SecuROM *new* 3.06.00.0003 Cracking (South Park Rally)

GAME: South Park Rally [https://en.wikipedia.org/wiki/South_Park_Rally]
Protection: SecuROM *new* 3.06.00.0003
Author: Luca D’Amico - V1.0 - 24 Marzo 2023

DISCLAIMER:
Tutte le informazioni contenute in questo documento tecnico sono pubblicate solo a scopo informativo e in buona fede.
Tutti i marchi citati qui sono registrati o protetti da copyright dai rispettivi proprietari.
Non fornisco alcuna garanzia riguardo alla completezza, correttezza, accuratezza e affidabilità di questo documento tecnico.
Questo documento tecnico viene fornito "COSÌ COM'È" senza garanzie di alcun tipo.
Qualsiasi azione intrapresa sulle informazioni che trovi in questo documento è rigorosamente a tuo rischio.
In nessun caso sarò ritenuto responsabile o responsabile in alcun modo per eventuali danni, perdite, costi o responsabilità di qualsiasi tipo risultanti o derivanti direttamente o indirettamente dall'utilizzo di questo documento tecnico. Solo tu sei pienamente responsabile delle tue azioni.

Cosa ci serve:

- Windows 10/11 (nativo, il gioco non funziona su VM)
- x64dbg (x32dbg) [https://x64dbg.com/]
- CFF Explorer [https://ntcore.com/?page_id=388]
- Disco di gioco originale (abbiamo bisogno del disco ORIGINALE)

Prima di iniziare:

Non sono riuscito in nessun modo a far partire questo gioco su una VM, probabilmente per problemi legati alla GPU virtuale di VMWare/Virtualbox. Per fortuna si avvia tranquillamente su Windows 11.
Come nel caso di SecuROM *new* 4.48.00.0004 (analizzata in uno dei miei documenti precedenti), il funzionamento di questo DRM consiste nel sostituire alcune chiamate alle varie API di Windows usate dal gioco, con una funzione che una volta effettuati i dovuti controlli, raggiungerà l’API richiesta con un jump. Il salto sarà diretto, senza passare per il thunk sulla IAT, ciò significa che quando ricostruiremo gli imports, dovremo scorrere la IAT alla ricerca del thunk corretto da inserire nel segmento .text patchando i bytes dove avviene la chiamata. È anche presente uno strato di cifratura inziale (per questo serve il disco originale) e varie tecniche anti-debug che complicheranno il raggiungimento dell’OEP.

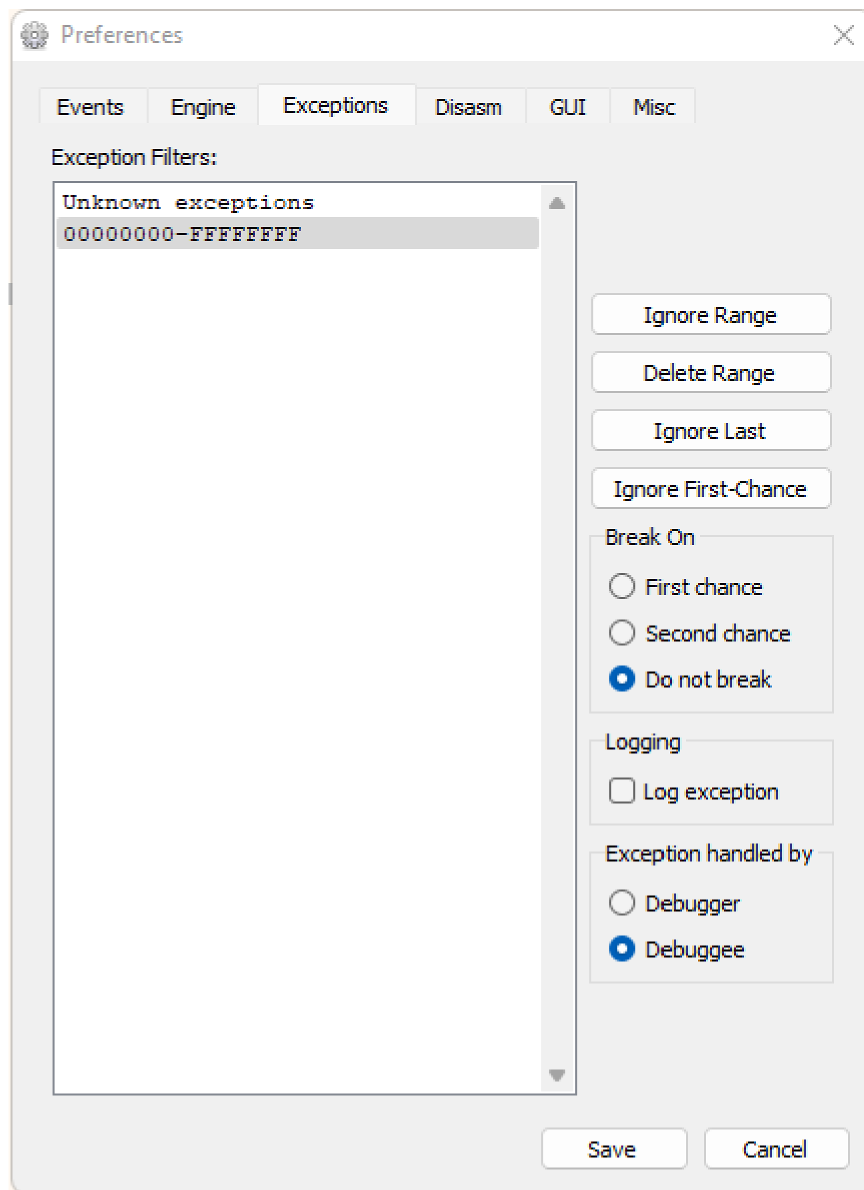
IMPORTANTE: Se l’eseguibile del gioco (sprally2.exe) viene avviato in modalità compatibilità, si avranno problemi durante la fase di dumping, in quanto verranno hookate varie API sulla IAT e Scylla non sarà in grado di riconoscerle!! Se durante questa fase riscontrate problemi di questo tipo, assicuratevi di NON aver eseguito il gioco in modalità retrocompatibilità. Se si verificano ugualmente problemi di API non riconosciute correttamente da Scylla perché hookate da ShimLib/aclayers.dll/apphelp.dll, provate a rinominare l’eseguibile del gioco (ad esempio da sprally2.exe a sprally2_orig.exe, come ho fatto io). Nei miei documenti precedenti non avendo piena consapevolezza di questa problematica, ho proceduto risolvendo manualmente le API hookate: questo sforzo non è in realtà necessario!

Iniziamo:

Installiamo il gioco e carichiamo l’eseguibile in x32dbg (sprally2.exe). Notiamo che l’entry point è situato a 0x004C3F82. Andando sulla Memory Map possiamo vedere che ci troviamo nella section .cms_t, ovvero la classica section del loader di SecuROM:

00400000	00001000	User	sprally2_orig.exe		IMG	-R---	ERWC-
00401000	0008D000	User	".text"	Executable code	IMG	ER---	ERWC-
0048E000	00005000	User	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00493000	00027000	User	".data"	Initialized data	IMG	-RWC-	ERWC-
004BA000	00011000	User	".cms_t"		IMG	ER---	ERWC-
004CB000	0001D000	User	".cms_d"		IMG	-RWC-	ERWC-
004E8000	00004000	User	".idata"	Import tables	IMG	-R---	ERWC-
004EC000	00002000	User	".rsrc"	Resources	IMG	-R---	ERWC-

Se avete già letto il mio documento su SecuROM 4.48, saprete che premendo su RUN nel debugger verremo inondati di eccezioni di ogni tipo. Procediamo configuriamo x32dbg per ignorarle, andando su Options -> Preferences -> Exceptions e scegliamo di ignorarle tutte:



Ricordate anche di togliere il segno di spunta da “Log exceptions”.

Ok, adesso il prossimo passo è ovviamente arrivare all’OEP.

Mettere un breakpoint hardware in esecuzione sul segmento .text non ci aiuterà, poiché come abbiamo già visto nell’altro documento sulla più recente versione di SecuROM, il loader ne rileverà la presenza mandando il processo in loop.

Possiamo utilizzare la stessa tecnica usata l’ultima volta: settiamo un breakpoint su una API che di solito viene chiamata vicino all’entry point, come la GetCommandLineA e ad ogni scatto, clicchiamo su run to user

code e verifichiamo in che segmento ci troviamo (ricordate che il nostro obiettivo è quello di arrivare dentro la sezione .text).

Dopo il terzo scatto del breakpoint, premendo run to user code, ci troveremo a 0x00484A1C proprio dentro il segmento .text!

```
     0048499F  DDD9               fstp st(1),st(0)
     004849A1  C3                 ret
     004849A2  55                 push ebp
     004849A3  8BEC               mov ebp,esp
     004849A5  6A FF              push FFFFFFFF
     004849A7  68 48EE4800        push sprally2_orig.48EE48
     004849AC  68 3C424800        push sprally2_orig.48423C
     004849B1  64:A1 00000000     mov eax,dword ptr fs:[0]
     004849B7  50                 push eax
     004849B8  64:8925 00000000   mov dword ptr fs:[0],esp
     004849BF  83EC 58            sub esp,58
     004849C2  53                 push ebx
     004849C3  56                 push esi
     004849C4  57                 push edi
     004849C5  8965 E8            mov dword ptr ss:[ebp-18],esp
     004849C8  FF15 E8C24C00      call dword ptr ds:[4CC2E8]
     004849CE  33D2               xor edx,edx
     004849D0  8AD4               mov dl,ah
     004849D2  8915 44FA4A00      mov dword ptr ds:[4AFA44],edx
     004849D8  8BC8               mov ecx,eax
     004849DA  81E1 FF000000      and ecx,FF
     004849E0  890D 40FA4A00      mov dword ptr ds:[4AFA40],ecx
     004849E6  C1E1 08            shl ecx,8
     004849E9  03CA               add ecx,edx
     004849EB  890D 3CFA4A00      mov dword ptr ds:[4AFA3C],ecx
     004849F1  C1E8 10            shr eax,10
     004849F4  A3 38FA4A00        mov dword ptr ds:[4AFA38],eax
     004849F9  33F6               xor esi,esi
     004849FB  56                 push esi
     004849FC  E8 3C380000        call sprally2_orig.48823D
     00484A01  59                 pop ecx
     00484A02  85C0               test eax,eax
     00484A04  75 08              jne sprally2_orig.484A0E
     00484A06  6A 1C              push 1C
     00484A08  E8 B0000000        call sprally2_orig.484ABD
     00484A0D  59                 pop ecx
     00484A0E  8975 FC            mov dword ptr ss:[ebp-4],esi
     00484A11  E8 59360000        call sprally2_orig.48806F
     00484A16  FF15 E8C24C00      call dword ptr ds:[4CC2E8]
EIP  00484A1C  A3 34944B00        mov dword ptr ds:[4B9434],eax
     00484A21  E8 17350000        call sprally2_orig.487F3D
     00484A26  A3 F8F94A00        mov dword ptr ds:[4AF9F8],eax
     00484A2B  E8 C0320000        call sprally2_orig.487CF0
```

Da notare però che NON ci troviamo subito sotto ad una call a GetCommandLineA, ma bensì a CALL dword ptr ds:[0x4CC2E8]. Sappiamo già il perché (sempre grazie all'analisi precedente di SecuROM 4.48): SecuROM rimpiazza varie API con una sua funzione che, dopo le opportune verifiche, si occuperà di chiamare l'API richiesta dal gioco.

Dall'ultimo screenshot è facile dedurre che l'OEP si trova a 0x4849A2 (inizio dell'attuale funzione).

Se riavviamo il debugger e torniamo a questo indirizzo, vedremo questo:

```
004849A2  837D F4 A8     cmp dword ptr ss:[ebp-C],FFFFFFA8
004849A6  283C5A         sub byte ptr ds:[edx+ebx*2],bh
004849A9  EB 88          jmp sprally2_orig.484933
004849AB  2E:B6 8E       mov dh,8E
004849AE  34 2A          xor al,2A
004849B0  BB BFE559B5    mov ebx,B559E5BF
```

Un disassemblato completamente diverso da quello precedente. Questo è dovuto al fatto che il loader di SecuROM ancora non ha sovrascritto questa area di memoria con le istruzioni decifrate che abbiamo visto prima.

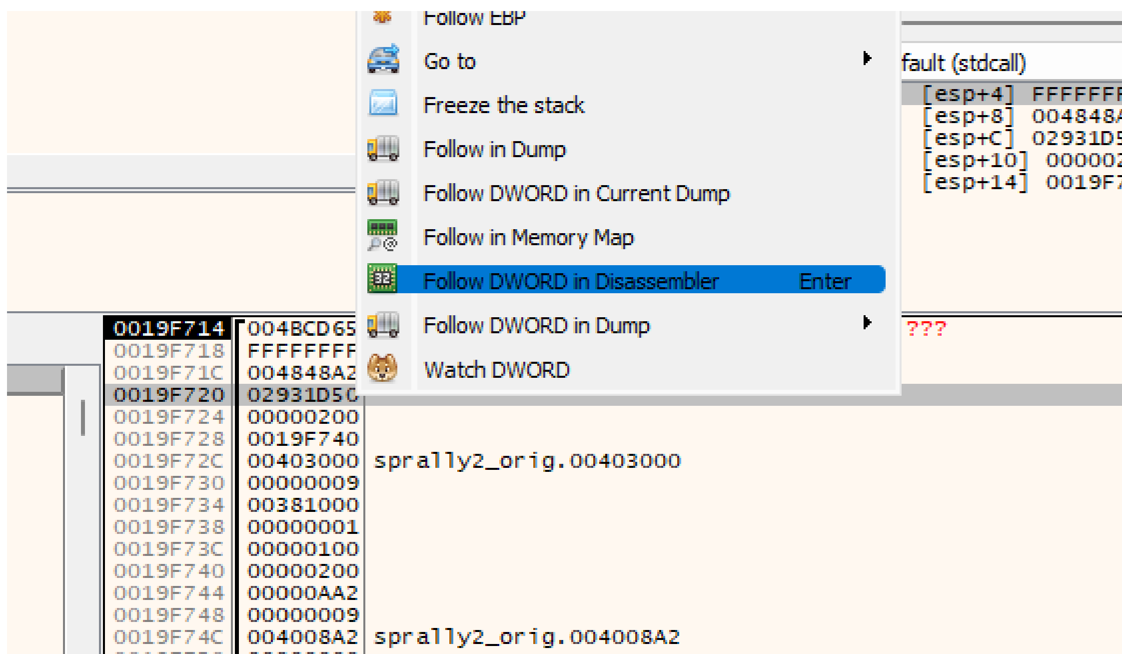
Come già sappiamo, su Windows i processi possono modificare la memoria grazie all'API WriteProcessMemory, che ha la seguente firma:

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

Quindi procediamo disattivando il breakpoint su GetCommandLineA, mettendo un breakpoint sulla WriteProcessMemory e avviamo l'esecuzione.

ATTENZIONE: questa versione di SecuROM (a differenza della versione successiva) sembra mandare il loader in loop quando rileva un breakpoint su WriteProcessMemory. La soluzione è semplice, ed è la stessa che abbiamo visto con Safedisc: mettiamo il breakpoint su WriteProcessMemory+0x2. In questo modo non verrà riconosciuto.

Ogni volta che scatta il breakpoint controlliamo sullo stack il parametro l'lpBaseAddress (ovvero l'indirizzo dove i dati andranno scritti), se è vicino al nostro OEP, è quello corretto! Alternativamente basta guardare il nostro lettore CD/DVD: quando vediamo attività di lettura, sapremo che la prossima chiamata a WriteProcessMemory scriverà i byte decifrati con il nostro OEP.



Ci troveremo in questa situazione. Guardando lo stack vediamo che 0x0019F71C è l'indirizzo sul quale verrà scritto il buffer situato a 0x0019F720. Quindi clicchiamo con destro su quest'ultimo e selezioniamo Follow DWORD in Disassembler. Trovare il nostro OEP dentro il buffer a questo punto è facile: possiamo sommare all'indirizzo attuale (0x02931D50) la differenza tra 0x4849A2 (indirizzo dell'OEP trovato in precedenza) e 0x4848A2 (base address dove verrà scritto il buffer): 0x2931E50.

Ed infatti, recandoci a 0x2931E50 (usando CTRL+G), possiamo vedere che c'è il nostro OEP pronto per essere scritto in memoria :D

```
02931E50   55                 push ebp
02931E51   8BEC               mov ebp,esp
02931E53   6A FF              push FFFFFFFF
02931E55   68 48EE4800        push sprally2_orig.48EE48
02931E5A   68 3C424800        push sprally2_orig.48423C
02931E5F   64:A1 00000000     mov eax,dword ptr fs:[0]
02931E65   50                 push eax
02931E66   64:8925 00000000   mov dword ptr fs:[0],esp
02931E6D   83EC 58            sub esp,58
02931E70   53                 push ebx
02931E71   56                 push esi
02931E72   57                 push edi
02931E73   8965 E8            mov dword ptr ss:[ebp-18],esp
02931E76   FF15 E8C24C00      call dword ptr ds:[4CC2E8]
02931E7C   33D2               xor edx,edx
02931E7E   8AD4               mov dl,ah
02931E80   8915 44FA4A00      mov dword ptr ds:[4AFA44],edx
02931E86   8BC8               mov ecx,eax
02931E88   81E1 FF000000      and ecx,FF
02931E8E   890D 40FA4A00      mov dword ptr ds:[4AFA40],ecx
02931E94   C1E1 08            shl ecx,8
02931E97   03CA               add ecx,edx
02931E99   890D 3CFA4A00      mov dword ptr ds:[4AFA3C],ecx
02931E9F   C1E8 10            shr eax,10
02931EA2   A3 38FA4A00        mov dword ptr ds:[4AFA38],eax
02931EA7   33F6               xor esi,esi
02931EA9   56                 push esi
02931EAA   E8 3C380000        call 29356EB
02931EAF   59                 pop ecx
02931EB0   85C0               test eax,eax
02931EB2   75 08              jne 2931EBC
02931EB4   6A 1C              push 1C
02931EB6   E8 B0000000        call 2931F6B
02931EBB   59                 pop ecx
02931EBC   8975 FC            mov dword ptr ss:[ebp-4],esi
02931EBF   E8 59360000        call 293551D
02931EC4   FF15 E8C24C00      call dword ptr ds:[4CC2E8]
02931ECA   A3 34944B00        mov dword ptr ds:[4B9434],eax
02931ECF   E8 17350000        call 29353EB
02931ED4   A3 F8F94A00        mov dword ptr ds:[4AF9F8],eax
02931ED9   E8 C0320000        call 293519E
02931EDE   E8 02320000        call 29350E5
02931EE3   E8 012F0000        call 2934DE9
02931EE8   8975 D0            mov dword ptr ss:[ebp-30],esi
02931EEB   8D45 A4            lea eax,dword ptr ss:[ebp-5C]
02931EEE   50                 push eax
02931EEF   FF15 E8C24C00      call dword ptr ds:[4CC2E8]
02931EF5   E8 93310000        call 293508D
02931EFA   8945 9C            mov dword ptr ss:[ebp-64],eax
```

C’è anche la CALL che abbiamo visto precedentemente (che sappiamo essere in realtà una GetCommandLineA) a 0x2931EC4.

Usiamo la tecnica del EBFE (salto sullo stesso indirizzo): patcheremo 0x2931E50 e 0x2931E51 con i byte EB ed FE in modo tale che dopo che questo buffer verrà scritto nella memoria, ci ritroveremo in loop proprio sull’OEP:

```
02931E50   EB FE          jmp 2931E50
02931E52   EC             in al,dx
02931E53   6A FF          push FFFFFFFF
02931E55   68 48EE4800    push sprally2_orig.48EE48
02931E5A   68 3C424800    push sprally2_orig.48423C
```

Lasciamo tutto il resto invariato e premiamo su RUN per continuare l’esecuzione (ricordatevi di disattivare il breakpoint su WriteProcessMemory+0x2).

Aspettiamo qualche secondo e clicchiamo sul tasto PAUSE del debugger. Saremo qui, proprio sull’OEP:

```
004849A2   EB FE          jmp sprally2_orig.4849A2
004849A4   EC             in al,dx
004849A5   6A FF          push FFFFFFFF
004849A7   68 48EE4800    push sprally2_orig.48EE48
004849AC   68 3C424800    push sprally2_orig.48423C
```

Il codice è il loop proprio grazie alla nostra patch dell’EBFE (guardate la freccia rossa).

Mettiamo subito un breakpoint e ripristiniamo i byte modificati (sostituiamo EB FE con 55 8B):

```
0048499F   DDD9                fstp st(1),st(0)
004849A1   C3                  ret
EIP EAX  004849A2   55                  push ebp
004849A3   8BEC                mov ebp,esp
004849A5   6A FF               push FFFFFFFF
004849A7   68 48EE4800         push sprally2_orig.48EE48
004849AC   68 3C424800         push sprally2_orig.48423C
004849B1   64:A1 00000000      mov eax,dword ptr fs:[0]
004849B7   50                  push eax
004849B8   64:8925 00000000    mov dword ptr fs:[0],esp
004849BF   83EC 58             sub esp,58
004849C2   53                  push ebx
```

Finalmente siamo fermi all’OEP!!

Ora possiamo pensare come ripristinare le API proxate da SecuROM e a fare un dump del nostro eseguibile dalla memoria.

Se fossimo stati su una VM, arrivati a questo punto sarebbe stato ideale effettuare uno snapshot per poterlo facilmente ripristinare successivamente in caso di problemi. Purtroppo, come già detto nell’introduzione di questo documento, questo gioco non funziona su VM. Ho deciso così di scrivere un piccolo script per x64dbg in modo tale da automatizzare quanto fatto sino ad ora e poter tornare facilmente all’OEP in caso di problemi:

```
// -------------------------------------
// variables
OEP_addr = 0x4849A2
wpm_buffer_base_addr = 0x4848A2


// start
run // run til the EntryPoint


// clear breakpoints
bc
bphwc


// find and hook WriteProcessMemory
wpm_addr = kernel32.dll:WriteProcessMemory
bp wpm_addr+0x2
SetBreakpointCommand wpm_addr+0x2, "scriptcmd call check_buffer"
erun
ret


check_buffer:
log "WriteProcessMemory({arg.get(0)}, {arg.get(1)}, {arg.get(2)}, {arg.get(3)}, {arg.get(4)})"
cmp [esp+8], wpm_buffer_base_addr
```

```
je patch_buffer
erun
ret

patch_buffer:
set [esp+C]+(OEP_addr-wpm_buffer_base_addr), #EB FE#
rtr
bc
bphwc
bp OEP_addr
SetBreakpointCommand OEP_addr, "scriptcmd call restore_oep_bytes"
erun
ret

restore_oep_bytes:
set eip, #55 8B#
bc
bphwc
ret
// ------------------------------------
```

Abbiamo visto in precedenza che la funzione di SecuROM che proxa le varie API si trova a [0x4CC2E8] ovvero all'indirizzo presente a 0x4CC2E8. Possiamo ottenere questo indirizzo semplicemente cliccando con il destro su una di queste call dword ptr ds:[0x004CC2E8] e scegliendo Follow in dump -> Costant. Guardiamo in basso nella finestra DUMP 1:

```
004CC2E8  60 D1 4B 00
```

Ok l'indirizzo della funzione di SecuROM è 0x004BD160 (questi valori vanno letti da destra verso sinistra).

Mettiamo un breakpoint sulla call all'indirizzo 0x484A16 (sappiamo che questa chiamata dovrà eseguire, dopo i controlli effettuati da SecuROM, l'API GetCommandLineA):

```
00484A0D   59              pop ecx
00484A0E   8975 FC         mov dword ptr ss:[ebp-4],esi
00484A11   E8 59360000     call sprally2_orig.48806F
00484A16   FF15 E8C24C00   call dword ptr ds:[4CC2E8]
00484A1C   A3 34944B00     mov dword ptr ds:[4B9434],eax
00484A21   E8 17350000     call sprally2_orig.487F3D
00484A26   A3 F8F94A00     mov dword ptr ds:[4AF9F8],eax
00484A2B   E8 C0320000     call sprally2_orig.487CF0
00484A30   E8 02320000     call sprally2_orig.487C37
```

Ed una volta raggiunto, facciamo click su STEP INTO per saltarci dentro.
Ci ritroveremo ovviamente a 0x004BD160:

```
004BD160   55                  push ebp
004BD161   8BEC                mov ebp,esp
004BD163   83EC 2C             sub esp,2C
004BD166   53                  push ebx
004BD167   56                  push esi
004BD168   57                  push edi
004BD169   8B45 04             mov eax,dword ptr ss:[ebp+4]
004BD16C   8945 F0             mov dword ptr ss:[ebp-10],eax
004BD16F   60                  pushad
004BD170   68 A0CD4B00         push spra1ly2_orig.4BCDA0
004BD175   64:FF35 00000000    push dword ptr fs:[0]
004BD17C   64:A1 00000000      mov eax,dword ptr fs:[0]
004BD182   8945 F8             mov dword ptr ss:[ebp-8],eax
004BD185   64:8925 00000000    mov dword ptr fs:[0],esp
004BD18C   8B45 F0             mov eax,dword ptr ss:[ebp-10]
004BD18F   83E8 04             sub eax,4
004BD192   8945 F0             mov dword ptr ss:[ebp-10],eax
004BD195   68 40364E00         push spra1ly2_orig.4E3640
004BD19A   FF15 18524E00       call dword ptr ds:[<&RtlEnterCriticalSection>]
004BD1A0   8B0D 882C4E00       mov ecx,dword ptr ds:[4E2C88]
004BD1A6   83C1 01             add ecx,1
004BD1A9   890D 882C4E00       mov dword ptr ds:[4E2C88],ecx
004BD1AF   8B55 F0             mov edx,dword ptr ss:[ebp-10]
004BD1B2   C1EA 10             shr edx,10
004BD1B5   81E2 FFFF0000       and edx,FFFF
004BD1BB   81E2 FFFF0000       and edx,FFFF
004BD1C1   8B45 F0             mov eax,dword ptr ss:[ebp-10]
004BD1C4   25 FFFF0000         and eax,FFFF
004BD1C9   33D0                xor edx,eax
004BD1CB   66:8955 D8          mov word ptr ss:[ebp-28],dx
004BD1CF   8B45 D8             mov eax,dword ptr ss:[ebp-28]
004BD1D2   25 FF000000         and eax,FF
004BD1D7   99                  cdq
004BD1D8   B9 09000000         mov ecx,9
004BD1DD   F7F9                idiv ecx
004BD1DF   33C0                xor eax,eax
004BD1E1   8A82 104F4E00       mov al,byte ptr ds:[edx+4E4F10]
004BD1E7   C1E0 08             shl eax,8
004BD1EA   66:8B4D D8          mov cx,word ptr ss:[ebp-28]
004BD1EE   66:33C8             xor cx,ax
004BD1F1   66:894D D8          mov word ptr ss:[ebp-28],cx
004BD1F5   8B45 D8             mov eax,dword ptr ss:[ebp-28]
```

Come potete vedere questa funzione è molto simile a quella di SecuROM che abbiamo già analizzato nel procedente documento (su SecuROM *new* 4.48), quindi non ripeterò tutta la spiegazione. L'importante è sapere che scorrendo alla fine di questa funzione c'è un jmp eax:

```
004BD6DC   5F        pop edi
004BD6DD   5E        pop esi
004BD6DE   5B        pop ebx
004BD6DF   8BE5      mov esp,ebp
004BD6E1   5D        pop ebp
004BD6E2   FFE0      jmp eax
004BD6E4   5F        pop edi
004BD6E5   5E        pop esi
004BD6E6   5B        pop ebx
004BD6E7   8BE5      mov esp,ebp
004BD6E9   5D        pop ebp
004BD6EA   C3        ret
004BD6EB   E5 D4     in eax,D4
004BD6ED   4B        dec ebx
004BD6EE   00FD      add ch,bh
004BD6F0   D4 4B     aam 4B
004BD6F2   000E      add byte ptr ds:[esi],cl
004BD6F4   D5 4B     aad 4B
```

Binary | Copy | Breakpoint | Follow in Dump | Follow in Memory Map | Graph G
Set Conditional Breakpoint Shift+F2 | Toggle F2 | Set Hardware on Execution

Mettendo un breakpoint hardware (attenzione, usate solo breakpoint hardware su questa funzione, altrimenti il processo andrà in crash) a quell'indirizzo, raggiungiamolo e guardiamo il contenuto del registro eax:

```
EAX   752AA2E0   <kernel32.GetCommandLineA>
EBX   00341000
ECX   4F819AC7
EDX   00000000
EBP   0019F760
ESP   0019F6E8
```

Ecco come viene raggiunta la GetCommandLineA.

Ok, l'idea è quella di scorrere tutto il segmento .text alla ricerca delle CALL alla funzione di SecuROM e dopo averle chiamate per recuperare le API, patcharle con gli indirizzi corretti appena ottenuti. Tuttavia, l'indirizzo contenuto in eax al momento del salto è un indirizzo assoluto alla API richiesta, che non passa dal relativo thunk della IAT. Non possiamo sostituirlo a quello della call a SecuROM in quanto cambierà essendo dinamico. A noi serve l'indirizzo del thunk corrispondente nella IAT.

Quindi, questo è quello che faremo:

1) Scorriamo il segmento .text alla ricerca delle call di SecuROM
2) Appena troviamo una call ci saltiamo dentro
3) Hookiamo il jump eax per tornare al nostro codice assembly (ottenendo l'indirizzo diretto della API che stiamo risolvendo)
4) Usando l'indirizzo diretto della API (contenuto in eax), scorriamo la IAT cercando il thunk corrispondente
5) Una volta trovato, patchamo la funzione nel segmento text per chiamare l'indirizzo del thunk.

Trovare l'indirizzo di inizio della IAT è un'operazione estremamente facile: possiamo andare nella relativa sezione in Memory Map, selezionare il segmento .rdata e cliccare su follow in dump e a quel punto vedere dove iniziano ad essere presenti gli indirizzi alle varie API (sottolineati in verde o in rosso):

```
0048E000 | 90 46 92 76 | 00 00 00 00 | 80 4E EF 76 | 00 00 00 00 | .F.v.....Nïv....
0048E010 | B0 B0 BF 76 | C0 65 BF 76 | E0 63 BF 76 | 20 5B BF 76 | °°¿vÀe¿vàc¿v [¿v
0048E020 | 40 62 BF 76 | 80 60 BF 76 | 20 64 BF 76 | 40 60 BF 76 | @b¿v.`¿v d¿v@`¿v
0048E030 | A0 5E BF 76 | 00 00 00 00 | 90 73 E1 76 | 90 9C E1 76 |  ^¿v.....sáv..áv
0048E040 | D0 41 E2 76 | 30 2D E2 76 | 20 96 E1 76 | 50 9C E1 76 | ÐAâv0-âv .ávP.áv
0048E050 | B0 88 E1 76 | 40 47 E1 76 | 60 08 E2 76 | 20 8D E1 76 | °.áv@Gáv`.âv .áv
0048E060 | E0 1C E1 76 | 80 3B E2 76 | 80 41 E2 76 | 70 42 E2 76 | à.áv.;âv.AâvpBâv
0048E070 | 80 08 E2 76 | 00 3B E2 76 | A0 9D E3 76 | 20 81 E3 76 | ..âv.;âv .ãv .ãv
0048E080 | 50 A0 E1 76 | 90 4D E1 76 | 80 83 E3 76 | F0 A2 E1 76 | P áv.Máv..ãvð¢áv
0048E090 | 70 ED E1 76 | D0 66 E1 76 | 30 40 E2 76 | 40 28 E2 76 | píávÐfáv0@âv@(âv
0048E0A0 | E0 99 E1 76 | A0 66 E1 76 | 10 42 E2 76 | D0 99 E3 76 | à.áv fáv.BâvÐ.ãv
0048E0B0 | 70 49 E1 76 | 30 8E E1 76 | 90 3F E2 76 | 20 0A E2 76 | pIáv0.áv.?âv .âv
0048E0C0 | D0 F8 E1 76 | C0 6D E1 76 | 10 FE E1 76 | B0 4D E1 76 | ÐøávÀmáv.þáv°Máv
0048E0D0 | 90 7F E1 76 | C0 66 E1 76 | 10 3B E2 76 | 50 F9 E1 76 | ..ávÀfáv.;âvPùáv
0048E0E0 | 80 95 E1 76 | E0 A0 E1 76 | F0 4C E6 77 | 20 2E E2 76 | ..ávà ávðLæw .âv
0048E0F0 | A0 73 E1 76 | 80 8D E1 76 | 70 1F E2 76 | E0 A2 E1 76 |  sáv..ávp.âvà¢áv
```

Ok l'indirizzo di inizio della IAT è 0x48E000 e finisce a 0x48E783:

```
0048E720 | 30 C1 D5 00 | 70 A0 D5 00 | 10 A1 D5 00 | 20 E9 D4 00 | 0ÁÕ.p Õ..¡Õ. éÔ.
0048E730 | 80 DA D4 00 | D0 DE D4 00 | 30 DF D4 00 | B0 E2 D4 00 | .ÚÔ.ÐÞÔ.0ßÔ.°âÔ.
0048E740 | F0 E0 D4 00 | 90 D6 D4 00 | 10 DC D4 00 | C0 D7 D4 00 | ðàÔ..ÖÔ..ÜÔ.À×Ô.
0048E750 | 70 D2 D5 00 | 00 DF D4 00 | 30 D6 D4 00 | 60 D6 D4 00 | pÒÕ..ßÔ.0ÖÔ.`ÖÔ.
0048E760 | 10 C7 D5 00 | 50 D6 D4 00 | 00 56 D4 00 | F0 9D D5 00 | .ÇÕ.PÖÔ..VÔ.ð.Õ.
0048E770 | 00 34 D5 00 | 40 A0 D5 00 | 00 00 00 00 | 40 89 9F 76 | .4Õ.@ Õ.....@..v
0048E780 | F0 74 CF 75 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ðtÏu............
0048E790 | 00 00 00 00 | 8D 19 60 38 | 00 00 00 00 | 02 00 00 00 | ......`8........
0048E7A0 | 35 00 00 00 | 00 00 00 00 | 00 00 0B 00 | 9A 99 19 3E | 5..............>
0048E7B0 | 9A 99 19 3F | 66 66 66 3F | CD CC CC 3E | 33 33 33 3F | ...?fff?ÍÌÌ>333?
```

Otteniamo la grandezza della IAT sottraendo l'indirizzo di inizio all'indirizzo di fine: 0x783.

Questi dati ci torneranno molto utili.

Adesso abbiamo bisogno di un po' di spazio libero all'interno della memoria dove scrivere il nostro codice assembly che si occuperà di eseguire le nostre operazioni. Andiamo su tab Memory Map e facciamo riservare un po' di memoria cliccando con il destro, selezionando Allocate Memory e infine su Ok (0x1000 bytes sono sufficienti).

Ci ritroveremo su DUMP1 all'indirizzo di partenza della nostra zona di memoria riservata. Non abbiamo bisogno di configurare i diritti di accesso a questa area di memoria in quanto ci ha già pensato il debugger, assicuratevi però che il segmento .text NON sia protetto in scrittura.

Clicchiamo con il destro sul primo indirizzo di questa area di memoria e scegliamo Follow in Disassembler. Siamo pronti a scrivere il codice che si occuperà di fixare le API proxate!

Scriviamo attentamente il seguente codice:

```
006A0000 | B9 00104000     | mov ecx,sprally2_orig.401000                | set ecx to .text start address
006A0005 | 8139 FF15E8C2   | cmp dword ptr ds:[ecx],C2E815FF             | check if the address in ecx is equals to the SecuROM call pattern (first 4 bytes)
006A000B | 75 2D           | jne 6A003A                                  |
006A000D | 8079 04 4C      | cmp byte ptr ds:[ecx+4],4C                  | check if the byte in [ecx+4] is equals to the last byte of the SecuROM call pattern
006A0011 | 75 27           | jne 6A003A                                  |
006A0013 | 890D 90006A00   | mov dword ptr ds:[6A0090],ecx               | store ecx to a temp memory location
006A0019 | FFE1            | jmp ecx                                     | execute the SecuROM call
006A001B | 8B0D 90006A00   | mov ecx,dword ptr ds:[6A0090]               | restore the previously stored ecx value
006A0021 | BB 00E04800     | mov ebx,<sprally2_orig.&GetUserNameA>       | move IAT start address to ebx
006A0026 | 3903            | cmp dword ptr ds:[ebx],eax                  | check if we found the thunk we are looking for
006A0028 | 74 0D           | je 6A0037                                   | if yes, jump and fix the call
006A002A | 83C3 04         | add ebx,4                                   | if not, increment ecx to point to the next thunk
006A002D | 81FB 83E74800   | cmp ebx,sprally2_orig.48E783                | check if we are still within the IAT address range
006A0033 | 7C F1           | jl 6A0026                                   | if yes, jump and check the next thunk
006A0035 | CD 03           | int 3                                       | ERROR: THUNK NOT FOUND :(
006A0037 | 8959 02         | mov dword ptr ds:[ecx+2],ebx                | replace the SecuROM call with the real/original API (ebx==thunk address)
006A003A | 41              | inc ecx                                     |
006A003B | 81F9 FFDF4800   | cmp ecx,sprally2_orig.48DFFF                | check if we reached the end of the .text segment
006A0041 | 75 C2           | jne 6A0005                                  |
006A0043 | CD 03           | int 3                                       | COMPLETED :)
```

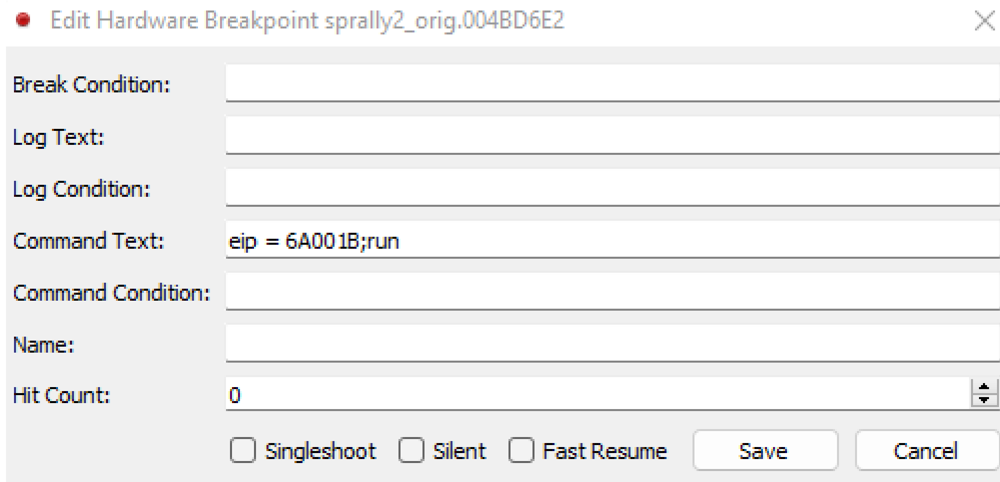
Il codice fa esattamente queste operazioni:

- L’indirizzo di inizio del segmento .text viene caricato nel registro ECX. Vengono quindi confrontati i byte attualmente puntati da ECX con la call di SecuROM (FF15E8C24C). Se non c’è una corrispondenza allora si procederà incrementando ECX di 1, per farlo puntare al prossimo byte del segmento .text, controllando se siamo arrivati al suo indirizzo finale.
- Quando viene trovata una chiamata alla funzione di SecuROM, salviamo l’indirizzo al quale siamo arrivati nel segmento .text in una posizione di memoria temporanea (io ho scelto 6A0090), così dopo averla eseguita, possiamo riprendere la ricerca da questo indirizzo.
- Effettuiamo un jmp su ECX in modo da effettuare la chiamata di SecuROM, e grazie al nostro hook (che imposteremo tra un attimo) torneremo all’indirizzo 6A001B subito prima che l’API originale venga chiamata. Avremo così l’indirizzo assoluto della nostra API nel registro EAX.
- Ripristiniamo il registro ECX con il valore salvato precedentemente, ovvero l’indirizzo al quale siamo arrivati nel segmento .text e carichiamo in EBX l’indirizzo di inizio della IAT (ricordatevi, abbiamo l’indirizzo assoluto dell’API in EAX, ora dobbiamo trovare il corrispettivo thunk).
- Scorriamo sulla IAT alla ricerca del thunk corrispondente all’API trovata, incrementando EBX di 4 byte alla volta (1 thunk è formato da 4 bytes). Se il thunk è stato trovato, saltiamo a 6A0037 e rimpiazziamo i byte della call di SecuROM con l’indirizzo del thunk corretto. Se non dovessimo trovare l’API in nessuno dei thunk della IAT, allora siamo in grossi guai (indirizzo 6A0035) e significa che abbiamo sbagliato qualcosa.
- Quando raggiungeremo l’indirizzo finale del segmento .text abbiamo completato il nostro lavoro e possiamo procedere con il dump.

Clicchiamo con il destro sull’indirizzo 0x6A0000 (indirizzo di partenza della memoria riservata per noi dal debugger) e selezioniamo Set EIP Here, in modo tale da impostare l’esecuzione partendo da questo indirizzo.

Rimane solo ha hookare il jmp eax nella funzione di SecuROM per riportare l’esecuzione al nostro codice (nello specifico all’indirizzo 0x6A001B).

Spostiamoci quindi all’indirizzo del jmp eax (0x4BD6E2) ed impostiamo un breakpoint hardware (i breakpoint software a questo indirizzo faranno crashare il gioco), cliccando con il destro e selezionando Breakpoint->Set Hardware on Execution. Adesso spostiamoci sul Tab Breakpoint, clicchiamo con il destro sul nostro breakpoint hardware appena impostato e selezioniamo Edit. Occorre valorizzare il campo Command Text in questo modo:



Così, ogni volta che il breakpoint scatterà, l'esecuzione continuerà dall'indirizzo 0x6A001B (il nostro codice).

Siamo pronti, torniamo all'area di memoria dove abbiamo scritto il nostro codice assembly e clicchiamo su RUN.

Dopo alcuni secondi, se tutto è andato bene, ci troveremo sull'ultima riga del nostro codice:

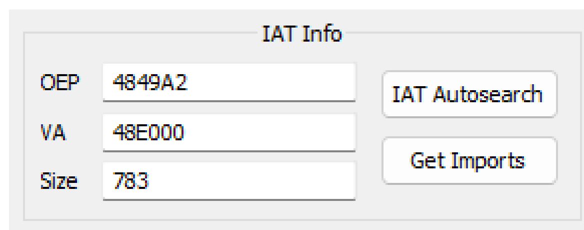
```
006A0041   75 C2    jne 6A0005
006A0043   CD 03    int 3                        COMPLETED :)
006A0045   0000     add byte ptr ds:[eax],al
```

Se ci spostiamo dove avevamo trovato la GetCommandLineA proxata dalla call di SeuROM (all'indirizzo 0x484A16), adesso vedremo una chiamata diretta all'API:

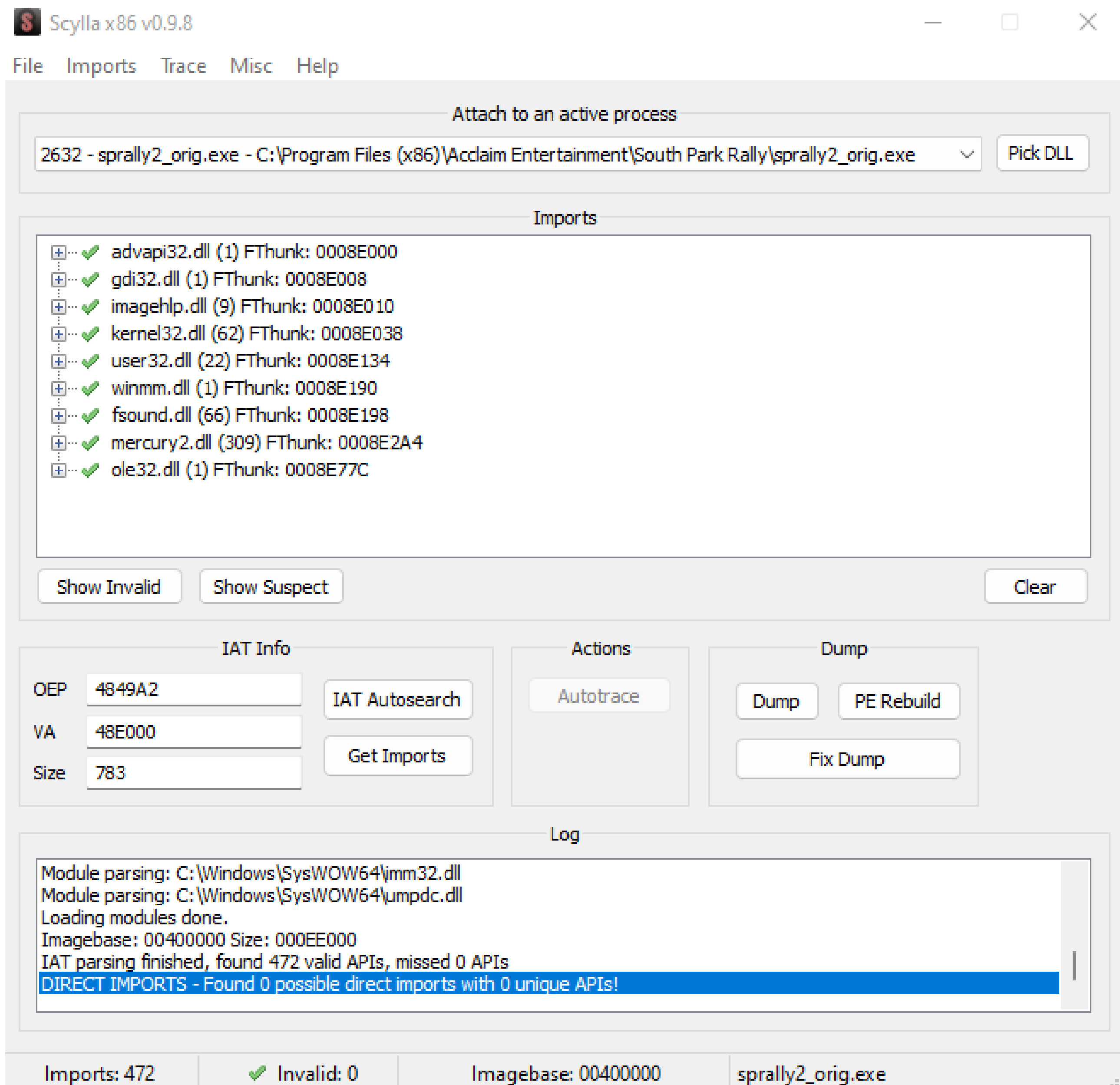
```
00484A0D   59            pop ecx
00484A0E   8975 FC       mov dword ptr ss:[ebp-4],esi
00484A11   E8 59360000   call sprally2_orig.48806F
00484A16   FF15 FCE04800 call dword ptr ds:[<&GetCommandLineA>]
00484A1C   A3 34944B00   mov dword ptr ds:[4B9434],eax
00484A21   E8 17350000   call sprally2_orig.487F3D
00484A26   A3 F8F94A00   mov dword ptr ds:[4AF9F8],eax
00484A2B   E8 C0320000   call sprally2_orig.487CF0
00484A30   E8 02320000   call sprally2_orig.487C37
00484A35   E8 012F0000   call sprally2_orig.48793B
```

Ottimo! Non resta altro che aprire Scylla (icona a forma di S) e dumpare il processo dalla memoria.

Configuriamo Scylla con i dati ottenuti precedentemente, relativi all'OEP, al VA della IAT e alla sua grandezza:



e clicchiamo su Get Imports.



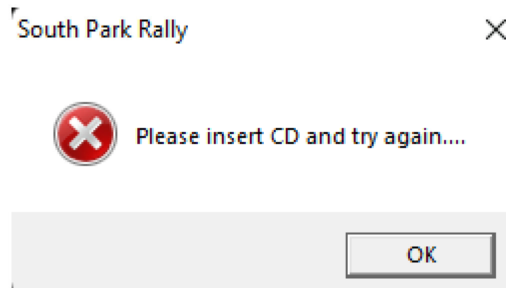
Se trovate API invalide, probabilmente è perché non avete disattivato la modalità di compatibilità di Windows. Altrimenti, se tutte le API risultano valide, potete procedere cliccando su Dump e infine su Fix Dump selezionando il file appena creato.

Ci ritroveremo così ad avere un eseguibile chiamato sprally2_orig_dump_SCY.exe sprotetto da SecuROM!

Completiamo il lavoro:

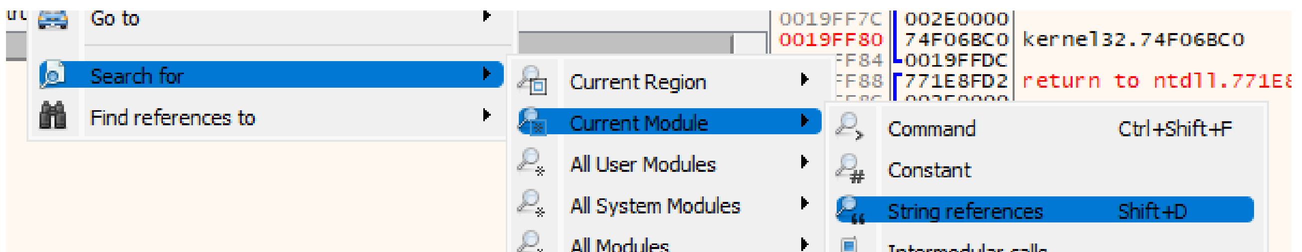
Avviamo il nostro nuovo eseguibile e controlliamo se tutto funziona correttamente.

Rimuoviamo il disco di South Park Rally dal lettore e riproviamo a lanciare il gioco. Otterremo il seguente errore:



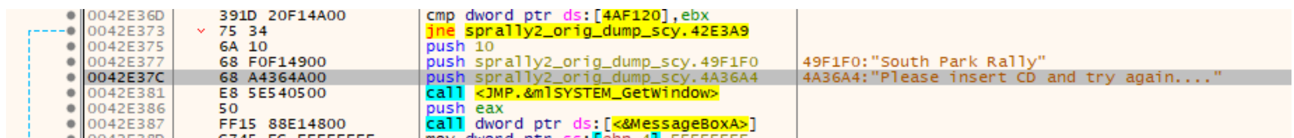
Nonostante questo controllo non sia parte di SecuROM, andiamo a rimuoverlo così potremo conservare il disco originale in un luogo sicuro e continuare a giocare senza doverlo tenere inserito nel lettore.

Apriamo il nuovo eseguibile sprally2_orig_dump_SCY.exe in x32dbg, premiamo su RUN e saremo fermi all'Entry Point del gioco (0x4849A2). A questo punto andiamo alla ricerca della stringa contente il messaggio di errore, cliccando con il destro e selezionando Search for->Current Module->String references.

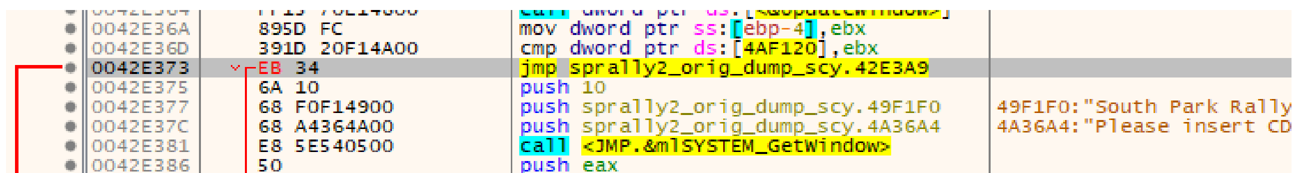


Inseriamo la stringa "Please insert" nel box Search in basso e clicchiamo due volte sul risultato:

Ci troveremo qui:



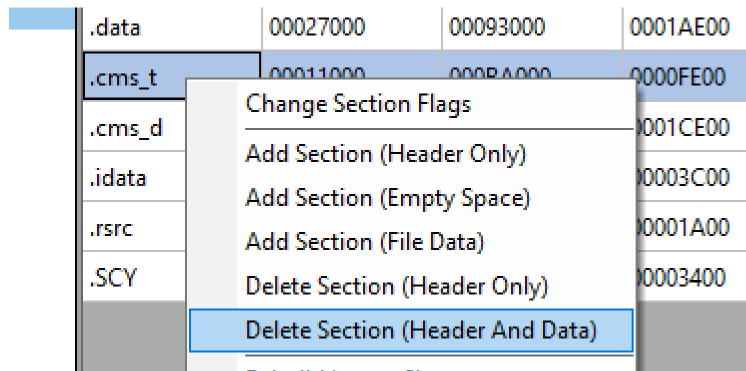
Sarà a questo punto sufficiente modificare il salto condizionale all'indirizzo 0x42E373 trasformandolo in un jmp:



Salviamo la nostra patch andando su File->Patch file… e cliccando su Patch File salviamo il nuovo eseguibile (io l'ho chiamato sprally2_nocd.exe).

Il lavoro non è ancora finito: il nostro eseguibile adesso funzionerà senza cd, ma se proprio vogliamo essere perfezionisti, restano da rimuovere quelle due sections dell'eseguibile che contengono i rimasugli di SecuROM e che ora occupano solo spazio inutilmente.

Apriamo l'eseguibile in CFF Explorer e spostiamoci su "Section Headers [x]". A questo punto selezioniamo le sezioni .cms_t e .cms_d, ovvero dove risiede il codice usato da SecuROM, e clicchiamo con il destro su "Delete Section (Header And Data)":



Alla fine, il nostro eseguibile avrà solo le seguenti sections:

Byte[8]	Dword	Dwor
.text	0008D000	00001
.rdata	00005000	0008E
.data	00055000	00093
.idata	00004000	000E8
.rsrc	00002000	000EC
.SCY	00004000	000EE

Salviamo le modifiche fatte.

Il risultato sarà un eseguibile che funziona senza cd e privo dello spazio occupato da SecuROM (circa 230Kb)!

Abbiamo finito 😊

<u>Credits:</u>

Grazie mille a **Rosario Camarda** per aver trovato e segnalato un paio di errori nel presente documento!

<u>Conclusione:</u>

Spero che anche questo documento sia stato di vostro gradimento. Come avrete potuto notare le differenze tra questa versione di SecuROM e quella precedentemente analizzata (la 4.48.00.0004) sono veramente poche.

In futuro mi piacerebbe provare ad automatizzare il processo qui descritto.

Grazie per la lettura!

Luca