GAME: Arabian Nights [https://en.wikipedia.org/wiki/Arabian_Nights_(2001_video_game)]
Protection: SecuROM *new* 4.48.00.0004
Author: Luca D’Amico - V1.0 - 5 May 2022 (English version 11 March 2023)

DISCLAIMER:

You will need:
- Windows XP VM (I used VMware [https://www.vmware.com/products/workstation-player.html])
- x64dbg (x32dbg) [https://x64dbg.com/]
- CFF Explorer [https://ntcore.com/?page_id=388]
- Original game disc (you need the ORIGINAL, otherwise this will not work)

Before you start:

SecuROM protected games may not work properly on Windows versions newer than XP.
As we already experienced with SafeDisc, once we removed this DRM, the game works perfectly even on Windows 11.
This DRM works by replacing (proxying) various Windows APIs used by the game, with a function that after running some checks, will reach the requested API with a jump instruction. This jump will be absolute, without passing from the relative *IAT* thunk, so when we rebuild the imports, we will need to loop through the *IAT* to find the correct thunk, get its address, and then replace the SecuROM call in the **.text** segment with it.
There’s also an initial layer of encryption (we will need the original game disc to decrypt it) and various anti-debugging techniques that will make it harder to reach the Original Entry Point (*OEP*).

Let’s begin:

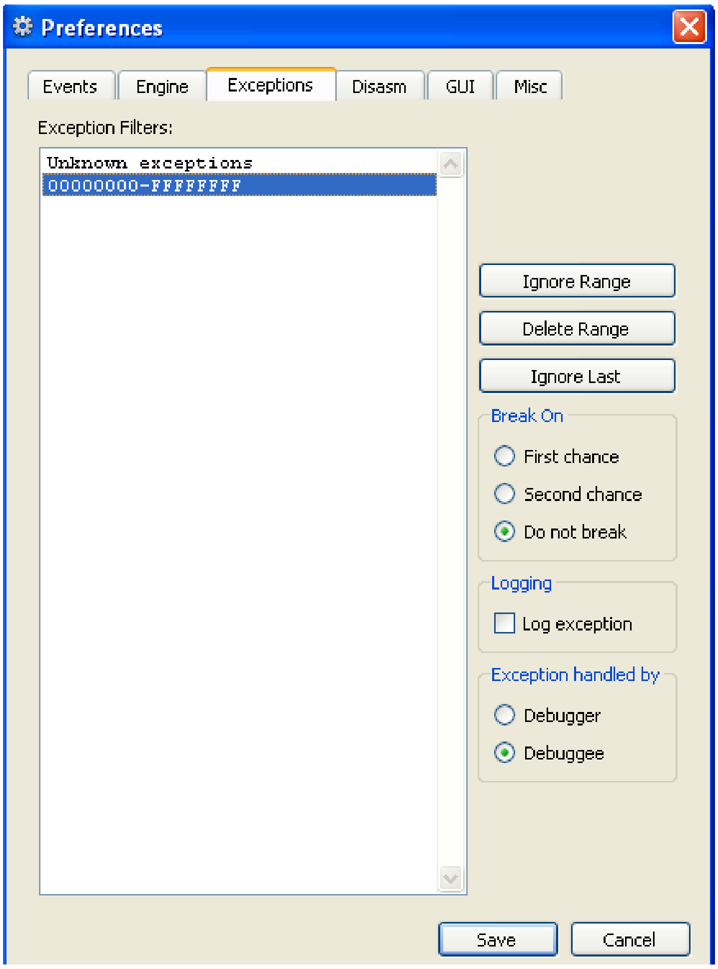
Install the game selecting the *FULL* install option. Once installed load the main executable (*_start.exe*) inside the debugger.

We can see that the entry point is located at **0x737CFD**.

If we go to the *Memory Map* tab, we can see that we are currently located in the **.cms_t** section:

00400000	00001000	_start.exe		IMG	-R---	ERWC-
00401000	00062000	".text"	Executable code	IMG	ER---	ERWC-
00463000	00003000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00466000	002C4000	".data"	Initialized data	IMG	-RWC-	ERWC-
0072A000	00002000	".idata"	Import tables	IMG	-RWC-	ERWC-
0072C000	00014000	".cms_t"		IMG	ER---	ERWC-
00740000	0002C000	".cms_d"		IMG	-RWC-	ERWC-
0076C000	00001000	".idata"	Import tables	IMG	-RW--	ERWC-
0076D000	00001000	".rsrc"	Resources	IMG	-R---	ERWC-
0076E000	00009000	".reloc"	Base relocations	IMG	-R---	ERWC-

We can suppose that the code of our game is in the **.text** segment and what we are going to execute is the SecuROM loader.
If we try to click on *RUN*, we will be constantly blocked with exceptions of various types: this is only one of the various techniques put in place to slow us down. This problem can be easily fixed by configuring the debugger to ignore all exceptions:



I also recommend unchecking *"Log exception"*, because the huge number of exceptions will extremely slow down the execution. Now we are ready to start.

As in most cases when trying to remove a protection of this type, the first step is to be able to reach the *OEP*.
My first attempt was to set a hardware breakpoint on the **.text** segment on execution: unfortunately thanks to the various anti-debugging techniques used, this operation will cause an endless loop of the SecuROM loader.

So, I decided to proceed in two steps:

1) Find out at what address the OEP is located
2) Find a way to reach said address

To get closer to the OEP, I've set a breakpoint on an API that is usually located near the Entry Point: **GetCommandLineA**.
Each time this breakpoint is triggered, we must click on *"Run to user code"* to see where the call originates from. Once we hit the breakpoint for the 3rd time, we are finally located inside the **.text** segment:

```
EIP  0044DD33  83C4 04            add esp,4
     0044DD36  85C0               test eax,eax
     0044DD38  75 0A              jne _start.44DD44
     0044DD3A  6A 1C              push 1C
     0044DD3C  E8 FF000000        call _start.44DE40
     0044DD41  83C4 04            add esp,4
     0044DD44  C745 FC 00000000   mov dword ptr ss:[ebp-4],0
     0044DD4B  E8 20780000        call _start.455870
     0044DD50  FF15 00137400      call dword ptr ds:[741300]
EIP  0044DD56  A3 18917200        mov dword ptr ds:[729118],eax
     0044DD5B  E8 F0780000        call _start.455650
     0044DD60  A3 30CE4700        mov dword ptr ds:[47CE30],eax
     0044DD65  E8 D6730000        call _start.455140
     0044DD6A  E8 81720000        call _start.454FF0
     0044DD6F  E8 5C1E0000        call _start.44FBD0
     0044DD74  C745 D0 00000000   mov dword ptr ss:[ebp-30],0
```

If you look closer, you will realize that the **GetCommandLineA** call was originated from **call dword ptr ds:[741300]**.
This is quite interesting, but for the moment let's focus on our current target (reaching the OEP).
Since we assumed that the **GetCommandLineA** API is in the function where the OEP resides, we can scroll up a little bit till the start of the current function, and we will finally be at our destination:

```
0044DCAB  CC                 int3
0044DCAC  CC                 int3
0044DCAD  CC                 int3
0044DCAE  CC                 int3
0044DCAF  CC                 int3
0044DCB0  55                 push ebp
0044DCB1  8BEC               mov ebp,esp
0044DCB3  6A FF              push FFFFFFFF
0044DCB5  68 70454600        push _start.464570
0044DCBA  68 D4AE4400        push _start.44AED4
0044DCBF  64:A1 00000000     mov eax,dword ptr fs:[0]
0044DCC5  50                 push eax
0044DCC6  64:8925 00000000   mov dword ptr fs:[0],esp
0044DCCD  83C4 A4            add esp,FFFFFFA4
0044DCD0  53                 push ebx
0044DCD1  56                 push esi
```

Perfect, now we know that **0x44DCB0** is the *OEP*. We need a way to halt the code execution right there.

<u>PAY ATTENTION:</u> it's mandatory to break the *OEP* when dumping from memory! Otherwise, the resulting executable will not work, as it will contain data related to the current execution.

If we try to set a breakpoint at that address, after restarting the debugger, the SecuROM loader will detect it and will cause and endless loop. No matter if the breakpoint is hardware-based: it will be still detected.
If we restart the debugger once more and we go to the address where the *OEP* resides BEFORE running the SecuROM loader, we will notice a very interesting thing:

```
0044DCAB  1863 31            sbb byte ptr ds:[ebx+31],ah
0044DCAE  E5 AB              in eax,AB
0044DCB0  6A E6              push FFFFFFE6
0044DCB2  46                 inc esi
0044DCB3  72 D7              jb _start.44DC8C
0044DCB5  2B30               sub esi,dword ptr ds:[eax]
0044DCB7  E5 28              in eax,28
0044DCB9  B8 6B49CC35        mov eax,35CC496B
0044DCBE  FE                 ???
0044DCBF  BF F040CFBA        mov edi,BACF40F0
0044DCC4  3A65 4E            cmp ah,byte ptr ss:[ebp+4E]
0044DCC7  C8 44D3 20         enter D344,20
0044DCCB  92                 xchg edx,eax
```

This code is completely different from what we expected to see! It is reasonable to think that this is encrypted memory and that it will be overwritten by the SecuROM loader during the boot phase.

On Windows, processes can modify memory thanks to the **WriteProcessMemory** API, which has the following signature:

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

Great, let's set a breakpoint on **WriteProcessMemory** and restart the debugger!
The 3rd hit is the correct one: we are sure about this since we see from the stack that the *lpBaseAddress* (the address where the data will be written) is right next to our *OEP*:

```
0012F760  0072F6E5  return to _start.0
0012F764  FFFFFFFF  return to FFFFFFFF
0012F768  0044DBB0  _start.0044DBB0
0012F76C  00CE8050
0012F770  00000200
```

At **0xCE8050** there is the buffer that will be written. Right click on it and select *"Follow DWORD in disassembler"*:

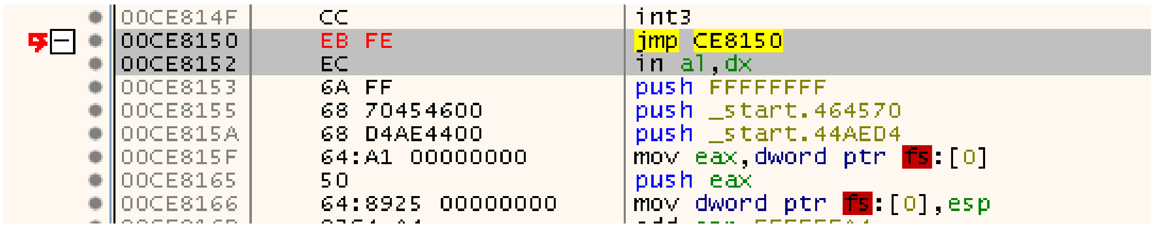
```
00CE8050  8CE1       mov ecx,fs
00CE8052  46         inc esi
00CE8053  0033       add byte ptr ds:[ebx],dh
00CE8055  D266 8B    shl byte ptr ds:[esi-75],cl
00CE8058  14 41      adc al,41
00CE805A  83E2 08    and edx,8
00CE805D  8955 EC    mov dword ptr ss:[ebp-14],edx
00CE8060  837D EC 00 cmp dword ptr ss:[ebp-14],0
00CE8064  74 0B      je CE8071
00CE8066  8B45 08    mov eax,dword ptr ss:[ebp+8]
```

We are exactly inside the buffer that will be written replacing the memory starting from **0x44DBB0**. Finding our *OEP* inside the buffer at this point is easy: we can add to the current address (**0xCE8050**) the difference between **0x44DCB0** (address of the *OEP*, found previously) and **0x44DBB0** (base address where the buffer will be written to): **0xCE8150**.

Indeed, finally at this address we find our *OEP* inside the buffer:

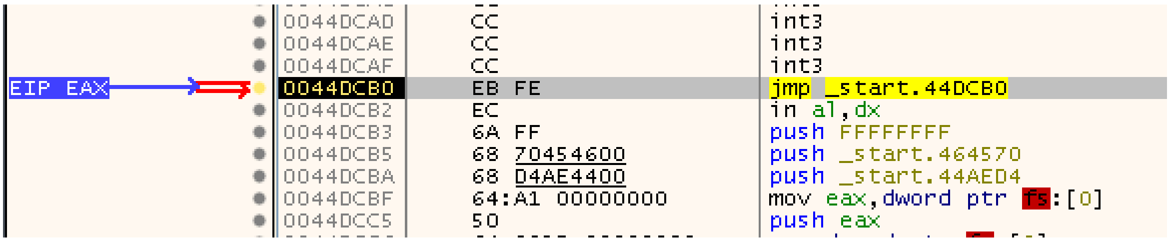
```
00CE814E  CC                  int3
00CE814F  CC                  int3
00CE8150  55                  push ebp
00CE8151  8BEC                mov ebp,esp
00CE8153  6A FF               push FFFFFFFF
00CE8155  68 70454600         push _start.464570
00CE815A  68 D4AE4400         push _start.44AED4
00CE815F  64:A1 00000000      mov eax,dword ptr fs:[0]
00CE8165  50                  push eax
00CE8166  64:8925 00000000    mov dword ptr fs:[0],esp
00CE816D  83C4 A4             add esp,FFFFFFA4
00CE8170  53                  push ebx
00CE8171  56                  push esi
00CE8172  57                  push edi
```

Great! To halt the execution at the *OEP*, we can modify the buffer, replacing the first two bytes with **EBFE** (infinite loop). In this way the buffer will be written with our patch and once the execution flow reaches the *OEP*, it will get stuck right there in our infinite loop. All that remains at that point is to set a breakpoint to block the execution and restore the original bytes. Let's proceed by patching the following bytes:

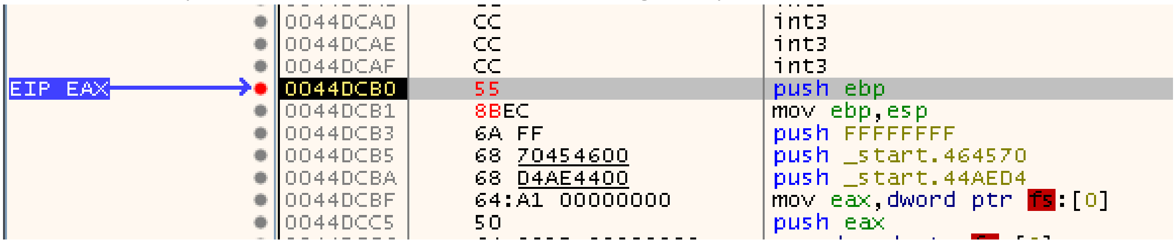


Just click on RUN and let the SecuROM loader complete its job. We can remove the breakpoint on **WriteProcessMemory** before we run our target executable again, as we don't need it any longer.

Wait a few seconds and then click on PAUSE. We will end up in a loop exactly at the OEP:



Let's set a breakpoint at 0x44DCB0 and restore the original opcodes (558B):

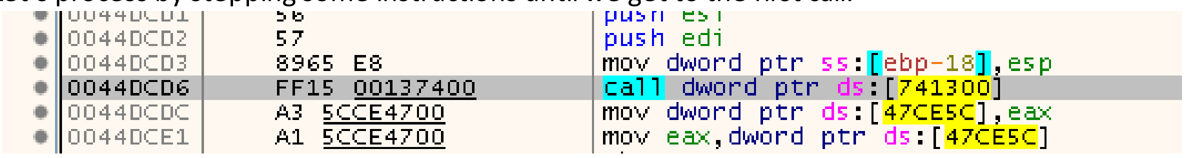


Perfect, the execution flow is currently stuck at the *OEP* ... just like we wanted!

If we now try to dump the binary with Scylla, we will obtain an executable that crashes at the first call (theoretically a **GetVersion**). We need to figure out what SecuROM did to the APIs used by the game (remember: we already found a suspicious call when we used a breakpoint on **GetCommandLineA** earlier).

If you are using a VM, I suggest creating a snapshot of the current state, so after understanding what happens to the APIs, we can quickly restore everything back to the *OEP* without having to start over (a special thanks goes to Antelox for recommending me this technique).

Let's process by stepping some instructions until we get to the first call:



It would have been reasonable to expect a call to **GetVersion**, but instead we are in front of a call that takes us to a function located in the **.cms_t** segment.
Let's click on step into to enter in this function to study what's happening here.

We realize that we are dealing with a particular function, there are also some calls to **timeGetTime** probably with the aim of detecting if we are spending time stepping between instructions with the debugger.

If we scroll down, we notice that before the classic *RET*, there is a very suspicious instruction, a **jmp eax**:

```
00730271   5E      pop esi
00730272   5B      pop ebx
00730273   8BE5    mov esp,ebp
00730275   5D      pop ebp
00730276 ^ FFE0    jmp eax
00730278   5F      pop edi
00730279   5E      pop esi
0073027A   5B      pop ebx
0073027B   8BE5    mov esp,ebp
```

This already gives us some clues about what is going on here (especially if you have read my technical paper about Laserlock!) 😊
We put a breakpoint on the jump and press RUN.
Once we hit the Breakpoint, we check the **EAX** register:

```
Hide FPU
EAX  7C81126A  <kernel32.GetVersion>
EBX  7FFD7000  &L"=::=::\\"
ECX  00144D30
EDX  7C98B140  ntdll.7C98B140
EBP  0012F7AC
ESP  0012F730
```

Here it is, the call we expected! It will be reached thanks to that jump!
We can run a second test by looking for another call at **0x00741300** immediately after the *OEP*. If you remember, while we were looking for the *OEP* we had set a breakpoint on **GetCommandLineA** and the call where execution was blocked originated from **0x0044DD50**. At that address we have a call dword ptr ds:[**741300**]:

```
0044DD44   C745 FC 00000000   mov dword ptr ss:[ebp-4],0
0044DD4B   E8 207B0000        call _start.455870
0044DD50   FF15 00137400      call dword ptr ds:[741300]
0044DD56   A3 18917200        mov dword ptr ds:[729118],eax
0044DD5B   E8 F0780000        call _start.455650
```

Let's continue the execution and follow this call. We will obviously come back to the function we have just analysed and in the end, we will find ourselves stuck again on the **jmp eax** (if you have kept the breakpoint active). Now let's check the **EAX** register:

```
Hide FPU
EAX  7C812FAD  <kernel32.GetCommandLineA>
EBX  7FFD7000  &L"=::=::\\"
ECX  7FFD7000  &L"=::=::\\"
```

Here is the reference to the **GetCommandLineA** API.

Now we no longer have any doubts: SecuROM has replaced the APIs used by the game with its own function (located in the **.cms_t** segment) which, based on the source address of the call, calculates the requested API and reaches it via a jump.

For those who want to understand in detail how the correct APIs are retrieved, you can study the disassembled part enclosed within the *CriticalSection* of that function, paying attention to the various precautions used to make debugging more complicated (such as **timeGetTime**).

At this point it would be reasonable to think of writing a few assembly lines to loop through the entire **.text** segment looking for the SecuROM calls, and after having called them to retrieve the right APIs, patch them with the correct addresses just obtained.
However, there is a big problem: the address stored in **EAX** at the moment of the jump does not pass from the relative thunk in the *IAT*, but instead it is an absolute jump to the requested function! We cannot replace it overwriting the one in the SecuROM call, because it will change (due to being dynamic).

We need the address of the corresponding thunk in the *IAT*.

So, this is the idea:

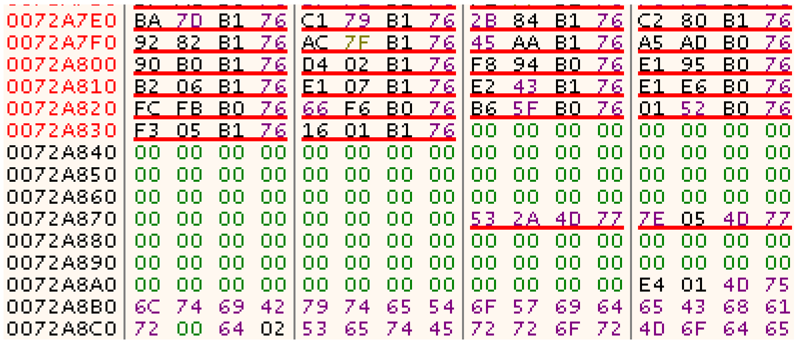
1) Let's loop through the **.text** segment looking for the SecuROM calls
2) As soon as we find a SecuROM call we jump into it
3) We hook the **jmp eax** instruction, with a jump to get back to our assembly code (getting the direct address of the API we are resolving)
4) Using the direct address of the API (stored in **EAX**), we loop through the *IAT* looking for the corresponding thunk
5) Once found it, we patch the function in the **.text** segment to call the address of the thunk, defeating SecuROM.

Finding the starting address of the *IAT* is an extremely easy operation: we can go to the relevant section in the **Memory Map** tab, select the *.idata* segment and click on *"Follow in dump"*. Now we will find the first occurrences of addresses to the various APIs used in our target:

```
0072A4B0 | A3 CC 6F 73 | 1B CB 6F 73 | 00 00 00 00 | 00 00 00 00
0072A4C0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A4D0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A4E0 | 00 00 00 00 | 26 B1 21 72 | 00 00 00 00 | 00 00 00 00
0072A4F0 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A500 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A510 | 00 00 00 00 | 3B 47 E8 73 | 00 00 00 00 | 00 00 00 00
0072A520 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A530 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A540 | 00 00 00 00 | C1 61 E4 77 | 69 5A E4 77 | 00 00 00 00
0072A550 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A560 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0072A570 | 00 00 00 00 | 00 00 00 00 | BD FD 80 7C | FA CA 81 7C
0072A580 | EE 94 80 7C | BF FC 80 7C | 28 1A 80 7C | 6B 23 80 7C
0072A590 | E2 10 83 7C | 12 FF 80 7C | AD 2F 81 7C | F5 60 83 7C
```

The *IAT* starts at **0x72A4B0** (**0x736FCCA3** is the address of **DirectDrawCreate** in this case). As we can see this *IAT* is a bit peculiar: it is full of free spaces.

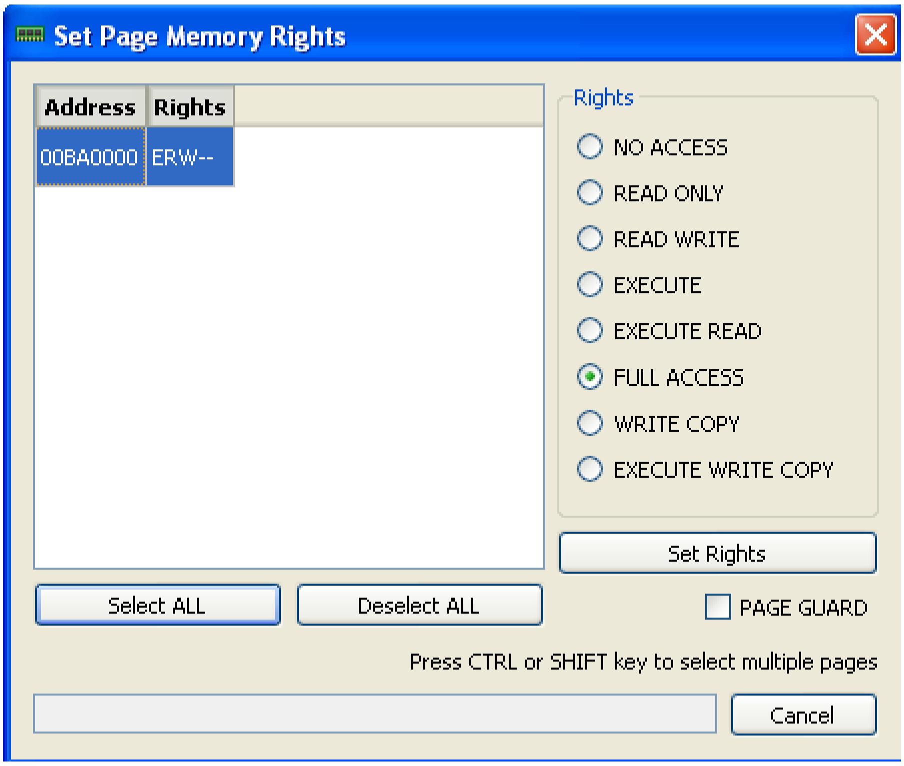
Alternatively, we could have used Scylla to look up the address of the *IAT* for us.
Scrolling down a bit in the dump we can see that the *IAT* ends up at the address **0x72A87F**:



0072A7E0	BA	7D	B1	76	C1	79	B1	76	2B	84	B1	76	C2	80	B1	76
0072A7F0	92	82	B1	76	AC	7F	B1	76	45	AA	B1	76	A5	AD	B0	76
0072A800	90	B0	B1	76	D4	02	B1	76	F8	94	B0	76	E1	95	B0	76
0072A810	B2	06	B1	76	E1	07	B1	76	E2	43	B1	76	E1	E6	B0	76
0072A820	FC	FB	B0	76	66	F6	B0	76	B6	5F	B0	76	01	52	B0	76
0072A830	F3	05	B1	76	16	01	B1	76	00	00	00	00	00	00	00	00
0072A840	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072A850	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072A860	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072A870	00	00	00	00	00	00	00	00	53	2A	4D	77	7E	05	4D	77
0072A880	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072A890	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072A8A0	00	00	00	00	00	00	00	00	00	00	00	00	E4	01	4D	75
0072A8B0	6C	74	69	42	79	74	65	54	6F	57	69	64	65	43	68	61
0072A8C0	72	00	64	02	53	65	74	45	72	72	6F	72	4D	6F	64	65

The last thing we need is to find a code cave where to put our own assembly code.

From the **MemoryMap** tab we choose a memory area that is marked as **PRV**, that is free and that is large enough for our purpose. I chose the one starting **0xBA0000**. Before continuing, remember to right-click on this section, choose **Set Page Memory Rights** and select **Full Access**:



We confirm by clicking **Set Rights**.

Since we're going to patch the calls in the text segment (replacing the usual SecuROM proxy calls with the real API addresses), let's make sure it has the writeable flag correctly set.

We are ready to write our code now, so let's go to **0xBA0000** and insert the following code:

```
00BA0000 | B9 00104000    | mov ecx,_start.401000               |
00BA0005 | 8139 FF150013  | cmp dword ptr ds:[ecx],130015FF     |
00BA000B | 75 2B          | jne BA0038                          |
00BA000D | 8079 04 74     | cmp byte ptr ds:[ecx+4],74          |
00BA0011 | 75 25          | jne BA0038                          |
00BA0013 | 890D 9000BA00  | mov dword ptr ds:[BA0090],ecx       | store ecx
00BA0019 | FFE1           | jmp ecx                             | jump to the securom call
00BA001B | 8B0D 9000BA00  | mov ecx,dword ptr ds:[BA0090]       | restore ecx (return from hook)
00BA0021 | BB B0A47200    | mov ebx,<_start.&DirectDrawCreate>  | mov ebx, 0x72A4B0 (IAT START)
00BA0026 | 3903           | cmp dword ptr ds:[ebx],eax          |
00BA0028 | 74 0B          | je BA0035                           |
00BA002A | 43             | inc ebx                             |
00BA002B | 81FB 7FA87200  | cmp ebx,_start.72A87F               |
00BA0031 | 75 F3          | jne BA0026                          |
00BA0033 | CD 03          | int 3                               | ERROR!! THUNK NOT FOUND!!
00BA0035 | 8959 02        | mov dword ptr ds:[ecx+2],ebx        |
00BA0038 | 41             | inc ecx                             |
00BA0039 | 81F9 F92F4600  | cmp ecx,_start.462FF9               |
00BA003F | 75 C4          | jne BA0005                          |
00BA0041 | CD 03          | int 3                               | COMPLETED!
00BA0043 | 0000           | add byte ptr ds:[eax],al            |
```

This is what the above code actually does:

- First it puts the starting address of the text section into the **ECX** register. The bytes are then compared against **0xFF15001374**, i.e., with the call to the SecuROM function (performed in two separate **cmp** instructions to check all the related bytes): If there is no match the **jne branch** will be followed and the address in **ECX** will be incremented by 1 to check for the next byte.

- If, on the other hand, we are in the front of a SecuROM call, we save the current address in the **ECX** register (which tells us which byte of the **.text** section we have arrived at) and jump into it.

- Once we reach the now famous **jmp eax** located in the SecuROM function we will set a hook (in a few minutes) to automatically jump back to **0xBA001B**. At this point we have the direct address of the requested API inside the **EAX** register.

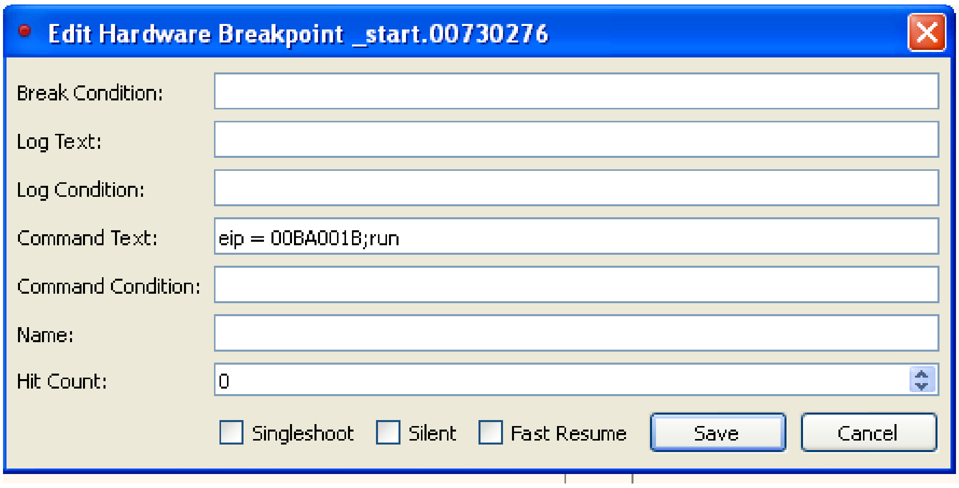
- Now we restore the **ECX** register and load the start address of the *IAT* into the **EBX** register.

- Let's loop through the *IAT* until we find the thunk that points to the address contained in **EAX** (**EBX** will obviously be incremented each time: If none of the thunks contain the API we're trying to fix, then we're in big trouble (*INT 3* at **0xBA0033**), but obviously this should NOT happen.

- If the thunk has been found, we will jump to address **0xBA0035**, where we will replace the bytes related to the SecuROM call with those of the correct thunk. At this point we can go back to continue scrolling the .text segment looking for the remaining calls.

- When the **.text** segment ends (**ECX** will be **0x462FF9**, i.e., the last address of the segment -6, which is the size of the call, in bytes), we are done and can proceed with the dump!

Before running the code, remember to right-click on the line **BA0000** and choose "*Set New Origin Here*". Now let's set our hook: move to **0x730276** (the address where the **jmp eax** is located), right click on it and choose "*Breakpoint*" -> "Set Hardware on Execution". Let's move to the **Breakpoints** tab, right click on the hardware breakpoint we just created and choose Edit. Let's configure it in this way:

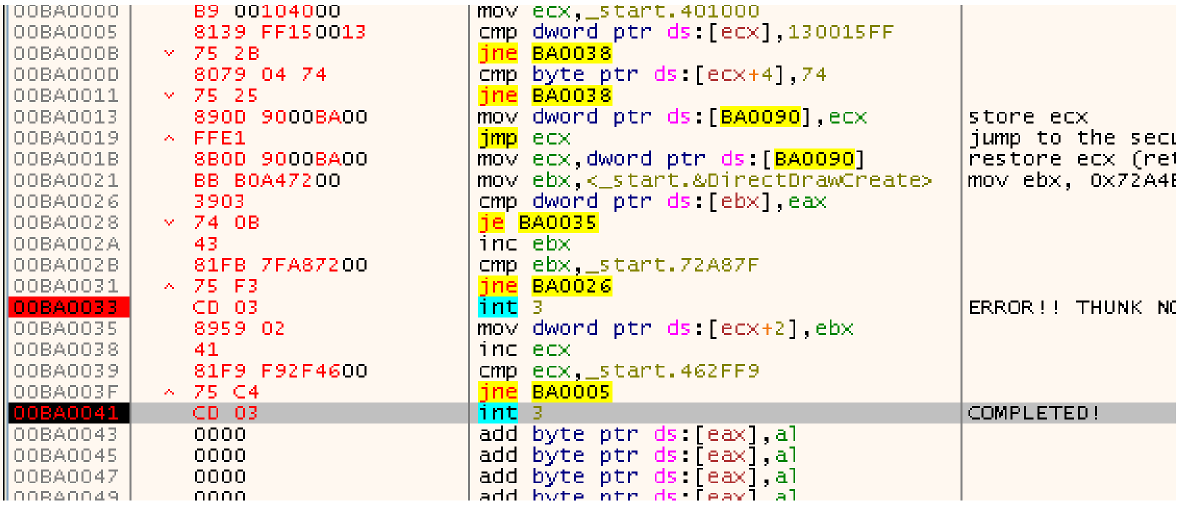


This way once the BP is triggered, we will automatically return to our code (at address 0xBA001B).

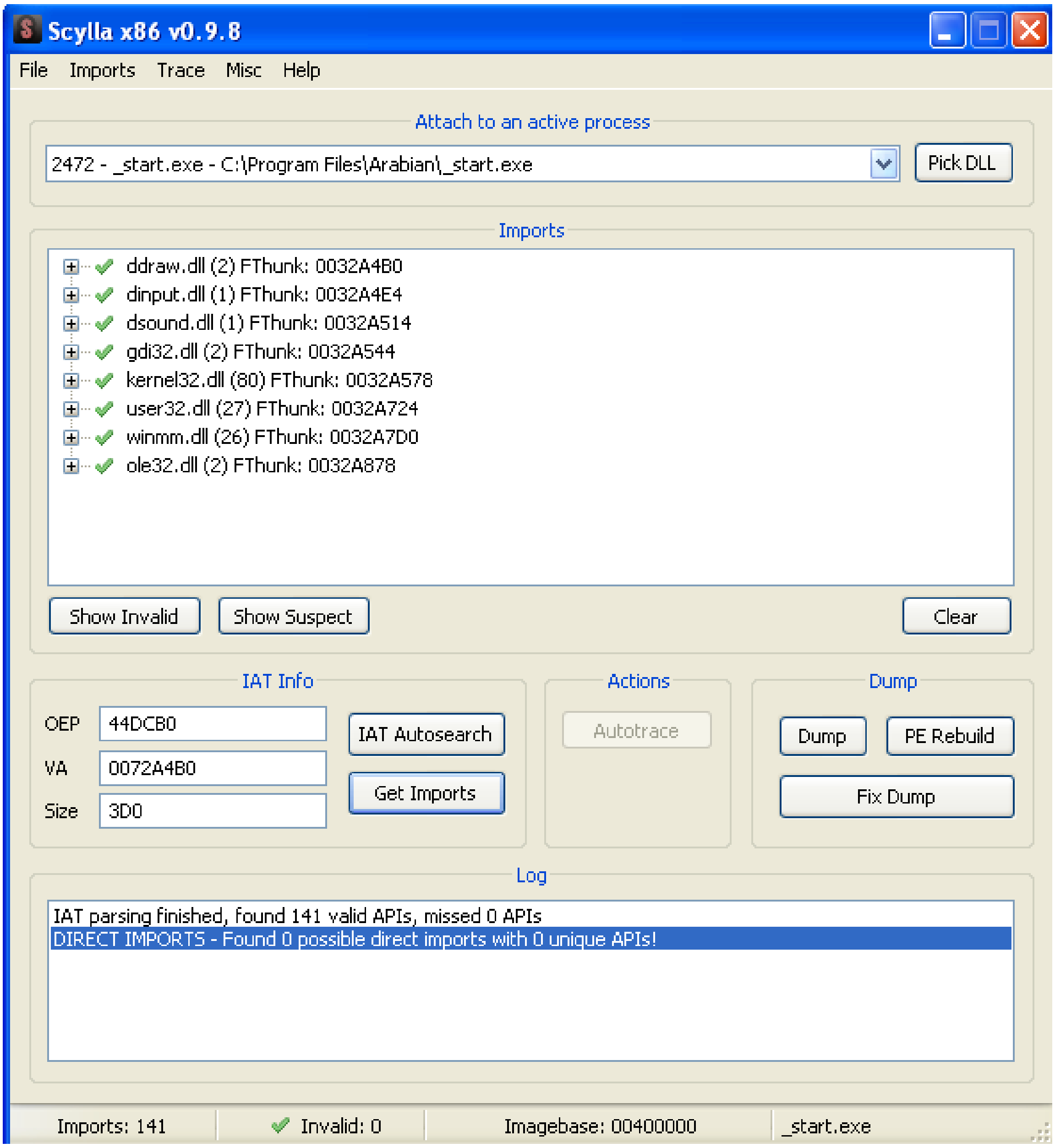
<u>PAY ATTENTION:</u> only use a hardware breakpoint in this case, otherwise the program will crash (the presence of software breakpoints will be detected).

We are ready to run our code, move to **0xBA0000** and click *RUN*.

Once the execution is complete, we will be stuck at **0xBA0041**:



We are almost there! Let's launch Scylla, choose the right process (*_start.exe*), set the *OEP*, the address of the *IAT* and its size:



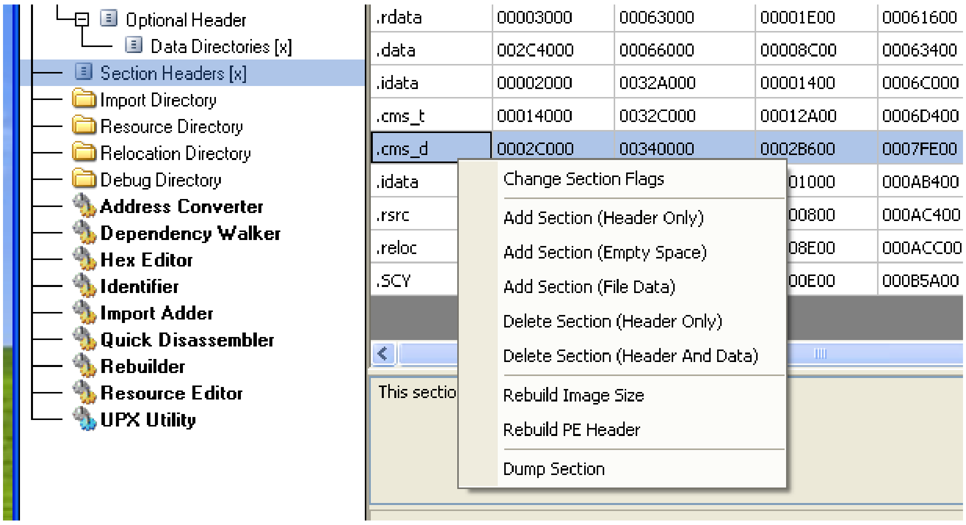
Let's click on *Get Imports*, then on *Dump* and finally on *Fix Dump*.

Finally, we will have an executable free from SecuROM 😀

Let's finish the job:

Our executable works perfectly and SecuROM is just past the memory addresses used by our new executable. However, if we want to be true perfectionists, we can make it slightly smaller by removing those two sections used by the SecuROM loader that now only take up useless space.

Let's load our binary into CFF Explorer and move to the *Section Headers* tab. Select the **.cms_t** and **.cms_d** sections, right click and choose *Delete Section (Header And Data)*:



Let's save the new executable (it will be much smaller) and we're **DONE**! 😀

Credits:

I would like to thank again the legendary Antelox for suggesting me the snapshot technique on VM. Undoubtedly a great way to save a lot of time.

A big thank you goes to m00k00 for reviewing and fixing the english translation of this technical paper!!

Conclusion:

As we have seen SecuROM *new* 4.48.00.004 conceptually shares something with Laserlock. Ultimately it is a very didactic DRM from which we have certainly learned something.

Thank you for reading this document 😊
Luca