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.

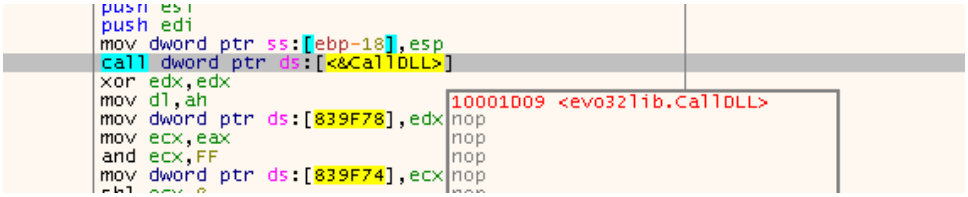
Cosa ci serve:
- Windows XP VM (ho usato VMware)
- x64dbg (x32dbg)
- Python 3
- Disco di gioco originale (abbiamo bisogno del disco ORIGINALE)

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), avviatelo e noterete che tutto funziona correttamente: non sono presenti controlli anti-debugger.
Riavviate il debugger, una volta all'entry point avremo questa situazione:

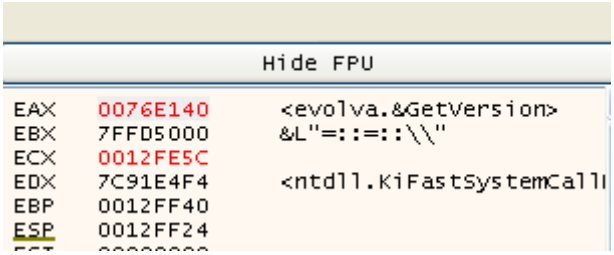
```
55              push ebp                                Entry
8BEC            mov ebp,esp
6A FF           push FFFFFFFF
68 F8827800     push evolva.7882F8
68 60B17000     push evolva.70B160
64:A1 00000000  mov eax,dword ptr fs:[0]
50              push eax
64:8925 00000000 mov dword ptr fs:[0],esp
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:[<&CallDLL>]
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:

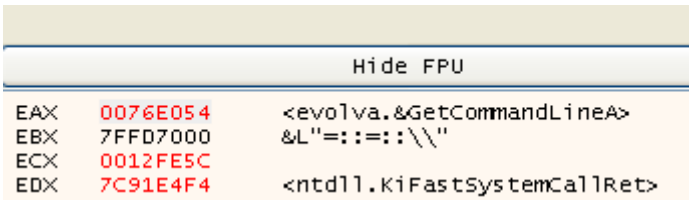


Proviamo a steppare sul disassembly ed entriamo dentro la funzione per vedere cosa fa.
Dopo una serie di nop, arriviamo alla parte realmente interessante:

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



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:



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:

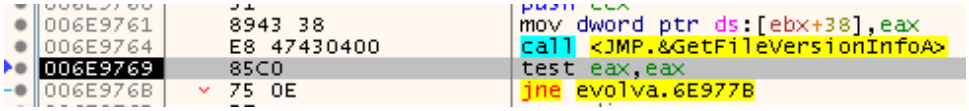
1) Sappiamo che le chiamate alle API sono state rimpiazzate tutte con chiamate alla stessa funzione CallDLL.
2) Sappiamo che CallDLL dopo le opportune verifiche recupererà l'API corretta in base alla posizione di origine della chiamata (presente sullo stack)
3) Il codice di CallDLL è tenuto sotto controllo per rilevare eventuali modifiche
4) Il codice del segmento .text del gioco viene anche esso monitorato per evitate 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.

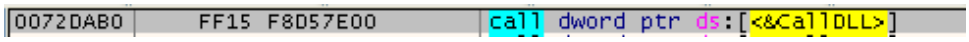
Il metodo più pratico per capire la differenza tra i due RET è quello di mettere un breakpoint hardware sull'indirizzo dell'ultimo, ovvero a 0x10001D8D:

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):



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

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



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!

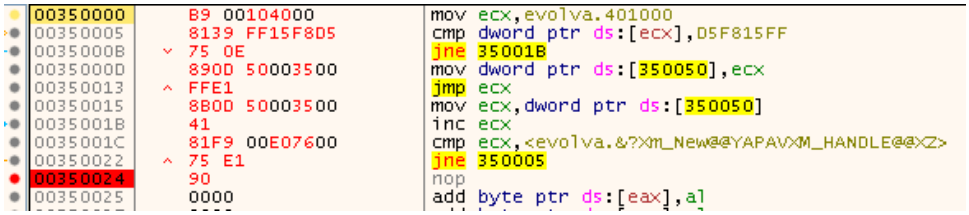
Dobbiamo fare particolarmente attenzione a questo, poiché quando sistemeremo le call, quelle che seguono quel salto condizionale andranno ulteriormente modificate in dei jump!

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 cosi 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 evo32lib.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:



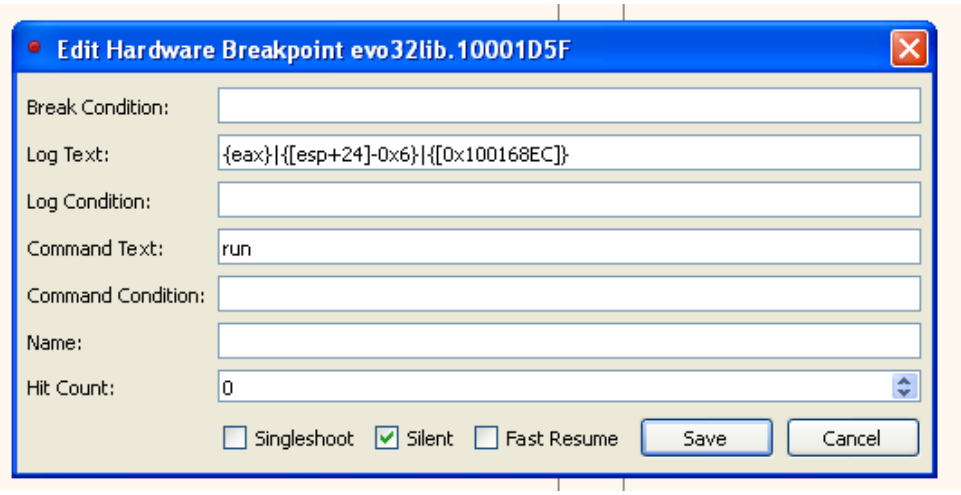
La prima riga assegna ad ecx l'indirizzo iniziale del segmento .text

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:

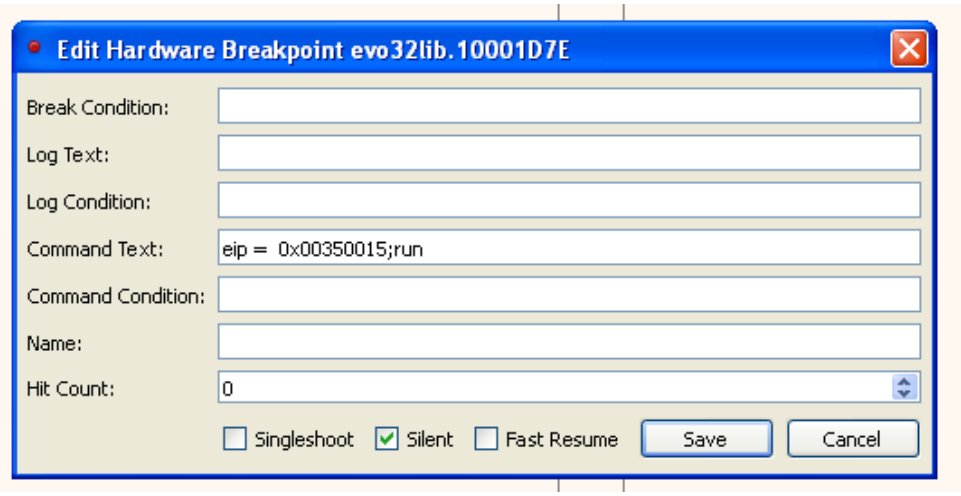
```
10001D4B    50              push eax
10001D4C    55              push ebp
10001D4D    8BEC            mov ebp,esp
10001D4F    50              push eax
10001D50    53              push ebx
10001D51    51              push ecx
10001D52    52              push edx
10001D53    56              push esi
10001D54    57              push edi
10001D55    36:8B45 08      mov eax,dword ptr ss:[ebp+8]
10001D59    50              push eax
10001D5A    E8 82FBFFFF     call evo32lib.100018E1
10001D5F    66:83C4 04      add sp,4
10001D63    3E:8B00         mov eax,dword ptr ds:[eax]
10001D66    803D EC680110 01  cmp byte ptr ds:[100168EC],1
10001D6D    0F84 0C000000   je evo32lib.10001D7F
10001D73    36:8945 04      mov dword ptr ss:[ebp+4],eax
10001D77    5F              pop edi
10001D78    5E              pop esi
10001D79    5A              pop edx
10001D7A    59              pop ecx
10001D7B    5B              pop ebx
10001D7C    58              pop eax
10001D7D    5D              pop ebp
10001D7E    C3              ret
10001D7F    36:8945 08      mov dword ptr ss:[ebp+8],eax
10001D83    5F              pop edi
10001D84    5E              pop esi
10001D85    5A              pop edx
10001D86    59              pop ecx
10001D87    5B              pop ebx
10001D88    58              pop eax
10001D89    5D              pop ebp
10001D8A    83C4 04         add esp,4
10001D8D    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:

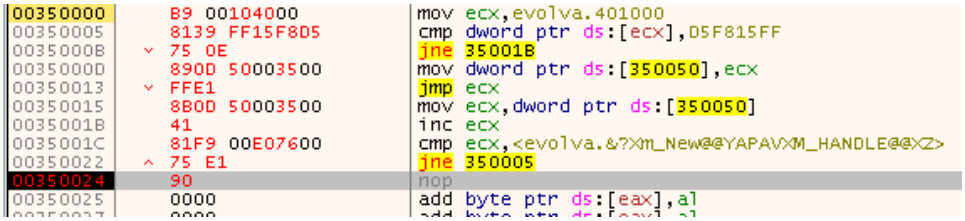


Se vi state chiedendo il motivo di quel 0x6 sottratto dall'indirizzo di ritorno, ricordatevi che a noi serve l'indirizzo da dove la chiamata è partita. In [ESP+24] è presente l'indirizzo della prossima istruzione DOPO la chiamata, quindi dobbiamo sottrarre 6 byte da esso per ottenere il valore che ci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

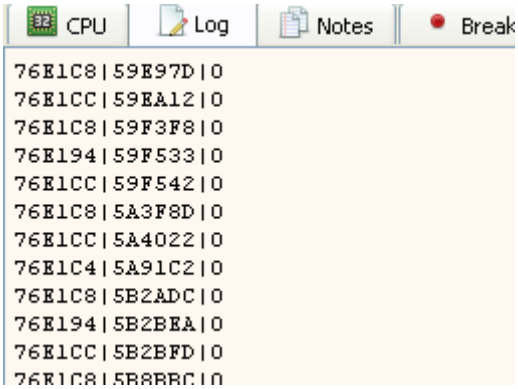
Adesso impostiamo i due breakpoint hardware sui RET, rispettivamente a 0x10001D7E e 0x10001D8D. Poiché l'esecuzione dovrà riprendere dal nostro codice assembly, entrambi vanno configurati cosi:



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



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



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:

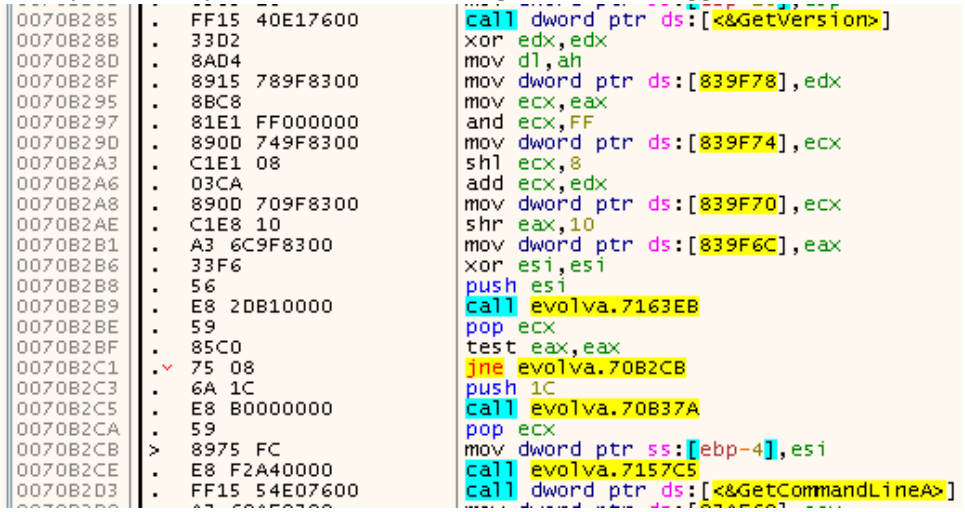
```
class Patch:
    def __init__(self, api_addr, call_addr, is_jmp):
        self.api_addr = api_addr
        self.call_addr = call_addr
        self.is_jmp = is_jmp

    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_jmp


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

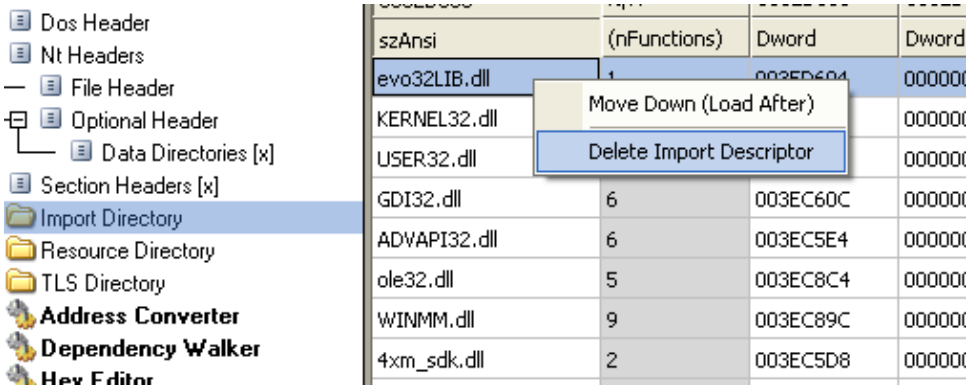
A questo punto possiamo aprire il nostro nuovo eseguibile nel debugger e fare un confronto:



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 accordiamo 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 orami non necessita più.
Apriamo quindi Evolva.exe con CFF Explorer, clicchiamo su Import Directory, selezioniamo evo32lib.dll e scegliamo Delete Import Descriptor:



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