

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

You will need:

- Windows XP VM (I used VMware)
- x64dbg (x32dbg)
- Python 3
- Original game disc (you need the ORIGINAL, otherwise this will not work)

Before you start:

Laserlock was a widely used copy-protection scheme during the late 1990s and early 2000s. The way it works is simple: some API calls will be replaced with a call to a function located in a DLL that is called like [gamearchlib].dll (in our case it is evo32lib.dll), that will retrieve the real API address, by checking the address where the call originated from.

To defeat this protection, we will need to obtain all the correct API addresses used by the game and replace all the calls to the Laserlock DLL inside the game's binary. Unfortunately, this is made a bit complicated due to numerous CRC checks in this library.

Let's begin:

Install the game and load Evolva.exe into the debugger (make sure to keep the original disc in the drive), click RUN and the game should start just like usually: there isn't any anti-debugging checks in place.

Reload the debugger. Once at the entry point of the module, you will see this:

. 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	

The first call is very suspicious: it would have been reasonable to expect a call to GetVersion, but instead we have a call to a function called CallDLL in the evo32lib.dll library:

```

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

```

Let's try to proceed by stepping within the disassembly and by entering into the call to check what's happening there.

After some NOPs we'll get to the interesting part:

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
10001D8E	90	nop

Let's keep stepping until we reach the instruction right after the call located at 0x10001D5A. Now check the return value stored in the EAX register:

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

As expected, this call was originally a GetVersion call 😊

If we continue stepping, we will notice that the conditional jump located at 0x10001D6D will not be taken, and the function will end at the RET located at 0x10001D7E. Afterwards GetVersion will be called (directly from the RET, since its address was moved on the stack)

We still don't know the meaning of that conditional jump, but we will be back to this soon. When we will be back at the main module, we can continue stepping over all the instructions until we can follow the second call to CallDLL.

Again, let's keep stepping over all the instructions until we reach the one just right after the call located at 0x10001D5A. Then check the EAX register, again:

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

Now it is quite clear what's happening: these APIs, called by the game, were all replaced with the same function called CallDLL located in the evo32lib.dll. The CallDLL function will check from where the call was originated and will provide the correct API address needed by the game, which will then be executed using the RET instruction (because that address was moved on the stack at the right position).

The first thing that would come to our minds is to find some free space to write a simple assembly routine to parse the .text segment finding all the calls to CallDLL, jumping into each of them and once we retrieved the correct API address (stored in EAX), patch the code to jump back to our assembly routine and finally fix the initial call with the correct address.

Unfortunately, this will not work...

Let's try to put breakpoint at 0x10001D5F (right after the call that retrieves the correct API address), click RUN on the debugger each time it will break at that address and after a couple of iterations, the game will crash...

Why? Because Laserlock will do some CRC checks on the game's binary in memory, and if it will notice any modification (like patches, hooks and software breakpoints) it will return wrong API addresses.

We can use hardware breakpoints (even if it only is possible to use at max 4 of them at the same time. They don't alter the code at all, so they will not be detected!) to break at the right address and to fix the calls to make them point to the right functions, but there is another problem:

Laserlock will also check the .text segment and if any patch is detected (like, obviously our patched bytes needed to fix the calls) the result will be a game crash.

Let's take a moment to assess the overall situation:

- 1) We know that the original API calls were all replaced with the same function called CallDLL inside the Laserlock DLL
- 2) We know that CallDLL, after its checks, will retrieve the correct API based on the address where the original call originated from (this value is retrieved from the stack).
- 3) The CallDLL code is checked against any modification.
- 4) The .text segment code is also checked against any modification preventing us to fix the calls by patching them.
- 5) Due to 3) and 4), we CAN'T use software breakpoints and we CAN'T patch anything.
- 6) We still don't know what's the purpose of the conditional jump that takes place just after the call that retrieves the original API addresses

This situation is a bit complex, huh? Welcome to the reverse engineering world 😊

Let's start from point number 6: once we will understand what's going on there, we can think of a way to solve everything else.

The most practical method to figure out the difference between these two RETs, is to put a hardware breakpoint on the address of the last one (0x10001D8D):

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

Once the execution will break, continue by stepping inside the API code until the return to the main module is reached. Once there, scroll up a few lines above and you will notice that the call to CallDLL has been modified (at 0x6E9764):

006E9760	54	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

It isn't the usual call, indeed if we follow it, we will find this:

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

Let's restart the debugger and go straight to that address, we will see:

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

During the execution, that call turned into a jump.

What does this mean? The conditional jump we were analysing decides whether the current API must be reached via a call or a jump!

We need to pay attention to this behaviour, because when we will fix all the calls, the ones that will take this conditional jump, needs to be turned into jumps!

We now have everything we need to know. Keep in mind that we CANNOT patch anything and we CANNOT use software breakpoints. Instead, we will use hardware breakpoints in a creative way. We will log the address of the correct API, the address where the call originated from and the boolean value that decides whether the current API should be reached by a call or by a jump. Next, we will write a little Python script that will patch the binary, removing Laserlock.

This is what we will do:

- 1) We will write some assembly code to parse the .text segment, looking for all the calls to fix
- 2) Once we will find a call to CallDLL we will jump at that address
- 3) We will set the first hardware breakpoint at 0x10001D5F (right after the call that will retrieve the correct API, inside the CallDLL function), in order to get a log of the data that we need to fix the call later.
- 4) We will set the second hardware breakpoint on the first RET, in order to alter the execution flow to jump back to our assembly code
- 5) We will set the third hardware breakpoint on the second RET, in order to alter the execution flow to jump back to our assembly code (remember that if this RET is executed, the API will be reached from a JMP)
- 6) Once we will have all the needed data logged, using some Python code we will “cold-patch” the game’s executable. Finally, we can remove the evo32lib.dll dependency from the EXE file.

The first thing to do is to look for some free space where we can put our assembly code. I’ve found a nice code cave starting from 0x350000. Let’s click on the Memory Map tab, select the right entry (the one that starts from 0x350000) and right-click and select “Set Page Memory Rights”. Then select FULL ACCESS and click on Set Rights.

Let’s return to the CPU View Tab, go to 0x350000, and carefully write this code:

00350000	B9 00104000	mov ecx, evolvva.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, <evolvva.&?xm_New@YAPAVxm_HANDLE@@@Z>
00350022	^ 75 E1	jne 350005
00350024	90	nop
00350025	0000	add byte ptr ds:[eax], al

The first line stores the starting address of the .text segment in the ecx register.

The CMP opcode checks if the dword pointed to by the address currently stored in ecx is a CallDLL call (if you pay attention to the bytes compared, you will notice that they are written in reverse, as the byte order is little endian). If there is no match, then ecx will be incremented by 1 in order to point to the next address.

Then there's another CMP, to check if we arrived at the end of .text segment. If there is a match, and we are in front of a call to CallDLL, we will first store the current address in some free memory (I've chosen 0x350050) and then we will jump into it, with the jmp ecx located at 0x350013. Once we will correctly set the hardware breakpoints inside CallDLL, we will jump back exactly at 0x350015 and we will restore the previously saved address in ecx in order to continue our parsing of the .text segment.

Let's right-click on 0x350000 and select Set New Origin Here, so we can tell the debugger that it has to start from this address when we will hit RUN. Let's put a breakpoint at 0x350024, so we can break here once all the APIs are logged. DO NOT USE int 3 here, otherwise the game will crash.

Before we launch our assembly code, we have to properly set the hardware breakpoints in CallDLL:

10001048	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

We have to set the first hardware breakpoint at 0x10001D5F in order to log the retrieved API, the address where the call was originated (located in the .text segment) from and the byte stored at 0x100168EC (which will tell us if the API should be reached from a call or a jump).

So, let's right-click that address and select Breakpoint->Set Hardware on Execution, then head over to the Breakpoint tab, right-click the newly created breakpoint and select Edit. Let's configure it this way in order to log all the data we need:

**Edit Hardware Breakpoint evo32lib.10001D5F**

Break Condition:

Log Text:

Log Condition:

Command Text:

Command Condition:

Name:

Hit Count:

☐ Singleshoot ☒ Silent ☐ Fast Resume

If you are wondering why 0x6 is subtracted from the return address remember that we need the address from which the call originated. In [ESP+24] the address of the next instruction AFTER the call is stored so we need to subtract 6 bytes in relation to this address in order to get the correct one we need. The call to CallDLL code is indeed 6 bytes long. The “run” command in Command Text is needed to automatically resume the execution after having logged the data we are looking for.

It's time to set the last two hardware breakpoints on the RET instructions respectively located at 0x10001D7E and 0x10001D8D. Since we want the execution flow to jump back to our assembly code, we need to configure both of them in this way:

**Edit Hardware Breakpoint evo32lib.10001D7E**

Break Condition:

Log Text:

Log Condition:

Command Text:

Command Condition:

Name:

Hit Count:

☐ Singleshoot ☒ Silent ☐ Fast Resume

We are ready to execute our code! Let's go to the 0x350000 address and click RUN. Once the execution is completed, we will be stopped at 0x350024:

00350000	B9 00104000	mov ecx, evolvva.401000
00350005	8139 FF15F8D5	cmp dword ptr ds:[ecx], D5F815FF
00350008	75 0E	jne 350018
0035000D	890D 50003500	mov dword ptr ds:[350050], ecx
00350013	FEE1	jmp ecx
00350015	8B0D 50003500	mov ecx, dword ptr ds:[350050]
00350018	41	inc ecx
0035001C	81F9 00E07600	cmp ecx, <evolvva.&?Xm_New@@YAPAVXM_HANDLE@@@Z>
00350022	75 E1	jne 350005
00350024	90	nop
00350025	0000	add byte ptr ds:[eax], al
00350027	0000	add byte ptr ds:[eax], al

Perfect, let's click on the Log tab and you will find all the calls to CallDLL logged there 😊

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			

Let's copy all the entries into a new document called calls.txt, and we are finally ready to write a Python script to patch the executable.



This is the Python code that I've written for our purpose:

```
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)
```

Everything here is really simple: we will read all the data from calls.txt. Every single line will be split using '|' character as a separator, and every patch will be applied to the .text segment.  
If the API should be reached by a jump, we will patch the corresponding byte turning that call in a jmp (by replacing the 0x15 opcode with 0x25).

Please keep in mind that we have to subtract the imagebase address (0x400000) from the address of the calls we will patch in order to obtain the corresponding file offset.

We can then load our new executable into the debugger and check it out:

0070B285	. FF15 40E17600	call dword ptr ds:[<&GetVersion>]
0070B28B	. 33D2	xor edx,edx
0070B28D	. 8AD4	mov dl,ah
0070B28F	. 8915 789F8300	mov dword ptr ds:[839F78],edx
0070B295	. 8BC8	mov ecx,eax
0070B297	. 81E1 FF000000	and ecx,FF
0070B29D	. 890D 749F8300	mov dword ptr ds:[839F74],ecx
0070B2A3	. C1E1 08	shl ecx,8
0070B2A6	. 03CA	add ecx,edx
0070B2A8	. 890D 709F8300	mov dword ptr ds:[839F70],ecx
0070B2AE	. C1E8 10	shr eax,10
0070B2B1	. A3 6C9F8300	mov dword ptr ds:[839F6C],eax
0070B2B6	. 33F6	xor esi,esi
0070B2B8	. 56	push esi
0070B2B9	. E8 20B10000	call evolvva.7163EB
0070B2BE	. 59	pop ecx
0070B2BF	. 85C0	test eax,eax
0070B2C1	. 75 08	jne evolvva.70B2CB
0070B2C3	. 6A 1C	push 1C
0070B2C5	. E8 B0000000	call evolvva.70B37A
0070B2CA	. 59	pop ecx
0070B2CB	. 8975 FC	mov dword ptr ss:[ebp-4],esi
0070B2CE	. E8 F2A40000	call evolvva.7157C5
0070B2D3	. FF15 54E07600	call dword ptr ds:[<&GetCommandLineA>]

Where the CallDLL calls used to be now there are the correct APIs.

Well done, you successfully removed Laserlock copy protection scheme from this executable 😊  
However, the work isn't yet finished...

#### Cutscenes fix:

If you launch the game without the disc in the drive you will find that the initial cutscenes aren't played back. Well, if you carefully check the game's directory, you will find that they aren't there.

Let's put the original game disc into the drive and copy the FMV directory to the game installation folder. Then open the Registry Editor (cmd+r -> regedit [RETURN]) and edit the value of this key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Computer Artworks\Evolva\1.0\FMVDir

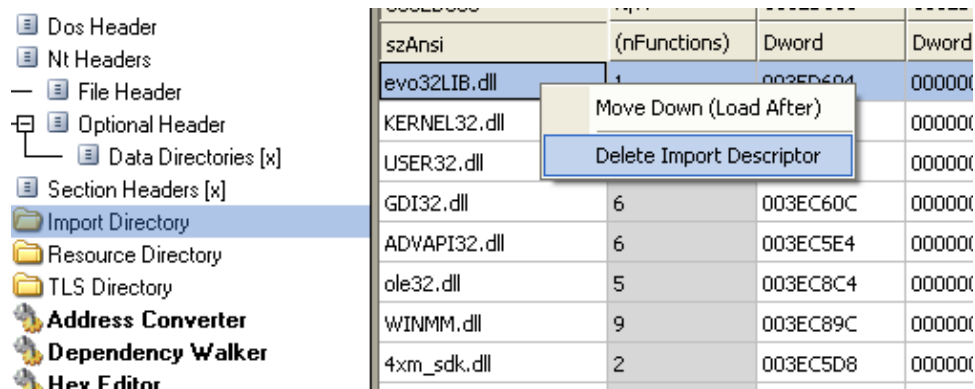
To: .\FMV

Now the cutscenes will be loaded from the game's directory.

#### Removing the evo32lib.dll dependency:

Everything is working perfectly but our binary still depends on the evo32lib.dll library that was once needed by Laserlock. We don't need this dependency anymore.

Let's load Evolva.exe into CFF Explorer and then click on Import Directory. Now select evo32lib.dll and click on Delete Import Descriptor:



Let's save it and we will finally have a totally Laserlock-free binary 😊

#### Credits:

I'd like to thank m00k00 who helped my A LOT by reviewing this paper and correcting all the spelling and grammar errors! You are AWESOME!

#### Conclusion:

I hope that you liked this technical paper. Reversing these very old DRMs is quite fun and you learn a lot!

Luca