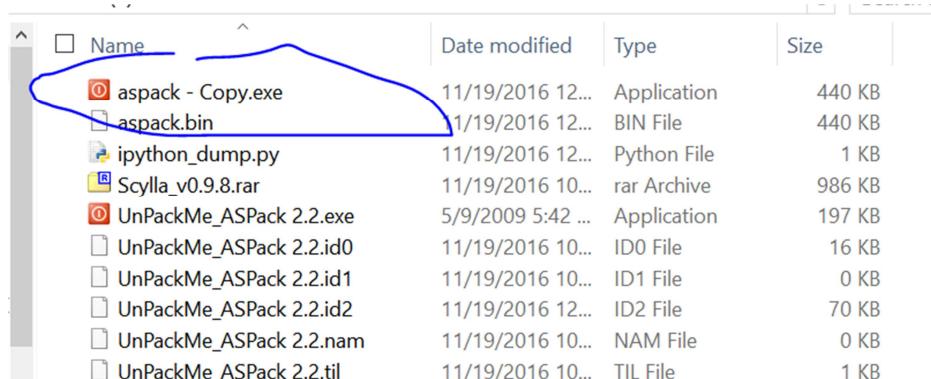
Now, with the menu FILE-SCRIPT FILE, I open it. It executes itself and creates an **aspack.bin** file. It doesn't have any icon, so I use PEEDITOR.



Name	Date modified	Type	Size
aspack.bin	11/19/2016 12...	BIN File	440 KB
ipython_dump.py	11/19/2016 12...	Python File	1 KB
Scylla_v0.9.8.rar	11/19/2016 10...	rar Archive	986 KB
UnPackMe_ASpack 2.2.exe	5/9/2009 5:42 ...	Application	197 KB
UnPackMe_ASpack 2.2.id0	11/19/2016 10...	ID0 File	16 KB
UnPackMe_ASpack 2.2.id1	11/19/2016 10...	ID1 File	0 KB
UnPackMe_ASpack 2.2.id2	11/19/2016 12...	ID2 File	70 KB
UnPackMe_ASpack 2.2.nam	11/19/2016 10...	NAM File	0 KB
UnPackMe_ASpack 2.2.til	11/19/2016 10...	TIL File	1 KB



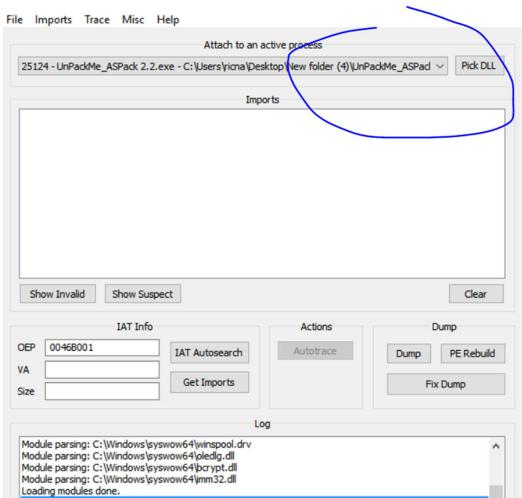
Right Click - DUMP FIXER.



Name	Date modified	Type	Size
aspack - Copy.exe	11/19/2016 12...	Application	440 KB
aspack.bin	11/19/2016 12...	BIN File	440 KB
ipython_dump.py	11/19/2016 12...	Python File	1 KB
Scylla_v0.9.8.rar	11/19/2016 10...	rar Archive	986 KB
UnPackMe_ASpack 2.2.exe	5/9/2009 5:42 ...	Application	197 KB
UnPackMe_ASpack 2.2.id0	11/19/2016 10...	ID0 File	16 KB
UnPackMe_ASpack 2.2.id1	11/19/2016 10...	ID1 File	0 KB
UnPackMe_ASpack 2.2.id2	11/19/2016 12...	ID2 File	70 KB
UnPackMe_ASpack 2.2.nam	11/19/2016 10...	NAM File	0 KB
UnPackMe_ASpack 2.2.til	11/19/2016 10...	TIL File	1 KB

When we rename it, its icon appears.

Let's open Scylla 0.98. In this part, I attached a newer version. I will find the process stopped at the OEP.

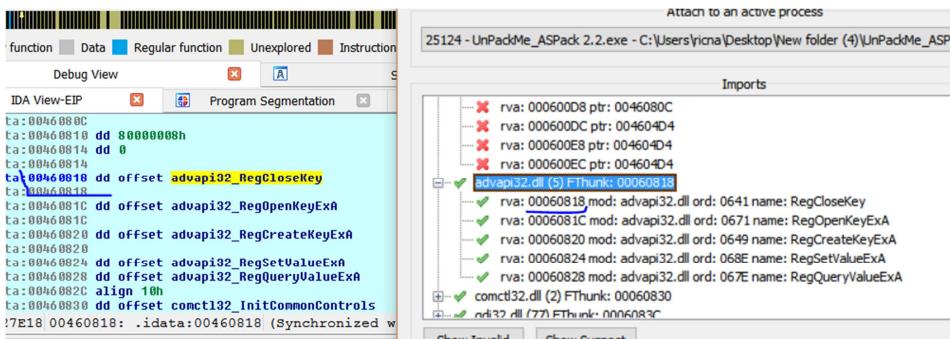


We enter the OEP that is 004271B0, press IAT AUTOSEARCH and GET IMPORTS.

If we press SHOW INVALID and select ADVANCED MODE, we see that there are some bad APIs. Let's try fixing them automatically.

No, it couldn't fix them.

Let's try it manually.

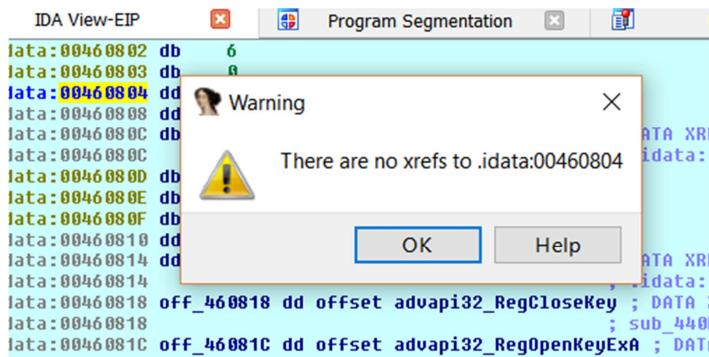


There, we see the first API at 0x460818. That is valid and matches. More above, the invalid APIs start. Let's see what there is in the first invalid API more above at 0x4600EC.

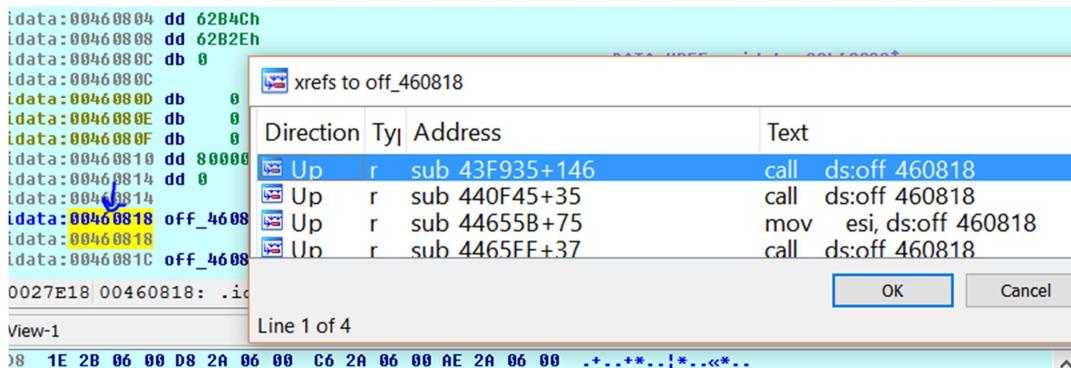
If I reorder then a little with D and then I regroup them.

```
.idata:00460802 db 6
idata:00460803 db 0
idata:00460804 dd 62B4Ch
idata:00460808 dd 62B2Eh
idata:0046080C db 0
idata:0046080C
idata:0046080C db 0 ; DATA XREF: .idata:004600C8t0
idata:0046080C ; .idata:004600D4t0 ...
idata:0046080D db 0
idata:0046080E db 0
idata:0046080F db 0
idata:00460810 dd 80000008h
idata:00460814 db 0 ; DATA XREF: .idata:004600B4t0
idata:00460814 ; .idata:004600C0t0 ...
idata:00460818 off_460818 dd offset advapi32_RegCloseKey ; DATA XREF: sub_43F935+146tr
idata:00460818 ; sub_440f45+35tr ...
idata:0046081C off_46081C dd offset advapi32_RegOpenKeyExA ; DATA XREF: sub_43F935+F9tr
00027E0C 0046080C ; .idata:0046080C (Synchronized with RTDI)
```

The content doesn't point to any valid address and if I also press CTRL+X, there are no references.

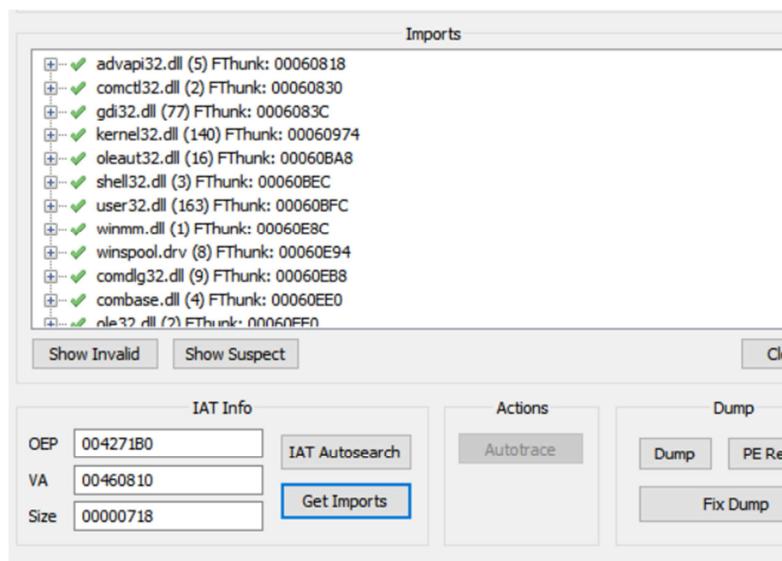


While in a real API, there will be references when the API is used. For example:



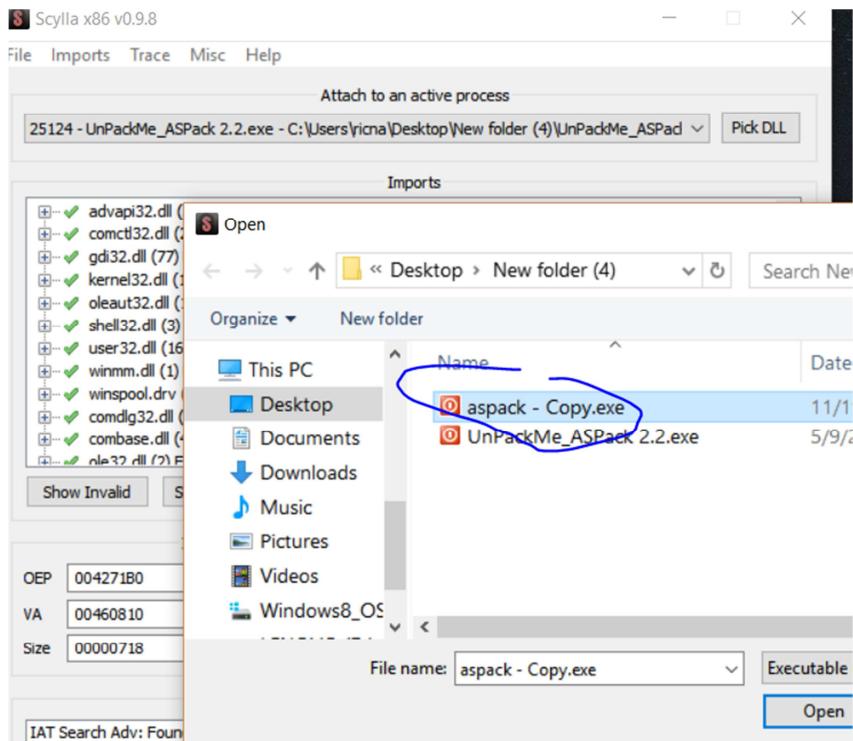
Those are no IAT APIs. We will delete them.

If I press CLEAR and IAT AUTOSEARCH again, but I tell it not to use the advanced mode, it finds them all too.

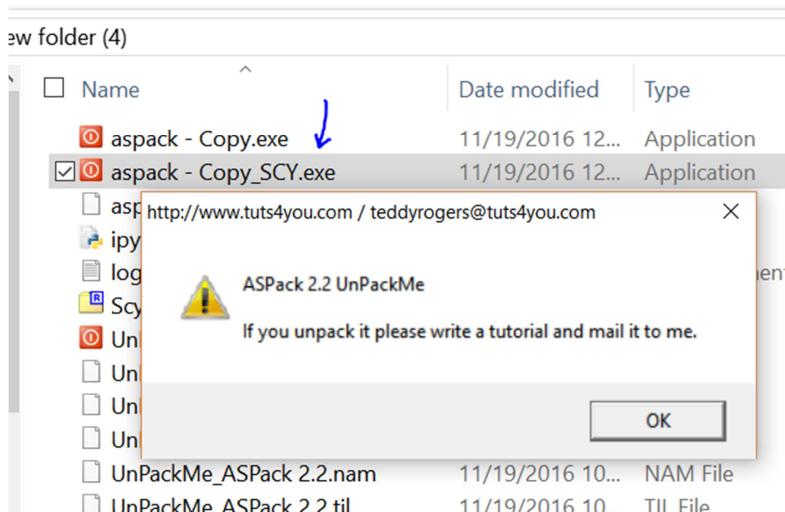


Now, the IAT starts at 0x460810. Clearing the ones that the Advanced Mode had added wrongly.

I can find the dumped file and press FIX DUMP on it.



And I can run the fixed file normally.



Ricardo Narvaja

Translated by: **@IvinsonCLS**