- DESMiTiFiCANDO SafeDisc 2 -

[Introduction]

Hello everyone. My nickname is KeopS. I had been wanting to do a tutorial on the SafeDisc 2 (hereinafter SD2) for some time but it was not until now that, encouraged by several colleagues, I decided to write something. It is not a tutorial that aims **to gut** SD2 but rather a short introduction to this protection.

[Aim]

As everyone knows by now (and if not that's what we're here for ;-) the guys at Macrovision use the TEA algorithm *(Tiny Encryption Algorithm)* to encrypt products protected with SD2. TEA is in the public domain, which means it can be used freely, and its operation is based on

encrypt 64 bits of data at a time using a 128-bit key.

The main program of the SD2-protected application (the .exe file
normally) is encrypted using the TEA algorithm. This encryption prevents the application from being executed directly, and forces it to be decrypted by the SafeDisc launcher or loader (SafeDisc Launcher) to

work properly. Said launcher verifies that a CD with a “signature
Authentic and correct SafeDisc” is present in the CD-ROM drive. If this signature is found, decrypt and run the application, otherwise a nice message will appear telling us to insert the original CD.

After reading the above it seems clear that:

1. It is not possible to run a program protected with SD2 without knowing the TEA key, and it can also be intuited that the TEA key is generated by some type of data loaded from the original CD; and,

2. It is impossible to run an SD2 protected program without the CD original.

However, throughout this tutorial we aim to demonstrate that:

1. The TEA key is generated without the help of any data loaded from the CD original; y,
2. Once this key is obtained, it is possible to run the game without the original CD.

To demonstrate all these statements I will use the games *HITMAN and SACRIFICE,* as well as the TRW2000 debugger. If you do not have these games it does not matter, since all the statements stated here are applicable to all games

protected with version 2.05.030 of SD2 (Alice, No One Lives Forever, NBA Live 2001, ...). Currently there is a new version of SD2, 2.10.030, (patch 2 of Sacrifice, Black&White, Undying, ...) that differs from the previous one in some things, although in essence it remains the same ;-)

Maybe someone is wondering, why am I interested in knowing the TEA key and being able to start the game without the original CD? I was thinking about how to present the tutorial, and I came to the conclusion that this was the best way to get people interested in this protection. With this tutorial I give you the opportunity to study SD2 in depth without having to have an original CD, just have a normal copy of the game in question.

Obviously, I am not advocating piracy but everything set forth here is for educational purposes only. So please, if you like a game, BUY IT as it is the best way to continue making quality games.

[Getting the TEA key]

1. The original key

This is the name given to the key that is originally found inside the file ~DE18B0.TMP *(HITMAN)* (which is actually the AuthServ.dll file), and which serves as the basis for generating the true TEA key. This key is always located at the address: *Image Base of the AuthServ.dll file + 02D660h.* In my case, the dll has an Image Base of B70000, and therefore the original key is

found on B9D660:

```
00B9D660 48 26 8D 5C C2 2E F5 51 A2 A3 2C 95 77 78 48 87 H&•\Â.õQ¢£,•wxH‡
```

On the other hand, this original key is always copied to address **100500F0,** as can be seen in the following routine:

```
100049EF MOV    WORD [EBP-04],00              ; sets the counter variable to 0
100049F5 JMP    SHORT 10004A03
100049F7 MOV    AX,[EBP-04]
100049FB ADD    AX,01
100049FF MOV     [EBP-04],AX
10004A03 MOV    ECX,[EBP-04]
10004A06 AND    ECX,FFFF
10004A0C CMP    ECX,BYTE +10 ; Have we copied the 10h bytes?
10004A0F JNC    NEAR 10004AA8 ; YES -> Finish the routine
10004A15 CMP    DWORD [EBP+08],BYTE +00 ; NO -> ¿Es [EBP+08] igual a 0?
10004A19 JZ     10004A66 ; YES -> Go to routine 2
10004A1B MOV    EDX,[EBP-04]                  ; NO -> EDX=00, 01, 02, ..., 0E, 0F
10004A1E AND    EDX,FFFF
10004A24 MOV    EAX,[EBP+08]                  ; EAX=B9D660 (source address)
10004A27 MOV    CL,[EAX]                      ; CL=byte that is in B9D660, B9D661,...
10004A29 MOV     [EDX+100500F0],CL            ; EDX+100500F0 (destination address)
10004A2F MOV    EDX,[EBP+08]
10004A32 ADD    EDX,BYTE +01                  ; Increase the source address by 1
10004A35 MOV     [EBP+08],EDX

        [Junk Code]

10004A64 JMP    SHORT 10004AA3
10004A66 MOV    EAX,[EBP-04]                  ; This routine places in the direction
10004A69 AND    EAX,FFFF                      ; 100500F0 values 000102...0E0F
10004A6E MOV    CL,[EBP-04]                   ; What is the TEA key that is used?
10004A71 MOV     [EAX+100500F0],CL            ; to decrypt tmp files

        [Junk Code]

10004AA3 JMP    100049F7
```

2. XOReos with 5 32-bit keys

Once the original key has been copied to address 100500F0, it is subjected to 5 XOR operations with 5 different 32-bit keys. But the funny thing is that **these keys are ALWAYS THE SAME!!!!! :-EITHER**

```
1001E160 MOV    BYTE [EBP-08],00 ; sets the counter variable to 0
1001E164 JMP    SHORT 1001E16F
1001E166 MOV    DL,[EBP-08]
1001E169 ADD    DL,04
1001E16C MOV     [EBP-08],DL
1001E16F MOV    EAX,[EBP-08]
```

```
1001E172 AND     EAX,FF
1001E177 CMP     EAX,[EBP+0C]       ; [EBP+0C]=10h have we XORed the 10h bytes?
1001E17A JNC     1001E194           ; YES -> Finish the routine
1001E17C MOV     ECX,[EBP-04]       ; NO -> ECX=100500F0
1001E17F MOV     EDX,[ECX]          ; EDX=content of 100500F0
1001E181 CHOIR   EDX,[EBP+10]       ; [EBP+10] = 32-bit key
1001E184 MOV     EAX,[EBP-04]       ; EAX=100500F0
1001E187 MOV     [EAX],EDX          ; Place the new value in 100500F0
1001E189 MOV     ECX,[EBP-04]       ; ECX=100500F0
1001E18C ADD     ECX,BYTE +04       ; Increase ECX address by 4
1001E18F MOV     [EBP-04],ECX
1001E192 JMP     SHORT 1001E166
```

The values that the address [EBP+10] has are:

1st 32-bit key:

```
0064ECEC AC FE 94 E0 0B 00 00 00 30 ED 64 00 CD 1C 66 00 ¬þ”à....0íd.Í.f.
```

2nd 32-bit key:

```
0064ECEC 3E FA 27 98 0B 00 00 00 30 ED 64 00 CD 1C 66 00 >ú'˜....0íd.Í.f.
```

3rd 32-bit key:

```
0064ECEC AD DE 16 52 0B 00 00 00 30 ED 64 00 CD 1C 66 00 -Þ.R....0íd.Í.f.
```

4th 32-bit key:

```
0064EC94 FC AE AB 3E 0B 00 00 00 D8 EC 64 00 CD 1C 66 00 ü®«>....Øìd.Í.f.
```

5th 32-bit key:

```
0064ECEC 3D 36 A2 42 0B 00 00 00 30 ED 64 00 CD 1C 66 00 =6¢B....0íd.Í.f.
```

After these 5 operations the key has a value of:

```
100500F0 B6 64 21 0A 3C 6C 59 07 5C E1 80 C3 89 3A E4 D1 ¶d!.<lY.\á€Ã‰:äÑ
```

3. XOReos with 2 64-bit and 2 128-bit keys

Perhaps this is the most complicated part of the tortuous path to the final TEA key. Let's go by parts. The key that has been obtained is now subjected to 4 XOR operations with 2 64-bit keys and 2 128-bit keys. This is the corresponding routine:

```
1001E30B MOV     DWORD [EBP-14],00            ; sets the counter variable to 0
1001E312 JMP     SHORT 1001E32F
1001E314 MOV     ECX,[EBP-14]
1001E317 ADD     ECX,BYTE +04
1001E31A MOV     [EBP-14],ECX
1001E31D MOV     EDX,[EBP-10]       ; EDX=address to XORear (100500F0)
1001E320 ADD     EDX,BYTE +04
1001E323 MOV     [EBP-10],EDX
1001E326 MOV     EAX,[EBP-08]       ; EAX=direction with morph key
1001E329 ADD     EAX,BYTE +04
1001E32C MOV     [EBP-08],EAX
1001E32F MOV     ECX,[EBP-14]
1001E332 CMP     ECX,[EBP-18]       ; [EBP-18]=10h have we XORed the 10h bytes?
1001E335 JNC     1001E374           ; YES -> Finish the routine
```

```
1001E337 MOV    EDX,[EBP-0C]      ; NO -> Routine that controls the length of
1001E33A ADD    EDX,[EBP+14]      ; the morph key:
1001E33D CMP    [EBP-08],EDX      ; [EBP+14]=08h for 64-bit keys
1001E340 JC     1001E34D          ; [EBP+14]=10h for 128-bit keys
1001E342 MOV    EAX,[EBP-08]      ; If the morph key is 64 bits:
1001E345 SUB    EAX,[EBP+14]      ; TEA key:          www xxxx yyyy zzzz
1001E348 MOV    [EBP-08],EAX      ;                         FREE
1001E34B JMP    SHORT 1001E337 ; key morph: aaaa bbbb aaaa bbbb

1001E34D MOV    ECX,[EBP+08]      ; This routine controls the length of the
1001E350 ADD    ECX,[EBP+0C]      ; TEA key, but [EBP+0C]=10h, so
1001E353 CMP    [EBP-10],ECX      ; so much does not make much sense
1001E356 JC     1001E363          ; that the TEA key always has a
1001E358 MOV    EDX,[EBP-10]      ; length of 10h...then the JC
1001E35B SUB    EDX,[EBP+0C]      ; always runs :-?
1001E35E MOV    [EBP-10],EDX
1001E361 JMP    SHORT 1001E34D

1001E363 MOV    EAX,[EBP-10]      ; EAX=direction to XORear
1001E366 MOV    ECX,[EBP-08]      ; ECX=key direction
1001E369 MOV    EDX,[EAX]         ; EDX=content of address to be XOReared
1001E36B XOR    EDX,[ECX]         ; [ECX] = 64 and 128 bit keys
1001E36D MOV    EAX,[EBP-10]      ; EAX=direction to XORear (100500F0)
1001E370 MOV    [EAX],EDX         ; Place the new value in 100500F0
1001E372 JMP    SHORT 1001E314
```

The key values at address [ECX] are as follows *(HITMAN):*

1st 64-bit key:

```
00773F70 D7 7A 47 A7 FC 9F 88 2B D7 7A 47 A7 00 00 00 00 ×zG§üŸˆ+×zG§....
```

2nd 64-bit key:

```
007732B0 D7 52 47 A7 D4 9F 88 03 D7 52 47 A7 00 00 00 00 ×RG§ÔŸˆ.×RG§....
```

1st 128-bit key:

```
007736C0 E5 2E 25 7E A4 42 5D AB 85 FA A3 A0 D0 0C A4 2E å.%~¤B]«…ú£ Ð.¤.
```

2nd 128-bit key:

```
00773930 F1 CA ED 83 CC 79 D0 D2 B7 77 E0 C6 31 07 CB 60 ñÊíƒÌyÐÒ·wàÆ1.Ë`
```

Now comes the million-dollar question: are these keys already in memory or are they loaded from the original CD? And if the answer is that they are already in memory, where are they and how are they generated? Interesting questions, right? Well, continue reading and we will reveal the mystery ;-)

Despite what you may think at first, these keys are located in memory within the same file as the original AuthServ.dll key, but they are encrypted. Specifically, the original keys are located at the address *Image Base + 78AEh* (in this case –*HITMAN*- B778AE since the dll has an Image Base of B70000):

```
00B778AE 80 EB 00 86 CB 2E AF 6A C0 83 40 C6 23 6E 6F 82 €ë.†Ë.¯jÀƒ@Æ#no,
00B778BE 26 DB 2A 22 7B 48 77 13 20 26 67 27 46 76 AC 61 &Û*"{Hw. &g'Fv¬a
00B778CE B2 BF 62 5F 93 F3 7A EA 92 2B A4 C1 27 FD 43 AF ²¿b_"ózê'+¤Á'ýC¯
```

and, using the following routine, they are copied to a buffer memory to proceed with their decryption:

```
00B77E12 MOV      EDX,[00BA1250] ; EDX=30h
00B77E18 PUSH     EDX                ; No of bytes to copy
00B77E19 MOV      EAX,[EBP-1C]       ; EAX=B778AE
00B77E1C PUSH     EAX                ; Origin address
00B77E1D MOV      ECX,[EBP-04]       ; ECX=CC1BD0
00B77E20 PUSH     ECX                ; destination address
00B77E21 CALL     00B88BB0           ; Copy from source to destination with number of bytes
```

We already have the original values of our keys placed in memory for decryption, so let's find out where it is and how the routine that performs this task works. It is actually the continuation of the previous routine:

00B77E45 MOV EDX,[00BA124C] ; EDX=C32FB757 (magic value)

```
00B77E4B MOV      [EBP-18],EDX
```

[Junk Code]

```
00B77E72 MOV      BYTE [EBP-0C],00 ; sets the counter variable to 0
00B77E76 JMP      SHORT 00B77E80
00B77E78 MOV      AL,[EBP-0C]
00B77E7B ADD      AL,01
00B77E7D MOV      [EBP-0C],AL
00B77E80 MOV      ECX,[EBP-0C]
00B77E83 AND      ECX,FF
00B77E89 CMP      ECX,[00BA1250] ; [BA1250]=30h have we XORed the 30h bytes?
00B77E8F JNC      00B77EC9           ; YES -> Finish the routine
00B77E91 MOV      EDX,[EBP-18]       ; NO -> EDX=C32FB757 (magic value)
00B77E94 AND      EDX,FF
00B77E9A MOV      EAX,[EBP-0C]
00B77E9D AND      EAX,FF
00B77EA2 MOV      ECX,[EBP-04]       ; ECX=CC1BD0 (source address)
00B77EA5 XOR      EBX,EBX
00B77EA7 MOV      BL,[ECX+EAX]       ; BL=byte that is in CC1BD0, CC1BD1, ...
00B77EAA FREE     EDX,EBX            ; Do these guys only know how to do XOReos? ;-)
00B77EAC MOV      EAX,[EBP-0C]
00B77EAF AND      EAX,FF
00B77EB4 MOV      ECX,[EBP-08]       ; ECX=CC1BA0 (destination direction)
00B77EB7 MOV      [ECX+EAX],DL       ; DL=new value in CC1BA0, CC1BA1, ...
00B77EBA MOV      EDX,[EBP-18]
00B77EBD IMUL     EDX,[00BA124C] ; Generation of a new magical value
00B77EC4 MOV      [EBP-18],EDX
00B77EC7 JMP      SHORT 00B77E78
```

In the end we have the decrypted keys in the destination address:

```
00CC1BA0 D7 7A 47 A7 FC 9F 88 2B D7 52 47 A7 D4 9F 88 03 ×zG§üŸˆ+×RG§ÔŸˆ.
00CC1BB0 F1 CA ED 83 CC 79 D0 D2 B7 77 E0 C6 31 07 CB 60 cÊíƒÌyÐÒ·wàÆ1.Ë`
00CC1BC0 E5 2E 25 7E A4 42 5D AB 85 FA A3 A0 D0 0C A4 2E å.%~¤B]«…ú£ Ð.¤.
```

On the other hand, one may wonder if the *magic value* with which the keys are decrypted is in memory or loaded from the CD. And here again I have to say that this value is already in memory. Let's see it. The variable that stores it is BA124C, so if we go in the reverse direction of the program flow we will reach a routine with the following code:

```
00B839F0 MOV      EDX,[EBP+10]       ; EDX=C32FB757
00B839F3 MOV      [00BA124C],EDX ; Put the magic value in our variable
```

but we need to continue investigating to know where that value is placed in the address corresponding to [EBP+10], and if that value depends on some other data or routine. In the end we arrive at the following code:

```
00B74813 PUSH     EAX
00B74814 JMP      SHORT 00B7481E
00B74816 ADD      [EAX],AL
00B74818 ADD      [EAX],AL
00B7481A ADD      [EAX],AL
00B7481C ADD      [EAX],AL
00B7481E PUSH     DWORD C32FB757 ; The magical value does not depend on ANYTHING! ;-)
00B74823 POP      EAX
```

After the previous operations our key has a value of:

```
100500F0 A2 A8 E9 F7 7C 57 D4 56 6E 44 C3 A5 40 31 8B B7 ¢¨é÷|WÔVnDÃ¥@1‹·
```

4. XOReo with a CRC

Finally, the key is again subjected to another XOR operation (and that's it... sorry, I've lost count of how many that are already ;-). Now it is the turn of a CRC that is calculated from the data in the .text and *stxt774* sections of the original file. The routine that is responsible for calculating said CRC is located at address **1001C28A.** Let's see how it works.

Basically, the routine takes blocks of data from the sections that I have
mentioned with a length equal to 1000h bytes. If the original length of the section or the number of bytes remaining to be processed is less than 1000h bytes, take a block of that length:

```
1001C308 CMP     DWORD [EBP+10],BYTE +00        ; [EBP+10]=section length
1001C30C JZ      NEAR 1001C4F8
1001C312 CMP     DWORD [EBP+10],1000            ; Are there more than 1000h bytes left?
1001C319 JNC     1001C326                       ; YES -> Skip to 1001C326
1001C31B MOV     EAX,[EBP+10]                   ; NO -> EAX=number of bytes remaining
1001C31E MOV     [EBP+FFFFFEE4],EAX             ; Enter number of bytes to process
1001C324 JMP     SHORT 1001C330
1001C326 MOV     DWORD [EBP+FFFFFEE4],1000 ; Mete 1000h bytes a procesar
1001C330 MOV     ECX,[EBP+FFFFFEE4]
1001C336 MOV     [EBP-08],ECX                   ; [EBP-08] = block length
                                                ; of data to process and subtract
                                                ; to the original length of the ; section contained in
                                                [EBP+10]
```

These data blocks are copied to a buffer where the CRC of that data is calculated. Then the CRCs of all the blocks are added and in the end the CRC of the complete section is obtained:

```
1001C45F MOV     CX,[EBP+FFFFFEFC]
1001C466 PUSH    ECX                      ; Number of bytes to process (WORD)
1001C467 MOV     EDX,[EBP+FFFFFEFF0]
1001C46D LEA     EAX,[EBP+EDX+FFFFFEFF8]
1001C474 PUSH    EAX                      ; Source address of the data block
1001C475 CALL    1001C558                 ; Routine that generates the CRC
```

After processing the entire section, the obtained CRC value is stored at an address:

```
1001C543 MOV     EDX,[EBP+18]             ; EDX=storage address
1001C546 MOV     EAX,[1004F2A0]           ; [1004F2A0]=CRC
1001C54B MOV     [EDX],EAX
```

CRC for .text section:

```
1004F2A0 E2 3F E4 2F 00 00 00 00 00 00 00 00 00 00 00 00 â?ä/............
```

CRC for section stxt774:

```
1004F2A0 EA B7 76 FA 00 00 00 00 00 00 00 00 00 00 00 00 ê·vú............
```

But now you may be wondering, wasn't the CRC calculated with the data in the .text and *stxt774 sections?* However, the above routine only calculates the CRC of one section! Effectively, what happens is that the value obtained from the .text section (which is the first to be processed) is copied to another memory address for later use. In this way they use the previous routine again, but this time with the data corresponding to the section

stxt774, to finally add both values and obtain the final CRC with the following code:

```
10017816 MOV   EAX,[EBP-10]      ; [EBP-10]=CRC from previous section
                                 ; (0 for the first time)
10017819 ADD   EAX,[EBP-18]      ; [EBP-18]=CRC of current section
1001781C MOV   [EBP-10],EAX      ; CRC FINAL
```

In my case the final CRC is at the address:

```
0064FA14 CC F7 5A 2A 88 02 00 00 01 00 66 00 00 C0 00 00 Ì÷Z*ˆ.....f..À..
```

Once this value is obtained, the key is XORed using the same routine with which the 5 XOReos were carried out with the 32-bit keys (see section 2. XOReos with 5 32-bit keys). Now the address [EBP+10] has the CRC that we had already obtained:

```
0064FA58 CC F7 5A 2A 0B F7 5A 2A A8 FA 64 00 CD 1C 66 00 Ì÷Z*.÷Z*¨úd.Í.f.
```

So we now have our nice **final TEA key** that will be used in the decryption routine to obtain the corresponding clean sections:

```
100500F0 6E 5F B3 DD B0 A0 8E 7C A2 B3 99 8F 8C C6 D1 9D n_³Ý° ?|¢³™•ŒÆÑ•
```

5. Conclusion

We have just demonstrated that none of the data necessary to calculate the TEA key is loaded from the original CD. Access to the original CD is made
solely for the purpose of determining that a CD with a correct SafeDisc name and signature is located in the CD-Rom drive.

6. Surprise!

This section is simply an exercise that I propose to see how
You have assimilated what you have just learned. The fact that I have chosen the HITMAN game to make this tutorial is not coincidental. In this game, in addition to the main file *(Hitman.exe)* being protected, one of the dlls *(System.dll)* is also protected . So my proposal is that you find out all the previous data but for the *System.dll file.* OK OK! I'm not going to be bad and I'm going to give you a little help, but it's not worth looking at the solution before you've tried it at least a few times ;-)

6.1 The original key

Archivo temporal: ~DE4335.TMP (Image Base=01E80000)

Original key value:

```
01EAD660 8C C7 0C 2B 06 CF 74 26 66 42 AD E2 B3 99 C9 F0 ŒÇ.+.Ït&fB-â³™Éð
```

The original key is copied to address **019600F0.**

Routine located at address:

```
01914A29 MOV        [EDX+019600F0],CL
```

6.2 XOReos with 5 32-bit keys

The 32-bit keys are the same as for *Hitman.exe.*

Routine located at address:

```
0192E181 XOR        EDX,[EBP+10]
```

TEA key value:

```
019600F0 72 85 A0 7D F8 8D D8 70 98 00 01 B4 4D DB 65 A6 r… }ø•Øp˜..´MÛe¦
```

6.3 XOReos with 2 64-bit and 2 128-bit keys

The 64 and 128 bit keys are the same as for *Hitman.exe.*

Routine located at address:

```
0192E36B XOR        EDX,[ECX]
```

TEA key value:

```
019600F0 66 49 68 80 B8 B6 55 21 AA A5 42 D2 84 D0 0A C0 fIh€¸¶U!ª¥BÒ„Ð.À
```

6.4 XOReo with a CRC

Valor CRC:

```
0064F5AC 08 16 DB 5D 0B 16 DB 5D FC F5 64 00 CD 1C 97 01 ..Û]..Û]üõd.Í.—.
```

Routine located at address:

```
0192E181 XOR        EDX,[EBP+10]
```

TEA final key value: (umh... this key sounds familiar ;-)

```
019600F0 6E 5F B3 DD B0 A0 8E 7C A2 B3 99 8F 8C C6 D1 9D n_³Ý° ?|¢³™•ŒÆÑ•
```

[Running an SD2 protected game without the original CD]

Once we have determined all of the above, it remains for us to demonstrate that it is possible to run an SD2-protected game without the original CD. For this we are going to use the game *SACRIFICE.*

This “trick” was originally published by R!SC on the ArthaXerXes forum (http://arthaxerxes.datablocks.net/forum); I only publish it here in a little more detail and with an example. So, with the acknowledgments made, let's get down to business.

The routine that handles all matters related to accessing the original CD to validate its authenticity is in the same dll as the original key (AuthServ.dll). The call to said routine is always located at the address: *Image Base of the file AuthServ.dll + E812h.* In my case, the dll has an Image Base of C50000, and therefore the call to the routine is located in C5E812:

```
00C5E812 CALL     00C54940        ; Upon return of this routine, EAX has 1
00C5E817 ADD      ESP,BYTE +04    ; If EAX=0 then the program ends
00C5E81A TEST     EAX,EAX
00C5E81C JNZ      00C5E82D
```

If we cancel that CALL we are canceling all accesses made to the CD and, therefore, we could start the game without the CD. But it is also true that by canceling this routine we cannot obtain the final TEA key because the XOReos are not carried out with the 32, 64 and 128 bit keys, only the XOReo is carried out with the CRC. So we have a little problem. But since we just

learn how to get the TEA key we can cheat the SafeDisc routine by inserting the final key “by hand”, that is, we first obtain the correct TEA key and then we no longer need the original CD at all. Let's analyze these operations one by one for *SACRIFICE:*

- bpx C5E812 (breakpoint in the routine that checks the CD)

-x (continue program flow)

- e eip B8 (change the original CALL (E8) for a MOV EAX (B8))

- bpm 100500F0 (TEA key memory access breakpoint)

-x (continue program flow)

- pret (trace the program until a RET is found)

- bc * (clear all breakpoints)

- e 100500F0 0A 66 9F 51 AE AE FF 6D 5A F2 37 86 86 34 2F 10 (engañar al SafeDisc by entering the correct TEA key into memory)

-x (continue program flow)

GREAT, IT WORKS!!! We have managed to run a game protected with SD2 without the original CD being in the CD-ROM drive. Therefore that means that you can RIP SD2 without needing the original CD.

So you are already spending hours like crazy to find out where they are and how to solve each and every one of the traps that the friends of Macrovision have prepared for you (hehehe ;-) Good luck!

[Final words]

Well I hope you enjoyed reading this tutorial as much as I did.
enjoyed writing it. My idea is to continue making tutorials about SD2, but it all depends on people's response to this tutorial. So I would like you to send me your criticisms or your suggestions for improvement, as well as topics that you would be interested in seeing in future tutorials.

There are very interesting topics about SD2 such as:

- Files added to the executable. How they are managed.
- TEA decryption routine. Prepared with morph routines.
- IAT redirection. Import encryption.
- CALLs "tricked" in the .text section. CALLs changed to JMPs.
- In certain games use of the SafeDisc API, that is, pieces of code encrypted within the .text section that are decrypted in

  execution time. (For example in Black&White ;-)

I hope to touch on these topics little by little with tutorials like this one. So stay tuned....hehehe ;-)

[Thanks]

I would like to say hello to:

- Gadix and Skuater for being such cool guys. ;-)
- Portia and Risc for helping me when I needed it.
- all the people from the #crackers channel on IRC-Hispano.
- and, in general, to all the people who are involved in cracking.

Greetings colleagues!

[How to contact the author]

Criticisms, suggestions and help can be sent to: joimbo@mixmail.com

20:45 on April 14, 2001

Copyright © KeopS