Crack of C- Dilla SafeDisc 2.70

Mafia game tutorial

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Crack and tutorial written by Peex

( September 24 , 2002)

Summary: This document presents a complete and detailed cracking method for SafeDisc 2.70, without pre-requisites on versions

prior to this protection. The solution chosen for this tutorial is to write a ' SafeDiscRemover ' patch in order to

restore the original program as that he was Before his protection. For the way followed in this tutorial, the main

The difficulty of crack lies in four anti-dumping protections that we will discuss in detail .

o File target : Game.exe ( Mafia games executable )

- Version: 1.0

- Size: 3,237,078 bytes

-CRC32:894A6B6C

o Software used :

-SoftIce v4.05 ( NuMega )

-IceDump v6.026 \_

- FrogsIce v1.10b (Frog's Print & Spath )

-Masm32 v7.0

- LordPE Deluxe v1.4 (Yoda)

- ADump v1.0 ( tHeRaiN / UCF)

- Import REConstructor v1.3 ( MackT / UCF)

- HexWorkshop v2.54 ( BreakPoint Software)

- Original Mafia game CD (Illusion Softworks)

o Difficulty : Quite easy ( but time consuming )

Tutorial Table of Contents

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

1. Introduction

2. General

OEP search : One technique among others

4. Fixing the reference issue \_ systems of . rdata rerouted , 1st try

5. Fixing the reference issue \_ systems of . rdata rerouted , 2nd try

6. Working solution for the reference problem systems of . rdata rerouted (Protection 1)

external calls rerouted in JMP (Protection 2)

rerouted internal calls (Protection 3)

9. Study and solution for code modified on the fly (Protection 4)

10. Fix the header and rebuild the import table

Game- specific CD-Check patch

12. Conclusion and outlook

13. Warning , author 's notes

14. Acknowledgments

15. Appendices: Patch source code (Part A &amp; B)

1. Introduction

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

This tutorial presents a method allowing you to remove all the protections of a program protected by SafeDisc . If this

technique is not particularly difficult to understand and carry out , it East on the other hand relatively long. A minimum of

knowledge is all the same necessary , particularly for writing assembly code , for dumping techniques as well as

for fixing these dumps. However , this tutorial is intended to be "self-contain", so I will detail all operations \_ until the most

simple to make it accessible to as many people as possible .

2. General

¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Let's start with a few general information about SafeDisc . You should know that this protection is a overlay applied to a program

finalized . That is to say that the designers of the game did not place no protection in their code. They call on C- Dilla to

protect their product finished . The protection generator will SO encrypt and protect the original program, then place a loader

at the end of the file .

In early versions of SafeDisc , the main program was encrypted and stored in an .ICD file and the loader placed

in an executable separated who presented themselves as the file to start to launch the application . Today everything fits into one

alone and the same executable , which does not change anything about the problem and its difficulty .

If we look more closely at the loader, we notice several things: It contains anti- SoftIce routines which are She even

encrypted , which means that we cannot \_ \_ not modify these anti- SoftIce routines on disk in with a view to neutralizing them

definitely . In Indeed , we will not find the code of these routines in the file since this code is encrypted . A solution

consists of coding a function that modifies the code memory after decryption function \_ Ouch finished his work , but

things get complicated singularly when we have multi- encrypted code (for the case studied , this East However feasible ).

debugger protections , the simplest solution is to hide OUR debugger ( SoftIce ) using \_

nice programs For example \_ FrogsIce (from Frog's Print). This program makes detection routines ineffective .

SafeDisc detection techniques \_ being nearly stay identical between versions, you can always use the preset

“C- Dilla SafeDisc " in FrogsIce ( optional , because the default mode works well too ). For information, the detection routines

used are the following :

- API Test\_Debug\_Installed / IsDebuggerPresent

- MeltIce (with \\.\SICE and \\.\SIWVID)

-INT 68h

FrogsIce goes very well all these protections. A big thank you to Frog's Print for their magnificent work. (To note that

YOU can also use IceDump as an anti- detector .)

Another general remark , after launch , the loader will Decrypt these protection routines and load them in memory as

a separate process . Furthermore , these routines remain \_ used throughout the execution of the main program (the game). We

can with these some findings suggest an anti-dumping process . The original code was modified to appeal to

protection routines, so that they must be in memory for the program to work correctly . In practice,

- Dilla 's protection generator replaces a call to a subroutine of the program (call <ref>) with a call to a routine

security SafeDisc . Once this safety routine has completed its work , it jump to the code of the original routine , and

the program continues . The only condition to respect is not to modify the system registers . If we dump simply the

main program, we will not have therefore no security routines which allow the program to operate , and the program

will crash of the that he will find a reference to one of the security functions . The purpose being to remove the CD verification , we can

follow two paths :

- The "loader/ patcher ": This method uses a loader which loads memory the main program ALREADY decrypted and which loads

in addition to security routines patched . In effect , a Once you are at the OEP (Original Entry Point), the loader of

SafeDisc has already carried out the CD-Check and decryption of the program, all that remains is the anti-dumping tricks. The routines of

loaded protections in memory are not there only to ensure the proper functioning of anti-dumping tricks.

- The “remover”: This method suggests restoring the program as it was before being modified by the protection generator

by C- Dilla . It is then necessary recode the instructions that have summer modified to remove anti - dumping processes . The starting point

of this method is to have the program at its OEP in order to proceed with its restoration. To rebuild the program, we

will use the SafeDisc security routines which do this work, but we will modify them slightly in order to put them to our

advantage .

(To be complete , it would be necessary Also talk about methods that emulate CD protection.)

In this tutorial, I chose the second method , namely the "remover" which is much cleaner than the first method a times

that the work has been completed . When protections become Really difficult to restore , the first method will become

maybe interesting . For this version of SafeDisc , reverse engineering is relatively easy as we will see . \_

The main difficulty of SafeDisc crack lies in four anti-dumping techniques. This tutorial therefore offers a solution

to remove these protections and thus allow the application to launch without the loader which East partner and who manages the

protection. In addition to controlling CD protection and decrypting the main program, the loader reconstructs bits of

code of the original program as it is executed . It redirects also calls \_ systems under various shapes Thus

that some internal calls . We will SO to land successively these four anti-dumping protections from SafeDisc which make its

difficulty .

You have to start by having the program at its OEP, but before you start looking the OEP , I will give you the few

information useful information from the Mafia game header ( Recovered with LordPE ).

ImageBase (IB) : 00400000

ImageSize (IS) : 0030225C

EntryPoint (EP) : 002FF05E

Section V\_Offset V\_Size R\_Offset R\_Size Flags

.text 00001000 002399BC 00001000 0023A000 60000020

.rdata 0023B000 0000AF5F 0023B000 0000B000 40000040

.data 00246000 000B4B7C 00246000 0000F000 C0000040

.rsrc 002FB000 00000D76 00255000 00001000 40000040

stxt774 002FC000 0000204F 00257000 00003000 E0000020

stxt371 002FF000 0000325C 0025A000 00004000 E0000020

OEP search : One technique among others

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯

[ Note: I will be insulting to some of among you , but I said that I detailed all , so I do it .]

If you look at the program entry point using ProcDump or LordPE , you \_ will get the entry point of the loader

SafeDisc and not the entry point of the game. When the loader finishes its work, namely checking the CD and decrypting the different

sections of the main program, it skips then to the OEP and the game starts. It is when we are at the OEP that we go power

to work . Before this point, the program is not decrypted , and afterward variables are initialized , the image East modified , and therefore

unusable in most cases . \_

For protection similar to SafeDisc ( with CD-Check and encryption ) but which would not have anti-dumping protections, a

dump to the OEP would give an executable functional a times fixed . That 's the power of the dump, and the people of C- Dilla we well

including , also , their research is no longer focused solely on improving encryption (because the dump short - circuits the

encryption ) but also in anti-dumping techniques (we can Also add the difficulty of copying CDs, but this does not

not interested here ).

To determine the OEP , there exists several techniques, and you in will find quite a few reading tutorials . \_ The most laborious

( not recommended ) is to trace the loader by watching when it leaves its segment to arrive in the .text section of the program (the

reference of the first instruction minus IB ( ImageBase ) = OEP ). We can Also monitor jmp , j\*, call, call[ref], jmp [ ref]

and ret, which is also very long. In remaining in the techniques rather average , we can also use a disassembler to

from a dump of the game course of execution ( therefore not valid for the reconstruction of the original program) to try to locate

some well- known program start structures , but this technique is risky and unsuccessful often if the program

contains snippets of code with anti- disassembly tricks that disrupt the disassembler .

I will now \_ YOU detail the technique I have used for this program, but This is not the only solution, too ,

choose the method you \_ want as much as you want find the OEP .

Once \_ SoftIce hidden , I launch the game and immediately switch to SoftIce to set a “ GetVersion ” breakpoint ( bpx

GetVersion ). GetVersion East a function of kernel32.dll which is often called at the very beginning of large applications. SO when

SoftIce stops at my breakpoint, there is a good chance that I am not very far from the start of the program that I risk

strong to recognize . Also note that the SafeDisc loader can also have this call system when initialized,

so you have to check in outgoing call \_ system ( GetVersion ) that we are in the .text section of the program

( section containing the game code). Generally the OEP is located towards the end of a section .text (finding/ rule empirical ).

00627338 5D POP EBP

00627339 C3 RET

(B) 0062733A 55 PUSH EBP

0062733B 8BEC MOV EBP,ESP

0062733D 6AFF PUSH FF

0062733F 68B8EF6300 PUSH 0063EFB8

00627344 688C956200 PUSH 0062958C

00627349 64A100000000 MOV EAX,FS:[00000000]

0062734F 50 PUSH EAX

00627350 64892500000000 MOV FS:[00000000],ESP

00627357 83EC58 SUB ESP,58

0062735A 53 PUSH EBX

0062735B 56 PUSH ESI

0062735C 57 PUSH EDI

0062735 D 8965 E8 MOV [EBP-18],ESP

(C) 00627360 FF 151CB16300 CALL [0063B11C]

(A) 00627366 33 D2 XOR EDX,EDX

Below is a screenshot \_ partial SoftIce \_ carried out after being returned from kernel32 GetVersion call . Point of

return is located in (A). Function \_ appellant GetVersion East SO in (C), and we typically recognize the start of a program in

(B). We are in the .text section, towards the end, and moreover a RET just above confirms the beginning of a

new routine. We see all this , we can say that it is won for the determination of the OEP .

Note: Once the OEP determined , you can search using a tracker \_ the instruction that points to the OEP . In using the

IceDump tracer (/ tracex 62733A), we discover that the instruction previous one was at 6FF119 , which times disassembled us

reveals a nice ' jmp 62733A'. To remark also that this instruction was not encrypted and that it was therefore simply visible

Since the p . Once we \_ \_ knows the SafeDisc technique , it is very easy to find manually this jump. SO nothing \_

hard, neither for us, nor for a SafeDiscRemover automatic .

4. Fixing the reference issue \_ systems of . rdata rerouted , 1st try

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Voila, the OEP East determined and our work would be nearly finished if there is did not have the additional anti-

dumping. Exactly , let's get to it, notice that the call system GetVersion East note :

CALL [0063B11C] ( in (C) i.e. 00627360)

while traditionally , it is note as this under SoftIce :

CALL [KERNEL 32!GetVersion ]

This is not a SoftIce error and this \_ deserved let's take a closer look . This Call points to the section. rdata which

contains in normal times (and among other things ) the call references external ( call to dlls external to the program). But if we

look at this reference (to 63B11C which is located In . rdata ), instead of finding a reference a call system ( here at kernel32), we

find this reference : 02BF1DE0

Here is a small extract from the section. rdata to give you an idea:

0063B11 C:> 02BF1DE0&lt; 02BF22A3 02BF2766 02BF2C29

0063B12C: 02BF30EC 02BF35AF 02BF3A 72 02 BF3F35

0063B13C: 02BF43F 8 02 BF48BB 02BF4D7E 02BF5241

0063B14C: BFF95DD 3 02 BF5BC7 02BF608A 02BF654D

And if we look at the code pointed by these references , we find always essentially the same thing, but these are not the

system routines expected . I you gives the code for the reference point in 63B11C i.e .: 02BF1DE0.

(D) 02BF1DE 0 688710 EABF PUSH BFEA1087

02BF1DE 5 9 C PUSHFD

02BF1DE 6 60 PUSHAD

02BF1DE7 54 PUSH ESP

02BF1DE8 68201EBF02 PUSH 02BF1E20

(E) 02BF1DED E857B54FFF CALL 020ED349

02BF1DF2 83C408 ADD ESP,08

02BF1DF5 6A00 PUSH 00

02BF1DF7 58 POP EAX

02BF1DF8 61 POPAD

02BF1DF 9 9 D POPFD

02BF1 DFA C 3 RET

We can already think as I have mentioned above to an anti-dumping mechanism , or the references of the appeals systems stored in

. rdata are replaced by pointers to SafeDisc security routines . Once \_ these SafeDisc routines launched , they

call in turn the true call system originally asked . \_

Also note that this part of memory does not belong to a process launched by the Windows loader, it is SO

probably a safety routine that has it allocated for its needs and who decrypted ( or calculated ) of the code inside . In

on the other hand, the call that we found inside this routine will inform us . \_ \_ Let's look at who owns the code that is located

in 20ED349. To do this , we run a small map32 on the reference :

map32 20ed349

Owner Obj Name Obj# Address Size Type

~DF394B .txt2 0001 0187:020A1000 0000F1E9 CODE RO

~DF394B .text 0002 0187:020B1000 0003AC35 CODE RO

~DF394B .txt 0003 0187:020EC000 00001E02 CODE RO

~DF394B .txt4 0004 0187:020EE000 0000011B CODE RO

~DF394B .txt5 0005 0187:020EF000 0000017F CODE RO

~DF394B .rdata 0006 018F:020F0000 00009E17 IDATA RW

~DF394B .data 0007 018F:020FA000 00018345 IDATA RW

~DF394B .reloc 0008 018F:02113000 00005382 IDATA RO

~DF394B stxt774 0009 018F:02119000 000009BC UDATA RO

This is the main security "DLL-like" (from SafeDisc ) which is launched by the SafeDisc loader . A passing remark,

~DF394B is stored on the hard drive as a file temporary (in a directory temporary ) and deleted after termination of the

program. It's not besides not the only one and you can YOU have fun dumping these routines to reverse them if That amuses you .

Let's return to our problem and let's start by plotting our call “CALL [0063B11C]” in (C). We notice that in outgoing call \_

"02BF1DED: CALL 020ED349" in (E) we arrive directly in the function kernel32 system . SO if I resume , the program Call

main in (C) goes to call a SafeDisc security routine , and this security routine will make the call system longed for . This

therefore call \_ been - rerouted - or even - rehoused - ( I will use both terms indifferently ). All calls \_ system (at kernel32.dll,

user32.dll and advapi32.dll) have Thus summer modified by the SafeDisc protection generator . He becomes logical that if we dump

the main program, we will not have the security routine which is in a separate process , and therefore the program

will crash from the first call system (absence of code, or invalid zone ). It is SO anti-dumping protection .

As we have selected to write a remover instead that a loader, he will SO to need replace calls to SafeDisc routines

by the values original calls \_ systems .

In fact , it's more complicated than that ... What I come to explain was valid for old SafeDisc protections

( THE people who have already tackled the first versions of SafeDisc must have recognized ). In the meantime, the system has evolved and

It's a bit more complex , but rather than giving you the solution directly , I chosen by you explain the path

towards the solution. Certainly the document will be longer, and the first patches of the "remover" will not be functional , but the aim of

this tutorial is yours \_ explain the thinking to be done to break this kind of protection, and not from you provide a solution

all cooked pre-chewed without explanation (for that , just look for a crack on the internet).

I repeat it again times : the first scans and patches will not work ! I detail them here for a purpose

pedagogic . (For people who are already slightly familiar with the versions previous versions of SafeDisc , you Go on widen them

eyes ... :o) Likewise , SafeDisc East consisting of several anti-dumping protections, and as I progress through this document, I

will do as if I didn't know none of the other protections still remain to be unsealed . In short , I will detail the evolution of crack

SafeDisc . This being said , let's continue .

My first idea to solve This problem was \_ to call each reference relocated of . rdata ( or again: each call system

rehoused by SafeDisc ) in modifying previously the SafeDisc function which calculates call references \_ \_ \_ systems in order to that she

returns the call reference in a register original system instead of calling it . Once we have the value \_

original of a call relocated , we replace him In . rdata and the turn is plays .

Let's analyze now the routine that calculates the value original of the call system to see how to do this :

(Call of this routine 02 BF1DED CALL 020ED349 in (E))

020ED 349 55 PUSH EBP

020ED34 A 8 BEC MOV EBP,ESP

020ED34C 83EC40 SUB ESP,40

020ED34F 53 PUSH EBX

020ED350 56 PUSH ESI

020ED351 57 PUSH EDI

020ED352 F0FF05440C1002 LOCK INC DWORD PTR [02100C44]

020ED359 740E JZ 020ED369

.

.

.

020ED499 50 PUSH EAX

020ED49A E82381FDFF CALL 020C55C2

020ED49F 59 POP ECX

020ED4A0 F0FF0D440C1002 LOCK DEC DWORD PTR [02100C44]

020ED4A7 780C JS 020ED4B5

020ED4A9 FF35F4DB1002 PUSH DWORD PTR [0210DBF4]

020ED4AF FF1548000F02 CALL [KERNEL32!SetEvent]

(F2)020ED4B5 8B650C MOV ESP,[EBP+0C]

020ED4B8 61 POPAD

020ED4B9 9D POPFD

(G2)020ED4BA C3 RET

.

.

.

020ED6E2 50 PUSH EAX

020ED6E3 E8DA7EFDFF CALL 020C55C2

020ED6E8 59 POP ECX

020ED6E9 F0FF0D440C1002 LOCK DEC DWORD PTR [02100C44]

020ED6F0 780C JS 020ED6FE

020ED6F2 FF35F4DB1002 PUSH DWORD PTR [0210DBF4]

020ED6F8 FF1548000F02 CALL [KERNEL32!SetEvent]

(F) 020ED6FE 8B650C MOV ESP,[EBP+0C]

020ED701 61 POPAD

020ED 702 9 D POPFD

(G) 020ED 703 C 3 RET

Some explanations on how this security routine works \_ go to use : When we arrive at (G), the pile

contains the address the call original system (in our case , kernel 32!GetVersion ) and the return address of the function caller

: 00627360 CALL [0063B11C ] in (C). YOU have it understood , we will simply to recover this reference system , and return to our

patch. ( Note: the routine arrives in (G2) if she has already done the calculation previously . So to be complete , it would be necessary to modify

these two places , but we \_ We'll deal with this later.)

Before continuing, I explain how a Call works for some of among YOU . When we make a call , the system changes

the eip to put it on the destination address , but it also puts the return address of the call in the stack , i.e. an instruction

after the Call ( which makes eip+5 if It is a Call <ref> and eip+6 if it's a Call[ref]). The called routine ends with a RET,

which removes the return value of the call from the stack and changes the eip to that value . We are SO income an instruction after the

Call. We could translate a Call <ref> by:

push eip+5

jmp <ref> \_

And a RET by:

pop eip

That's it for this parenthesis . SO when the system makes the first RET, he jumps on the call system (kernel 32!GetVersion ), and

When the GetVersion routine ends and reaches its final RET, we return to the main program, just after Call[0063B11C].

( From one simplicity childish , and I am not likely to go further down into the details of the explanations... :o)

So to retrieve the value of the call original system , just do a POP edx in (G), and the reference system which

should have been called passes into register edx . The second reference in the stack will be used to return to the caller , our patch. We

could SO insert our POP in place of the RET in (G) and add a RET after, but we would at the same time modify the code which is

find below (G) (one byte because a pop takes one byte). The trick consists therefore to place a JMP in (F) to bring the program

in ADump Or we have as much space as we need wanna . Once we \_ \_ is in our memory segment , we restore the

instructions that the jump has deleted and we add a POP edx before the RET (The jump consumes 5 bytes, that 's why that we don't put it anymore

in back ).

Let's see so what does the end of the calculation function ( 020ED349) look like after modification ( in order what returns the value of

call original system in edx .)

020ED6F 0 780 C JS 020ED6FE

020ED6F 2 FF 35F4DB1002 PUSH DWORD PTR [0210DBF4]

020ED6F 8 FF 1548000F02 CALL [KERNEL32!SetEvent]

020ED6 FE E 931493081 JMP 833F2034

And at the address pointed by the JMP, we restore the instructions deleted by the jump and we add our patch:

833C 2034 8 B650C MOV ESP,[EBP+0C]

833C 2037 61 POPAD

833C2038 9D POPFD

833C2039 5A POP EDX &lt;&lt;&lt; Notre superbe patch

833C203A C3 RET

We will then type a little code in SoftIce to create the fix program that we has conceived . For my part, I had everything

first started by writing my little programs at the OEP , but this was not very clean and I SO used ADump to create a

free memory area to store my code. A significant advantage \_ is that if the game crashes , the code is always stored in

ADump . (Note that we can Also use IceDump to create a free memory segment .)

Here is therefore the program which traverses . rdata looking for call system relocated by the SafeDisc protection routine . THE

calls relocated are recomposed in calling the specific SafeDisc routine , then stored in ADump , a little after the code. It will be necessary

then dump these references to paste them into the section. rdata from the dumper executable to the OEP .

833C 2000 B 800B06300 MOV EAX,0063B000

833C 2005 BB 00606400 MOV EBX,00646000

833C200 A 81380000 BE02 CMP DWORD PTR [EAX],02BE0000

833C 2010 720 E JB 833C2020

833C2012 81380000C002 CMP DWORD PTR [EAX],02C00000

833C2018 7306 JAE 833C2020

833C201A FF10 CALL [EAX]

833C201C 8913 MOV [EBX],EDX

833C201E EB04 JMP 833C2024

833C2020 8B10 MOV EDX,[EAX]

833C2022 8913 MOV [EBX],EDX

833C2024 0504000000 ADD EAX,00000004

833C2029 83C304 ADD EBX,04

833C202C 3D50B26300 CMP EAX,0063B250

833C2031 75D7 JNZ 833C200A

833C2033 CC INT 3

This screenshot from SoftIce corresponds to the more readable code (jump and comments ) below :

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Start of RemoveSafeDiscPatch code < test 1="">

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

mov eax , 63B000h ; Start of section. rdata Or we will scan the references .

mov ebx , 833C3000h ; Start of the area where we will store references original ( ADump zone + 1000h).

\_loop:

cmp dword ptr [ eax ], 02BC0000h ; Checked if it's a call relocated (zone 2BC0000 to 2C00000).

jb \_copy ; If this is not the case , we will go just copy the reference .

cmp dword ptr [ eax ], 02C00000h ;

jae \_copy ;

call dword ptr [ eax ]; Calls the SafeDisc security function \_ \_ patched .

; edx contains then the reference to the call original system .

mov dword ptr [ ebx ], edx ; We copy this reference in . data+n .

jmp \_next ; And we continue the treatment .

\_copy:

mov edx , dword ptr [ eax ] ; This is not a reference relocated , we copy it simply .

mov dword ptr [ ebx ], edx ;

\_next:

add eax , 4 ; We move forward one dword in . rdata .

add ebx , 4 ; Likewise for the ebx pointer \_

cmp eax , 63B250h ; Continue analysis up to .rdata+250h.

jnz \_loop ;

int 3 ; Break here when the patch has finished .

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; End of RemoveSafeDiscPatch code < test 1="">

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Explanation on how the program works : (Reminder: This patch is not functional ! )

This program scans . rdata to .rdata+250h (.rdata+250h = end of calls relocated ; located under SoftIce ) in testing for each

reference if She is between 2BE0000 and 2C00000 (Area of SafeDisc routines which manage this protection). If this

reference East relocated , so we're going to call this reference to retrieve the reference original of the call system SO stored

in edx . It is of course necessary previously patch routine 020ED349 so that it returns the value of the reference original in edx .

Voila, the patch is written and ready to work , we launch the Mafia game in setting a breakpoint at the OEP (62733A) then a times what

SoftICe break, we change the EIP to put it at the beginning of our program stored in ADump . We then place a breakpoint on the

interrupt 3 ( bpint 3) and we launch the program. We can also plot a few cycles to see if everything goes like this foreseen .

The patch runs , analyzes the 0x250h values that we him had requested and SoftIce break on interrupt 3. Okay, let's take a quick look

eye in memory to see the changes before testing the game.

Call reference values \_ \_ system originals stored in ADump :

833C3000: BFE815 EA BFE 8167D BFE81534 BFE81644

833C3010: BFE819AA 00000000 00981040 00000000

833C3020: 7CD73DF 7 00000000 009C4860 00000000

833C3030: BFFA19F 5 BFF 77B30 BFF764C0 BFF77B8F

833C3040: BFF77B 57 BFF 76FD1 BFF7713B BFF77C1C

833C3050: BFF7C8 BD BFFA 0AAB BFF779AE BFFABDEA

.

.

.

833C3100: BFFC63C 0 BFF 77D63 BFF77D81 BFF77D9F

833C3110: BFF76F3 B BFF 76DA8 BFFA0ACB &gt;BFF92F1B&lt; ( In 64611C, or 63B11C for . rdata , we have the call reference system

833C3120: BFF8C5 DA BFF 777AD BFF77716 BFF77A90 kernel32!GetVersion called and (C))

833C3130: BFF 77741 BFF 779D5 BFF96347 BFF76A38

...

Before continuing with the tests, these must be data be In . rdata . As always , we have several solutions: we can do

a memory copy using SoftIce , or \_ \_ \_ dump the references and then reload them in . rdata . The second case

present the advantage of keeping the references that we have calculated if things go wrong while the first method East

faster . \_ I you gives both :

- Memory copy : \_

m 833C3000 L 250 63B000

- Dump with IceDump :

/ dump 833C3000 250 c:\rdataFix.bin

/ load 63B000 250 c:\rdataFix.bin

We therefore now have our changes in . rdata . To have This result , we can think to have restored calls \_ systems , because that 's it

looks good. Besides if we take a look at the first call system after the OEP , we find the reference original to

the call system GetVersion written in clear in SoftICe :

00627360 FF 151CB16300 CALL [KERNEL32!GetVersion]

Instead of:

00627360 FF 151CB16300 CALL [0063B11C]

But unfortunately , it is one of the rare case valid ... In Indeed , if we now launch the program we see a nice crash.

It's not so not as simple as that , what worked in the past no longer works , and this protection has evolved a lot .

5. Fixing the reference issue \_ systems of . rdata rerouted , 2nd try

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

To understand the problem , you need to find an invalid case and debug it . Find an invalid case \_ is not very hard,

SoftIce go away will load for you as soon as the program crashes . We can then compare the call system that causes the crash in two

case : (the crash is sometimes a little after the call system himself )

- The case functional , where we n / A carried out no modification of call references system of . rdata .

- The case Or we fixed it . rdata with our patch.

simply surrender realize that the program is not calling the correct system routine ...

For example , the call system in 0062CC35 (from the unpatched functional program )

0062CC 35 FF 15ECB06300 CALL [0063B0EC]

calls the reference BFF77716 which corresponds to the call system GetModuleHandleA .

But our (poorly) patched program calls the reference BFFA1392 which corresponds to the call system GetEnvironmentVariableA :

0062CC 35 FF 15ECB06300 CALL [KERNEL32!GetEnvironmentVariableA]

The program will therefore crash when calling this function or a little later.

Obviously , our reference correction patch of . rdata is wrong somewhere . If we trace the call from the program

original ( unpatched ) , this returns the correct reference , whereas when the call has been done since our patch, the reference East

erroneous .

If we think about it , logically , this cannot be due to context in which the system is located when the call to

function East carried out . But the context is essentially made up of two things: the registers systems , and the battery. In proceeding

by elimination , we realize that the registers don't have no influence on the determination of the appeal original system . By

against , the stack will influence the calculation of the call reference system . In fact , the SafeDisc calculation routine uses the

reference placed on the stack by the Call instruction to determine the reference system to call (She uses Also of a other

value placed in the stack by the routines referenced by . rdata : example in (D)). We go so make a verification manual for

check OUR assumption .

For example , let's modify the first 'Call [ ref . rdata ]' found, namely the Call [ GetVersion ] located in (C). Arriving at his eip , we

assemble code for another call system , for example our CALL [0063B0EC] (a 627360: call [63B0EC]) then trace this call ,

a times entered into the function , we will immediately modify the first value of the stack (set by the call). We go so do

believe in this routine that it has been called from another point, or more precisely from the call in 62CC35. We go SO edit This dword has

SoftIce help and modify it to put it in 62CC35+5 or 62CC3A (under SoftICe , e esp , then passing bytes in dword , or

write in BigEndian )

And indeed , the modified call refers to the call system valid expected : GetModuleHandleA . He will SO need to modify

our program so that it do trust the SafeDisc security routine that the call was thrown at the location of the call. This go

train an essential modification in our program, it will to need sweep the entire .text section (the code of the

program) to find rerouted calls in order to find cases valid to modify. rdata . Also , our patch will no longer do

call calls systems rerouted , but 'push < fakeref >; jmp <ref. rdata ="">'.

It is what the program intends to do follow . But we are not at the end of our surprises, and this version

of the patch doesn't work either. Besides she doesn't treat only one protection out of the four (and treats it badly).

Like the patches are starting to take a long time to write , as writing code under SoftIce East painful and that I something else to

do that calculate manually the references of my jumps under SoftIce , we will use a compiler assembler to code

our programs (masm32 for example ). You might as well immediately use this method because protections are becoming more and more

complex, and patches Also . We win SO in time and patience .

Here is the procedure to follow : We compile our written patch in assembler to get a file executable , then we load this

executable in ADump . Once \_ Once we have reached the OEP of the game, we change the EIP to put it at the beginning of the code of our patch. The patch being

relatively simple, the header takes 0x200h bytes and the patch code begins therefore at +0x200h relative to the location where we have it

loaded into ADump . (This root's patch will therefore never be executed such which , and will therefore be always loaded with memory manually to

be used under SoftIce .) As before , it will be necessary always modify the SafeDisc reference calculation routine .

Now the code for this second attempt at fixing of the . rdata :

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Begin the RemoveSafeDiscPatch code < essay 2="">

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

title RemoveSafeDiscPatch <essay2>

.386

.model small, stdcall

option casemap :none

.code

; This patch compiled is to be loaded into ADump ( or any free memory segment ).

ADump equ 833DE000h ; Memory segment start address created by ADump

; ( to be modified in depending on its initialization).

; Constant dependent on the target program to be set ( Values for the game Mafia: Game.exe).

\_IB equ 00400000h ; ( target program ImageBase )

\_text equ 00001000h ; (section making part of the target program )

\_textSize \_ equ 002399BCh ; "

\_rdata \_ equ 0023B000h ; "

\_rdataSize \_ equ 000B4B7Ch ; "

start:

; == > \_ Procedure main scan of the .text section. We are looking for calls of type "Call [ref]" & lt ;= =

pushfd ; We save the registers .

pushad ;

mov edx , (\_IB + \_text ) ; Start of .text section

\_ mainLoop :

cmp word ptr [ edx ],15FFh ; Is this a “Call [ref]” (FF 15)?

i \_check1 ; If yes , we push parsing at \_check1.

\_ returnCheck : ; Return point for subsequent routines .

inc edx ; Increments the index edx for analysis next .

cmp edx , (\_IB+\_text+\_ textSize ); End of .text section?

jne\_mainLoop \_ \_ ; If not, then we continue.

popad ; We restore the registers .

popfd

int 3 ; End of analysis , return to SoftIce for dumping.

; == > \_ First verification procedure : Zone validity test for reference & lt ; ==

\_check1:

cmp dword ptr [edx+2], (\_IB+\_ rdata ) ; The call points to . rdata to .rdata+250? ( Relocated routines area )

jb\_returnCheck \_ \_ ; No, then This is not a call system relocated by SafeDisc

cmp dword ptr [edx+2], (\_IB+\_rdata+250h );

jae\_returnCheck \_ \_ ;

; Here we have a call [. rdata to .rdata+250]

; It remains to be verified if this call East rehomed by SafeDisc .

; == > \_ Second verification procedure : Pointed zone validity test & lt ; ==

\_check2:

mov eax , dword ptr [edx+2] ; Sets the pointer value to . rdata+n in eax

; If the call East between 02BE0000 and 02D00000 it is a call to

; SafeDisc protection routine : ~DF394B

; Cases already processed ( appeal \_ system +BC000000) are ignored at the same time.

cmp dword ptr [ eax ],02BE0000h ;

jb\_returnCheck \_ \_ ; If this is not a call to the SafeDisc protection routine , we

cmp dword ptr [ eax ],02D00000h ; do n't touch anything , because this address is “healthy”

jae\_returnCheck \_ \_ ;

; Arrived here , we restore the call original system .

; == > \_ Procedure for restoring call values systems & lt ; ==

\_ sysCallFix :

; Saving records \_ Before unnecessary call , SafeDisc function is clean.

; The protection routine is made to believe that it was called by a function

; of the game (Fake). (for info: CALL [ref] = push (eip+6), jmp [ref ])

add edx , 6 ; Call return address for the call to set.

pushedx ; \_ The SafeDisc protection routine uses the return address of the

; call calling (in the stack) to determine the call original system .

subedx , 6 ; We put back edx as before because we serves as pointer in the main scan loop.

jmp dword ptr [ eax ] ; Calling a SafeDisc routine which calls the general conversion routine in (E).

; ATTENTION ! Before running this patch, modify the SafeDisc conversion routine .

; For what return the address of the call original system in ECX.

\_keep on going: ; Return point for end of conversion routine changed .

pop ebx ; We trash the return value which was used for the conversion function (call fake).

mov dword ptr [ eax ], ecx ; Correct . rdata+n with the value of ecx ( call original system ).

jmp\_returnCheck \_ \_ ; Ok, we have dealt with a case , we continue.

\_ jumpHere : ; The modification (JMP) of the general conversion routine should point here .

no ; Reference for manual modification in SoftIce of the general conversion routine.

mov esp ,[ ebp+0Ch]; We replace the code that we removed | (8B 65 0C)

popad ; | (61)

popfd ; | (9D)

pop ecx ; Originally : RET (C3). The stack contains SO the address of the call system ,

; we replaces this ret by a pop ecx to recover this value .

jmp \_continue ; We return to our patch.

end start

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; End of code RemoveSafeDiscPatch < try 2="">

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Before continuing with the execution of this patch, a quick note on the access rights to the different sections of the program. If

YOU look at the .text and .text section attributes . rdata , you will notice that they are read only . \_ But our patch needs

describe In . rdata to break the first protection ( rewriting call references systems originals ). YOU have

as always several solutions and there is not just one way to proceed as we say always Larry Wall. YOU can either

modify the access rights on the memory ranges on the fly , or modify the attributes of these two sections in the header of the

program using LordPE for example . \_ This second technique is not valid only if no checksum is carried out on the program

And that's the case here . So don't you in deprive not because this method is more persistent (no handling to be repeated each time times what

YOU start the program).

Let's launch the patch then let's start by looking what conversions have data . Here is a small extract from the calls table

systems external In . rdata :

0063B000: BFE 81644 BFE 81644 BFE81534 BFE81644

0063B010: BFE8167 D 00000000 00981040 00000000

0063B020: 7CD73DF7 00000000 009C4860 00000000

0063B030:&gt;02BE04EF&gt; BFFA1B27 BFF764C0 &gt;02BE1338&lt;

0063B040: BFF77B57 BFF777AD &gt;02BE2181&lt;&gt;02BE2644&lt;

0063B050: BFF77B57 &gt;02BE2FCA&lt; BFF771D4 BFF77B8F

0063B060: BFF8516B BFF76DA8 BFFA13D7 BFFC6389

0063B070: BFFA1C7C BFF8516B BFF841CB &gt;02BE5F68&lt;

0063B080: BFF8E150 BFF75770 BFF9C947 BFFA1B27

0063B090: BFF76FD1 BFFA1B4A BFF77CC6 BFF77917

0063B0A0:&gt;02BE8A43&lt; BFF7E06D BFF77D9F BFF76A38

0063B0B0: BFF77D3D BFF7CE18 BFFC6055 BFF9DA56

0063B0C0: BFF75881 &gt;02BEB51E&lt; BFF9C63C &gt;02BEBEA4&lt;

0063B0D0: BFF8ABBC BFFA1B27 BFF9600D BFF9DB34

0063B0E0: BFF88BF2 BFF76EA3 BFF9160B BFF77716

0063B0F0: BFFA19F5 BFF75770 BFFA19F5 BFF7755A

The observation that we can do, it's that he stay a few references (02BE0000 - 02D00000) not processed ( indicator & gt ;& lt ;). We

may wonder if simply \_ these calls are not used by the program. It is nevertheless unlikely , because if the

compiler put them there , it's that he had a reason to do it (ok ok , the Microsoft compiler does not always react

logically , let's move on ...). It remains that this is not very normal and not clean. Let's move on to the testing phase, put put the code in place

that you have modified in the safety routines as a safety measure , as soon as times that one other protection uses it (and it is the

case ).

YOU YOU in doubt it , the program crashes miserably . I'm a donkey ... Well not entirely , because I could n't have suspected

of a thing before coming across it , and having first achieved all this . Now it 's this little detail that we are going to see .

6. Working solution for the reference problem systems of . rdata rerouted (Protection 1)

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯

First of all , when we analyze some cases on all calls \_ relocated of . rdata , we realize that one part of

addresses recalculated is all the same valid . A more in-depth analysis (reverse-engineering ) of the general routine for calculating

references systems original is not necessary , and it is sufficient to have a little imagination to understand what is happening.

Taken independently , our calculations are correct, but these reference calculations are not valid for all calls of the

program. In the unpatched game , two calls different to the same reference of . rdata give references \_ calls systems

different .

Reread these last two sentences, it must be -tilt-. It is what we \_ calls a phenomenon of insight ( phenomenon opposed to

a understanding logical and progressive). Let's move on ...

We can not so not one change the values once and for all of . rdata by pointing them to calls system originals . There

SafeDisc security routine \_ manages two parameters , and the method that we come to use don't takes in account one in staring

as constant the second (the .rdata table ). It's not so not possible as that .

He will SO need to use a little more our head. The solution I found (there are surely some others ) consists of

build a table. rdata temporary containing the references calls systems restored as for test 1 ( so false for one

part of the calls ). Then, for each call of the program to a reference relocated by SafeDisc , we look at the function system

original pointed out by this call and we recalculate the 'Call [ref]' in function of our new table. Finally , we copy OUR

new table on the old one . I didn't go into detail to give you the idea general , we will NOW see

this technique more closely .

- Stage 1: We start by building a new table. rdata Or each call system relocated East replaced by the reference of

the call original system . Of course , as we just said, this table will not be valid for all program calls ,

but it doesn't matter . This table is built from a small patch like the one we saw in test 1. (We did not SO

not written for nothing ;) That is to say that we sweeps . rdata at .rdata+250h (end of calls relocated ) in appellant each call rehoused . There

SafeDisc calculation function \_ \_ East as previously modified to return to our patch with the value of the call system

original in a register .

- Stage 2: The table constructed in step 0 is invalid for most calls (call [. rdata ] ) of the program. But we will

here correct all Calls in the program to make them valid . It's simple, when we finds a Call [. rdata ], we look if This

last is relocated , and if this is the case , we retrieve the call reference original system as usual . \_ We search SO

this reference retrieved from our new reference table . Once \_ this one found in the new table, we know at what

place to point the processed Call so that it call the right reference system . And the turn is plays ! In practice, as we

wanna to correct . rdata and not build a new section. rdataBis , we recover the index of the reference found in the news

table to add it to the start address of . rdata . We thus fix the calls in relation to . rdata , but we still have to copy the

values from our new table in . rdata so that the program calls are valid .

- Stage 3: All calls [. rdata ] rehoused we were modified , all that remains is to copy the new table onto the old one , and this

protection will only be one bad memory ( finally almost ...).

Once the patch carrying out the above operations has been launched , we now obtain :

0187:0062CC 35 FF 15ECB06300 CALL [KERNEL32!GetModuleHandleA] [ref: BFF77716]

and no more:

0187:0062CC 35 FF 15ECB06300 CALL [KERNEL32!GetEnvironmentVariableA] [ref: BFFA1392]

This first protection is well unleaded .

One remark, nothing tells us that the original program before being protected had the same table. rdata that our new table,

but that n / A no importance because all the calls are fixed and this therefore does not change nothing to the proper functioning of the program. He is

moreover probable that this is not the case ( but still one times , it doesn't matter).

I should here YOU present the patch code < tests 3=""> which is functional and which removes the first protection well, but I

won't do it not because if I continue as much detail and explain all my patches, I will write a book. I will so I

satisfy to explain the principles and resolutions of the other protections to save a little space and you find the code

three first protections united in a single patch which is located in appendix: ' RemoveSafeDiscPatch v1.0 Part A'. This patch is

enough commented to avoid a detailed explanation of how the code works . One more thing, the method we

we have just seen remove the first protection, but this patch alone will not work not because he is linked to a second protection,

it will be necessary SO remove protection one and two at the same time.

In the source ' RemoveSafeDiscPatch v1.0 Part A' provided in the appendix, the functions that we just talked about are processed in the

procedures :

- BuildTable ( Calculates the new call referencing table system )

- CallRefFix ( Fixes Call[ref] in function of the new referencing table )

- CopyTable ( Copies the new referencing table in place of the old one (in .rdata ) )

remain to be addressed , but two are close to the techniques mentioned above, including a very similar one :

- JmpFix ( Very similar )

-CallFix \_

Let's start by treating the one that most closely resembles the case previous .

external calls rerouted in JMP (Protection 2)

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

The second protection we are going to now see East linked to the first as I already said . She is managed by the same

reference calculation routine \_ systems that we have seen previously , so that we could very well have consider these two

protections like a alone . The reason for which I distinguish these two cases is that in this second protection the code of

.text section has been amended while for the case Previously , it was the section. rdata which had undergone modifications.

As for the last times , you will be able to spot this protection during a crash. Once the previous protection removed ,

he is legitimate to make a dump of the program and run this dump to see if our patch works well and if there is no remaining

protection. Of course , as there are still protections and as the SafeDisc security routines are not in effect . memory , play

crash. The point of origin of the crash (easy to trace ) is not Besides not far from the OEP and it is This case we are going to study .

The call that caused the crash is located in 6273 EA:

0187:006273 AE E 978600D00 JMP 006FD42B

This jump points to the stxt774 section like YOU can see it in YOU referring to the sections table. After having trace

a little code, we realize that it is still ~DF394B which is called in the end, and if we look closer we find \_ OUR

good old reference calculation routine system , this same routine that we just patched. This means that this JMP will in

called \_ a system routine . So , the principle of this system East as previously to reroute Calls [ ref. rdata ] to a

C- Dilla 's safety routine , except that in this case , the protection generator will transform the Call [ ref. rdata ] en

Jmp <ref stxt774="">. He will SO to need scan the program looking for these specific jumps to replace them with the

Call [ ref. rdata ] program initials before its protection by SafeDisc .

Here is the illustration in code of our example of JMP in 6273AE:

006273 AE E 978600D00 JMP 006FD42B --------|

|

006FD42B 53 PUSH EBX &lt;-------|

006FD42C E893FCFFFF CALL 006FD0C4 ----------|

006FD431 B8F166A704 MOV EAX,04A766F1 |

006FD436 8D12 LEA EDX,[EDX] |

|

006FD0C4 870424 XCHG EAX,[ESP] &lt;--------|

006FD0C7 9C PUSHFD

006FD0C8 058AFAFFFF ADD EAX,FFFFFA8A On se balade un peu dans stxt744...

006FD0CD 8B18 MOV EBX,[EAX]

006FD0CF 6BDB08 IMUL EBX,EBX,08

006FD0D2 035804 ADD EBX,[EAX+04]

006FD0D5 9D POPFD

006FD0D6 58 POP EAX

006FD0D7 EBA9 JMP 006FD082 -------|

|

006FD082 871C24 XCHG EBX,[ESP] &lt;-----|

006FD085 C3 RET -----|

|

02BFF251 0000 ADD [EAX],AL &lt;----|

02BFF253 68B4736200 PUSH 006273B4

02BFF258 68DC12EABF PUSH BFEA12DC

02BFF25D 9C PUSHFD

02BFF25E 60 PUSHAD

02BFF25F 54 PUSH ESP

02BFF260 6893F2BF02 PUSH 02BFF293

02BFF 265 E 8DFE04EFF CALL 020ED349 & lt ;& lt ;& lt ; But finally , we come back on familiar ground :o)

02BFF26 A 83 C408 ADD ESP,08

02BFF26 D 6 A00 PUSH 00

02BFF26 F 58 POP EAX

02BFF 270 61 POPAD

02BFF 271 9 D POPFD

02BFF 272 C 3 RET

To break this second protection, we proceed roughly in the same way way as before . We patch the general routine of

calculation of references systems so that it places the reference of the call original system in a registry and redirect it towards

our patch ( make sure this this time to modify the two return cases ). Then, the scanning phase of the .text section can

to start. Once we \_ \_ East on to have a Jmp valid ( i.e. a \_ reference call system transformed in Jmp ) we calculate

the destination address of the jump and we go there . Thanks to our modification, we will collect the reference original and return to

our patch. All that will remain is to transform the Jmp in Call [ref. rdata ]. To calculate the address of . rdata to point to, we

search in our new reference table system the index of the reference to call and we add the start address of . rdata .

If you do n't understand everything, read the function code JmpFix , that should clear up . If we now look at the values

hexadecimals of the code, it is necessary to replace E9???????? ( jmp <ref>) by FF15???????? ( call [ref]). YOU note that the call [ref] takes

one more byte, but this is taken into account accounted by the protection generator which does not touch this byte when it operates the

transformation (no change in code size). Once the call \_ system has been carried out , the return address is not located

not an instruction after the jmp , but one instruction plus one byte, which corresponds to the case of the unprotected program (call [ref]

original). Ahh, it’s still well done :o) We can therefore without risk overwrite the byte when we redo the calculation for our

call and put things back in order .

So that the patch of this protection is effective , you must have copied the new reference table system on the old

( in .rdata ) . Also , the first two protections being linked and they must be done successively without copying the new table

on the old one between the two patches , otherwise it would be impossible to find the references system originals .

Here is the diagram to follow :

call BuildTable ; 1) Calculate the new call referencing table system .

call CallRefFix ; 2) Corrects Call[ref] in function of the new referencing table .

call JmpFix ; 3) Converts jmp <ref> to Call [ref] with the new referencing table .

call CopyTable ; 4) Copy the new referencing table in place of the old one (in .rdata ) .

Once we have rotated \_ \_ our patch, in place of

006273 AE E 978600D00 JMP 006FD42B

we will get the call [. rdata ] corresponding :

006273 AE FF 1520B16300 CALL [KERNEL32!GetCommandLineA]

A small note in passing regarding the code of the functional patch ( RemoveSafeDiscPatch v1.0 Part A ): Rather than modify

manually each time times the general routine for calculating references system to adjust the jmp which points to the address of

return of our patch, it is more interesting to write a little piece of code that does this work automatically . It is especially

advantageous when we debug the patch, we win so a lot of time. Note that the modifications made are made in

memory segment read - only :

~DF394B .txt 0003 0187 :020EC000 00001E02 RO CODE

therefore necessary , before launching the patch, to modify the attributes of this hot zone with SoftIce to add the rights in

writing . We can of course automate this procedure with a little code in using the API VirtualProtect , or even do the

calculation of the table pages and modify the second bit of the page concerned , but This is beyond the scope of this tutorial. With IceDump ,

simply use the command /PAGEFLAG < addr > PWUC.

Protections 1 and 2 on rehousing calls systems external are now out. YOU can dump the program

and use it later as a backup. With IceDump , process as follows: (We can also use LordPE with a jmp eip to the OEP and

subsequently correct the EB FE to 55 8B using an editor hexadecimal . If you use IceDump , remember to disable the

breapkpoints which point to code to be dumped, otherwise YOU Go on YOU find with int3 (CC) in the dump file

locations of the bpx .)

/ dump 400000 30225c c:\tmp\Game\_12.bin

To take crack again or YOU have it left , launch the game ( the original , with CD in the reader ) and a times arrived at the OEP under

SoftIce ( always the same bpx 62733A) load the image you have done just Before :

/ load 400000 30225c c:\tmp\Game\_12.bin

Implementation \_ guard : The RemoveSafeDiscPatch offers to take care of the first three protections in succession if we look at the code,

but you must first launch the ' TableCallSkip ' routine and exit the game for this to happen. works . If you start from the image

Game\_12.bin to continue on the third protection, you will be able to use the same code but it will not be necessary execute that

functions ' TableCallSkip ' ( then exit or crash) and then ' CallFix ' ( These routines and their functioning are treated in the

following chapter ).

That 's it, we can now move on to the third protection, we're getting down to business now ! Nah, just kidding , she East

very simple to crack, besides there is nothing really difficult to crack in SafeDisc .

rerouted internal calls (Protection 3)

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

One more times , you will discover this protection as YOU have find the others , that is, by observing the crashes. THE

first effects of this third protection lies also not far from the OEP . I move on to the implementation evidence of the problem for

explain directly what the protection does ( you need to start knowing music). It is for this times of a Call

internal rerouted in internal call (403389). This internal call will in turn call a SafeDisc security routine ( a \_ other this

times ) to calculate the reference original of the internal call .

Here is the first example find :

00627394 E 8F0BFDDFF CALL 00403389

who, a times past the SafeDisc calculation routines brings us in : 62CDA0. (The original Call was therefore a call 62CDA0.)

If you study several case , you Notice that the replacement call still points to the same routine: CALL 00403389.

Now if \_ we enter this \_ function and we trace a little code, we will arrive finally in ~DF394B that we knows

GOOD. Be careful however , this is not the same reference calculation function \_ \_ original than previously (It is necessary to change a

little , otherwise That becomes monotonous :o).

Here is a small extract of what we found under these famous CALL 00403389:

00403389 51 PUSH ECX

0040338 A 50 PUSH EAX

0040338 BE 854FCFFFF CALL 00402FE4 ----------

... |

|

00402FE4 B83B950000 MOV EAX,0000953B &lt;--

00402FE9 59 POP ECX

00402FEA 03C1 ADD EAX,ECX

00402FEC 8B00 MOV EAX,[EAX]

00402FEE FFE0 JMP EAX ----------

... |

|

020AFF1E 58 POP EAX &lt;---------

020AFF1F 59 POP ECX

020AFF20 6800004000 PUSH 00400000

020AFF25 9C PUSHFD

020AFF26 60 PUSHAD

020AFF27 8B442428 MOV EAX,[ESP+28]

020AFF2B 8BCC MOV ECX,ESP

020AFF2D 83C124 ADD ECX,24

020AFF30 50 PUSH EAX

020AFF31 51 PUSH ECX

(H) 020AFF 32 E 8D2FFFFFF CALL 020AFF09 & lt ;& lt ;& lt ; The other reference calculation function \_ \_ originals .

(I) 020AFF 37 59 POP ECX (for internal references )

020AFF 38 58 POP EAX

020AFF 39 61 POPAD

020AFF3 A 9 D POPFD

020AFF3 BC 3 RET

Always with the same logic , guess what we \_ go TO DO ? Modify the routine above of course in order to recover the value of

the original internal call and return to our program. We go so place our jump in (I) and restore the code that we overwrote

with our jump one times arrived in our program. Instead of the RET we will place a pop edx to have the value of the call

original internal and the turn is plays .

Here we \_ gets after modification:

020AFF 32 E 8D2FFFFFF CALL 020AFF09

(I2)020AFF 37 E 931243481 JMP 833F236D

And of course it is necessary rewrite the code overwritten by the JMP. This is done in our patch:

833F236 D 59 POP ECX

833F236 E 58 POP EAX

833F236 F 61 POPAD

833F 2370 9 D POPFD

833F 2371 59 POP ECX

833F 2372 EBE 2 JMP 833F2356

Some explanations on how the patch works for this third protection:

We scan the .text section (which contains the game code) looking for call 403389. But be careful, to search for this

string , you should know that the Call <ref> reference is not encoded statically , but relative to the position of this call

( like a jmp ). That is to say that the four bytes following E8 are given in relation to the position of this call+5 ( address of

return of the call). We go SO scan the program looking for 'E8' and a times this byte identified , we will take the next four bytes

to calculate dynamically the reference point . If the reference calculated East equal to 403389, then we found our call .

This method is not perfect , because we can very well fall between two instructions and believe having found a call 403389 then

that he don't East Nothing . To avoid this , it would be necessary to use a function ' IsValidInstruction ' to ensure that we did not fall

between two instructions during our scan . However , the probabilities of coming across a case of call 403389 in a call program

less than 3 MB is very small and this simple check East amply sufficient.

Let's continue , a once a call is spotted , we will jump to routine 403389 in putting previously the return address of the processed call

in the stack. It's a fake as we already did for the previous protection . Once the value has been calculated carried out by the

SafeDisc routine , we will return to our patch thanks to the modification made in (I2). ( Noticed : This procedure of

modification is done automatically in the final patch RemoveSafeDiscPatch v1.0 Part A.)

Once the patch is executed , the third protection should be removed . But there you go, in the series of good surprises, we have a

case call special that fucks up shit . It turns out that that he exists in the CALL 403389 program which , times rerouted , point

on a further instruction. Here is an example :

0062 DFDB E 8A953DDFF CALL 00403389

In effect , when we look at Or we arrive after passing the SafeDisc routines , we find the value 0062DFE0, which is

in fact the address the instruction next . It is perfectly stupid , but it is especially problematic to the extent Or That

disrupts the internal call reference calculation routine . In effect , if we launch the calculation routine with this address

Then that we then use it on a other address valid (from call 403389), the reference of this second treatment go be erroneous .

In fact the reference is not going to be totally false , but shifted .

I resume , when we calls a call 403389 which will point to eip+5 (I would call it now 'call stupid ') , all

references later calculated will be shifted by a value n which varies in call function . \_ What I just said is

partially false, because, if all valid call 403389 references will be modified , however the call references

morons remain good ( she point always at eip+5). Indeed , the SafeDisc routine always manages to determine the

stupid call references :o) It's important because it's which will allow us to correct the problem . \_

Indeed , we can correct all calls 403389 without problem if we know avoid stupid calls so as not to create

offsets . He will SO to need create a table of stupid calls to avoid . The problem is that we ca n't know in advance

if a call is stupid or not (taken out of context , this tutorial would be really very stupid , then please , don't quote me :o) To find out

if a call is stupid , we have to go see (ok, ok I adds on purpose a little ; ). If the reference obtained worth eip+5

( instructions following the processed call ), we add it to the ' BlackList '. Of course, a offset value \_ East added , but this does not

It doesn't concern our stupid calls . Once the list \_ built , it will be necessary restart the game to avoid the lag problem . We

will be able to SO correct all calls 403389 in avoiding stupid calls . I recognize that this method is far from clean and it

must exist a best solution, but She works and I didn't dig much more (seen differently , it's a good report

crack/time).

In done , if YOU have done some tests, you have of will notice one thing, stupid calls are always preceded by a RET.

This could be a heuristic to spot them , but to her alone , this tracking technique is a little light and it can be

that we end up with unprocessed internal call cases . Since I have n't tried this method , I wouldn't go any further.

YOU can always try this way to see Or That led . I have Also note a few others oddities in the program like

CALL 403387, which does not point to a valid instruction . These calls do not seem however never be executed .

Okay, all that That’s all well and good, but what to do with stupid calls ? It's simple, since they point to a further instruction and

that we cannot leave them ( because they call the SafeDisc routine ), we will simply nope them . Five little nods for a call

stupid , pack It is weighed .

If I resume the operation of the patch for this third protection, you must first start by calling a routine

specific to our patch which will build the ' BlackList '. This is the ' TableCallSkip ' routine. Then , kill and restart the game

to finally call the ' CallFix ' routine which will properly remove the protection 3. If you want to do the first three

protections in one go , it is therefore necessary previously run ' TableCallSkip ' and kill the game.

If everything went well, the call:

00627394 E 8F0BFDDFF CALL 00403389

must be replaced by:

00627394 E 8075A0000 CALL 0062CDA0

Here is a small extract from the BlackList to give you an idea of the number of stupid calls to avoid :

833F4000: 0040CC3 B 0040 F2EB 004108AB 0041093B

833F4010: 004109CB 0041245 B 0041278 B 0042537B

833F4020: 00425FBB 0042FBCB 0043271B 004335DB

833F4030: 0043A25B 0043ABEB 0043B0FB 0043BFDB

.

.

.

833F4270: 006064FB 00606DBB 00607F4B 006089EB

833F4280: 0060B17B 0060CB7B 00610C6B 0061155B

833F4290: 0061680B 006184DB 0061DF4B 0061E62B

833F42A0: 006245AB 0062DFDB 00000000 00000000

We can ask ourselves if these stupid calls are a form of protection or not. Did the protection generator intentionally

created these calls with the aim of deceiving us Or is this only a side effect of protection? Personally , I do n't think

that this either protection , but if that's it East one is \_ Really stupid protection ; o )

If you want to experiment manually to test this protection, here is a few case that I have spotted :

Valid case of call 403389:

627394 CALL 403389 -& gt ; 62CDA0

62D4A4 CALL 403389 - > 626B03

NOW the call that causes the first crash if we don't skip stupid calls ( always in cases valid ):

54976B CALL 403389 - > 549F00 ( Reference original , or if we avoided stupid calls \_ previous )

- > \_ 54AF6C ( Reference wrong , if we did n't avoid the stupid calls previous )

Invalid case ( or stupid call ):

62DFDB CALL 403389 -& gt ; 62DFE0 (causes an offset of +32)

( And the extract from the BlackList East enough for me not to lie down ...)

That's it for this third protection including YOU find the code attached in ' RemoveSafeDiscPatch v1.0 Part A'. He doesn't

all that remains is attack the last one . But before you start , dump the game and call it randomly 'Game\_123.bin'. This

image will serve as a backup, but she will serve us also to rebuild the future Game\_1234.bin. Proceed also a few

tests: Start the original game, load the dump memory and check that:

- the Calls [. rdata ] appear correctly with their equivalence name (ex: 627360: CALL [KERNEL32!GetVersion])

- the Jmp that we have treated have disappeared to make way for Call [. rdara ] (ex: 006273AE:

CALL [KERNEL 32!GetCommandLineA ]

- internal calls relocated were \_ modified (ex: 627394: CALL 0062CDA0)

And of course the patched game (123) works well.

We can now \_ approach the last protection, remember your breath... To push a big sigh because there is Really Nothing

hard.

9. Study and solution for code modified on the fly (Protection 4)

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

This fourth and final anti -dumping protection is my most interesting opinion to analyze. It's about with a little trick

allowing you to redirect the program's eip on a SafeDisc routine . This nice little capture system will start the routine of

protection when he goes meet interrupt 3 for example . But interrupt 3 is not the only instruction to be used

and there is all a panoply of instructions which are used to redirect execution of the program.

If we look at how this protection works , we discover that one once these specific instructions \_ are executed and

security routine has finished its work, the program returns to the same point ! ( and not an instruction further) And also

that the instruction specific trigger \_ its transformed as if by magic in a other instruction. The instruction of

triggering therefore \_ summer She even be modified , and this replacement instruction East actually the program instruction

original ( unmodified by the protection generator ).

The protection generator will SO choose some instructions ( randomly maybe ) to replace them with instructions

specific triggers ( such as INT 3). When the program passes these instructions, it will execute the routine

protection that goes put things back order Then return to the same point, the program can then continue as normal . We can

Also note that these changes are definitive . When launched , the game contains a lot of code to modify at the time .

stolen , and as it is used , the modifications to be made become fewer .

We can so immediately think about dumping the .text section after a or two hours of games, but It is a very

bad idea, because we will forget always a few case that we will end early or later by taking into account Full head...

I have enough spoken , let's see right away some examples of these famous changes:

- To start, a case of SGDT:

005920 FD E 8522C0900 CALL 00624D54

00592102 &gt;0F01C4& lt; SGDT ESP

00592105 A 3A4F86E00 MOV [006EF8A4],EAX

who a times amended will give the following code :

005920 FD E 8522C0900 CALL 00624D54

00592102 &gt;83C404& lt; ADD ESP,04

00592105 A 3A4F86E00 MOV [006EF8A4],EAX

- Case of RSM ( which screws up the SoftIce disassembly engine )

0055CD 30 E 8B17F0C00 CALL 00624CE6

0055CD35 &gt;0FAA RSM

0055CD 37DB 4B ESC

0055CD 39 3 B<f7 cmp ="" esi,edi ="" 0055cd3b="" e3c1="" jecxz ="" 0055ccfe="" 0055cd3d="" ea038956085f5e="" jmp ="" 5e5f:08568903="" 0055cd44="" 5b="" pop="" ebx ="" which=" " once ="" modified ="" will give ="" the following =" " code="" :="" ( 5="" bytes="" modified )="" 0055cd30="" e8b17f0c00="" call="" 00624ce6="" 0055cd35="">B8398EE338&lt; MOV EAX,38E38E39

0055CD3A F7E3 MUL EBX

0055CD3C C1EA03 SHR EDX,03

0055CD3F 895608 MOV [ESI+08],EDX

0055CD42 5F POP EDI

0055CD43 5E POP ESI

0055CD44 5B POP EBX

- Yet another case , on the INT 3 this times :

004713A 2 E 859331B00 CALL 00624700

004713A7 > CC INT 3

004713A 8 CC INT 3

004713A 9 CC &lt ; INT 3

004713AA 83EC0C SUB ESP,0C

who a times amended will give the following code :

004713A 2 E 859331B00 CALL 00624700

004713A7 &gt;8D4D20& lt; LEA ECX,[EBP+20]

004713AA 83EC0C SUB ESP,0C

Now let's look at the routine that is called to do the substitution work :

020BAEC 4 9 C PUSHFD & lt ;& lt ;& lt ; Entry point into the routine (when executing RSM for example)

020BAEC 5 57 PUSH EDI ( Interesting to monitor )

020BAEC 6 56 PUSH ESI

020BAEC 7 52 PUSH EDX

020BAEC8 51 PUSH ECX

020BAEC9 53 PUSH EBX

020BAECA 50 PUSH EAX

020BAECB E8B4FDFFFF CALL 020BAC84

020BAED0 58 POP EAX

020BAED1 5B POP EBX

020BAED2 59 POP ECX

020BAED3 5A POP EDX

020BAED4 5E POP ESI

020BAED5 5F POP EDI

020BAED6 9D POPFD

020BAED7 8D64240C LEA ESP,[ESP+0C]

020 BAEDB FF 6424FC JMP [ESP-04] & lt ;& lt ;& lt ; Return to instruction address in course

He is interesting to monitor this routine to see what are the instructions that cause it to be called . Once we have \_ \_

treaty a small series of cases , we can try to find a generalization . ( Note: to see directly the code of the instruction

to trigger under SoftIce , use the following macro :

macro VIEW = 'd @(esp+8); u @(esp+8);'

with a: bpx 020BAEC4 do 'VIEW')

Here are the instructions I used noted :

SGDT EAX / SGDT ECX / SGDT EDX / SGDT EBP / SGDT ESP

SIDT EAX / SIDT ECX / SIDT EDX / SIDT EBP / SIDT EDI

RSM / UD2 / INT 3

Here is the generalization I found for performing the hexadecimal search in the program code:

SGDT &amp; SIDT - > 0F 01 C?

RSM - > 0F AA

UD2 - > 0F 0B

INT 3 - > CC

He will in effect to need Browse the entire .text section to search all occurrences that the generator

protection generated . Once we have detected a case , we jump on it to let the transformation take place all alone and we

returns to our patch in previously modifying the end of function 020BAEC4 .

But here there is a problem middle finger , chains wanted are very short , and the cases invalid are many . A simple

'MOV EAX, 0FAA' will be detected as string to modify. Lucky for us, the folks at C- Dilla write great routines

clean with a lot of checks , so that if we start processing a case wrong , the routine will not no modification.

So , we are going use and abuse it :o) Well, we're not going to push the envelope too much when even in passing him all the code... We're going him

the work a little easier by carrying out a first filtering , that is to say by looking for strings 0F01C ? / 0FAA / 0F0B / CC.

Like since always , we will use the strength of the adversary ( or instead energy ) as in I don't know what martial art anymore .

Except that here , we reuse the code that has already been wrote to our advantage . As the game works , there is necessarily somewhere one \_

function that does the work, all we have to do is locate it and find when it is easiest to

call to get the job done. Null need to reverse all the functions , it just takes a little observation and the tour is plays .

If you have Understood that , you you understood everything ! Furthermore, we are programmer , and a programmer it's lazy, then if we can in

do as little as possible, we wo n't bother = )

On this occasion , we had a lot of luck because the functions were clean , which saved us time. If this did not have

not been the case , it would have necessary to write a function ' IsValidInstruction ' to check that we is of course on an instruction and not

between two. Perhaps future versions of SafeDisc will not be as ' Safe', so as an exercise for the next

times , write Me this function :)

A small remark in passing, the few values hexadecimals that are after the trigger instruction don't have none

relationship with the replacement instruction , this are in some sort of padding bytes .

Let's see NOW using the latest patch because it is a bit special. YOU note that he been set apart in

' RemoveSafeDiscPatch v1.0 Part B' and this is not for nothing . In Indeed , it is necessary to dump the program between part A and part

B because to launch the second patch (part B), because we will need the game in course execution to be able to make the second patch. THE

program will therefore already be initialized and the image memory will no longer be valid . It will be necessary rebuild cracked game from dump \_

intermediate and the new patched .text section .

Before launching the ' RemoveSafeDiscPatch v1.0 Part B', the 20BAEC4 protection routine must also be in a state

allowing its good execution , which is not the case at the OEP . To enable the patch to work , the trick consists of launching

the game in posing a breakpoint at 20BAEC4, and a times income from the protection routine in the game ( treatment of the first case in

55CD35), we change the eip to point it to our patch B. This is even in fact a little more complicated than that , because we distinguish

two cases :

- the SGDT / SIDT / RSM / UD2

- INT 3

It is necessary to treat these two cases in depending on their context respective . If we stop in 55CD30 ( first case ), we can to treat

all the SGDT/SIDT/RSM/UD2 ( in short 0F???) but not the INT 3. We will so for this times launch the routines: ReconstructP1,

ReconstructP2 and ReconstructP3. He's staying then the case of INT 3 to be treated . For this , it will be necessary let the program continue and

interrupt on a case valid CC. As there will only be these case , a simple breakpoint in 20BAEC4 will suffice . Once \_ This case

processed and back in the game code, we can then launch the last routine of the patch which takes care of INT 3. That's it for

use of patch B which is I you grant it , a little twisted . To treat everything in one piece , it would be necessary push further

searches and a little reversing the functions . If you you have time, why not...

I do not detail how the program works , although YOU are arrived so far , you YOU doubt the way he proceeds ,

and if This is not the case , take a look at the code.

10. Fix the header and rebuild the import table

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯

now have a nice dump of the game where the protection of C- Dilla East removed . He will SO to need give back this working dump

and this on all machines. Let's get started Simply , we will first make it work on our PC . For that , he goes

First of all have to change the EP of the program which is remained on the loader EP . Using LordPE \_ \_ or ProcDump , so change

the EP of the dump to put it in the OEP less the ImageBase , i.e. in 22733A (62733A - 400000) for the game Mafia. At this point, if YOU

rename the dump to .exe and if YOU launch the game, everything will go well until the first library call external that we will not have

not rerouted (LS3DF.dll for example ). The game will crash then because this library was not \_ loaded when starting the game . This East

perfectly normal because the import table is sucks . ( Calls to kernel32, user32 and advapi32 work well

only because their \_ references are encoded statically (by our patch) and that these dll are always in memory .)

Okay, back to our problem , we are now going rebuild the imports table to render our game cracked a little more

portable. Here again, there are several solutions, The root's solution which consists of doing half the work by readjusting

manually the import table pointers and then call a rebuilder which rebuilds the 'hint name array' and

function names \_ \_ corresponding from the physical addresses of the functions ( encoded statically by our patch) again

called 'import address table'. I will re-explain the problem so that you get it right. In our dump, the import table

contains addresses \_ static for kernel32, user32 and advapi32 dlls . The program works so good on our machine, but

not on one another platform because these \_ addresses static change from one version to another . To enable compatibility , in time \_

normal, the loader rebuilt these addresses static from function names ( ex : \_ \_ GetVersion ). But in our dump, these

information are missing . He will SO have to reconstruct them from the addresses static , and we will use for this a

rebuilder ( except if YOU have want to take the trouble of searching and adjusting 100 references ...).

In short , in all cases we will need a program to reconstruct the import table, so as much

choose the simplest and fastest method . We go therefore opt for the whole solution automated , for lazy people

as me (I remind you , less I have do , the better I feel :o)

For this solution I use 'Import REConstructor v1.3' from MackT (UCF). This little program is perfect for what we wanna

TO DO. Here's how to do it : Launch the original game (with the CD in the drive ) and a times arrived at the OEP , load the dump

Game\_1234.bin (with OEP modified , it will already be done for the rest as That ). YOU can SO let the game run . Switch

then on ImportREC and select the process target . Run an AutoSearch IAT then a GetImport . YOU should to notice all

dlls ( and function calls ) supported \_ account by the game. All that remains is to launch a FixDump in selecting the file \_

your dump ( little imported if This is not an executable ). ImportREC will use this \_ \_ file for \_ build a new one staring

the import table. The dump fixed has an extra '\_' at the end of its name). There you have it , the import table of our

dump is fixed automatically in less than 30 seconds . Thank you to MackT and UCF for their friendly proggies.

Okay, last modification, let 's now rebuild this dump with LordPE ( or ProcDump ). For LordPE ensure that in the

rebuilder options the ' Dumpfix ' and ' ValidatePE ' options are activated . YOU can Also select a realignment of the

dump, your dump will only be smaller. YOU can then launch 'Rebuild PE' and admire the work. Your executable is more

little one, he found its beautiful icon and above all, it works , without the SafeDisc protection routines and on all

platforms . \_ It is won !

Note that the crack fixed now contains \_ one more section than ImportREC to add . If you find that it doesn't look clean,

YOU can always repatch the fix, to reinstall the modifications in . rdata . But in any case , remember to thank the

authors of the tools you use , otherwise This is not very fair play.

Another alternative, rather than going through an external rebuilder , you can also use IceDump ( command /PEDUMP), this one

included the Phoenix engine written by G-Rom. We find thus the very familiar ProcDump options in what concerns the part \_

reconstruction of the import table (/option P).

Game- specific CD-Check patch

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

For a little , I would have it forgotten ... The biggest difficulty of this crack :o) If you launch your crack fixed blazing nine out of one

other bike , you will be entitled to the following message : 'Please insert MAFIA CD 1' which earned me a crazy laugh until my jaw drops .

( well , yes , the CD is not in the player since I have changed PC...) After SafeDisc protection , a bogus CD-Check from bogus!

In fact that should be the protection of the game before Illusion Softworks did call on C- Dilla to protect their product , and

they must have forgotten Or be too lazy to take it off . Finally all this was very funny :) I'll give you a shot all the same a copy screen

for you go up to what looks like :

00624490 56 PUSH ESI

00624491 8 B350CB26300 MOV ESI,[USER32!MessageBoxA]

00624497 E 864000000 CALL 00624500

0062449 C 84 C0 TEST AL,AL

0062449 E 7525 JNZ 006244C5 & lt ;& lt ;& lt ; It was really hard to find ...

006244A 0 A 128926F00 MOV EAX,[006F9228] Jump Me That fissa and we don't talk about it anymore.

006244A 5 85 C0 TEST EAX,EAX

006244A7 7406 JZ 006244AF

006244A9 8B08 MOV ECX,[EAX]

006244AB 50 PUSH EAX

006244AC FF513C CALL [ECX+3C]

006244AF 6A15 PUSH 15

006244B1 6A00 PUSH 00

006244B3 68F8176500 PUSH 006517F8

006244B8 6A00 PUSH 00

006244BA FFD6 CALL ESI

006244BC 83F802 CMP EAX,02

006244BF 75D6 JNZ 00624497

006244C1 32C0 XOR AL,AL

006244C3 5E POP ESI

006244C4 C3 RET

006244C 5 B 001 MOV AL,01

006244C 7 5 E POP ESI

006244C 8 C 3 RET

When we finished On that note, we say to ourselves that it really was n't hard ;)

No mysteries , no magic , just logic ! For me , it's That ’s cracking. YOU will notice also that in addition to the fact that

the game launches faster at startup , the cracked game must logically run a little faster than the protected game due to

that he no more security routines to execute . Morality , crack then your games before from there play , you will save time

startup and a few FPS ;) Finally , you can write an automatic remover if YOU are motivated , there are a few

difficulties interesting that you wait .

12. Conclusion and outlook

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

We have now reached the end of this tutorial, and if YOU have could learn something , I would be happy . The protection of

C- Dilla is certainly long to remove , but there is nothing difficult if we knows how to operate the management functions of

protection that they have writing . We could moreover, imagine anti-dumping systems that are much more complex and more difficult to

destroy only those we \_ just found . \_ For example , the difficulty would increase if the rerouting of internal functions and

external no longer passed through an alone function but by hundreds . More difficult of course, but Nothing impossible since he

enough to write a function patcher automatic which identifies these functions and modifies them . Even more protection

interesting would be SO to include polymorphism in these hundreds of functions of sorts that they are even harder to

spot (I remind that we must return to our patcher in all cases for our method ). But still in times , nothing

impossible since we \_ can write a tracer scan that monitors the sp . Concerning modifying the code on the fly , the

C- Dilla people would have could have done better , because they only replace invalid code with valid code . They would have could do

a double process , on the one hand rebuilding to the eip and on the other side ' blur ' elsewhere so that we can never have

a memory image valid . Finally , never say never because it is simply enough to short- circuit the part of the routine that

destroys the code elsewhere , the only difficulty being then to find this place ( that’s what reverse- engineering is for ). More

complex now , let's imagine a system that does hold several routines of size n in one space memory of size n in

substituting them each in turn so that all the routines are never in memory . That 's no longer a problem

difficult to solve , but to put also in place since these routines must not be done mutually reference , which \_

comes down to solving a problem reachability in a graph . And this is not as simple to solve on a program that

weighs several megas . From this side we are quiet ( examples toy can still \_ function ) and we have time to see

come . All this reminds me of the glorious Atari/Amiga era :) Ahhhhhh , I'm living again ! Finally , after so many years of mediocrity in

commercial protections , it's starting to become interesting . I can't wait to see the next ones . (Note that it there are very good

CrackMe ingenious which far exceed commercial protections )

All this makes me think of something , there are CrackMe , ReverseMe , KeyGenMe ... And if we was creating DumpMes ?! We

could easily flouting trade protections in term of complexity of anti-dump technique and their put 10km in the

view :) At the moment I write these lines , I have not yet seen one that offers this concept ( on the other hand, some CrackMe include

already anti-dumping protections in preventing for example a simple reading of their memory zone to dump them passing through

a windows API). SO if That YOU sing , Let's code!

13. Warning , author 's notes

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

If you YOU wait here a disclaimer, it's not won ... I do n't want to tell the spiel usual : be careful...

thing ... it's not good... over to you risk and peril ... provided for educational purposes only ... the author clear everything

responsibility ... does not encourage no one beyond passing the law ... patati et patata ...

I want just to specify one thing I have cracked I game Mafia certainly, but I have bought This game ! Yes I have paid to play , and I

plays ! I enjoyed finishing the game , and like somewhere I have Also paid protection for the game, I'm having fun with it. Normal

No ?! :o) This tutorial is not Nothing else only help for those who want Also enjoy as Me . There you go, so I am a

cracker ;) And I don't consider not that what I \_ do either illegal since I reward software authors by buying

their products , and I do not distribute any crack that would reduce their profits. All I spread is knowledge, knowledge .

Free to you to do with it what you want .

Peex

14. Acknowledgments

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

I want to thank :

Yoda, for his excellent programs as well as for his explanations.

All the team that designed and created the excellent IceDump , the do-it-all plug-in.

Z0mbie, for these very good articles, these research , its translation effort and the dissemination of its code.

Frog's Print &amp; Spar , which have wrote the program that gave me definitely saved the most time.

C- Dilla , who suggested to us here for hours of fun .

Illusion Softworks, for the development of the game Mafia which I really liked .

so all the authors of the proggies that I use .

And I especially thank :

Cork, for his help and encouragement .

Greetings to the entire French-speaking community of coders and see you soon .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

A N N E X E S

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Start of code for RemoveSafeDiscPatch v1.0 Part A

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

title RemoveSafeDiscPatch v1.0 Part A (Crack & Tutorial by Peex )

.386

.model small, stdcall

casemap option :none

.coded

; This patch compiled is to be loaded into ADump ( or any free memory segment ).

ADump equ 833DE000h ; Memory segment start address created by ADump

; ( to be modified in depending on its initialization).

ADumpPatchCode equ (ADump+200h); Start of patch code at 200 (512 bytes for the PE header).

ADumpPatchData equ (ADump+1000h); Data start at 1000 ( used to store the new referencing table ).

ADumpPatchData2 equ (ADump+2000 h) ; Data start at 2000 ( used to store the blacklist).

; Addresses of SafeDisc routines to modify (see tutorial).

fct2fix1A equ 020ED4B5h ; (Routine for calculating the reference of Call[. rdata ] and jmp <ref>)

fct2fix1B equ 020ED6feh ; (same, but used When the address has already been calculated )

fct2fix2 equ 020AFF37h ; (Routine fixing Call <ref>)

; Constant depending on the target program to be set ( Values for the game Mafia: Game.exe).

\_IB equ 00400000h ; ( target program ImageBase )

\_text equ 00001000h ; (section making part of the target program )

\_textSize \_ equ 002399BCh ; "

\_rdata \_ equ 0023B000h ; "

\_rdataSize \_ equ 000B4B7Ch ; "

\_stxt774 equ 002FC000h ; "

\_stxt774Size equ 0000204Fh ; "

; EntryPoint du RemoveSafeDiscPatch.

start:

pushfd ; On sauvegarde les registres.

pushad ;

call BuildTable ; 1) Calculate the new call referencing table system .

call CallRefFix ; 2) Corrects Call[ref] in function of the new referencing table .

call JmpFix ; 3) Converts jmp <ref> to Call [ref] with the new referencing table .

call CallFix ; 4) Fix rerouted Call <ref> .

call CopyTable ; 5) Copy the new referencing table in place of the old one (in .rdata ) .

popad ; We restore the registers .

popfd ;

int 3 ; The games are here fixed , the dump is to be done at this time (from \_IB to \_IB+IS).

; Once \_ dumped , put the eip to the OEP (of the game) and verify that the game is working

; correctly .

call TableCallSkip ; Call blacklist calculation routine <ref>

; This routine should be started before all others to build the list and the game

; must be restarted .

int 3 ;

BuildTable proc

; == > \_ 1 & lt ;= = First part Calculates the new call referencing table system ( stored in ADump ).

mov eax , \_ cleanReturnPoint\_ BuildTable ; Patch the call determination routine ( SafeDisc ) original system

call PatchMagicSafeDiscFct 1 ; and returns it to \_cleanReturnPoint\_BuildTable .

mov eax , (\_IB+\_ rdata ) ; Start of section. rdata .

mov ebx , ADumpPatchData ; Destination for copying references relocated (in ADump , following the program).

\_loopBuildTable : \_

cmp dword ptr [ eax ], 02BE0000h ; Checked if the call system has been relocated by SafeDisc (Routines between 2BE0000 - 2C00000).

jb\_copyRef \_ \_ ; If this is not a call to the SafeDisc protection routine , we

cmp dword ptr [ eax ], 02C00000h ; do n't touch anything , because this address is “healthy”.

jae\_copyRef \_ \_ ;

; Arrived here we are on to have a call system rehomed by SafeDisc .

jmp dword ptr [ eax ] ; We search SO the call corresponding original system in calling the SafeDisc routine

; patched previously ( this is in fact the subroutine which is patched ). We have then the reference

; of the call system in ecx when we returns to \_returnPoint\_BuildTable .

\_ returnPoint\_BuildTable :

mov dword ptr [ ebx ], ecx ; We store the reference to the call system in our referencing table .

; ( in the part Adump data : ADumpPatchData ) \_

jmp\_continueBuildTable \_ \_ ; And we continue.

\_ copyRef :

mov ecx , dword ptr [ eax ] ; It 's not a call relocated , we copy it such what .

mov dword ptr [ ebx ], ecx ;

\_continueBuildTable : \_

add eax , 4 ; We move forward one dword in . rdata .

add ebx , 4 ; Likewise for our new referencing table in ADump .

cmp eax , (\_IB+\_rdata+250 h) ; We loop as long as we did not arrive at .rdata+250h (End of calls rehoused )

jne\_loopBuildTable \_ \_ ;

ret ; End of routine 2, return to caller .

; Return point function SafeDisc patched . First of all , you need to restore the data

; overwritten by the jmp . ( cost of 5 bytes)

\_cleanReturnPoint\_BuildTable : \_

mov esp , [ebp+0 Ch] ; We replace the code that we removed .

popad ; |

popfd ; |

pop ecx ; Originally : “Ret”. The stack contains the address of the call system ,

; a ret at this moment therefore launches the call system . We replace it with a pop ecx to

; to recover this value .

jmp\_returnPoint\_BuildTable \_ \_ ; We return to our patch (Everything is clean).

BuildTable endp

CallRefFix proc

; == > \_ 2 < = = Second part Corrects the program's "Call[ref]" references (.text section ) function of the

; new call referencing table system .

mov eax , \_ cleanReturnPoint\_ CallRefFix ; Patch the call determination routine ( SafeDisc ) original system

call PatchMagicSafeDiscFct 1 ; and returns it to \_cleanReturnPoint\_CallRefFix .

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopCallRefFix : \_

cmp word ptr [ edx ], 15FFh ; Is this ( possibly ) a “Call [ref]”?

i \_checkCallRef1 ; If yes , we push parsing to \_checkCallRef1.

\_ returnCheckCallRef : ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne\_loopCallRefFix \_ \_ ;

ret ; End of routine 2, return to caller .

; First verification procedure : Zone validity test for reference .

\_checkCallRef1:

cmp dword ptr [edx+2], (\_IB+\_ rdata ); The call points to . rdata to .rdata+250? ( Relocated routines area )

jb\_returnCheckCallRef \_ \_ ; No, then This is not a call to a call system ( relocated or not).

cmp dword ptr [edx+2], (\_IB+\_rdata+250h );

jae\_returnCheckCallRef \_ \_ ;

; Arrived here , we have a Call [. rdata to .rdata+250].

; It remains to be verified if this call East rehomed by SafeDisc .

; Second verification procedure : Pointed zone validity test . \_

\_checkCallRef2:

mov eax , dword ptr [edx+2] ; Sets the pointer value to . rdata+n in eax .

; If eax East between 02BE0000 and 02D00000 it is a call to

; SafeDisc protection routine : ~DF394B.

cmp dword ptr [ eax ], 02BE0000h ;

jb\_returnCheckCallRef \_ \_ ; If this is not a call to the SafeDisc protection routine , we

cmp dword ptr [ eax ], 02D00000h ; do n't touch anything , because this address is “healthy”.

jae\_returnCheckCallRef \_ \_ ;

; Arrived here we are on to have a call system rehomed by SafeDisc .

; Procedure for determining appeal \_ original system .

\_CallRefFix : \_

; Saving records \_ Before unnecessary call , SafeDisc function is clean.

; The protection routine is made to believe that it was called by a function

; of the game. (for info: CALL [ref] = push (eip+6), jmp <ref>)

add edx , 6 ; Calculated the return address of the processed Call .

pushedx ; \_ The SafeDisc protection routine uses the return address of the

; call calling (in the stack) to determine the call original system .

subedx , 6 ; We restore edx because we don't serves as pointer in the main scan loop.

jmp dword ptr [ eax ] ; We start the execution of the SafeDisc routine pointed which in turn calls the routine

; general conversion which has been patched previously . (She goes so jump on our

; patch to \_cleanReturnPoint\_CallRefFix . We will be able to SO to recover the call original system .

\_ returnPoint\_CallRefFix :

; We now have the call reference original system in ecx for the processed Call .

pop ebx ; We trash from the stack the return value of the processed call which served as fake.

; It remains to modify the call treaty in function of the new table.

call GetRefTable ; Calculation of the call reference system for the new referencing table .

mov dword ptr [edx+2], ebx ; Corrects the call processed with the new reference calculated ( in function of the news

; table ). (Reminder: edx = call processed or even function caller in .text)

; This call is NOW fixed for the new table ( that it still needs to be copied)

jmp\_returnCheckCallRef \_ \_ ; Continue patching.

\_cleanReturnPoint\_CallRefFix:

mov esp,[ebp+0Ch] ; On replace le code que l'on a enlevé

popad ; |

popfd ; |

pop ecx ; Originally : “Ret”. The stack contains the address of the call system ,

; a ret at this moment therefore launches the call system . We replace it with a pop ecx to

; to recover this value .

jmp\_returnPoint\_CallRefFix \_ \_ ; We return to our patch (Everything is clean).

CallRefFix endp

JmpFix proc

; == > \_ 3 < = = Third part Converts jmp & lt ;. \_ stxt ref&gt ; in Call[. rdata ] en function of the new table

; SEO . \_

mov eax , \_cleanReturnPoint\_JmpFix ; Patch the call determination routine ( SafeDisc ) original system

call PatchMagicSafeDiscFct 1 ; and returns it to \_cleanReturnPoint\_JmpFix .

mov edx ,( \_ IB+\_text ); Start of .text section

\_loopJmpFix : \_

cmp byte ptr [ edx ],0E9h ; Is this a “ Jmp ref”?

I \_ CheckJmp ; If yes , we push parsing to \_CheckJmp .

\_ returnCheckJmp : ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx ,( \_IB+\_text+\_ textSize ) ; End of .text section?

jne\_loopJmpFix \_ \_ ;

ret ; End of routine 3, return to caller .

\_CheckJmp : \_

; You must ensure that this jump points to the stxt774 section (stxt774 contains the management of this protection with jmp )

mov eax , dword ptr [edx+1] ; Calculation of the destination address of the processed jump .

add eax , edx ;

add eax, 5 ;

cmp eax,(\_IB+\_stxt774) ; On vérifie si ce saut pointe dans la section stxt774.

jb \_returnCheckJmp ;

cmp eax,(\_IB+\_stxt774+\_stxt774Size) ;

jae \_returnCheckJmp ;

; Here we are on to have a call[ ref . rdata ] modified by SafeDisc in jmp <ref .stxt774="">

jmp eax ; We simulate the launch of this function to get the call original system .

\_ returnPoint\_JmpFix :

pop ebx ; We trash from the stack the reference that the SafeDisc routine had calculated for the

; return from jmp (because the function system ends with a ret when it has finished ).

call GetRefTable ; Calculates the reference to the call system for the new table ( in with a view to transforming the

; jmp in Call.

mov word ptr [ edx ], 15FFh ; We transform the jmp in Call[ref].

mov dword ptr [edx+2], ebx ; And we correct the reference of call[ref] with the new reference calculated for the

; new referencing table .

; ( callback edx = function caller in .text)

; This call is NOW fixed for the new table ( that it still needs to be copied).

jmp\_returnCheckJmp \_ \_ ; Continue patching.

\_ cleanReturnPoint\_JmpFix :

mov esp ,[ ebp+0Ch]; We replace the code that we removed

popad ; |

popfd ; |

pop ecx ; Originally : “Ret”. The stack contains the address of the call system ,

; a ret at this moment therefore launches the call system . We replace it with a pop ecx to

; to recover this value .

jmp\_returnPoint\_JmpFix \_ \_ ; We return to our program

JmpFix endp

CallFix proc

; == > \_ 4 < = = Fourth part Call Rerouting &lt;403389&gt; in call <ref>

mov eax , \_ cleanReturnPoint\_ CallFix ; Patch the other routine ( SafeDisc ) for determining the original call

call PatchMagicSafeDiscFct 2; and returns it to \_cleanReturnPoint\_CallFix .

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopCallFix : \_

cmp byte ptr [ edx ],0E8h ; Is this a “Call ref”?

I\_CheckCall \_ ; If yes , we push parsing to \_CheckCall .

\_ returnCheckCall : ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne\_loopCallFix \_ \_ ;

ret ; End of routine 3, return to caller .

\_CheckCall : \_

mov eax , dword ptr [edx+1] ; Calculation of the destination address of the processed Call .

add eax , edx ;

add eax , 5 ; (Note: 5 is the number of bytes that the call takes same : E8 ?? ?? ?? ??)

; ( same thing for jmps .)

cmp eax , 00403389h ; Is this a jump to reroute ( 403389 = internal call rerouting routine ).

jnz\_returnCheckCall \_ \_ ;

; Check that this reference a trap: call at n+5 ( called stupid call in the tutorial)

call FindCallSkip ; Search for the reference in the ban tables (blacklist in ADumpData2).

cmp ebx , 1 ; If we found this reference , it must be deleted instead of converting it .

i\_nopCall \_

; Restoration of the original call in place of call 403389.

; We go simulate the processed call (Call Fake) to recover the original call .

add edx , 5 ; We put the return address of the call in the stack, to simulate the call point .

pushedx ; \_

subedx , 5 ; We restore edx , because we do n't is used for scanning progress.

jmp eax ; We jump to 403389, with the return value (fake) in the stack.

\_ returnPoint\_CallFix :

; We now have the reference of the routine to call in ecx , we must therefore recalculate

; the call in function of the routine to call and the position of this call.

pop eax ; Trash the fake value from the stack. ( eax no longer used )

sub ecx , edx ; Calculates the value of the jump to be made from the position of the processed call .

sub ecx , 5 ;

mov dword ptr [edx+1], ecx ; And corrects the call processed with the value of the jump to be made to reach the

; original internal routine .

jmp\_returnCheckCall \_ \_ ; Continue patching.

\_ nopCall :

mov dword ptr[edx], 90909090h ; On noppe le call trap (5 nop)

mov byte ptr[edx+4], 90h ;

jmp \_returnCheckCall ; Continue le patch.

\_cleanReturnPoint\_CallFix:

pop ecx ; On replace le code que l'on a enlevé

pop eax ; |

popad ; |

popfd ; |

pop ecx ; Originally : “Ret”. The stack contains the address original ,

; a ret at this moment therefore launches the call . We replace it with a pop ecx to

; to recover this value .

jmp\_returnPoint\_CallFix \_ \_ ; We return to our program

CallFix endp

CopyTable proc

; == > \_ 5 < = = Fifth part Copy of the new call referencing table systems on the old (. rdata )

mov eax , (\_IB+\_ rdata ) ; Start of section. rdata .

mov ebx , ADumpPatchData ; Start of new reference table calls system calculated .

\_loopCopyTable : \_

mov ecx , dword ptr [ ebx ] ; We copy the values from the new table into . rdata .

mov dword ptr [ eax ], ecx ;

add eax , 4 ; Advance one dword in . rdata and in our table.

add ebx , 4

cmp eax , (\_IB+\_rdata+250 h) ; We loop as long as we did not copy the 250 values .

jne\_loopCopyTable \_ \_ ;

ret

CopyTable endp

GetRefTable proc

; == > \_ Annex 1 : GetRefTable < == \_

; Function that searches a value in our referencing table and returns the index relative to . rdata .

; Entrance : ecx : Value searched in the table (A call system ).

; Exit : ebx : Value of the future . rdata referencing this call system .

mov ebx , ADumpPatchData ; Start of the new calls table systems ( stored in in ADump ).

\_ scanMatchRef : ; We look for the correspondence in our table with the call system given in ecx .

cmp dword ptr [ ebx ], ecx ; We compare the call system given to the value in our table at the index ebx .

i\_matchRef \_ ; If it matches, we found our hint .

add ebx , 4 ; Otherwise , we continue browsing the table one dword further.

cmp ebx , (ADumpPatchData+250 h) ; We continue the search until .rdata+250.

jne\_scanMatchRef \_ \_ ;

int 3 ; Not found ! There is a bug... :(

\_ matchRef :

sub ebx , ADumpPatchData ; We calculate the offset value . \_

; ebx contains SO the table index or we found the match .

add ebx , (\_IB+\_ rdata ) ; We add then the base value of . rdata to this hint .

ret ; Return to caller .

GetRefTable endp

TableCallSkip proc

; == > \_ Appendix 2: TableCallSkip & lt ; ==

; Function which builds the table of CALL 403389 to avoid ( called call trap or stupid calls )

mov eax , \_ cleanReturnPoint\_ TableCallSkip ; Patch the other routine ( SafeDisc ) for determining the original call

call PatchMagicSafeDiscFct 2; and returns it to \_cleanReturnPoint\_TableCallSkip .

mov ebx , ADumpPatchData2 ; Memory segment Or we stores the table of calls not to be processed and not to be skipped .

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopTableCallSkip : \_

cmp byte ptr [ edx ],0E8h ; Is this a “Call ref”?

i\_CheckCallSkip \_ ; If yes , we push parsing to \_CallSkip .

\_ returnCheckCallSkip : ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne\_loopTableCallSkip \_ \_ ;

ret ; End of routine 3, return to caller .

\_CheckCallSkip : \_

mov eax , dword ptr [edx+1] ; Calculation of the destination address of the processed Call .

add eax , edx ;

add eax , 5 ; (Note: 5 is the number of bytes that the call takes same : E8 ?? ?? ?? ??)

; ( same thing for jmps .)

cmp eax , 00403389h ; Is this a jump to reroute ( 403389 = internal call rerouting routine ) .

jnz\_returnCheckCallSkip \_ \_ ;

; We go simulate the processed call (Call Fake) to recover the original call .

add edx , 5 ; We put the return address of the call in the stack, to simulate the call point .

pushedx ; \_

subedx , 5 ; We restore edx , because we do n't is used for scanning progress.

jmp eax ; We jump to 403389, with the return value (fake) in the stack.

\_ returnPoint\_TableCallSkip :

; We now have the reference of the routine to call in ecx , we must therefore recalculate

; the call in function of the routine to call and the position of this call.

pop eax ; The fake value placed in the stack goes be reused for future comparison . \_

; Detection of cs:n call cs:n+5 (call trap or stupid call ).

cmp ecx , eax ; Compare function address \_ called at function return address caller

jne\_returnCheckCallSkip \_ \_ ; If they are equal , we have a cs:n call cs:n+5 (call trap)

mov dword ptr [ ebx ], edx ; This is a call to avoid , we add it to our list in ADumpPatchData2

; ( for nopping ulterior ).

add ebx , 4 ; And we advance one dword in our list .

jmp\_returnCheckCallSkip \_ \_ ; Continue patching.

\_cleanReturnPoint\_TableCallSkip : \_

pop ecx ; We replace the code that we removed

pop eax ; |

popad ; |

popfd ; |

pop ecx ; Originally : “Ret”. The stack contains the address original ,

; a ret at this moment therefore launches the call . We replace it with a pop ecx to

; to recover this value .

jmp\_returnPoint\_TableCallSkip \_ \_ ; We return to our program

TableCallSkip endp

FindCallSkip proc

; == > \_ Appendix 3: FindCallSkip & lt ;==

; Function which searches for a reference in the bans table ( blacklist )

; Entrance : edx : Reference to search in the table.

; Exit : ebx : True/False

mov ebx , ADumpPatchData2 ; Memory segment Or we stores the table of calls not to be processed .

\_ loopFindRef : ;

cmp dword ptr [ ebx ], edx ; Comparison of the searched reference with that of the index table ebx .

i\_findRef \_ ;

add ebx , 4 ; Otherwise , we continue browsing the table one dword further.

cmp ebx , (ADumpPatchData2+400 h) ; We continue the search up to 1024 references . ( Readjust if necessary )

jne\_loopFindRef \_ \_ ;

\_ notFindRef :

mov ebx , 0 ; Return 0 in ebx if not found.

ret ;

\_ findRef :

mov ebx , 1 ; Return 1 in ebx if found.

ret ;

FindCallSkip endp

PatchMagicSafeDiscFct1 proc

; == > \_ Appendix 4.A : Patch of a SafeDisc routine : Call calculation systems for Call[. rdata ] and jmp <ref stxt =""> rehoused & lt ;==

; (Avoid manual modifications under SoftIce )

; When this function East called , eax contains the desired return point . But this value East given in relation to the

; patch compiled (its IB) and not compared to the patch in ADump . We must therefore start by readjusting this value .

sub eax , 401000h ; We subtract the IB of the patch.

add eax , ADumpPatchCode ; We add the start address of the patch code stored on ADump .

mov ebx , eax ; We make a copy for a second modification.

sub eax , fct2fix1A ; We calculate the value of the jump to make from fct2fix1A to arrive in eax .

sub eax , 5 ; (fct2fix1A = address where we must exit the SafeDisc routine to recover the

; call value \_ original system and avoid its launch .)

mov byte ptr [ cs:fct 2fix1A], 0E9h ; We write a jmp to fct2fix1A (E9 ?? ?? ?? ??).

mov dword ptr [ cs:fct 2fix1A+01], eax ; And we set the value of the jump ( previously calculated ).

sub ebx , fct2fix1B ; There are two return points for this SafeDisc routine . The second East used

sub ebx , 5 ; when the value of the call system has already been calculated Before .

mov byte ptr [ cs:fct 2fix1B], 0E9h ; (Same, only address changes)

mov dword ptr [ cs:fct 2fix1B+01], ebx ;

ret

PatchMagicSafeDiscFct1 endp

PatchMagicSafeDiscFct2 proc

; == > \_ Appendix 4.B: Patch of a other SafeDisc routine : Call calculation \_ system originals for Call 403389 & lt ;==

; (Avoid manual modifications under SoftIce )

sub eax, 00401000h ; Idem.

add eax, ADumpPatchCode ;

sub eax, fct2fix2 ;

sub eax, 5 ;

mov byte ptr[cs:fct2fix2], 0E9h ;

mov dword ptr[cs:fct2fix2+01], eax ;

ret

PatchMagicSafeDiscFct2 endp

end start

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Find the code of RemoveSafeDiscPatch v1.0 Part A

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯ ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Start the RemoveSafeDiscPatch v1.0 Part B code

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

title RemoveSafeDiscPatch v1.0 Part B (Crack &amp; Tutorial by Peex)

.386

.model small, stdcall

option casemap :none

.code

; This patch compiled is to be loaded into ADump ( or any free memory segment ).

ADump equ 83524000h ; Memory segment start address created by ADump

; ( to be modified in depending on its initialization).

ADumpPatchCode equ (ADump+200h); Start of patch code at 200 (512 bytes for the PE header).

; Addresses of the SafeDisc routine to modify (see tutorial).

fct2fix equ 020BAED7h ; - > \_ Routine for calculating and rewriting the modified code

; Constant dependent on the target program to be set ( Values for the game Mafia: Game.exe).

\_IB equ 00400000h ; ( target program ImageBase )

\_text equ 00001000h ; (section making part of the target program )

\_textSize \_ equ 002399BCh ; "

; EntryPoint of the RemoveSafeDiscPatch .

start:

pushfd ; We save the registers .

pushad ;

call ReconstructP 1; Fixes changes based on 0F 01 C? (SGDT/SIDT).

call ReconstructP 2 ; Fixes modifications based on 0F AA (RSM).

call ReconstructP 3; Fixes modifications based on 0F 0B (UD2).

call ReconstructP 4; Fix CC based modifications (Int 3).

popad ; We restore the registers .

popfd ;

no ; The game is here fixed , the dump of the .text section must be done at this time.

; Once \_ dumped , put the eip to the OEP (of the games) and check that the games work

; correctly .

ReconstructP1 proc

mov eax , \_cleanReturnPoint\_ReconstructP1

call PatchMagicSafeDiscFct ; Patch the call determination routine ( SafeDisc ) original system

; and returns it to \_cleanReturnPoint\_ReconstructP1.

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopReconstructP1:

cmp word ptr [ edx ], 010Fh ; Is this ( possibly ) a “0F 01”: SGDT/SIDT?

i \_checkReconstructP1 ; If yes , we push parsing to \_checkReconstructP1.

\_returnCheckReconstructP1: ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne \_loopReconstructP1 ;

ret ; End of routine, return to caller .

\_checkReconstructP1:

mov eax , dword ptr [edx+2] ; Checking the 3rd byte: Is this a C?

and eax , 0F0h ; We only take care of the first part of the byte ( xxxx 0000).

cmp eax , 0C0h ; And we make the comparison to find out if this 1st part is a C.

I \_continueReconstructP1 ; If yes , we treat This case of modified code .

jmp \_returnCheckReconstructP1 ; If not, it's probably a “ normal ” instruction .

\_continueReconstructP1:

; Arrived here we have a string 0F 01 C?.

jmp edx ; Starts the execution of this instruction, and therefore of the recalculation routines of the

; code ( SafeDisc routines ).

\_returnPoint\_ReconstructP1: ; We will be back here after \_ modified the code ( SafeDisc routine patched ).

jmp \_returnCheckReconstructP1 ; And we continue the treatment .

\_cleanReturnPoint\_ReconstructP1:

lea esp , [esp+0 Ch] ; We recode the deleted instructions with our jump.

jmp \_returnPoint\_ReconstructP1 ; We return to our patch (everything is clean).

ReconstructP1 endp

ReconstructP2 proc

mov eax , \_cleanReturnPoint\_ReconstructP2

call PatchMagicSafeDiscFct ; Patch the call determination routine ( SafeDisc ) original system

; and returns it to \_cleanReturnPoint\_ReconstructP2.

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopReconstructP2:

cmp word ptr [ edx ], 0AA0Fh ; Is it ( possibly ) a “0F AA”: RSM?

I \_continueReconstructP2 ; If so , we handle the case .

\_returnCheckReconstructP2: ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne \_loopReconstructP2 ;

ret ; End of routine, return to caller .

\_continueReconstructP2:

; Arrived here we have a string 0F AA.

jmp edx ; Starts the execution of this instruction, and therefore of the recalculation routines of the

; code ( SafeDisc routines ).

\_returnPoint\_ReconstructP2: ; We will be back here after \_ modified the code ( SafeDisc routine patched ).

jmp \_returnCheckReconstructP2 ; And we continue the treatment .

\_cleanReturnPoint\_ReconstructP2:

lea esp , [esp+0 Ch] ; We recode the instructions deleted with our jump.

jmp \_returnPoint\_ReconstructP2 ; We return to our patch (Everything is clean).

ReconstructP2 endp

ReconstructP3 proc

mov eax , \_cleanReturnPoint\_ReconstructP3

call PatchMagicSafeDiscFct ; Patch the call determination routine ( SafeDisc ) original system

; and returns it to \_cleanReturnPoint\_ReconstructP3.

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopReconstructP3:

cmp word ptr [ edx ], 0B0Fh ; Is this ( possibly ) a “0F 0B”: UD2?

I \_continueReconstructP3 ; If so , we handle the case .

\_returnCheckReconstructP3: ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne \_loopReconstructP3 ;

ret ; End of routine, return to caller .

\_continueReconstructP3:

; Arrived here we have a string 0F 0B.

jmp edx ; Starts the execution of this instruction, and therefore of the recalculation routines of the

; code ( SafeDisc routines ).

\_returnPoint\_ReconstructP3: ; We will be back here after \_ modified the code ( SafeDisc routine patched ).

jmp \_returnCheckReconstructP3 ; And we continue the treatment .

\_cleanReturnPoint\_ReconstructP3:

lea esp , [esp+0 Ch] ; We recode the instructions deleted with our jump.

jmp \_returnPoint\_ReconstructP3 ; We return to our patch (everything is clean).

ReconstructP3 endp

ReconstructP4 proc

mov eax , \_cleanReturnPoint\_ReconstructP4

call PatchMagicSafeDiscFct ; Patch the call determination routine ( SafeDisc ) original system

; and returns it to \_cleanReturnPoint\_ReconstructP4.

mov edx , (\_IB + \_text ) ; Start of .text section.

\_loopReconstructP4:

cmp word ptr [ edx ], 0CCCCh ; Is it ( possibly ) a "CC": INT3 (x2 x3 xN )?

I \_continueReconstructP4 ; If so , we handle the case .

\_returnCheckReconstructP4: ; Return point for subsequent routines .

inc edx ; If not, we continue the scan.

cmp edx , (\_IB+\_text+\_ textSize ) ; End of .text section?

jne \_loopReconstructP4 ;

ret ; End of routine, return to caller .

\_continueReconstructP4:

; Arrived here we have a CC CC chain .

; We proceed to count the number of CCs which follow the start of the chain .

mov ebx , 0 ; We initialize the counter ebx

\_loopCounter : \_

cmp byte ptr [edx+ebx+1], 0CCh ; We look if we still have a "CC" ebx after the start.

jne\_countFinish \_ \_ ; If this is not a CC, counting East finished .

inc ebx ; Otherwise , we increment the counter and continue.

jmp\_loopCounter \_ \_

; ebx now contains the number of CCs after edx .

\_countFinish : \_

cmp byte ptr [edx-1], 0C3h ; Checked if This is not a RET (C3) in edx-1.

i \_skipReconstructP4 ; If this is the case , there is no point in starting the treatment , it is a bogus case .

; Arrived here we have a case valid to modify.

jmp edx ; Starts the execution of this instruction, and therefore of the recalculation routines of the

; code ( SafeDisc routines ).

\_returnPoint\_ReconstructP4: ; We will be back here after \_ modified the code ( SafeDisc routine patched ).

\_skipReconstructP4:

add edx,ebx

jmp \_returnCheckReconstructP4 ; And we continue the treatment .

\_cleanReturnPoint\_ReconstructP4:

lea esp , [esp+0 Ch] ; We recode the instructions deleted with our jump.

jmp \_returnPoint\_ReconstructP4 ; We return to our patch (Everything is clean).

ReconstructP4 endp

PatchMagicSafeDiscFct proc

; ==&gt; Annexe : Patch d'une routine de SafeDisc &lt;==

; (Evite les modifications manuelles sous SoftIce)

sub eax, 401000h ; cf. RemoveSafeDiscPatch v1.0 Part A

add eax, ADumpPatchCode ;

sub eax, fct2fix ;

sub eax, 5 ;

;

mov byte ptr[cs:fct2fix], 0E9h ;

mov dword ptr[cs:fct2fix+01], eax ;

ret

PatchMagicSafeDiscFct endp

end start

;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

; Fin du code du RemoveSafeDiscPatch v1.0 Part B

;¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯