Reversing Games Serie :

(Offline and Online Games)

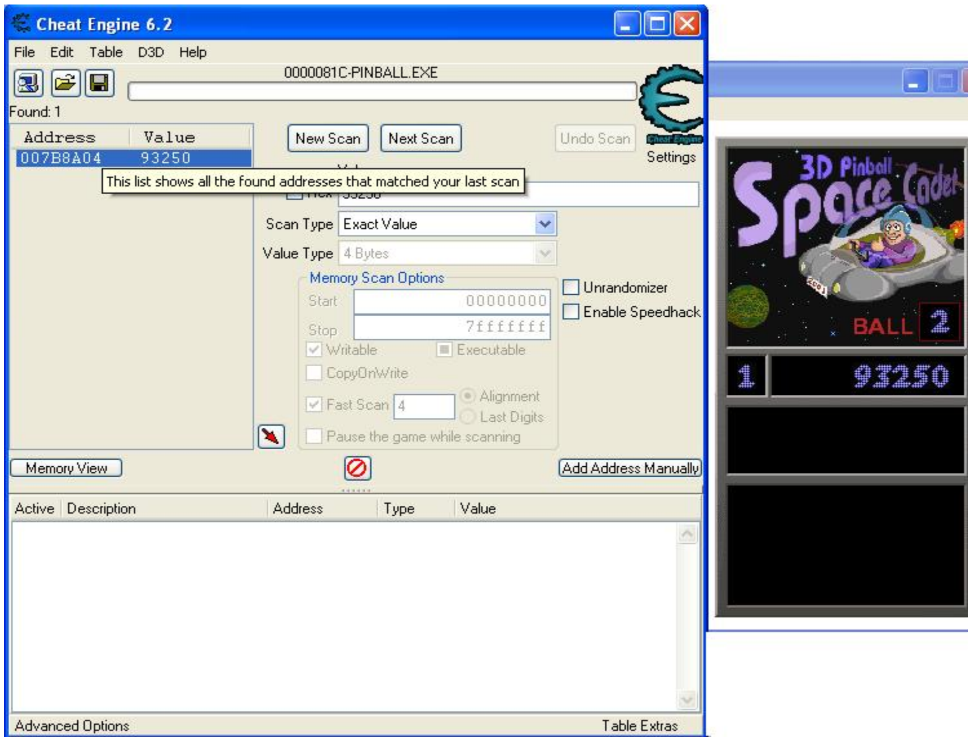
I - Reversing Offline Games :

1st Example : Reversing of Windows 3D Pinball game.

By Souhail Hammou (Dark-Puzzle)

Welcome to the first article of the serie “How to reverse games”. We’ll be covering in this serie how to reverse both online and offline games . To start I will show a simple example on how to locate the exact routines then do some edits to score more when you hit a certain object in the game Pinball . I tought about starting with such a game that is available in almost every windows XP so that everybody can practise what is explained in this article.

What will we need in this tutorial ?:
In this tutorial we will need CE (Cheat Engine) that is a free open-source tool that can be downloaded from :
http://www.cheatengine.org/
this tool will help us locate the exact routine that is adding score gained when the ball hits a certain object .
We will also need a debugger (OllyDBG , Immunity) in order to examine the disassembly and also memory for arguments and instructions that gives us a score and also to save the new patch into a new executable making the changes permanent.
What are we waiting for let’s get started !
- Finding addresses :
First we will need to locate the exact routine that adds the score when the ball hits whatever object , it’s simple open CE , attach pinball.exe process (PINBALL.exe is available by default under this directory “**C:\Program Files\Windows NT\Pinball\PINBALL.EXE**”, play a little bit until the ball falls between the two bars or just use speed hack to pause the game to get a stable score . Enter that score into CE in a form of a DWORD (4bytes) , the score will be searched . In most cases you’ll find one address with the first scan .



0000081C-PINBALL.EXE

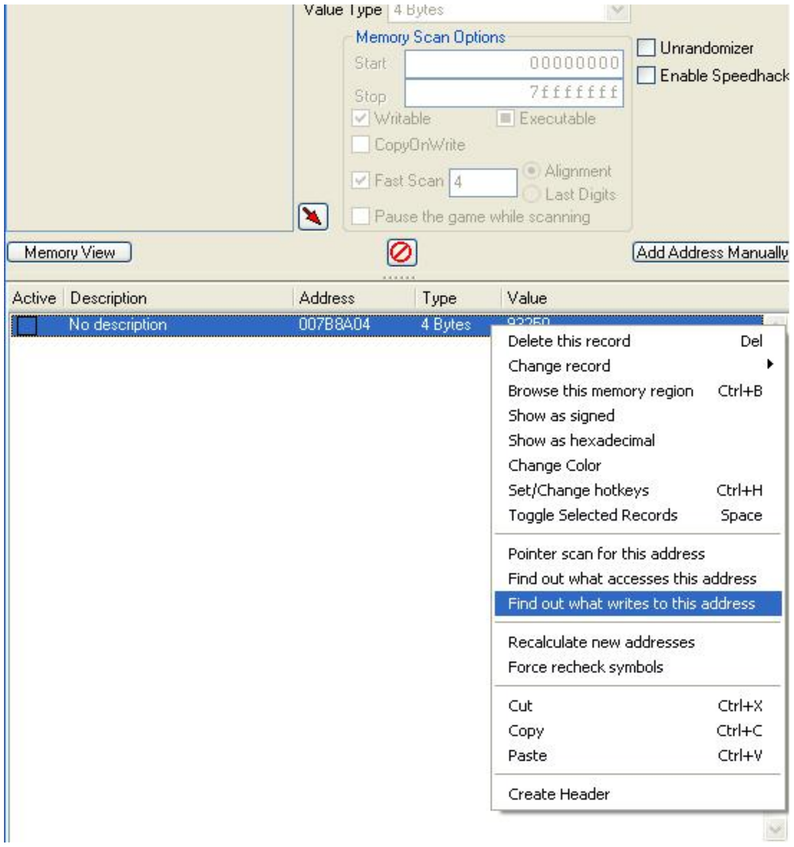
Found: 1

Address	Value
007B8A04	93250

This list shows all the found addresses that matched your last scan

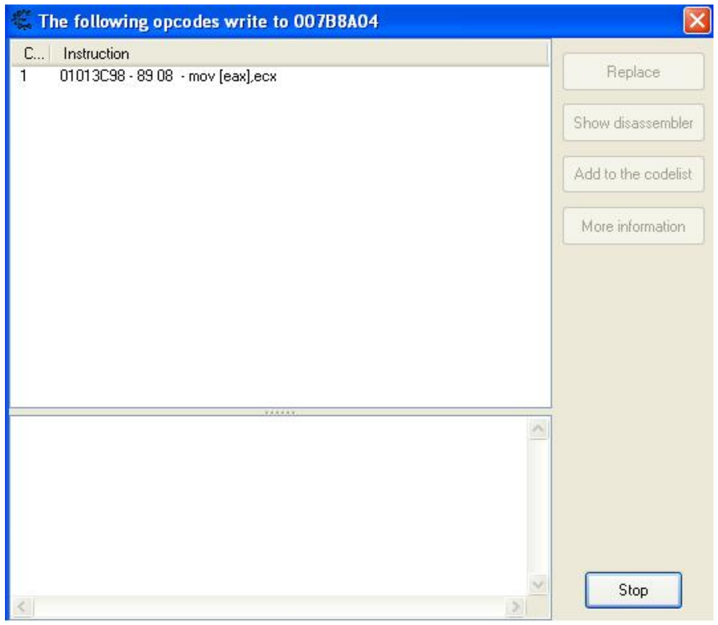
Scan Type: Exact Value
Value Type: 4 Bytes
Memory Scan Options
Start: 00000000
Stop: 7fffffff
[✓] Writable
[ ] Executable
[ ] CopyOnWrite
[✓] Fast Scan 4
Alignment
Last Digits
[ ] Pause the game while scanning
[ ] Unrandomizer
[ ] Enable Speedhack

If not play again then you will recognize that one of the addresses changes to your current score . So that score will be at a certain address (007B8A04 in our case) , but that address is nothing but the memory location where the score is stored , so we will use it to find where the program exactly writes to that scores and adds what's gained after the ball hits a certain object in the game. To do so , we will need to add that address to the table (Double click the address) then click "Find out what writes to that address" .



Active	Description	Address	Type	Value
	No description	007B8A04	4 Bytes	93250

An empty box will pop up , that's because we need to hit another object with the ball to find out what opcodes write the score / write to 007B8A04.
After playing another round an instruction will pop in that box.

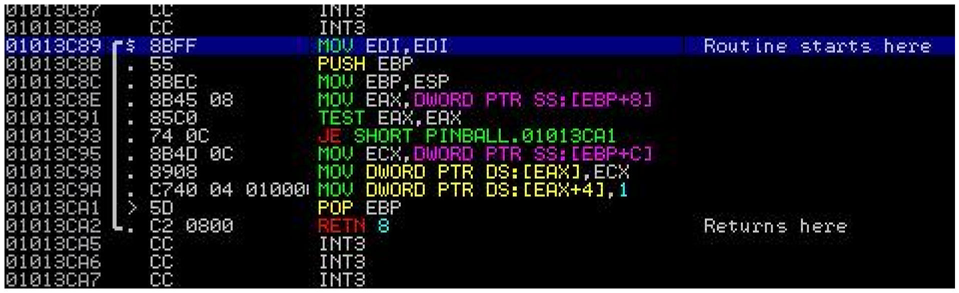


C...	Instruction
1	01013C98 - 89 08 - mov [eax],ecx

01013C98 - 89 08 - mov [eax], ecx . what this instruction do is copy the value of ECX register into a DWORD in the memory location pointed by EAX , this must be our total score right ? but in this tutorial we will try to analyze what happens before all of this ! How this score is being generated ? When the ball hits a certain object , how the score of hitting that certain object is being set then passed to this routine to be added to the previous score.
To figure out all of this , all we have to do is keep that address in mind , switch to a debugger (Immunity debugger in my case) then start doing things backward .

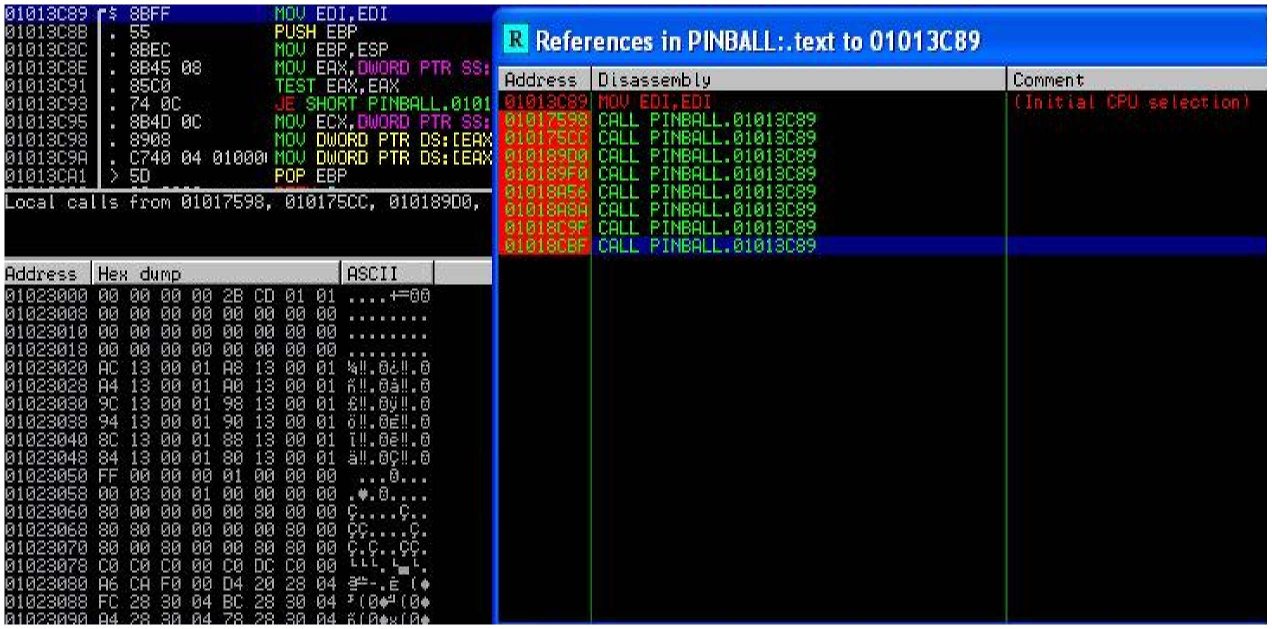
- Debugging the game :

Now, to successfully debug the game you have to detach it from CE , start it again or start a new round and play for some seconds , then attach the process pinball.exe to the debugger . I found a problem attaching it but I figured out this solution , it may be the case for you it may be not , who knows ? So after attaching the debugger to the game all we need to do is go to that address that writes to 007B8A04 which is 01013C98 . Before you'll find yourself in a different Module (ntdll) , go to Executable modules and double click Pinball.exe module . Now press "CTRL+G" then paste the address 01013C98 to go directly to that expression. Now we will try to locate ourselves inside of code , all we have to do is figure out where that routine starts. The main goal from doing this is to go to the root of the calculation that sets the score gained whenever the ball hits a certain object .



```
01013C87    CC             INT3
01013C88    CC             INT3
01013C89 r$ 8BFF           MOV EDI,EDI                         Routine starts here
01013C8B  . 55             PUSH EBP
01013C8C  . 8BEC           MOV EBP,ESP
01013C8E  . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
01013C91  . 85C0           TEST EAX,EAX
01013C93  . 74 0C          JE SHORT PINBALL.01013CA1
01013C95  . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]
01013C98  . 8908           MOV DWORD PTR DS:[EAX],ECX
01013C9A  . C740 04 01000( MOV DWORD PTR DS:[EAX+4],1
01013CA1  > 5D             POP EBP
01013CA2  . C2 0800        RETN 8                              Returns here
01013CA5    CC             INT3
01013CA6    CC             INT3
01013CA7    CC             INT3
```

Ok , what to do now ? We will need to locate which calls are referencing to that beginning of the routine , right click "MOV EDI, EDI" -> find references to -> selected commands. Before doing this you will need to analyze code in order to locate all the calls , loops and routines ... by the debugger , to do so press "CTRL+A" .
Okey after getting all the refering addresses as shown in this capture ,



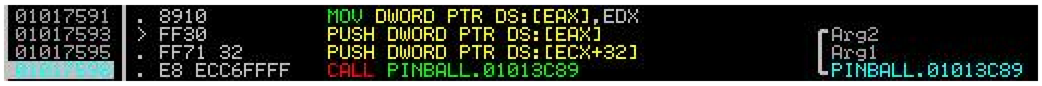
```
01013C89 r$ 8BFF           MOV EDI,EDI
01013C8B  . 55             PUSH EBP
01013C8C  . 8BEC           MOV EBP,ESP
01013C8E  . 8B45 08        MOV EAX,DWORD PTR SS:
01013C91  . 85C0           TEST EAX,EAX
01013C93  . 74 0C          JE SHORT PINBALL.0101
01013C95  . 8B4D 0C        MOV ECX,DWORD PTR SS:
01013C98  . 8908           MOV DWORD PTR DS:[EAX
01013C9A  . C740 04 01000( MOV DWORD PTR DS:[EAX
01013CA1  > 5D             POP EBP
Local calls from 01017598, 010175CC, 010189D0,

Address  Hex dump                 ASCII
01023000 00 00 00 00 2B CD 01 01 ....+=00
01023008 00 00 00 00 00 00 00 00 ........
01023010 00 00 00 00 00 00 00 00 ........
01023018 00 00 00 00 00 00 00 00 ........
01023020 AC 13 00 01 A8 13 00 01 ¼‼.0¿‼.0
01023028 A4 13 00 01 A0 13 00 01 ñ‼.0á‼.0
01023030 9C 13 00 01 98 13 00 01 £‼.0ÿ‼.0
01023038 94 13 00 01 90 13 00 01 ö‼.0É‼.0
01023040 8C 13 00 01 88 13 00 01 î‼.0ê‼.0
01023048 84 13 00 01 80 13 00 01 ä‼.0Ç‼.0
01023050 FF 00 00 00 01 00 00 00  ...0...
01023058 00 03 00 01 00 00 00 00 .♥.0....
01023060 80 00 00 00 00 80 00 00 Ç....Ç..
01023068 80 80 00 00 00 00 80 00 ÇÇ....Ç.
01023070 80 00 80 00 00 80 80 00 Ç.Ç..ÇÇ.
01023078 C0 C0 C0 00 C0 DC C0 00 └└└.└▄└.
01023080 A6 CA F0 00 D4 20 28 04 ª╩≡.╘ (♦
01023088 FC 28 30 04 BC 28 30 04 ⁿ(0♦╝(0♦
01023090 A4 28 30 04 78 28 30 04 ñ(0♦x(0♦
```

References in PINBALL:.text to 01013C89

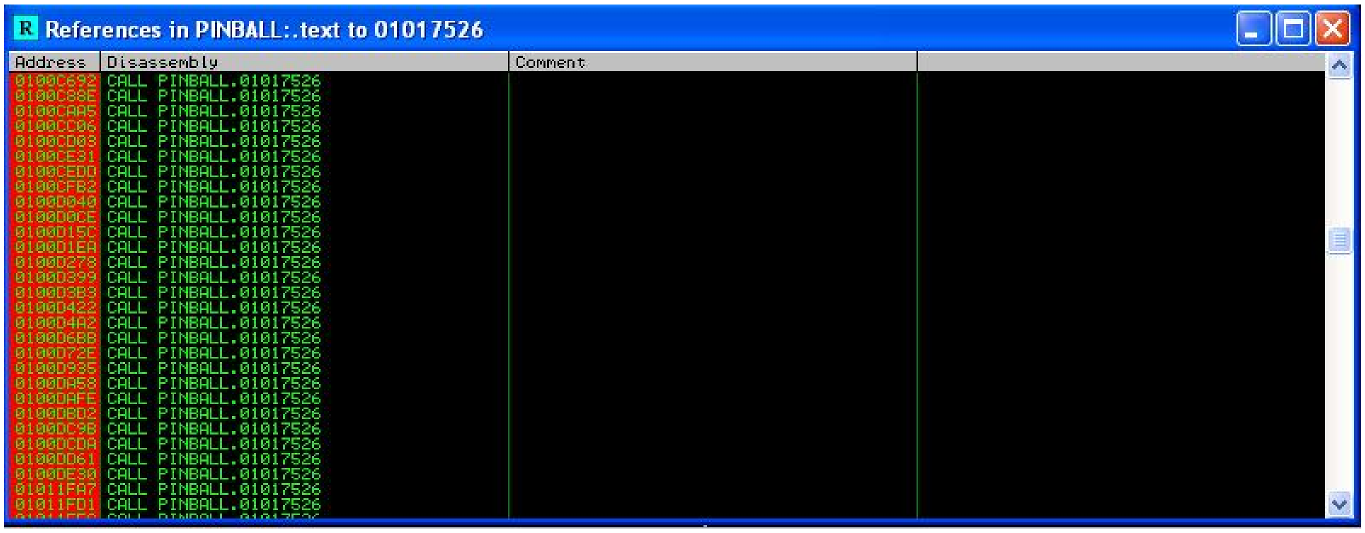
```
Address   Disassembly              Comment
01013C89  MOV EDI,EDI              (Initial CPU selection)
01017598  CALL PINBALL.01013C89
010175CC  CALL PINBALL.01013C89
010189D0  CALL PINBALL.01013C89
010189F0  CALL PINBALL.01013C89
01018A56  CALL PINBALL.01013C89
01018A8A  CALL PINBALL.01013C89
01018C9F  CALL PINBALL.01013C89
01018CBF  CALL PINBALL.01013C89
```

I set a bp in all the calls referring to 01013C89 to locate the exact address that calls the routine that we saw . In fact all those addresses are calling “MOV EDI, EDI” but we are just interested in that call that is related to the score gained while hitting an object not while getting bonus ... etc
Let’s go playing a little bit now , the call that we want will be hit as soon as we will hit an object in the game or go through some point where we get score (entering between the two small gates when shooting the ball). Ok , now I broke into this address 01017598 , we will delete all the other breakpoints as we don’t need them anymore in our task.



```
01017591 . 8910          MOV DWORD PTR DS:[EAX],EDX
01017593 > FF30          PUSH DWORD PTR DS:[EAX]               ┌Arg2
01017595 . FF71 32       PUSH DWORD PTR DS:[ECX+32]            │Arg1
01017598 . E8 ECC6FFFF   CALL PINBALL.01013C89                 └PINBALL.01013C89
```

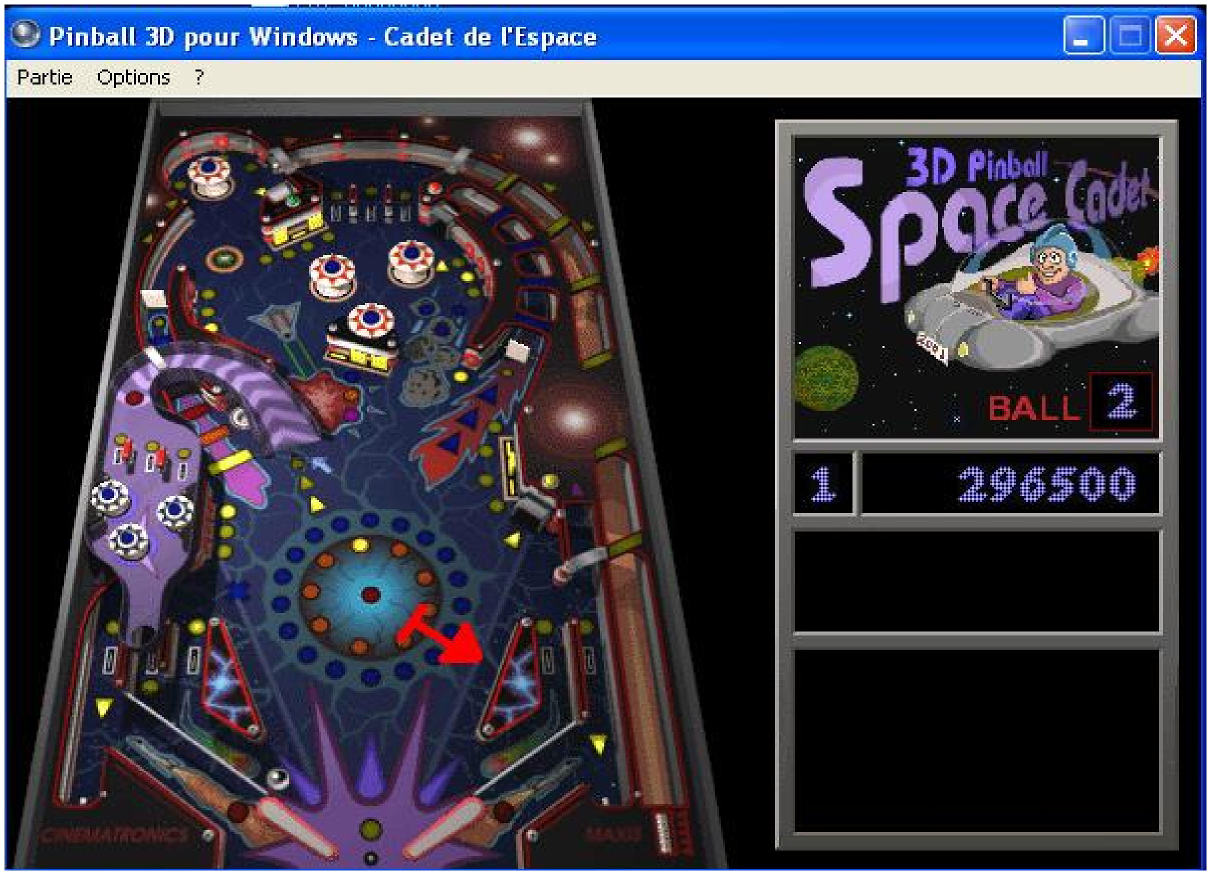
So , basically before that call we have two arguments pushed into the stack, let’s see what’s going now under the hood with those arguments , so first a 4 bytes DWORD as known will be pushed as a 2nd arguments basically because the stack works backwards So the stack will see this as : CALL address --> Arg1 --> Arg2 means from ESP-4 to ESP+4 . This second argument refers to a DWORD that is located in the memory location pointed by EAX , without going through memory dump and searching for that address we will look a little bit up in that routine. Before that, we can clearly see that 01017593 was reached by a short conditional jump after comparing EDX to a value . before that jump and comparison there’s an instruction that moves that DWORD in the memloc pointed by EAX into EDX so we can check EDX register and get the 2nd argument which is in my case : 0003EED6 which represents 257750 in decimal and that is the total score that I will get. The 1st argument can be gathered by checking the memory dump after the DWORD is pushed , after doing this I got : 0003ECE2 which represents 257250 , we’ll see how to do this later in this tutorial . Now , the things are beginning to be clear 257750 - 257250 = 500 . which means 500 points. Then those arguments will be used in the first call that we saw to put the total score into the game after some other routines...
Now all we have to do is to go more deeper , we will do as we did last time go to the starting of this routine , check the referencing calls to the first address then set a bp on all those calls , after that play the game a little bit again . Don’t forget to remove the previous breakpoint so when you want to pass it the executable will continue normally without stopping you again.



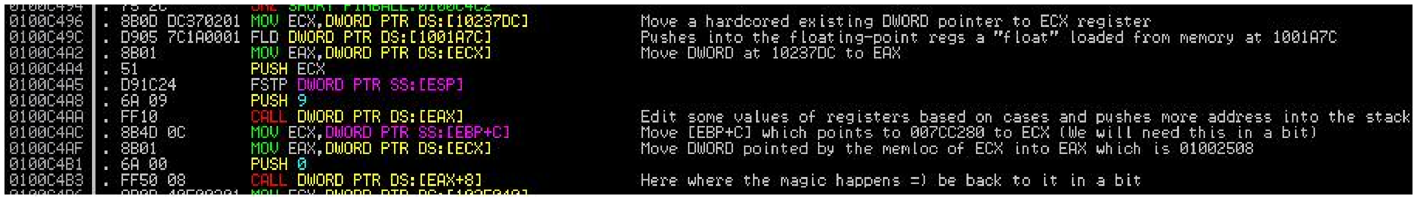
References in PINBALL:.text to 01017526

```
Address   Disassembly                Comment
0100C692  CALL PINBALL.01017526
0100C88E  CALL PINBALL.01017526
0100CAA5  CALL PINBALL.01017526
0100CC06  CALL PINBALL.01017526
0100CD03  CALL PINBALL.01017526
0100CE31  CALL PINBALL.01017526
0100CEDD  CALL PINBALL.01017526
0100CFB2  CALL PINBALL.01017526
0100D040  CALL PINBALL.01017526
0100D0CE  CALL PINBALL.01017526
0100D15C  CALL PINBALL.01017526
0100D1EA  CALL PINBALL.01017526
0100D278  CALL PINBALL.01017526
0100D399  CALL PINBALL.01017526
0100D3B3  CALL PINBALL.01017526
0100D422  CALL PINBALL.01017526
0100D4A2  CALL PINBALL.01017526
0100D6BB  CALL PINBALL.01017526
0100D72E  CALL PINBALL.01017526
0100D935  CALL PINBALL.01017526
0100DA58  CALL PINBALL.01017526
0100DAFE  CALL PINBALL.01017526
0100DB02  CALL PINBALL.01017526
0100DC9B  CALL PINBALL.01017526
0100DCDA  CALL PINBALL.01017526
0100DD61  CALL PINBALL.01017526
0100DE30  CALL PINBALL.01017526
01011FA7  CALL PINBALL.01017526
01011FD1  CALL PINBALL.01017526
```

To spoil you a little bit about the importance of those calls , I can tell that each one refers to an object inside the game (Right and left electro shocking object , those circle buttons ...etc) . So for this tutorial I chose the Right electro shocking object to study , the suitable call in its routine from this list is the CALL in this address 0100C4BD . Let's put a bp on it .
Now make sure that you removed all other breakpoints , so when you hit another object than the electro shocking object (situated at the bottom right of the pinball table) you won't break at the previous Calls.

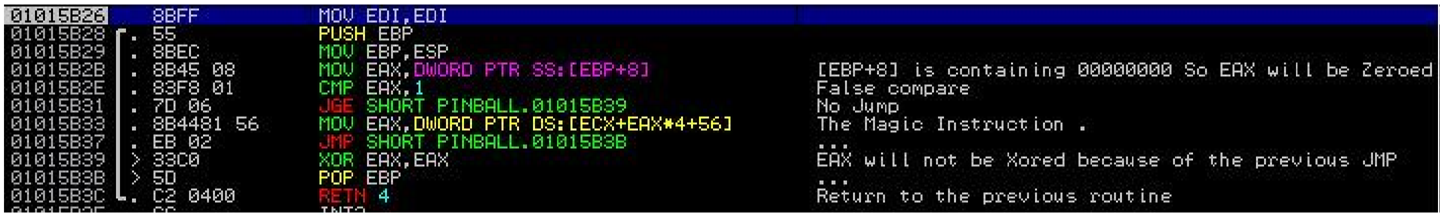


Here where the fun part comes , just keep on reading ... =)
After playing a little bit and hitting the electro shocking object we will suddenly break on the call, all we have to do now is analyze that routine to get into what we want.
Here we are , I just broke at the CALL address 0100C4BD that is calling the previous routine that we saw (just to remind you).
Here's an capture showing the routine with some short comments explaining what the important instructions do :



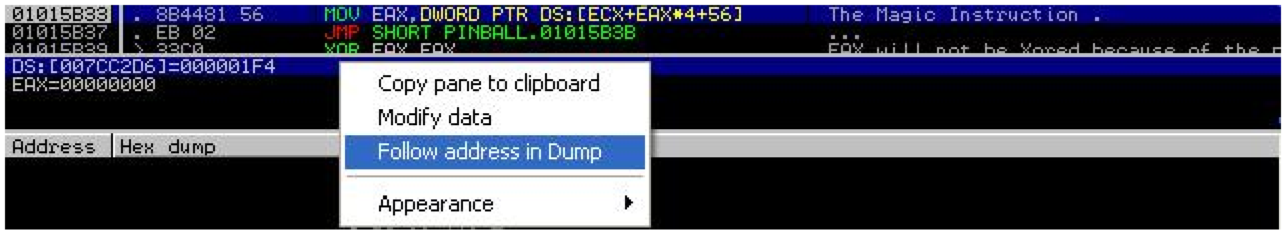
```
0100C494  . 75 2C          JNZ SHORT PINBALL.0100C4C2
0100C496  . 8B0D DC370201  MOV ECX,DWORD PTR DS:[10237DC]          Move a hardcored existing DWORD pointer to ECX register
0100C49C  . D905 7C1A0001  FLD DWORD PTR DS:[1001A7C]              Pushes into the floating-point regs a "float" loaded from memory at 1001A7C
0100C4A2  . 8B01           MOV EAX,DWORD PTR DS:[ECX]              Move DWORD at 10237DC to EAX
0100C4A4  . 51             PUSH ECX
0100C4A5  . D91C24         FSTP DWORD PTR SS:[ESP]
0100C4A8  . 6A 09          PUSH 9
0100C4AA  . FF10           CALL DWORD PTR DS:[EAX]                 Edit some values of registers based on cases and pushes more address into the stack
0100C4AC  . 8B4D 0C        MOV ECX,DWORD PTR SS:[EBP+C]            Move [EBP+C] which points to 007CC280 to ECX (We will need this in a bit)
0100C4AF  . 8B01           MOV EAX,DWORD PTR DS:[ECX]              Move DWORD pointed by the memloc of ECX into EAX which is 01002508
0100C4B1  . 6A 00          PUSH 0
0100C4B3  . FF50 08        CALL DWORD PTR DS:[EAX+8]               Here where the magic happens =) be back to it in a bit
```

Just keep in mind the address pointed by ECX is 007CC280 . This address represents a memory location that was set in EBP+C after the CALL DWORD PTR DS:[EAX] .
So now , we have to set a bp on the starting of that routine again 0100C48D and step over each instruction . After the CALL DWORD PTR DS:[EAX+8] we will notice that the value of EAX has changed to 000001F4 which represents 500 in decimal . This is the score relative to hitting that object . Now after knowing which call is exactly controlling the score "500" after hitting the electro shocking object, let's step into it using F7 after sitting a bp on that CALL.



```
01015B26     8BFF           MOV EDI,EDI
01015B28  ┌. 55             PUSH EBP
01015B29  │. 8BEC           MOV EBP,ESP
01015B2B  │. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]          [EBP+8] is containing 00000000 So EAX will be Zeroed
01015B2E  │. 83F8 01        CMP EAX,1                             False compare
01015B31  │. 7D 06          JGE SHORT PINBALL.01015B39            No Jump
01015B33  │. 8B4481 56      MOV EAX,DWORD PTR DS:[ECX+EAX*4+56]   The Magic Instruction .
01015B37  │. EB 02          JMP SHORT PINBALL.01015B3B            ...
01015B39  │> 33C0           XOR EAX,EAX                           EAX will not be Xored because of the previous JMP
01015B3B  │> 5D             POP EBP                               ...
01015B3C  └. C2 0400        RETN 4                                Return to the previous routine
```

So we noticed before that after this call EAX was containing our score , the last instruction in that call moving something to EAX that something is our "Magic Instruction" that we will deal with it with an interesting way in order not to play with memory and ruin other instructions that may call those locations. Before doing that I will explain what this instruction does :
So ECX was set in the previous call as we noticed : ECX = 007CC280 and EAX =00000000 as the DWORD in [EBP+8] = 00000000 , And 56 which is 86 in decimal is added to that memory location address at ECX , all of that happens for one purpose copying an existing DWORD at [ECX+56] to EAX . We can already say that this value is 000001F4 which is represented by F4 01 00 00 in memory (So it's automatically converted to little endian by the debugger).
So to know where exactly to look we will add 56 to the actual value of ECX , EAX isn't important cause 0*4 = 0. Let's do the calculation : 007CC280 + 56 = 007CC2D6 and this is the address that will be in EAX after that instruction (MOV EAX, [ECX+EAX*4+56]).
Our mission now is to change this instruction and make our score higher than 500 , our objective is to follow 007CC2D6 in memory dump , look for another DWORD already available in memory dump higher or lower than 007CC280 so we can put either (-) or (+).
To do so , a capture is describing how :



```
01015B33  . 8B4481 56   MOV EAX,DWORD PTR DS:[ECX+EAX*4+56]   The Magic Instruction .
01015B37  . EB 02       JMP SHORT PINBALL.01015B3B            ...
DS:[007CC2D6]=000001F4
EAX=00000000
```

This will take us in the Hex Dump windows directly to this :

Ok now , I took a look a little bit down to find another DWORD that has a greater value from 500 to increase our score .
I came across that address 007CC3C9 which has : C9 7C 00 00 --> little endian 00007CC9 and in decimal 31945 .
So now we have to know the difference between the two addresses 007CC280 and 007CC3C9 .
So we'll do this calculation 007CC3C9 - 007CC280 = 149 .
Now it's time to assemble that instruction and make a patch :
MOV EAX,DWORD PTR DS:[ECX+EAX*4+149]
which will add a score of 31945 instead of 500 =) .
And all this without editing any value in memory.
Now to save all those changes , remove all the breakpoints then right click anywhere in CPU --> Copy to executable --> All modifications then click yes and save it wherever you like.
This is how to make our changes permanent so you can benefit from a huge score whenever you play .

P.S :
the positive thing that you can also benefit from is that this small routine is called by other routines referring to different objects, raising the value added to [ECX] from 56 to 149 will make us benefit more and more certainly if the value of [ECX] is different from our case which will call another DWORD that may help us gain 1 millions or 2 millions of score.
And also , this tutorial makes the task a little bit hard and long because we can just edit the score at the first step using Cheat Engine and make it infinite , but the purpose of this paper is to know that cheating is more than editing memory , it's is also about analyzing routines , instructions which makes it more fun.

About the author :
Souhail Hammou known as Dark-Puzzle is a security researcher and a reverse engineering fan from Morocco. He discovered many 0day vulnerabilities in many software of famous companies (Tonec Inc , Huawei , FLstudio , Microsoft , Mozilla, Safari ...) and he's also working on webapplication vulnerabilities in his free-time. Souhail made a blog where he's submitting his latest research , lessons and news and is available at http://www.dark-puzzle.com/.
You can contact him at : dark-puzzle@live.fr
Or : https://www.facebook.com/dark.puzzle.sec