The Twilight of Safedisc

Fully reversing a generic protection

by ArthaXerXès
Version 1.0, 25th January 2000

Copyright ©2000 ArthaXerXès and all the contributors to "*The Twilight of Safedisc*". All rights reserved.

This document is provided "as is" without express or implied warranty, including, but not restricted to, the **merchantability** and the **fitness for a particular purpose**.

ArthaXerXès and the contributors to this document assume **no liability** for **direct** or **indirect** damages resulting from its use, even if the damage results from errors in the document and even if one or several authors have been advised of the possibility of such damage.

As well, illegal uses of this document are **your entire responsibility**. We **do not condone piracy**, this document has been written for **educational purposes**.

You can distribute this document freely in any way, as long as it is kept in its original form. Should you wish to quote this document, do not forget to precise the title, the author and the version.

Front picture : Hans Weiditz, *An Alchemist*

Contents

Chapter 1

Introduction

I see Fate calls: let me on your soft bosom lie,
There I did wish to live, and there I beg to die.

"Venus & Adonis"

1.1 A note from the author

Welcome to this essay reverser !

It is the result of hard and long work, so I really hope you will appreciate it, and most of all, that you will learn something.

You will see, that generic protection is not a good option for software authors, since once it is broken, all the other programs using the protection can be broken too. I shall not discuss my motivations for fully reversing Safedisc and writing this essay. It is neither unethical nor illegal. Of course "bad" uses of this Knowledge exist, but the developers can only blame their sloth and their stupidity. We are reversers, it is our purpose to reverse.

You will notice that the layout is quite unusual. Well, from now on I shall write all my essays using the famous LaTeX2e (Just try to obtain the same result with Word. . .). I really should have used it before.

If you own a printer ideally a postscript printer), the best solution is of course to print the EPS version of this document that you should find on my site. [1] Keep in mind, however, that this is a *two-sided* layout.

Otherwise, if you wish to read this essay in its electronic form, the PDF version is a good choice.

Again, I really hope that you will enjoy this essay. And remember, transmit your Knowledge, for if you do not, our caste will die out.

[1] `http://altern.org/xerxes/` or `http://woodstok.incyberspace.com/arthaxerxes`

1.2 What is in this essay

You are certainly wondering *what* is this essay all about, since there are already well done tutorials about "Safedisc cracking" available.

In this essay, you will not only learn how to crack safedisc with and **without** the original CD, but also how the protection internally works. This knowledge is totally missing in the existing tutorials, and this is the main reason why I am writing this. You will also see that I use a different method to crack the protection.

1.3 The requisites

This is not an essay for beginners, there is no doubt about it. Even an advanced cracker may have some difficulties to understand some parts.

Here is a non exhaustive list of the knowledge that isrequiredd to fully understand this document :

1. A good cracking experience. There is no greater teacher than experience.
2. The *basics* of the PE structure.
3. A solid knowledge of anti-debugging.
4. A basic knowledge of memory management under Windows.
5. A good knowledge of the `x86` assembly language and of the processor itself.
6. Manual ICD files unwrapping. (Read Black Check's or R!SC tutorials)

1.4 Software used

I strongly encourage you to use the software listed, there are equivalents of course, but it will make harder (impossible without Softice or IDA) to follow my steps if you use different tools.

Since they have been chosen for their qualities, you should not be disappointed by their performances. The version used for each program is not necessarily the latest one, it is just the one I have.

Name	Ver	Category	URL
Softice	4.01	debugger	`http://www.compuware.com`
IDA Pro	3.8b	disassembler	`http://www.datarescue.com`
WinHex	8.85	hex editor	`http://www.winhex.com`
Icedump	5	dumper	`http://protools.cjb.net`
Procdump	1.6	dumper & PE editor	`http://procdump.cjb.net`
PEWizzard	1.10	PE editor	`http://protools.cjb.net`
FileMonitor	4.1	system spy	`http://www.sysinternals.com`

Table 1.1: Software used

Not all these programs are free of use. Keep this in mind please, and act in consequence. I shall explain in due time why we need all these programs, for now ensure that you installed them on your computer.

1.5 Some due credits and thanks

I would like to thank the following persons, for the direct or indirect help they provided to me.

Black Check for the papers he wrote, and also for the additional information he was willing to give to provide.

R!SC also for the papers he wrote about the Safedisc protection.

Tola he also wrote some good papers about Safedisc.

Pedro for the hint on the DR7 check. (even if it was a long time ago, and for a complete different matter)

Fravia+ for the marvellous web site he created (which is unfortunately no more), from which I learnt a lot.

The Prestige team for their patience, their lack of arrogance (very rare in the *scene*), and also for being my *method testers*.

Some close friends which I will obviously not name, that helped me a lot in providing me software and also in testing my work.

1.6 Contacting the author

You can only reach me through e-mail. My e-mail is `xerxes@altern.org`. You will find my public PGP key on my web site. If you find someone on IRC claiming to be ArthaXerXès, it is a lie : I do not use IRC at all.

Before you mail me, make sure that the answer to your question is not in this document or other documents I talk about. Stupid questions will be ignored. Do not waste your time with a crack request : **I am not a cracking service**.

Naturally, you should feel free to contact me if you have additional information to provide, or if you found a mistake in this paper. As well, clever questions and remarks are appreciated.

I speak French and English, French being my natural language.

Chapter 2

Copying the CD

What can be avoided
Whose end is purpos'd by the mighty Gods ?

"Julius Caesar"

2.1 Full copy

Contrary to what you may think, it is not impossible to make a working copy of a CD protected by Safedisc. The Safedisc protection relies on unreproducible sectors.

Generally, when you copy a CD, it reads the data of a sector, an regenerates the CRC on-the-fly, this prevents generation loss. The only way to prevent this, is to read the whole sector, including its CRC, and writing it "as is". This is called RAW copy.

2.1.1 RAW Copy

To succeed you need two things :

1. a CD recorder that is able to do DAO (Disc-At-Once) RAW writing.
2. a CD recording software that supports DAO RAW writing.

The software is obviously not a problem. Most of the programs allow RAW writing, the most famous is of course Clone CD, but Discjuggler also works fine for example.

However, if your CD recorder does not support DAO RAW there is little to do. Perhaps that hacking the firmware may give good results (DAO RAW is generally only disabled in the firmware), but this is not an easy hack.

To determine if your hardware supports DAO RAW, you may either check your documentation or look in the *supported functions* menu command that most software have. It should inform you if DAO RAW is supported. (TAO RAW will not work for obvious reasons)

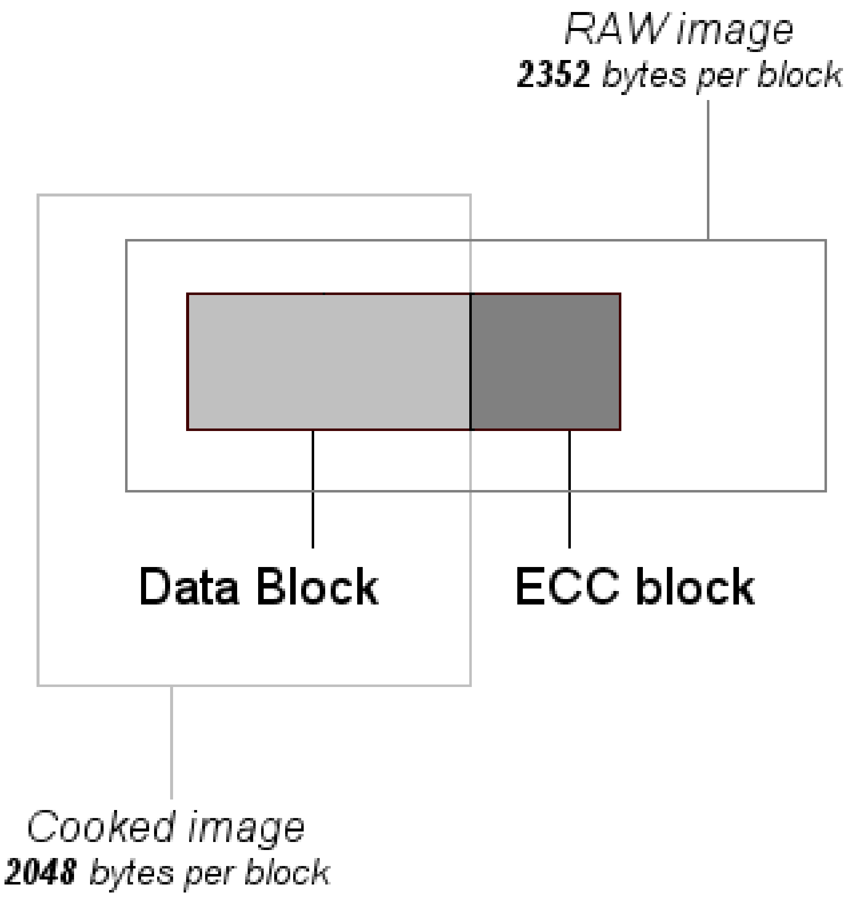

Figure 2.1: MODE 1 Block structure

2.1.2 Recording

The process is very simple. I recommend to follow these steps :

1. Know that reading speed and recording speed may affect the quality of the copy. For optimal results 1:1 copy is advised. However, with good hardware and good software, higher speeds should not affect the quality.
2. Do not try to make an on-the-fly copy ! This will not work.
3. At first, you need to create an image of the Safedisc CD. To obtain a correct image, you must ensure that errors are **ignored**, that RAW mode is **enabled** and that the CD reader with which you are making the image supports RAW reading. The whole reading process is a bit lengthy, so be patient.
4. Once the image is done, the only thing left to do is to launch recording. Again, check that RAW mode is **enabled** and that errors will be written **uncorrected** (i.e. ignored).

2.1.3 The limits

You certainly wonder why I fully reversed safedisc if copying it is so easy. The explained method has got some limits that you certainly noticed.

1. Few CD recorders support DAO RAW.
2. DAO RAW copy is unreliable, generation loss is very important.
3. Safedisc protection is **very slow**, the check performed before the game starts is time consuming. Over more, the CD is required to play the game, which is annoying.

I think it is important that software developers understand that buying a protection is a waste of money. In my opinion, a well done manual is a much better protection. It costs too much to copy a manual, and it is far to replace the original (whereas there is no difference between a copied and an original CD).

Chapter 3

Removing the safedisc protection

Wayward sisters, you that fright
The lonely traveller by night,
Who, like dismal ravens crying,
Beat the windows of the dying,
Appear ! Appear at my call, and share in the fame
Of a mischief shall make all Carthage flame.
Appear !

"Dido & Aeneas"

3.1 Working with the original CD

Our example is the game Rayman 2. It is protected with a recent version of Safedisc, I think it is r3. (but a brand new one just has been released apparently...)

3.1.1 Camouflaging Softice

Safedisc use several ways to detect a debugger, therefore you have to camouflage Softice.

- Change the int 68h return value to 4300h.
- Change all drivers' name.
- The debug register check will be removed later.

There are a lot of documents that explain how to do it, it is very easy (there are even programs doing it for you). I do not advise to use Frogsice, it is an excellent detection tool, but it is better to modify Softice directly.

3.1.2 Retrieving the deciphered data

There is a lot of data to dump before we actually start cracking. As you already know, the `ICD` contains three parts that are ciphered :

1. the `.text` section
2. the `.data` section
3. the `.rdata` section (we shall deal with this section later)

Furthermore, the `.data` section is modified during the execution of the `ICD` after it has been unwrapped, this is why we can dump the `.data` section only during a limited period : after it has been decoded and before the execution of the program starts.

The best way is probably to put a breakpoint on FreeLibrary. The lengths of all the sections and their respective start addresses can be determined with the help of Procdump. Simply open the file `rayman2.icd`, and choose `''PE Editor''`. You also need to write down the entry point of the executable of course.

With my method, we also need to hack `dplayerx.dll` and `rayman2.exe`. Both files have ciphered `.txt` sections, we have to dump them too.

Time for dumping !

1. Make sure Softice is loaded, hit `CTRL-D` and type the commands : `bc * ; bpx FreeLibrary ; bd * ; g` (`;` stands for `ENTER`).
2. Run `rayman2.exe` with either the original or a DAO RAW copy of the Rayman 2 CD.
3. Wait for the logo to disappear, hit `CTRL-D` and enable the breakpoint (`be *`).
4. You should find yourself in Softice few seconds after the breakpoint has been activated. Hit `F11` (or type `g @ss:esp`) to return to the caller (which should be within `dplayerx`).
5. Disassemble the entry point of `rayman2.icd`, in typing `u 45fcc0`. You should see this code :

   ```
   push    ebp
   mov     ebp, esp
   push    ffffffffh
   push    0049c9d0h
   push    0045ff90h
   mov     eax, dword ptr FS:[00000000]
   push    eax
   ...
   ```

   If you do not, this means the `.text` section is not deciphered yet. Hit `g` and wait for the debugger to break again.

6. Dump all the sections :
   (a) `rayman2.exe.txt` (`pagein d 401000 c800 r2_txt.dmp`)
   (b) `dplayerx.txt` (`pagein d 8ee000 9e31 dx_txt.dmp`)
   (c) `rayman2.icd.data` (`pagein d 9f000 7c00 r2_data.dmp`)
   (d) `rayman2.icd.text` (`pagein d 401000 9a200 r2_text.dmp`)

   You can also see the size of the sections with the help of the command `map32`. Do not forget to switch pages when dumping `rayman2.exe.txt`, with the help of the command `addr`. Switch back when it is done.

7. You now have **4** files : `r2_txt.dmp`, `dx_txt.dmp`, `r2_data.dmp` and `r2_text.dmp`.

3.1.3 Rebuilding the files

There is a last thing to do before we crack the protection, you have of course to rebuild `rayman2.exe`, `dplayerx.dll` and `rayman2.icd` with the dumped sections. You can use either Procdump or PEWizzard. In my example I used PEWizzard.

Here is how to rebuild `rayman2.icd`, the method is identical for `rayman2.exe` and `dplayerx.dll`, except that the sections' name change of course.

1. Create a directory called `icd_rebuild`. Copy `r2_data.dmp`, `r2_text.dmp` and `rayman2.icd` in this directory. If PEWizzard is not in your path, also copy it in this directory.
2. Open a DOS prompt in this directory. Split the `ICD` file (`pew -split rayman2.icd`).
3. Rename `section0.bin` to `r2_text.cod`, copy `r2_text.dmp` to `section0.bin`. Both files must have the **exact same** size in bytes.
4. Rename `section2.bin` to `r2_data.cod`, copy `r2_data.dmp` to `section2.bin`. Both files must have the **exact same** size in bytes.
5. Rename `rayman2.icd.pe` to `icd.pe`. Rename `rayman2.icd` to `uncracked.icd`.
6. Rebuild the `ICD` file (pew -join icd.pe rayman2.icd). Both `ICD` files should have roughly the same size. (it is possible the rebuilt executable is a bit smaller)

Once you have done this with all files, you can copy them to the main installation directory or Rayman 2. It is of course advised to back up the original files in another directory before doing so.

3.1.4 Unprotecting `rayman2.exe`

The `.txt` issue

> *You probably noticed, that the program crashes if you put breakpoint in it. This is because it performs checksums on its sections, and it uses the value of these checksum to decode the `.txt` one. Once the program will be deprotected, you will see that you can put breakpoint wherever you wish.*

If you try to run the hacked `rayman2.exe` now it will crash, since it tries to decode sections that are already decoded. It is time to disassemble `rayman2.exe` and `dplayerx.dll` with IDA. Choose 686 for the processor. The whole disassembling process takes few minutes, depending on the speed of your computer.

Once done, enable faults (`faults on`) in Softice, and run the executable. You should be within `rayman2`, this is what we want. Put a breakpoint on the first memory address of the `.txt` section (`bpm 401000 R`). Run the program again. You will be in this copy memory routine :

```
...
mov     ecx, [esp+8+arg_8]  ; how many to copy
mov     esi, [eax+4]
mov     edi, [esp+8+arg_4]
mov     eax, ecx
add     esi, edi   ; copy from
mov     edi, [esp+8+arg_C]  ; copy to
shr     ecx, 2  ; how many to copy div 4
repe    movsd ; <- this is probably where you will break
mov     ecx, eax
mov     ax, 1
and     ecx, 3  ; how many to copy mod 4
repe    movsb
pop     edi
pop     esi
retn
```

It is now wise to put a breakpoint on edi-4 (`bpm edi-4`). You may now type `g`. You will break again, but in a different place, which looks like a checksum routine.

```
...
mov     eax, [ebp+8]
mov     ecx, [eax]
mov     [ebp+var_8], ecx ; <- you will certainly break here
mov     edx, [ebp+8]
mov     eax, [edx+4]
mov     [ebp+var_C], eax
mov     ecx, ds:dword_42F010
mov     [ebp+var_10], ecx
...
```

You probably noticed that there are a lot of useless jumps. All these jumps are here to make tracing and disassembling more difficult, they are totally useless.

> *This is where IDA comes useful, remember, IDA stands for Advanced Interactive Disassembler. Interactive : this is the important keyword. I really hope that you know how to use IDA, because I am not going to teach you this : it would take too much time. Read your IDA manuals or some tutorials on the matter if you feel lost.*
>
> *I also strongly advise you to rename the functions, labels and variables to explicit names, it makes reversing much easier.*

Of course, the checksum routine in which we ended is located in the `.txt2` section. Since the entry point of the executable is in the `.text` section, we can assume that the function that does the whole decoding of the `.txt` section is called in the `.text` section.

We have two alternative : we can trace from the entry point and monitor if changes are performed on the `.txt` section, or we can use cross reference to find what is the main function. The later method is the one I used, but you can use the other one if you prefer, however, in my opinion it is less reliable in this case.

In IDA go at the top of the "`checksum`" function in which you ended. Rename it "`checksum_0`". Go to the caller, and rename it "`checksum_1`", and so on until you find yourself into a function which is called from the `.text` section.

> **note** : it is highly probable that some functions are not detected by IDA. If that is the case, you have to scroll up until you find this sequence :
>
> ```
> push    ebp
> mov     ebp, esp
> push    ...
> ```
>
> This marks the beginning of the function. Rebuild it from here (hit `P`) or use the menu.
> Also, IDA can be confused by the following code :
>
> ```
> ...
> jmp     short near ptr loc_whatever+1
>
> ...
>
> loc_whatever:
>  wrong disassembled code
> ...
> ```
>
> To overcome this difficulty, you have to undefine the code at location `loc_whatever`, and redefine it as code starting at location `loc_whatever+1`.

The whole process may take some time, and you will probably want to test within softice if you found the right caller. Eventually, you should find the "big" call at location 00410a94, the function is 004270ad. You probably noticed that the program uses a checksum performed on the `.txt2` section to decode the `.txt` section. Removing the call to the function will therefore solve two problems.

```
...
loc_410A94:  ; CODE XREF: .text:00410A90
 call    checksum_6
 add     esp, 4
...
```

Removing this call will solve our little problem. Use WinHex, search for the string `e8 14 66` (you will see that it is unique) and replace it with `eb 03`. We did most of the hard job concerning `rayman2.exe`, but there are still one or two things to do.

Local files check

If you try to run now the executable, you will see (with FileMonitor for example) that it loads `dplayerx.dll` and `rayman2.icd` from the CD. What it does is checking the files, and if they were modified, it loads them from the CD instead.

This protection is easy to find and to remove. To find it, put a breakpoint on `GetFileAttributesA`, and wait until it is called with `dplayerx.dll` or `rayman2.icd` as parameters. `GetFileAttributesA` is called by the shell and by `clokspl.exe` before it is called by `rayman2.exe`.

You should break in this code :

```
...
mov     esi, [ebp+arg_4]
mov     edi, DS:GetFileAttributesA
push    esi ; <- esi points to the name of the file to check
call    edi ; checking the file ...
cmp     eax, 0FFFFFFFFh
...
```

If you analyse the function carefully, you will see it is easy to crack. This code does the check :

```
...
loc_4052E0: ; CODE XREF: check_file+96
cmp     word ptr [ecx], 0
jnz     loc_40541F ; here we jump if the local file is ok
...
```

Therefore, we merely have to make the jump unconditional. You will see that we shall jump to a location in which eax is set to 1 (true). The string to search is `0f 85 35 01`, the string to put is `e9 36 01 00 00`. Now the locals file are loaded.

More checksums

There are still checksums performed, and it is a good idea to remove them, isn't it ? As usual, if you put a breakpoint on 401000, you will find them.

The easier to remove is the one that is called from "clean code" (i.e. without 10^{31337} jumps around). Just after the call, eax is tested. Unless you are completly stupid, you probably guessed that this can be easily removed in modifying the called function.

```
...
loc_4028FD: ; CODE XREF: CD_illa_check_2+AC
 push    2
 call    checksum_2_5 ; the nasty checksum
 add     esp, 4
 test    ax, ax ; eax == 0 -> no soup for you !
 jnz     short loc_40294F
...
```

To remove the function, search for `55 8B EC 83 EC 0C 53 56 57 EB 01 EE` (shortest unique string) and replace with `33 c0 40 c3` or `b8 01 00 00 00 c3`, as you prefer. Now the function will always return 1.

The other checksums are performed at location 4115d6 :

```
...
the_checks:  ; CODE XREF: .text:004115C9
 mov     eax, [ebp-18h]

loc_4115D9:  ; CODE XREF: .text:004115B9
 push    offset a_txt_1  ; pushing txt1 section
 mov     ecx, [ebp-14h]
 sub     esp, 0Ch
 mov     edx, esp
 mov     [edx], eax
 mov     eax, [ebp-10h]
 mov     [edx+4], ecx
 mov     [edx+8], eax
 call    near ptr nasty_checksum  ; here we check section .txt
 add     esp, 10h
 mov     edx, [ebp-18h]
 mov     eax, [ebp-14h]
 push    offset a_txt2_1  ; pushing txt2 section
 sub     esp, 0Ch

loc_411607:  ; CODE XREF: .text:0041163B
 mov     ecx, esp
 mov     [ecx], edx
 mov     edx, [ebp-10h]
 mov     [ecx+4], eax
 mov     [ecx+8], edx
 call    near ptr nasty_checksum  ; here we check section .txt2

loc_411619:  ; CODE XREF: .text:004115CE
 add     esp, 10h
 xor     esi, esi
...
```

Here my choice was to “jump over” the two calls. Search for `02 EB 0B EB FC D7 E1 4A 76 BA 98 05 A5 0D 8B` and replace `02` with `53`. The checksums are now annihilated !

The mundane CD check

There is a first CD protection layer, that allows Safedisc to quickly detects if the wrong CD or no CD at all is inserted. If you put a breakpoint on `GetDriveTypeA` you will find it at once. It is better to remove this protection if you want to make a “no CD crack”. If you do not remove it, you will need a perfect copy of the CD, and you will not be allowed to have the crack on the CD.

The function is 4124f0, here is the code :

```
first_CD_check proc near ; CODE XREF: .text:0040F6AF

var_104= byte ptr -104h
arg_0= dword ptr  4

 sub     esp, 104h
 lea     eax, [esp+104h+var_104]
 push    104h
 push    eax
 push    0
 call    ds:GetModuleFileNameA
 lea     ecx, [esp+104h+var_104]
 push    ecx
 call    ds:GetDriveTypeA
 cmp     eax, 5
 jnz     short CD_drive_found
 mov     edx, [esp+104h+arg_0]
 lea     eax, [esp+104h+var_104]
 push    edx
 push    eax
 call    sub_412780
 add     esp, 8
 add     esp, 104h
 retn

CD_drive_found:  ; CODE XREF: first_CD_check+26
 mov     ecx, [esp+104h+arg_0]
 lea     edx, [esp+104h+var_104]
 push    ecx
 push    edx
 call    CD_sub_check ; this function returns 1
 add     esp, 8 ; when everything is fine
 add     esp, 104h
 retn
first_CD_check endp
```

This protection is mundane. To remove it, you just need to replace the string `81 EC 04 01 00 00 8D 44` with `33 c0 40 c3` or `b8 01 00 00 00 c3`.

The real CD check

Now, we have to remove the **real** CD check. Locating it is very easy, since it is so slow you just have to trace until you have some long CD operation performed. If you narrow your search, you will quickly find the routine, which is 4064d0. This function is called twice, from the same function (4055b0). Here is the code corresponding to the calls :

```
...
 push    esi ; see, the argument ?
 push    offset byte_436100 ; and this one ?
 mov     ecx, [ebp+arg_14]
 sub     esp, 0Ch
 mov     edx, esp
 mov     [edx], eax
 mov     eax, [ebp+arg_18]
 mov     [edx+4], ecx
 mov     [edx+8], eax
 call    near ptr safedisc_CD_check ;<-- called here
 add     esp, 14h
 mov     ebx, eax ; the result of the function is in eax
 mov     [ebp+var_C], edi
 jmp     loc_40581A


loc_4056F3: ; CODE XREF: big_CD_check+E8 big_CD_check+FC ...
 mov     edx, [ebp+arg_10]
 push    edi ; here we have a different argument
 push    offset byte_436100 ; same argument than above
 mov     eax, [ebp+arg_14]
 sub     esp, 0Ch
 mov     ecx, esp
 mov     [ecx], edx
 mov     edx, [ebp+arg_18]
 mov     [ecx+4], eax
 mov     [ecx+8], edx
 call    near ptr safedisc_CD_check ;<-- and here
 add     esp, 14h
 mov     ebx, eax ; again, the result of the function is in eax
 mov     [ebp+var_C], edi
 jmp     loc_40581A
...
```

Use the technique we used many times : make the function always return 1. For this, you have to search for the string `55 8b ec 53 56 57 bb` and replace with `33 c0 40 c3` or `b8 01 00 00 00 c3`. There is no more CD protection in `rayman2.exe`, congratulations reverser !

> **note** : you will notice that the LED of your CD drive lights, this is because there is still some CD check performed around 405628 (the result does not matter since the protection is cracked). It is not required to modify anything for the protection to be fully removed, **but** you can remove the access if you wish. It is easier that what we did, is not it ?

The debug register check

As I told you, the protection checks for the content of the debug register DR2 in our example. It is not necessary to remove this check since I think there is a bug in it. `:-)`

However, I perfectly remember that other Safedisc versions used a similar protection, but instead of checking DR2, DR7 was checked. The DR7 check is very disturbing since you cannot use any breakpoint. Future protection may use the DR7 check again, therefore I think it is relevant to explain how to remove it.

> *The eight debug registers control the debug operation of the processor. These registers can be written to and read using the move to or from debug register form of the* `mov` *instruction. A debug register may be the source or destination operand for one of these instructions. The debug registers are privileged resources; a* `mov` *instruction that accesses these registers can only be executed in real-address mode, in smm, or in protected mode at a* `cpl` *of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).*
>
> Intel Architecture Software Developers's Manual

Since the program runs in ring 3, and there is no VXD, the programmers used a very clever (yes, sometimes protection makers are clever !) trick to switch to ring 0 : they change the division by 0 handler and perform a division by 0. When they are within the handler, they are running in ring 0, enabling them to check the debug registers...

We now have to locate the debug register access, and this is very easy ! As explained in the later quotation, the debug registers can only be acceded with the help of a `mov` instruction. And a `mov r32, dr0-dr7` instruction is always assembled as `OF 21 xx`. Look at the tables to determine the value of xx.

Opcode structure	7	6	5	4	3	2	1	0
	1	1	y	y	y	z	z	z

debug registers

register	y	y	y
DR0	0	0	0
DR1	0	0	1
DR2	0	1	0
DR3	0	1	1
DR4	1	0	0
DR5	1	0	1
DR6	1	1	0
DR7	1	1	1

general registers

register	z	z	z
eax	0	0	0
ecx	0	0	1
edx	0	1	0
ebx	0	1	1
esp	1	0	0
ebp	1	0	1
esi	1	1	0
edi	1	1	1

Table 3.1: Third opcode format for mov to/from debug registers

In our example, `mov eax, DR2` is assembled `0f 21 d0` ($d0 = 11010000$). If you search for this short string in `rayman2.exe`, you will find it only once. Replace `0f` with `CC` and redirect the int 3 to Softice (`i3here on` or `bpint 3`). This way you will break in the DR2 check routine.

```
...
pusha
mov     [ebp+var_1C], cs
sidt    [ebp+var_14]
mov     eax, dword ptr [ebp+var_14+2]
mov     esi, [eax+4]
mov     si, [eax]
mov     edi, [esi]
mov     [ebp+var_4], edi
mov     edi, [esi+4]
mov     [ebp+var_8], edi
mov     dword ptr [esi+1], 0CF530E58h
mov     byte ptr [esi], 58h
lea     ebx, div0_handler ; see, we change the handler...
xor     eax, eax
div     eax ; division by zero...

div0_handler: ; DATA XREF: dr2_check+1BA
mov     eax, [ebp+read_write_dr2] ; we are in ring 0...
cmp     eax, 1
jz      short write_to_dr2 ; shall we read or write to dr2 ?
mov     eax, DR2    ; we read from dr2
mov     [ebp+dr2_content], eax
jmp     short div0_quit

write_to_dr2: ; CODE XREF: dr2_check+1CA
mov     eax, [ebp+dr2_content]
mov     DR2, eax ; we write to dr2

div0_quit: ; CODE XREF: dr2_check+1D2
sidt    [ebp+var_14]
mov     eax, dword ptr [ebp+var_14+2]
mov     esi, [eax+4]
mov     si, [eax]
mov     dword ptr [esi+1], 0CF535158h
mov     byte ptr [esi], 58h
xor     ecx, ecx
mov     cx, [ebp+var_1C]
lea     ebx, div0_exitpoint ; establishing exit point...
xor     eax, eax
div     eax

div0_exitpoint: ; DATA XREF: dr2_check+1F7
mov     edi, [ebp+var_4] ; we are back to ring 3 again.
mov     [esi], edi
mov     edi, [ebp+var_8]
mov     [esi+4], edi
popa
cmp     [ebp+read_write_dr2], 0
jnz     short dr2_I_wrote ; if we wrote to dr2, then we quit
mov     eax, [ebp+p_dr2_content]
mov     ecx, [ebp+dr2_content]
mov     [eax], ecx ; here we save the value of dr2
...
```

Of course, DR2 should contain 0, and it does, even when using it within Softice (type `cpu` to know the value of the registers). This is why I consider this protection has got a bug ! Unless I misunderstood something... If you encounter a DR7 check instead of a DR2 check, the expected value for DR7 is 400, not 0.

> **Important** : if ever you encounter a DR7 check, this is the **first** protection you should remove.

There is no need to remove the DR2 check in our case since the function that calls it has been either modified or not called at all thanks to our previous modifications. However, to do it you would have to search for the string `8b 4d e8 89` and replace it with `c7 00 00 00 00 00 33 c0 40`.

Now `rayman2.exe` is fully deprotected, and I think it is time for us to take a break. What about a game of Unreal Tournament ? Very relaxing. Perhaps you would prefer to eat something or to have a drink ? Your brain probably needs glucose now (especially if English is not your natural language), therefore a fruit juice or some biscuits may be a good idea. I do not recommend alcoholic beverage, for contrary to what you feel, your capacities are decreased. You can also stop for now, and resume reading another day.

Of course, if you master the very ancient art of meditation, the moment is ideal.

If you prefer, you can continue to read this document, but I really advise you to take a break so as to take fully advantage of what you just learnt.

3.1.5 Deprotecting `dplayerx.dll`

This is the last thing we have to do, once performed, Safedisc protection will be no more. `dplayerx.dll` takes care of deciphering the `ICD` file with the help of the key extracted from the CD (actually bad sectors). It also interfaces the modified `.rdata` with the system. Indeed, instead of directly accessing the system function, the program goes through `dplayerx.dll` which redirects it to the right location.

This is why, in other tutorials you had to rebuild the `.rdata` section manually, which was sometimes producing unreliable cracks. The whole point of my method is here, we modify `dplayerx.dll`, so that it **still** interfaces the system functions, even if there is **no** CD at all in the drive.

However, you remember that we rebuilt the `ICD` file. Why ? Simply because decoding the `ICD` is slow. We shall therefore modify `dplayerx.dll` so that it does not deciphers the `ICD` anymore.

The .txt issue

You can proceed the same way that we did for rayman2.exe, since this is the same type of routines, except that they are located in dplayerx.dll.txt2 this time. All that you have to do, is to follow the same procedure and you will eventually find this call :

```
...
mov     [eax+4], edx
mov     ecx, [ebp+arg_8]
mov     [eax+8], ecx
call    near ptr a_w_checksum_3 ; here we decode...
add     esp, 28h
jmp     short loc_8F9564
...
```

Removing this is mundane (but finding it was not, was it ? ;-)). Search for e8 90 00 00 and replace with eb 03.

Debug registers

There is the exact same routine that checks the content of DR2 in dplayerx.dll. As I said, do not waste your time with it : it is inefficient. However, should you face a DR7 check, you must act as explained previously in that case.

Changing the behaviour of dplayerx.dll

To speed up loading, we built an ICD file with decoded .data and .text sections. Of course, dplayerx.dll will try to decipher them anyway, therefore when executing rayman2.icd, you will not have the clear code anymore.

What you have to do, is to remove the call to the function that takes care of the whole decoding process. An easy task, for to detect it you just need to trace from the entry point of the dll, and monitor the content of rayman2.icd.text. You will quickly find this piece of code to be interesting :

```
...
call    ds:GetCurrentProcess
push    offset dword_905030
mov     dword_909740, eax
call    sub_8F8BBA
add     esp, 4
mov     dword_909744, eax
push    1
push    offset dword_909740
call    decipher ; this function does the whole thing
...
```

Step into decipher, you will finally reach this piece of code :

```
...
 mov    eax, esp
 mov    [eax], ecx
 mov    ecx, dword_908980
 mov    [eax+4], edx
 mov    [eax+8], ecx

loc_8F40B0: ; CODE XREF: whole_ICD_stuff+20EC
 call   section_decipher ; here we decipher sections...
 add    esp, 0Ch
 jmp    short loc_8F40C1
...
```

You will see later, when we shall study how to crack without the CD, that the heart of Safedisc lies deep in this function. For now, however, all we need to do is to remove the call. What we shall do is jump over the code, but where to ? If you jump just after it will crash since we are within a loop.

But do we know where the whole `.rdata` process takes place ? No we do not. Run `rayman2.exe` with faults enabled (`faults on`), you will quickly find yourself in `dplayerx.dll`, near this code :

```
 ...
loc_8F1AF9: ; CODE XREF: create_api_interface+738
 mov    ecx, [ebp+key] ; store the key in ecx
 mov    edx, [ebp+rdata_value] ; edx contains a value
; from .rdata
 xor    edx, [ecx]

loc_8F1B01: ; CODE XREF: create_api_interface+7CB
 add    edx, [ebp+base_call]
 mov    [ebp+rdata_value], edx ; we also rebuild the
; rdata in the temp memory
 mov    eax, [ebp+rdata_value]
 xor    ecx, ecx
 mov    cx, [eax] ; here you have a general protection fault
 and    ecx, 0FFh
 push   ecx
 mov    edx, [ebp+rdata_value]

loc_8F1B19: ; CODE XREF: create_api_interface+7BC
 add    edx, 2
 push   edx
 mov    eax, [ebp+var_8]
 push   eax
 call   alloc_api_interface
 ...
```

The problem is that ecx should contain the key extracted from the original CD (actually "computed" is a more accurate term), since we hacked the protection, the memory contains garbage instead of the correct key, therefore the program access an invalid memory emplacement.

How to guess the correct key value ? Well, this is very simple, you do not guess it, you *steal it*. Note that the key is stored in 909473. If you put a breakpoint on this value and run the program again (`bpm 909473 W`), you will eventually break here :

```
store_key proc near ; CODE XREF: create_api_interface+4FD

arg_0 = dword ptr  8
arg_4 = word ptr  0Ch

 push   ebp
 mov    ebp, esp
 mov    eax, [ebp+arg_0]
 mov    ecx, [eax]
 mov    key_backup, ecx ; here we copy to 909473
 mov    dx, [ebp+arg_4]
 mov    word_909477, dx
 pop    ebp
 retn

store_key endp
```

This is perfect. All you have to do is to disable all breakpoints but the one on 909473, move the hacked files (`rayman2.exe`, `dplayerx.dll` and `rayman2.icd`) elsewhere and restore the original files. Insert the original CD, wait for the program to break and smile, for the correct value of the key should be in ecx. In our case, the value is `c15cf2e5`.

Change the code so that `c15cf2e5` is always stored in 909473, will bypass the check. To do so, simply search for `8B 45 08 8B 08 89 0D` and replace with `b9 e5 f2 5c c1`. Now the program does not crash anymore within `dplayerx.dll`, but within `rayman2.icd`.

We know why : this is because we have not removed the call to the function that decodes the `.text` and `.data` sections. We also know where this function is called, at the location 8f40b0. All we have to do is to jump just before the call to the function that maps the API. Within IDA, if you scroll down from 8f40b0, you will find this piece of code to be attractive :

```
...
loc_8F48E3: ; CODE XREF: whole_ICD_stuff+28B4
 mov    ecx, [ebp+8]
 push   ecx
 push   offset key_1
 call   rdata_chks ; we call this function once the
; ICD is deciphered.
; It will take care of the whole .rdata remapping stuff...
 add    esp, 8
...
```

Well, do I really have to tell you what to do ? Instead of calling the function, we shall directly jump to 8f48e3. To do this, search for `E8 EB 0D 00` and replace with `83 c4 0c e9 2b 08 00 00`.

Ready for the test ? Remove the original CD from the drive, ensure that the hacked version of `rayman2.exe`, `dplayerx.dll` and `rayman2.icd` are in the main directory, and run `rayman2.exe`...

...congratulations, it works. You just cracked Safedisc.

Of course, the program still asks you for the CD, this is not the Safedisc protection which is in cause but the game itself. If you insert a copy of the CD, it will work fine, for the checks are mundane.

Removing the non-safedisc protection of Rayman 2 is trivial matter, and I shall not discuss it here.

3.1.6 Generalization

We annihilated Safedisc, did not we ? What is best, is that we do not have to take care of `.rdata`, since it will be rebuilt **as if the original CD were present** !

If you try this method on a different Safedisc game, you will notice that **dplayerx.dll** and the main executable have got the **same size** (if not, this certainly means it is a different version of safedisc). You will also notice that the protection functions are located at the **same position**. Cracking should not take more than 10 minutes.

If you encounter a different version of Safedisc, what you learnt here should prove useful, since chances are the protection is much alike. If you follow this method, you should be able to defeat any future Safedisc releases.

3.2 Working without the original CD

You work will not be very different, except that

- you cannot dump the `.text` and `.data` section of `rayman2.icd`
- you cannot *steal* the key.

3.2.1 Few words about the protection

What you know, is that it inspects the CD and searches for the bad sectors. Afterward, with the help of the sectors found, it computes a key, which will be used to decode the sections of `rayman2.icd`. We also noticed that the `.rdata` section is still dependant of `dplayerx.dll` after the operation, this is to make cracking harder.

We need to reverse the ciphering algorithm, and we also need to create a brute force cracker. Afterward, we shall extract the decoded section to obtain the same `ICD` than before (you will see that it is a slow process, therefore it is better to have the sections deciphered)

3.2.2 Working with the main executable

You may follow the same steps to build a working a `rayman2.exe`. Dump the `.txt` section, rebuild the exe and remove the protections. If you already have a hacked `rayman2.exe`, there is nothing to do.

3.2.3 Reversing the algorithm

Do you remember this piece of code ?

```
...
mov     eax, esp
mov     [eax], ecx
mov     ecx, dword_908980
mov     [eax+4], edx
mov     [eax+8], ecx

loc_8F40B0: ; CODE XREF: whole_ICD_stuff+20EC
call    section_decipher ; here we decipher sections...
add     esp, 0Ch
jmp     short loc_8F40C1
...
```

As I told you, the heart of Safedisc lies deep within. After a lot of tracing and code analysis, you will discover that the sections are decode per 4096bytes-sized blocks, and that each block is decoded per 8bytes-sized blocks. Here is the decryption algorithm :

```
decode_8bytes proc near
; CODE XREF: decoding_it_real+27 decoding_it_real+8B ...

to_decode= dword ptr  8
key= dword ptr  0Ch

push    ebp ; this routines decodes 8 bytes
mov     ebp, esp
mov     eax, DS:rdata_0 ; the first 4 bytes of dplayerx.dll.rdata
push    ebx
push    esi

loc_8F800A:
mov     esi, [ebp+to_decode]
push    edi
mov     edi, DS:rdata_4 ; the 4-8th bytes of dplayerx.dll.rdata
mov     edx, [esi]
mov     ecx, [esi+4]
shl     eax, 5

[garbage]

mov     ebx, edi
dec     edi
test    ebx, ebx
jbe     end_routine
mov     esi, [ebp+key]
inc     edi
mov     [ebp+key], edi

start_loop: ; CODE XREF: decode_8bytes+C0
mov     ebx, [esi+8] ; we loop 20 times
mov     edi, edx
```

```
 shl     edi, 4
 add     edi, ebx
 mov     ebx, edx
 shr     ebx, 5
 add     ebx, [esi+0Ch]
 xor     edi, ebx
 lea     ebx, [eax+edx]
 xor     edi, ebx
 sub     ecx, edi

[garbage]

 mov     ebx, [esi+4]
 mov     edi, ecx
 shr     edi, 5
 add     edi, ebx
 lea     ebx, [eax+ecx]
 xor     edi, ebx
 mov     ebx, ecx
 shl     ebx, 4
 add     ebx, [esi]
 xor     edi, ebx
 sub     edx, edi

[garbage]

 sub     eax, DS:rdata_0

[garbage]

 mov     edi, [ebp+key]
 dec     edi
 mov     [ebp+key], edi
 jnz     short start_loop
 mov     esi, [ebp+to_decode]

end_routine:  ; CODE XREF: decode_8bytes+35
 mov     [esi], edx  ; we save edx

[garbage]

 mov     [esi+4], ecx  ; we save ecx

[garbage]

 pop     edi
 pop     esi
 pop     ebx
 pop     ebp
 retn
decode_8bytes endp
```

The algorithm is not unknown anymore. We have ciphered data, we know the algorithm, but we ignore the key and do not have clear data. To brute force the algorithm we need to guess clear data, and we can do it.

At the end of the `.data` section you should see this pattern `CD DE 58 A2 75 01 7F` to be repeated a lot of time. It does not require a lot of PE knowledge to know that `.data` sections end with 0, for alignment purposes. Therefore, we know that `CD DE 58 A2 75 01 7F` corresponds to `00 00 00 00 00 00 00 00`.

We have the algorithm, clear text and coded text. We can easily determine the key now. But there is still a problem, trying from 00000000h to FFFFFFFFh will take some time. Fortunately we have a way to narrow our search. You certainly remember that the `.rdata` section is handled in a different manner, and that the algorithm is very different. Here it is :

```
 ...
loc_8F1AF9: ; CODE XREF: create_api_interface+738
 mov     ecx, [ebp+key] ; store the key in ecx
 mov     edx, [ebp+rdata_value] ; (1) edx contains a value
; from .rdata
 xor     edx, [ecx]

loc_8F1B01: ; CODE XREF: create_api_interface+7CB
 add     edx, [ebp+base_call] ; edx now contain the correct value
 ...
```

If you are careful, you will see that at the point (1), edx always contains something like `c1 xx xx xx`. Since this result is xored with the value of the key, and since there is no way that the result of the operation can be greater than FFFFFFh, we know that the two first bytes of the key are `c1` (because $a \oplus a = 0$). We divided the number of possibilities by 256, this imply cracking will be 256 times faster !

You have to program something that will try to find the correct key. Here is some source code to help you :

```
brute_loop proc uses esi, addr :DWORD

 mov     eax, start_value
 ; put c1000000 - 1 in start_value
 mov     esi, addr

bl_loop:
 inc     eax
 jz      bl_end

 push    eax
 push    esi

 call    decode_8bytes

 or      ecx, edx
 jnz     bl_loop

bl_end:
```

```
 int     3 ; when we are here, eax should contain the correct key.

 ret

brute_loop endp

decode_8bytes proc uses ebp eax ebx esi edi, to_decode : DWORD, key : DWORD

; optimized the original safedisc routine a little bit
; do not know if I can do better

 mov     esi, to_decode
; to_decode should point to CD DE 58 A2 75 01 7F
 mov     edx, [esi]
 mov     ecx, [esi+4]

 mov     eax, rdata_val
 ; put the 4 first bytes of dplayerx.dll.rdata here
 shl     eax, 5

 mov     esi, key

 mov     ebp, how_many ; we use ebp as a counter
; put the 4-8th bytes of dplayerx.dll.rdata in how_many
 test    ebp, ebp
 jbe     end_routine


start_loop: ; we loop 20h times
 mov     ebx, esi ; key+8
 mov     edi, edx
 shl     edi, 4
 add     edi, ebx
 mov     ebx, edx
 shr     ebx, 5
 add     ebx, esi ; key+C
 xor     edi, ebx
 lea     ebx, [eax+edx]
 xor     edi, ebx

 sub     ecx, edi

 mov     ebx, esi ; key+4
 mov     edi, ecx
 shr     edi, 5

 add     edi, ebx
 lea     ebx, [eax+ecx]
 xor     edi, ebx
 mov     ebx, ecx
 shl     ebx, 4
 add     ebx, esi ; key+0
 xor     edi, ebx
 sub     edx, edi
```

```
 sub     eax, rdata_val

 dec     ebp
 jnz     start_loop

end_routine:

 ret

decode_8bytes  endp
```

You have of course to add more to obtain a full program, but the hardest part is explained here. I suppose that you understood that the algorithm uses a 32 bytes long key, but that this key is actually 4 times the same 8 bytes long key.

After a short period, if your program works fine, you should find the same key than before : `c15cf2e5`.

3.2.4 Hacking the `dplayerx.dll`

You can follow the very same steps for hacking `dplayerx.dll`, except that you should not remove the `ICD` deciphering. It is now needed to alter the key before this function is called.

A very easy task, all that you have to do, is to put a breakpoint on 8F40B0 (`bpx 8f40b0` when you are within dplayerx.dll). When you will break, put these bytes starting at 908988 : `e5 2f 55 c1 e5 2f 55 c1 e5 2f 55 c1 e5 2f 55 c1` (`ed ds:908988 c15cf2e5,c15cf2e5,c15cf2e5,c15cf2e5`, afterward check with `db ds:908988` that bytes are in the reversed order.)

> **Caution** : make sure you have no breakpoints within `rayman2.icd.text` before proceeding.

Once the memory has been modified, step over the call, both sections are now decoded. You may dump them and rebuild the ICD file. Once this is done, all you have to do is to hack `dplayerx.dll` so that the `.rdata` section get deciphered correctly.

The final result should be the same than the one we obtained with the original CD. Test it. . .

. . . congratulations, you just cracked Safedisc **without** the original CD.

3.2.5 Minimal cracking

You learnt how to crack without the CD. What are the files you need for cracking ? Here is the list :

1. `rayman2.exe`
2. `dplayerx.dll`
3. `rayman2.icd`

This should be enough for cracking. `:-)`

> **note** : if you are working on IRC, for an ISO release for example (thou knave !), you only need these three files. However, if the provider can dump the sections for you, it is of course better.
>
> The Prestige's fellows told me that the whole ISO was transmitted through ftp to the cracked. Try my method...

Chapter 4

Conclusion

But Death, alas ! I cannot shun;
Death must come when he is gone
"Dido & Aeneas"

4.1 Safedisc is no more

This protection is obsolete. Not only the original CD can be copied, but as I showed you, it is possible to crack the protection with the help of only three files.

And there is more to investigate ! I have not tried it yet, but I suppose that cracking the drivers used to read the bad sectors of the CD could give good results too. I fear, however, that the protection might be blind at this point (i.e. not aware of what it should find). I also fear that modifying the driver's dll might be detected, which would force us to hack the main executable and `dplayerx.dll`, leading to an equivalent solution.

Do you remember Pedro's generic Safedisc crack ? It simulates bad sectors so as to fool the protection, but it only works with older Safedisc revision. I suppose that if the crack is resident, there is no generic solution to modify the dll.

Everything I explained here can be performed by Black Check's brute forcer, except that the `.rdata` will be rebuilt. This is 99% reliable, the only 100% reliable solution being, in my humble opinion, to hack `dplayerx.dll` as we did.

R!SC's unwrapper is also a nice tool, but it has not been updated for a long time now, and needs the presence of the original CD. The method used is completely different here, since it **dumps** the data, it does not decode it.

4.2 The future

I am certain that Safedisc will continue to release new versions of Safedisc (well until developers realize they are wasting their money), and I shall continue to reverse them.

Right now, I have Unreal Tournament to play with, it uses an even more recent version of Safedisc. Future versions of this essay will include the results of my *inverstigations !* (as well as corrections I suppose `:-)`).

However, I doubt that major changes will occur, therefore if you master the techniques described, you should also be able to crack future versions of Safedisc.

What I expect to change :

- Some changes in the deciphering algorithm. (find the new one and reverse it)
- New anti-debug tricks. (use Frogsice and your brain, remember the debug registers)
- More checksums. (`bpm` is your friend)
- Some reorganizations within the code. (just analyse the code with IDA)

As you can see, they can not really catch you by surprise. . .

4.3 Reliable CD protection does not exist

It is impossible to create a reliable CD protection on PC, since the hardware is generic. Therefore, you can only put the protection that CD drives are able to read, and if they are able to read it, you can copy it.

Worst, the best that can do the protection makers is hiding the protection code, but they cannot remove it. Indeed, at a moment or another, the executable must be deciphered into memory so that it can be executed. As well, checksums routines are code that is executed, as I showed you, it is possible to bypass them.

The security is equivalent to a very solid door, but with the key hidden nearby. . .

Looks like a blank page, does not it ? :-)