

GAME: Evolva
Protection: Laserlock
Author: Luca D'Amico - V1.0 - 20 Aprile 2022

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 XP VM (ho usato VMware)
- x64dbg (x32dbg)
- Python 3
- Disco di gioco originale (abbiamo bisogno del disco ORIGINALE)

Prima di iniziare:

Laserlock è stata una protezione molto usata durante la fine degli anni '90 e nei primi anni del 2000. Il funzionamento di questa protezione è semplice: alcune API vengono rimpiazzate con una chiamata ad una funzione contenuta nella dll di Laserlock (che prende il nome di giocoarchlib.dll, in questo caso specifico quindi evo32lib.dll), che recupererà l'indirizzo reale dell'API in base alla posizione da cui si è originata la chiamata. Per sconfiggere questa protezione occorrerà ottenere gli indirizzi corretti delle API usate e sostituirli al posto di quello della dll di Laserlock. Purtroppo questo procedimento è reso un po' complicato data la presenza di numerosi CRC in questa libreria.

Iniziamo:

Installate il gioco e aprite Evolva.exe con il debugger (assicuratevi di avere il disco originale ancora inserito), avviate e noterete che tutto funziona correttamente: non sono presenti controlli anti-debugger.

Riavviate il debugger, una volta all'entry point avremo questa situazione:

55	push ebp	Entry
8BEC	mov ebp,esp	
6A FF	push FFFFFFFF	
68 F8827800	push evolva.7882F8	
68 60B17000	push evolva.70B160	
64:A1 00000000	mov eax,dword ptr fs:[0]	
50	push eax	
64:8925 00000000	mov dword ptr fs:[0],esp	
83EC 58	sub esp,58	
53	push ebx	ebx:&
56	push esi	
57	push edi	
8965 E8	mov dword ptr ss:[ebp-18],esp	
FF15 F8D57E00	call dword ptr ds:[%&Call1DLL>]	
33D2	xor edx,edx	
8AD4	mov dl,ah	

La prima call è decisamente sospetta: sarebbe stato lecito aspettarsi una chiamata a GetVersion, ma invece c'è una call ad una funzione chiamata CallDLL nella libreria evo32lib.dll:

```

push esi
push edi
mov dword ptr ss:[ebp-18],esp
call dword ptr ds:[<&Call0LL>]
xor edx,edx
mov dl,ah
mov dword ptr ds:[839F78],edx
mov ecx,eax
and ecx,FF
mov dword ptr ds:[839F74],ecx
nop
nop
nop
nop
nop

```

Proviamo a steppare sul disassembly ed entriamo dentro la funzione per vedere cosa fa. Dopo una serie di nop, arriviamo alla parte realmente interessante:

```

1000104B 50          push eax
1000104C 55          push ebp
1000104D 8BEC       mov ebp,esp
1000104F 50          push eax
10001050 53          push ebx
10001051 51          push ecx
10001052 52          push edx
10001053 56          push esi
10001054 57          push edi
10001055 36:8B45 08  mov eax,dword ptr ss:[ebp+8]
10001059 50          push eax
1000105A E8 82FBFFFF call evo32lib.100018E1
1000105F 66:83C4 04  add sp,4
10001063 3E:8B00     mov eax,dword ptr ds:[eax]
10001066 803D EC680110 01 cmp byte ptr ds:[100168EC],1
1000106D 0F84 0C000000 je evo32lib.1000107F
10001073 36:8945 04  mov dword ptr ss:[ebp+4],eax
10001077 5F          pop edi
10001078 5E          pop esi
10001079 5A          pop edx
1000107A 59          pop ecx
1000107B 5B          pop ebx
1000107C 58          pop eax
1000107D 5D          pop ebp
1000107E C3         ret
1000107F 36:8945 08  mov dword ptr ss:[ebp+8],eax
10001083 5F          pop edi
10001084 5E          pop esi
10001085 5A          pop edx
10001086 59          pop ecx
10001087 5B          pop ebx
10001088 58          pop eax
10001089 5D          pop ebp
1000108A 83C4 04    add esp,4
1000108D C3         ret
1000108E 90          nop

```

Continuiamo a steppare sino a subito dopo la call a 0x10001D5A. Controllate il valore di ritorno, contenuto nel registro EAX:

Hide FPU		
EAX	0076E140	<evolve.&GetVersion>
EBX	7FFD5000	&L"=::=:\\"
ECX	0012FE5C	
EDX	7C91E4F4	<ntdll.KiFastSystemCall1
EBP	0012FF40	
ESP	0012FF24	
EIP	00000000	

Come ci aspettavamo, questa call in origine era una GetVersion 😊

Se continuiamo a steppare, noteremo che il salto condizionale a 0x10001D6D non verrà preso, e la funzione terminerà con il RET situato a 0x10001D7E, quindi la GetVersion sarà chiamata (direttamente dal RET poiché l'indirizzo è stato posizionato sullo stack). Ancora non sappiamo il significato di questo salto, ma presto lo scopriremo.

Tornando al modulo principale, continuiamo a steppare le istruzioni ed entriamo nella seconda chiamata a CallDLL. Nuovamente continuiamo ad eseguire istruzione per istruzione sino a subito dopo la call a 0x10001D5A. Controlliamo nuovamente EAX:

Hide FPU		
EAX	0076E054	<evolve.&GetCommandLineA>
EBX	7FFD7000	&L"=::=:\\"
ECX	0012FE5C	
EDX	7C91E4F4	<ntdll.KiFastSystemCallRet>

Adesso è piuttosto chiaro quello che sta succedendo: queste API, chiamate dal gioco, sono state sostituite tutte con la stessa funzione CallDLL contenuta nella libreria evo32lib.dll. CallDLL controllerà da dove la chiamata si è originata e fornirà la relativa API corretta necessaria al gioco, che verrà eseguita al momento del return.

La prima idea che potrebbe saltarci in mente è quella di trovare un po' di spazio libero e scrivere qualche riga di assembly per parsare il segmento .text alla ricerca di tutte le chiamate a CallDLL, saltarci dentro e una volta ottenuta l'API corrispondente (in EAX) patchare la funzione per tornare al nostro codice e a quel punto modificare l'indirizzo iniziale della call con quello corretto.

Purtroppo questo non funzionerà...

Provate a mettere un breakpoint a 0x10001D5F (subito dopo la call che recupera l'API corretta), premete RUN sul debugger ogni volta che arriverà a quell'indirizzo e dopo un po' il gioco andrà in crash...

Laserlock effettua controlli CRC su questo codice e se vengono rilevate modifiche (patch, hook e breakpoint software) verranno ad un certo punto recuperate API errate.

Possiamo usare i breakpoint hardware (anche se è possibile usarne al massimo 4 contemporaneamente, essi non modificano il codice) per fermarci all'indirizzo giusto e correggere le call per farle puntare alle funzioni corrette, ma c'è un ulteriore problema:

Laserlock controlla anche il segmento .text e se rileva modifiche (come ovviamente i byte che andremo a cambiare per sistemare le call) anche in questo caso il risultato sarà il crash del gioco.

Facciamo un attimo il punto della situazione:

- 1) Sappiamo che le chiamate alle API sono state rimpiazzate tutte con chiamate alla stessa funzione CallDLL.
- 2) Sappiamo che CallDLL dopo le opportune verifiche recupererà l'API corretta in base alla posizione di origine della chiamata (presente sullo stack)
- 3) Il codice di CallDLL è tenuto sotto controllo per rilevare eventuali modifiche
- 4) Il codice del segmento .text del gioco viene anche esso monitorato per evitare che possiamo patchare e aggiustare le call
- 5) A causa del punto 3 e 4 NON possiamo usare i breakpoint software e NON possiamo applicare nessuna patch al gioco
- 6) Ancora dobbiamo capire il significato del salto condizionale che avviene dopo che le API vengono recuperate

Situazione un tantino complessa eh? Benvenuti nel mondo del reverse engineering 😊

Partiamo dal punto 6: una volta compreso questo ultimo quesito, possiamo pensare ad un modo per risolvere tutto il resto.

Il metodo più pratico per capire la differenza tra i due RET è quello di mettere un breakpoint hardware sull'indirizzo dell'ultimo, ovvero a 0x10001D8D:

10001D8A	83C4 04	add esp, 4
10001D8D	C3	ret
10001D8E	90	nop
10001D8F	90	nop
10001D90	90	nop

Appena l'esecuzione sarà bloccata, continuate a steppare dentro il codice dell'API sino a ritornare al modulo principale. Una volta lì, salite su di qualche riga e noterete che la chiamata a CallDLL è stata modificata (a 0x6E9764):

006E9760	31	push ecx
006E9761	8943 38	mov dword ptr ds:[ebx+38],eax
006E9764	E8 47430400	call <JMP.&GetFileVersionInfoA>
006E9769	85C0	test eax,eax
006E976B	75 0E	jne evolva.6E977B

Non si tratta della solita call, infatti entrandoci dentro troveremo questo:

0072DAB0	FF25 B4E27600	jmp dword ptr ds:[<&GetFileVersionInfoA>]
----------	---------------	---

Riavviando il debugger e tornando a questo stesso indirizzo, troviamo:

0072DAB0	FF15 F8D57E00	call dword ptr ds:[<&CallDLL>]
----------	---------------	--------------------------------

La call è diventata un jump.

Cosa significa questo? Il salto condizionale che stavamo analizzando decide se l'API attuale deve essere raggiunta tramite una call o un jump!

Dobbiamo fare particolarmente attenzione a questo, poiché quando sistemere le call, quelle che seguono quel salto condizionale andranno ulteriormente modificate in dei jump!

A questo punto abbiamo tutto ciò che ci serve sapere. Ricordandoci che NON possiamo patchare nulla e NON possiamo usare breakpoint software, sfrutteremo i breakpoint hardware in modo creativo. Ci faremo loggare l'indirizzo dell'API richiesta, l'indirizzo da dove viene effettuata la chiamata e infine il valore che stabilisce se l'API deve essere raggiunta tramite call o jump. Successivamente scriveremo un piccolo script in python che patcherà il binario, liberandolo così da Laserlock.

Questo è quello che faremo:

- 1) Scriveremo qualche riga di assembly per parsare il segmento text alla ricerca delle chiamate da sistemare
- 2) Una volta trovata una chiamata a CallDLL ci salteremo dentro
- 3) Useremo il primo breakpoint hardware all'indirizzo 0x10001D5F (subito dopo la call che recupera l'API, dentro la funzione CallDLL) per farci loggare i dati necessari per sistemare la chiamata
- 4) Useremo il secondo breakpoint hardware sul primo RET per far tornare l'esecuzione al nostro codice assembly (in questo caso l'API è raggiunta tramite call)
- 5) Useremo il terzo breakpoint hardware sul secondo RET per far tornare l'esecuzione al nostro codice assembly (in questo caso l'API è raggiunta tramite jump)
- 6) Una volta loggati tutti i dati che ci servono, con qualche riga di python patcheremo "a freddo" l'eseguibile del gioco. A questo punto possiamo rimuovere anche la dipendenza da evos2lib.dll

La prima cosa da fare è trovare un po' di spazio libero per il nostro codice assembly, io l'ho posizionato a 0x350000. Quindi andiamo su Memory Map, selezioniamo il blocco che parte da 0x350000 e clicchiamo con il destro su Set Page Memory Rights e per finire selezioniamo FULL ACCESS e su Set Rights.

Posizioniamoci a quell'indirizzo nella CPU View e scriviamo attentamente il seguente codice:

00350000	B9 00104000	mov ecx,evolva.401000
00350005	8139 FF15F8D5	cmp dword ptr ds:[ecx],D5F815FF
00350008	75 0E	jne 35001B
0035000D	890D 50003500	mov dword ptr ds:[350050],ecx
00350013	FFE1	jmp ecx
00350015	8B0D 50003500	mov ecx,dword ptr ds:[350050]
00350018	41	inc ecx
0035001C	81F9 00E07600	cmp ecx,<evolva.&?xml_New@YAPAVXM_HANDLE@XZ>
00350022	75 E1	jne 350005
00350024	90	nop
00350025	0000	add byte ptr ds:[eax],a1

La prima riga assegna ad ecx l'indirizzo iniziale del segmento .text

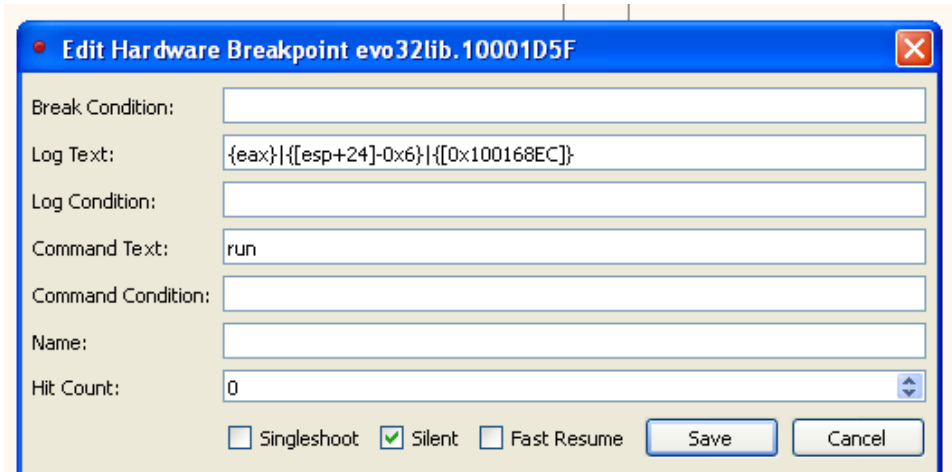
Il CMP controlla se la dword contenuta all'indirizzo attuale in ecx corrisponda effettivamente una call a CallDLL (se notate i byte relativi alla dword sono scritti al contrario, poiché l'ordine dei byte è little endian). Se non c'è corrispondenza, allora verrà incrementato ecx di 1, per andare al prossimo indirizzo e verrà effettuato un controllo per vedere se siamo arrivati all'indirizzo finale del segmento .text. Se invece la dword presente all'indirizzo che stiamo analizzando è una chiamata a CallDLL, prima salveremo l'indirizzo attuale su della memoria libera (ho scelto 0x350050) e poi ci salteremo dentro, con il jump ecx a 0x350013. Una volta impostati correttamente i breakpoint hardware dentro CallDLL, faremo in modo di ritornare a 0x350015, dove l'indirizzo di ecx sarà ripristinato per poter continuare.

Premete con il destro su 0x350000 e cliccate su Set New Origin Here, in modo da dire al debugger che vogliamo far partire l'esecuzione da questo indirizzo. Mettiamo anche un breakpoint su 0x350024 per fermarci qui una volta che tutte le API saranno state loggate. Non usate int3, altrimenti il programma andrà in crash.

Prima di poter avviare il nostro codice assembly, andiamo ad impostare correttamente i breakpoint hardware su CallDLL:

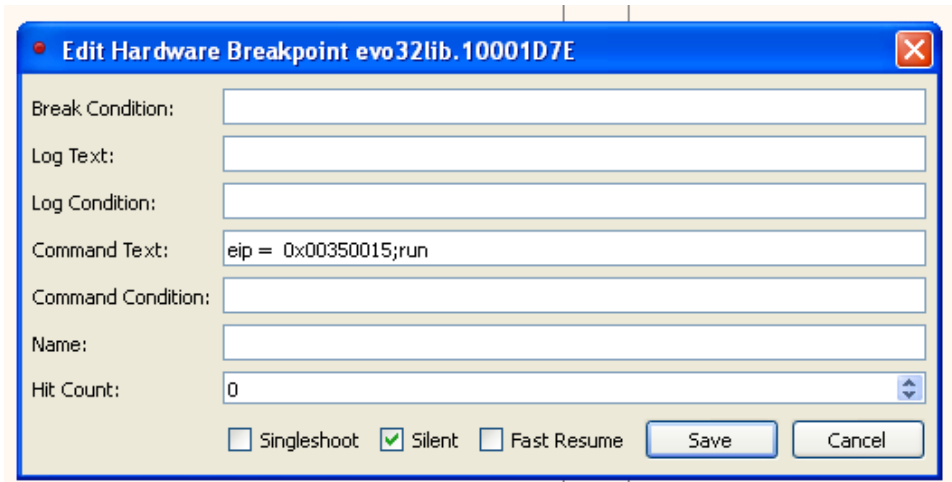
●	1000104B	50	push eax
●	1000104C	55	push ebp
●	1000104D	8BEC	mov ebp,esp
●	1000104F	50	push eax
●	10001050	53	push ebx
●	10001051	51	push ecx
●	10001052	52	push edx
●	10001053	56	push esi
●	10001054	57	push edi
●	10001055	36:8B45 08	mov eax,dword ptr ss:[ebp+8]
●	10001059	50	push eax
●	1000105A	E8 82FBFFFF	call evo321ib.100018E1
●	1000105F	66:83C4 04	add sp,4
●	10001063	3E:8B00	mov eax,dword ptr ds:[eax]
●	10001066	803D EC680110 01	cmp byte ptr ds:[100168EC],1
●	1000106D	0F84 0C000000	je evo321ib.10001D7F
●	10001073	36:8945 04	mov dword ptr ss:[ebp+4],eax
●	10001077	5F	pop edi
●	10001078	5E	pop esi
●	10001079	5A	pop edx
●	1000107A	59	pop ecx
●	1000107B	5B	pop ebx
●	1000107C	58	pop eax
●	1000107D	5D	pop ebp
●	1000107E	C3	ret
●	1000107F	36:8945 08	mov dword ptr ss:[ebp+8],eax
●	10001083	5F	pop edi
●	10001084	5E	pop esi
●	10001085	5A	pop edx
●	10001086	59	pop ecx
●	10001087	5B	pop ebx
●	10001088	58	pop eax
●	10001089	5D	pop ebp
●	1000108A	83C4 04	add esp,4
●	1000108D	C3	ret

Dobbiamo mettere il primo breakpoint hardware su 0x10001D5F per poter loggare: l'API recuperata, l'indirizzo di origine della chiamata (nel segmento .text), e il byte contenuto in 0x100168EC che ci dirà se l'API deve essere raggiunta tramite call o jump. Per fare tutto ciò, clicchiamo l'indirizzo con il destro e scegliamo Brakpoint->Set Hardware on Execution, dopodiché andiamo nella tab Breakpoint, clicchiamo con il destro sul breakpoint appena inserito e clicchiamo su Edit. Configuriamolo in questo modo per poter loggare ciò che ci serve:



Se vi state chiedendo il motivo di quel 0x6 sottratto dall'indirizzo di ritorno, ricordatevi che a noi serve l'indirizzo da dove la chiamata è partita. In [ESP+24] è presente l'indirizzo della prossima istruzione DOPO la chiamata, quindi dobbiamo sottrarre 6 byte da esso per ottenere il valore che ci serve. La chiamata a CallDLL infatti è grande 6 byte. Il comando "run" in Command Text ci serve per far riprendere l'esecuzione automaticamente dopo aver effettuato il log.

Adesso impostiamo i due breakpoint hardware sui RET, rispettivamente a 0x10001D7E e 0x10001D8D. Poiché l'esecuzione dovrà riprendere dal nostro codice assembly, entrambi vanno configurati così:



Siamo pronti per eseguire il nostro codice, torniamo a 0x350000 e clicchiamo su RUN. Una volta terminata l'esecuzione saremo fermi sull'indirizzo 0x350024:

```

00350000  B9 00104000  mov ecx, evolv.401000
00350005  8139 FF15F8D5  cmp dword ptr ds:[ecx], D5F815FF
00350008  v 75 0E      jne 350018
00350000  890D 50003500  mov dword ptr ds:[350050], ecx
00350013  v FFE1      jmp ecx
00350015  8B0D 50003500  mov ecx, dword ptr ds:[350050]
00350018  41          inc ecx
0035001C  81F9 00E07600  cmp ecx, <evolv.&?Xm_New@YAPAVXm_HANDLE@XZ>
00350022  ^ 75 E1      jne 350005
00350024  90          nop
00350025  0000      add byte ptr ds:[eax], a1
00350027  0000

```

Perfetto, clicchiamo sul tab Log e avremo tutte le chiamate a CallDLL loggate 😊

```
CPU | Log | Notes | Break
76E1C8|59E97D|0
76E1CC|59EA12|0
76E1C8|59F3F8|0
76E194|59F533|0
76E1CC|59F542|0
76E1C8|5A3F8D|0
76E1CC|5A4022|0
76E1C4|5A91C2|0
76E1C8|5B2ADC|0
76E194|5B2BEA|0
76E1CC|5B2BFD|0
76E1C8|5B8BFC|0
```

Copiamole su un documento chiamato calls.txt ed iniziamo a scrivere uno script in python per poter patchare l'eseguibile.

Il codice che ho scritto è il seguente:

```
class Patch:
    def __init__(self, api_addr, call_addr, is_jump):
        self.api_addr = api_addr
        self.call_addr = call_addr
        self.is_jump = is_jump

    def get_api_addr(self):
        return self.api_addr

    def get_call_addr(self):
        return self.call_addr

    def is_jump(self):
        return self.is_jump

def read_patches_from_file(file_path):
    f = open(file_path, 'r')
    lines = f.readlines()
    f.close()
    return lines

def parse_patches(txt_patches, imagebase):
    patches = []
    for txt_patch in txt_patches:
        patch_parts = txt_patch.split('|')
        patches.append(Patch(int(patch_parts[0], 16),
int(patch_parts[1], 16) - imagebase, bool(int(patch_parts[2]))))
    return patches

def apply_patches_to_file(file_path, patches):
    f = open(file_path, 'r+b')
    for patch in patches:
```

```

f.seek(patch.get_call_addr() + 0x2)
f.write(patch.get_api_addr().to_bytes(4, "little"))
if(patch.is_jump()):
    f.seek(patch.get_call_addr() + 0x1)
    f.write(bytes([0x25]))
f.close()

```

```

if __name__ == "__main__":
    txt_patches = read_patches_from_file('calls.txt')
    patches = parse_patches(txt_patches, 0x400000)
    apply_patches_to_file("Evolva.exe", patches)

```

Il tutto è molto semplice: vengono letti i dati del log dal file calls.txt, le varie parti di ogni singola linea vengono divise tramite il carattere '|', e una dopo l'altra vengono applicate al segmento .text. Se l'API deve essere raggiunta tramite un jump, verrà anche patchato il byte corrispondente trasformando quella call in un jmp (sostituendo l'opcode 0x15 con 0x25).

Inoltre ricordate che l'ibase va sottratta dall'indirizzo delle chiamate da patchare, in modo tale da ottenere l'offset corretto del segmento .text nel file.

A questo punto possiamo aprire il nostro nuovo eseguibile nel debugger e fare un confronto:

<pre> 0070B285 . FF15 40E17600 0070B28B . 33D2 0070B28D . 8AD4 0070B28F . 8915 789F8300 0070B295 . 8BC8 0070B297 . 81E1 FF000000 0070B29D . 890D 749F8300 0070B2A3 . C1E1 08 0070B2A6 . 03CA 0070B2A8 . 890D 709F8300 0070B2AE . C1E8 10 0070B2B1 . A3 6C9F8300 0070B2B6 . 33F6 0070B2B8 . 56 0070B2B9 . E8 2DB10000 0070B2BE . 59 0070B2BF . 85C0 0070B2C1 . 75 08 0070B2C3 . 6A 1C 0070B2C5 . E8 B0000000 0070B2CA . 59 0070B2CB > 8975 FC 0070B2CE . E8 F2A40000 0070B2D3 . FF15 54E07600 </pre>	<pre> call dword ptr ds:[&&GetVersion] xor edx,edx mov dl,ah mov dword ptr ds:[839F78],edx mov ecx,eax and ecx,FF mov dword ptr ds:[839F74],ecx shl ecx,8 add ecx,edx mov dword ptr ds:[839F70],ecx shr eax,10 mov dword ptr ds:[839F6C],eax xor esi,esi push esi call evolvva.7163EB pop ecx test eax,eax jne evolvva.70B2CB push 1C call evolvva.70B37A pop ecx mov dword ptr ss:[ebp-4],esi call evolvva.7157C5 call dword ptr ds:[&&GetCommandLineA] </pre>
--	---

Dove prima erano presenti le chiamate a CallDLL, adesso ci sono le API corrette.

Complimenti, avete rimosso Laserlock da questo eseguibile 😊

Tuttavia il lavoro non è ancora finito...

Cutscenes e chiavi di registro:

Se avviate il gioco senza avere il CD nel lettore, noterete che i filmati iniziali non vengono riprodotti. Effettivamente controllando nella cartella del gioco ci accorgiamo che non sono presenti. Inseriamo nuovamente il CD di Evolva e copiamo la cartella FMV nella directory di installazione del gioco.

Apriamo regedit e modifichiamo il valore di questa chiave:

HKEY_LOCAL_MACHINE\SOFTWARE\Computer Artworks\Evolva\1.0\FMVDir

In

.\FMV

In questo modo i filmati verranno letti dalla cartella di Evolva.

Dipendenza da evo32lib.dll:

Tutto funziona alla perfezione, ma il nostro binario è ancora dipendente dalla libreria evo32lib.dll usata da Laserlock e che oramai non necessita più.

Apriamo quindi Evolve.exe con CFF Explorer, clicchiamo su Import Directory, selezioniamo evo32lib.dll e scegliamo Delete Import Descriptor:

Import Name	Ordinal	Address	Signature
szAnsi	(nFunctions)	Dword	Dword
evo32LIB.dll	1	003EC604	00000000
KERNEL32.dll			00000000
USER32.dll			00000000
GDI32.dll	6	003EC60C	00000000
ADVAPI32.dll	6	003EC5E4	00000000
ole32.dll	5	003EC8C4	00000000
WINMM.dll	9	003EC89C	00000000
4xm_sdk.dll	2	003EC5D8	00000000

Salviamo il tutto e avremo finalmente un binario completamente libero da Laserlock 😊

Conclusioni:

Spero che questo documento sia stato di vostro gradimento. Reversare questi vecchissimi DRM risulta essere una attività parecchio istruttiva e divertente.

Luca