

**Manual unpacking of the SafeDisc protector version 2.30.33 on**

**example MaxPayne version 1.0**

**Author: mr\_magic**

**Date: 11/22/2004**

**Translation and editing: DillerInc**

**Experiment victim: MaxPayne.exe (preferably have the original disk)**

**Size: 5,799,256 bytes**

**Tools :**

* **SoftIce**
* **IceExt**
* **P E E ditor (part of PETools or LordPE )**
* **ImpRec**
* **Hex Workshop**
* **Hiew**

**Category: manual unpacking**

|  |
| --- |
| **Content** |
| Introduction |
| Analysis of the entry point ( EP) , search for the original entry point ( OEP) |
| Import system analysis |
| Fixing the Import Address Table ( IAT), part one |
| Correction of the import address table ( IAT), part two |
| Import fix, part one |
| Import fix, part two (digression from the main topic) |
| Import fix, part three |
| Have you reached the final? |
| Dumping and recompilation |
| Some conclusions |
| Personal wishes (it is advisable to read first) |
| Disassembled PE Header |

|  |
| --- |
| **Introduction** |

**The following is for educational purposes only‼**

The purpose of this article is not to provide any step-by-step guide on how to unpack games protected by SafeDisc II. version. Much more attention will be paid here to the very principle of this particular work, which, on the one hand, does not guarantee the absolute effectiveness of the attempts described here, but, on the other hand, helps to place some important accents necessary to achieve success.

So, we have to neutralize the already slightly moss-covered SafeDisc protector (“SD” hereinafter ) of the second version, and this is not accidental, because by virtue of due to some negligence of the Macrovision developers , the fundamental protective mechanisms described in this article still remain relevant in newer versions of this protector. Apparently, the old protective mechanisms are only supplemented by new ones, which, however, break down according to the same principle as the old ones. But that’s still for us remains to be explored ☺.

*“Please insert the correct CD-ROM, select OK and restart application”*

I think many people have seen a similar phrase when trying to launch a game without a disk or from an inaccurate copy.

You can find out the exact version of the tread using the PEiD program , but if you at least sometimes act on the “ Do it yourself” principle! ” , then know that by opening our executable file in Hex Workshop and looking at the end of *the PE header* , we will find the version number after the line “ BoG\_ \*90.0&!! Yy> ” and it is represented there as three values of the Unsigned Long type. In the case of MaxPayne 1.0 this is:

020000001E00000021000000h

...or 2.30.33.

Before we begin, let's do some standard preparations.

Let's open our victim in PE Editor and take a look at the sections, but first an educational digression:

|  |
| --- |
| The two lower sections are stxt774 and stxt371 - belong to SafeDisc. The upper one (stxt774) is responsible for certain protective mechanisms, and continues to be used during the game. The lower one (stxt371) is the boot section, and after decrypting the executable file is no longer used by the game. |

Let's get back to business. This means that we need to change the write permissions in several significant sections. And here we are again faced with that strange negligence of Macrovision programmers - PE Header is not checked for changes, i.e. no checksum checks or anything like that. Here the author suggests using PE Editor to change the characteristics of the .text and .rdata sections as follows:

.text = E0000020

.rdata = C0000040

...apparently, this is done only to set the sections to *Writeable mode* .

And finally, in order to be able to somewhat calmly debug our executable file, it is necessary to activate the IceExt *!Protect on* command in SoftIce , because SD actively uses various anti-debugging techniques, and if the debugger is detected, which unfortunately happens quite often, you will have to reboot the entire system.

|  |
| --- |
| **Analysis of entry point ( EP), search for original entry point (OEP)** |

To take a closer look at SD protection, we first need to get to the entry point in the debugger. There are several ways to do this, the simplest of which is probably using the “Break & Enter” option of the same PE Tools. Once in the debugger, we should see this:

|  |
| --- |
| **008CB04A 55 PUSH EBP**  **008CB04B 8BEC MOV EBP,ESP**  **008CB04D 60 PUSHAD**  **008CB04E B8B9B08C00 MOV EAX,008CB0B9**  **008CB053 2D4AB08C00 SUB EAX,OFFSET MaxPayne.{ModuleEntryPoint}**  **008CB058 0305BAB08C00 ADD EAX,DWORD PTR DS:[8CB0BA]**  **008CB05E C7054AB08C00E9 MOV DWORD PTR DS:[{ModuleEntryPoint}],0E>**  **008CB068 A34BB08C00 MOV DWORD PTR DS:[8CB04B],EAX**  **008CB06D 6809B08C00 PUSH 008CB009**  **008CB072 A02DB08C00 MOV AL,BYTE PTR DS:[8CB02D]**  **008CB077 3C01 CMP AL,1**  **008CB079 7407 JE 008CB082**  **008CB07B B800000000 MOV EAX,0**  **008CB080 EB03 JMP 008CB085**  **008CB082 8B4508 MOV EAX,DWORD PTR SS:[EBP+8]**  **008CB085 50 PUSH EAX**  **008CB086 E833000000 CALL 008CB0BE ; Procedure decryption games**  **008CB08B 83C408 ADD ESP,8**  **008CB08E 83F800 CMP EAX,0 ; If EAX = 1, then error**  **008 CB 091 741 C JE 008 CB 0 AF ;if decryption went smoothly**  **008CB093 C7054AB08C00C2 MOV DWORD PTR DS:[{ModuleEntryPoint}],0C**  **008CB09D C7054BB08C000C MOV DWORD PTR DS:[8CB04B],0C**  **008CB0A7 50 PUSH EAX**  **008CB0A8 A129B08C00 MOV EAX,DWORD PTR DS:[8CB029]**  **008CB0AD FFD0 CALL EAX**  **008CB0AF 61 POPAD ;... then let's jump here**  **008 CB 0 B 0 5 D POP EBP ; tidy up the stack**  **008CB0B1 EB06 JMP 008CB0B9**  **008CB0B3 7216 JB 8CB0CB**  **008CB0B5 61 POPAD**  **008CB0B6 13600D ADC ESP,DWORD PTR DS:[EAX+D]**  **008CB0B9 E97645EAFF JMP 0076F634 ;...и прыгаем на OEP!!!** |

In passing, it is worth noting that by simply changing the jump command to address 008 CB 091 owners of a non-original CD will not achieve anything, because... the game (including the OEP itself) are decrypted and calculated using the digital signature of the CD and the so-called digital signature created with the help of this signature. *TEA-Key* .

If you don't believe me, try it yourself and see what happens.

However, the above was presented purely for informational purposes, and in the future we will act a little differently to get to the OEP .

As you can see, after starting the game, a small window briefly appears on the screen with the image of Max with his legendary Beretta. So, at the moment when this window disappears, you need to quickly press Ctrl + D to activate SoftIce , and enter in it:

: addr MaxPayne ; the hike is necessary only under XP

: bpx 008CB0AF ; I, working on w2k SP4, used only this command here

... F5... and we find ourselves back in the debugger before switching to OEP.

Thus, the simplest part has been overcome ☺.

|  |
| --- |
| **Import system analysis** |

If we now recall the good old SafeDisc of the first version, it should be noted that then the functions imported from the Kernel32 and User32 libraries seemed to be “ overwritten ” ( or “ cut off ” - whichever you prefer), i.e. were redirected to SD procedures , which, in turn, then called these same imported functions. It would be even surprising if in the second version of the protector this would look somehow different. And we really won’t be wrong in our assumptions by tracing a little code:

**0076F661 FF1550937C00 CALL DWORD PTR DS:[7C9350]**

**...tracing the code below the OEP in the debugger, we will not see any names of imported API functions. Instead, we will quite often come across similar “ reference ” calls, as shown above.**

**For brevity, we call them *Call [ Ref ]* ( *Ref* - from the English reference - link).**

**If we go into this procedure (F8), we will see the following:**

|  |
| --- |
| **01975759 68 EA 12 EABF PUSH BFEA 12 EA ; Pushes the first parameter onto the stack**  **0197575E 9C PUSHFD**  **0197575F 60 PUSHAD**  **01975760 54 PUSH ESP**  **01975761 6899579701 PUSH 01975799 ; Import address+40 h — second parameter**  **01975766 E 8 DC 25 F 200 CALL 00 F22C90 ; Calling SD procedure**  **0197576B 83C408 ADD ESP,8**  **0197576E 6A00 PUSH 0 ; ... A Here**  **01975770 58 POP EAX ; here**  **01975771 61 POPAD ; We never**  **01 97 5772 9D POPFD ; we get there**  **01 97 5773 C3 RET** |

*Note: I will explain why the addresses are colored blue a little later.*

SD procedure is passed two parameters, with the help of which it subsequently calculates a certain relative virtual address ( RVA) .

* The first parameter is a double word ( DWORD), always starting with *BF* .
* The second is the address of the truncated import (for example, in the case above it is

address 01975759) + 40 h.

The resulting RVA is nothing more than the address at which one or another imported function is located.

In other words, the SD procedure seems to restore the import, but, apparently, only at the moment of its call. The rest of the time, this same import, as you might guess, is something cut off and erased (remember Call [ Ref ]'s) .

If we continue talking about the first version of SafeDisc , then as an example we need to give the code that could be used to fix all this import that was jammed by the protector:

|  |
| --- |
| **mov e ax , $ IAT\_Start\_Offset - 4**  **\_start : \_**  **add eax,4**  **cmp eax, $End\_IA T**  **ja\_end \_**  **cmp dword ptr [ eax ], $lower\_bound\_of\_trimmed\_import**  **jl\_start \_**  **cmp dword ptr [ eax ], $truncated\_import\_upper\_bound**  **j\_start \_**  **call [ eax]**  **mov dword ptr [eax], edx**  **jmp\_start**  **\_end:**  **jmp eip** |

In general, it was necessary to use code that would examine the entire IAT for calls leading to the SD procedure . If such a thing was found, the actual RVA was called , i.e. in fact, the SD procedure itself was called, which carefully calculated the correct address of the specific imported function and returned it in the EDX register.

All this later became a little more complicated, because... the SD procedure no longer returned the calculated address in this way, which, however, was very easily corrected with the help of a small patch. But, as they say: “ It’s a dry theory, my friend ,” so we need to try to see if this method still works. To do this, we must first get to the end of this notorious SD procedure.

So, let’s go into it (F8) at the address 01975766 and start cutting through this jungle of self-modifying code using F10 for now We won't end up in the following place:

|  |
| --- |
| **00F23243 F0:FF0D F48F1201 LOCK DEC DWORD PTR DS:[1128FF4] ; LOCK prefix**  **00F2324A 780D JS 00 F 23259**  **00F2324C 8B0DEC311301 MOV ECX,DWORD PTR DS:[11331EC]**  **00F23252 51 PUSH ECX**  **00F23253 FF15 64401101 CALL KERNEL32.SetEvent**  **00F23259 EB07 JMP 00F23262**  **00F2325B 8BC0 MOV EAX,EAX**  **00F2325D 7806 JS 00F23265**  **00F2325F 90 NOP**  **00F23260 7903 JNS 00F23265**  **00F23262 EBF7 JMP 00F2325B**  **00F23264 88 --Junk Byte--**  **00F23265 8B650C MOV ESP,DWORD PTR SS:[EBP+C] ; Vosstanavlivaet**  **0 0F2 3268 61 ATTACK ; registry from**  **0 0F2 3269 9 D POPFD ; the stack**  **0 0F2 326 A C 3 RET ; ...and jumps to the correct import**  **00F2326B 76 04 JBE 00F23271**  **00F2326D 90 NOP**  **00F2326E 90 NOP** |

Standing on the RET command , it makes sense to look into the stack - there we will see the correct RVA of the import as the topmost parameter, and immediately after it will be our return address, and not a simple one, but received through the system ( *CALL DWORD PTR DS offset : [7 C 9350]* ) *+* 6.

By tampering with the code here a little, we will force the SD procedure not to call the correct RVA of the import, but to return it to us for further processing. Thanks to the rich garbage code, we have enough space to place our patch:

: a 00F23259

|  |
| --- |
| **00F23259 8B650C MOV ESP,DWORD PTR SS:[EBP+C] ; We restore**  **00 F 2325 C 61 POPAD ; registers**  **00 F 2325 D 9 D POPFD ; from the stack**  **00 F 2325 E 5 A POP EDX ; and place the correct import RVA in EDX**  **00F2325F C3 RET** |

...if we now “ jump ” through RET , we will find ourselves at the return address, and the EDX register will contain the RVA to our import.

Looking again at the code that was used in the good old SafeDisc of the first version to restore imports erased by the protector, we will see that we are missing a couple more values. Namely:

* The beginning of IAT , and more specifically the beginning of the .rdata section (look in PE Editor) - 007С9000
* The length of this same IAT is 1000h (the table of import addresses will actually be larger in size, but according to the author of the article, you are unlikely to find anything interesting beyond this limit of a thousand bytes)
* The limits of the address space of the protector... so... stop! Here we need to make a little reservation...

It is better to move the bottle of beer to the other end of the table for a while, because, in my opinion, this is a fairly important point to understand (albeit quite simple).

By the phrase “ protector address space ” I personally mean the address space, those addresses to which our Call [ Ref ]s refer (remember those?), i.e. that area where SD does its dark deeds on our poor imports.

In other words, if we go (F8) to these Call [ Ref ] 's, we will see these characteristic addresses, based on which we can approximately determine the limits of the protector's address space, i.e. its lower and upper boundaries.

Now remember about our blue addresses - these are what they are - the addresses of our protector in a particular case. And if you go not only to the Call [ Ref ] described above, but also to others, then, as I said, you can approximately outline the boundaries of these addresses.

For example, for me these addresses looked either

0197ХХХХ,

...or

0198ХХХХ,

...accordingly, I decided to mark the boundaries as follows:

01900000 - lower limit

01 A00000 - upper limit

...like with a reserve ☺.

|  |
| --- |
| **Fixing the Import Address Table ( IAT), part one** |

Now we seem to have everything ready to start fixing our IAT. All that remains is to find a suitable place to implement our correction code. The author of the article decided to use the very beginning of the PE file for this, namely the place at offset 00000500, because There are so many zeros there that you could probably land an entire Boeing.

Here's some sketch:

|  |
| --- |
| **mov ecx ,007 C 8 FFC ;Start IAT -4**  **\_start : \_**  **add ecx ,4; Continue with the next RVA**  **cmp ecx .007 CA 000 ; Have we reached the end of the IAT ?**  **ja\_end \_ \_**  **cmp dword ptr [ ecx ],01900000; If not, then check whether RVA indicates**  **jl\_start \_ \_**  **cmp dword ptr [ ecx ],01A00000; tread area?**  **j\_start \_ \_**  **call dword ptr [ ecx ]; If yes, then call him to receive**  **mov dword ptr [ ecx ], edx ; correct RVA, replacing the old RVA in IAT with it**  **jmp\_start \_**  **\_end \_**  **jmp eip ; We loop to give us control** |

But however, there are a couple more points that need to be taken into account. Namely, if we carefully trace the SD procedure ( which we, by the way, call indirectly in our sketch code), then we will come across places like this:

|  |
| --- |
| **...**  **00F22E38 E8F3E0FFFF CALL 00 F2 0F30**  **00F22E3D 83C40C ADD ESP,0C**  **00F22E40 8945F4 MOV DWORD PTR SS:[EBP-C],EAX**  **00 F 22 E 43 837 DF 400 CMP DWORD PTR SS :[ EBP - C ],0 ; The import address has not been restored yet?**  **00 F 22 E 47 0 F 847 B 010000 JE 00 F 22 FC 8 ; If not, we restore it !**  **00 F 22 E 4 D 7507 JNZ 00 F 22 E 56 ; Otherwise, we use the already obtained RVA**  **...**  **00 F 2319 B 837 DF 400 CMP DWORD PTR SS :[ EBP - C ],0 ; The import address has not been restored yet?**  **00 F 2319 F 7529 JNZ 00 F 231 CA ; If it is still restored, we use the already received RVA**  **00F231A1 8B45E0 MOV EAX,DWORD PTR SS:[EBP-20] ; Happening here**  **00F231A4 50 PUSH EAX ; calculation**  **00F231A5 8B4DE8 MOV ECX,DWORD PTR SS:[EBP-18] ; correct RVA**  **00F231A8 51 PUSH ECX ; imported**  **00F231A9 E882F4FFFF CALL 00F22630 ; functions**  **...** |

As can be seen from the above code, there are two checks for the restoration of the import.

In order to avoid emergency situations, it is better for us to patch this matter so that our correct RVA is calculated anew each time, i.e. change the conditional jump at address 00 F22E47 to unconditional, and the second conditional jump at address 00 F2319F simply for *NOP* it.

As for the calculation of the correct RVA itself, we have already encountered the two parameters used in this operation - they are pushed onto the stack before calling the SD procedure.

If we now trace (F10) through the call to address 00 F231A9 and see what we have in the EAX register...

:what eax

... then, unfortunately, we won’t see anything worthwhile. Although in this case we will be dealing with importing the MSVCRT library , but It will be unclear which one.

This leads us to the conclusion that the two parameters that we talked about above are not the only ones. And in fact, if we call the SD procedure under normal conditions (i.e. from the program itself, and not from our stump sketch) and at the same time look into the stack, we will see a clearer picture:

99 57 97 01 ... XX ... EA 12 EA BF **67 F6 76 00**

XX are intermediate bytes (basically providing the stack through the commands PUSH, PUSHAD, PUSHFD ...) . And since the stack is read “ back to front ” , the missing parameter in this case is the value 0076 F667 - the return address after calling the current Call [ Ref ]'a ( Call [ Ref ]\_offset + 6) !!!

This means that SD also uses the offset of the actual Call [ Ref ] + 6 as a third parameter to calculate the RVA of the imported function.

Well, this forces us to assemble new, modernized code...

***A small lyrical digression...***

It seems to me that at this stage some readers may have quite fair questions about the events described in this article, especially if these readers have not yet had experience with such unpackings.

One of the first questions that arose for me personally concerned the choice of a tool for introducing our correction code into the executable file of the game - do not write opcodes in some hex editor. In this case, I personally decided to use the well-known HIEW, because . there you can do it all very clearly, i.e. Feel free to write using assembler commands.

Next, one of the unclear points may be the launch of our correction code at the right time.

So, there is this option: after we have stopped in the debugger not far from the jump on OEP , we trace to this jump and, standing on it, enter the following...

: r eip = 00400500

... i.e. we change the state of the EIP register to the starting address of our code, press F10 and we find ourselves where we need to be. Well, then it’s a matter of technique... ☺.

|  |
| --- |
| **Correction of the import address table ( IAT), part two** |

What we intend to do now is to comb through the entire program code in search of Call [ Ref ]s, having found which we must find out whether they point to the area where our IAT is located . If successful, we make sure whether we are dealing with a worn-out import, i.e. do our found Call [ Ref ] s refer to the tread area. Next, again if successful, we have in our hands the offset of Call [ Ref ] , by which the SD procedure is called, and thus the treasured third parameter for calculating the correct RVA import.

All code is in the .text section, which starts at address 00401000

and ends with 007С8 FFF.

|  |
| --- |
| **mov eax , 00400 FFF ; Start of the .text section - 1**  **\_start : \_**  **inc eax ; EAX increment - let's move on...**  **cmp word ptr [ eax ], 15 FF ; Current function - Call [ Ref ] ?**  **je\_FirstCheck ; \_ \_ ...if yes, let's check it further**  **cmp eax , 007С8 FFF ; Have we reached the end of the .text section yet ?**  **jne\_start \_ \_**  **jmp eip**  **\_FirstCheck : \_**  **cmp dword ptr [ eax +2], 007 C 9000 ; does Call[Ref ] point to**  **jl\_start**  **cmp dword ptr [eax+2], 007CA000 ; region IAT?**  **ja\_start**  **\_SecondCheck:**    **mov ecx, dword ptr [eax+2] ; In ECX we place the import offset in the address table**  **cmp dword ptr [ ecx ], 01900000; Import offset refers**  **jl\_start \_ \_**  **cmp dword ptr [ ecx ], 01А00000; on the tread area?**  **ja\_start**  **; From now on we have a 100% redirected call,**  **; those. worn out import**  **push \_ CallFixReturnPoint ; Our address where we want to return after the SD procedure**  **add eax, 6**  **push eax ; Third parameter ( Call [ Ref ] offset + 6)**  **sub eax, 6**  **jmp ecx ; JMP instead of CALL, so as not to touch the stack again**  **\_AfterCallFixReturnPoint : \_**  **mov dword ptr [ ecx ], edx ; Replace the RVA of the SD procedure in the IAT with the correct RVA**  **jmp\_start \_** |

But first , we need to tweak a few more things. Let's remember our little patch at the end of the SD procedure. This time we need to add it a little, namely:

|  |
| --- |
| 0 0F2 3259 8B650C MOV ESP,DWORD PTR SS:[EBP+C]  0 0F2 325 C 61 POPAD  0 0F2 325 D 9 D POPFD  0 0F2 325 E 5 A POP EDX ; As is already known, in EDX - correct RVA  **00 F2325F 83C404 ADD ESP, 4 ; Here we add a stack correction!**  00F23262 C3 RET ; ...and back to our correction code |

The stack correction was required because in our corrective code, before calling the SD procedure, we added an additional parameter to the stack using the PUSH command - our return address where we want to land.

Let's launch...

Everything should have gone smoothly this time...or what?

If we now look at IAT in the debugger, we will sadly see that the lion's share of addresses are still not corrected, despite the constant calling of the SD procedure , which was supposed to restore them.

What's the problem?

With the help of our code, we imitated the normal functioning of the SD procedure very well. If we did not make mistakes in our code, then maybe the Macrovision programmers set some cunning traps for us.

What if they specifically decided to use Call [ Ref ]'s offset as the third parameter...?

We ourselves witnessed how the SD procedure , called from different places, returned different RVAs . Now it turns out that Call [ Ref ]'s read the same IAT address in order to receive different imported functions. These same Call [ Ref ]'s were previously different API functions, but now they look like the same ones to us.

In short, this means that we will never be able to achieve the desired result only by one-time correction of the import address table alone.

|  |
| --- |
| **Import fix, part one** |

Although these arguments are the logical conclusions of our mental work, the method described below was already previously published by the authors r!sc and Peex, for which we give them a special bow.

In general, we must fix not only the IAT, but also the .text section, for which we will need to always have the “ original ” available throughout the execution of our code IAT , i.e. containing trimmed imports so that we could then clean up the correct RVA of the imported function for each Call [ Ref ] .

The solution to such a problem is to create a “ temporary ” IAT , which we will then, after restoring all imports, be able to replace the original cropped IAT. To do this, we will use the code from our first attempt in a slightly modified form.

Now the main question remains where to find a thousand-byte space for our temporary IAT. The answer is quite logical - the boot section SD (stxt371) ceases to be used after the game is decrypted - that’s where we will be located.

|  |
| --- |
| **mov ecx , 007 C 8 FFC ; Start of IAT -4**  **mov ebx , 008 CAFFC ; Start of loading section SD -4**  **\_start : \_**  **add ecx , 4; Let's move on...**  **add ebx , 4; We are moving further also in the temporary IAT**  **cmp ecx , 007 CA 000; Have we reached the end of the IAT ? ?**  **ja\_EndBuildIAT**  **mov edx, dword ptr [ecx] ; In EDX we enter the RVA of the current import**  **cmp edx , 01900000; Is RVA**  **jl\_BuildIAT \_ \_**  **cmp edx , 01А00000; in the tread area?**  **ja\_BuildIAT \_ \_**  **call edx ; If yes, I call the already familiar SD procedure**  **\_BuildIAT : \_**  **mov dword ptr [ ebx ], edx ; Entering the correct RVA into the temporary IAT**  **jmp\_start**  **\_EndBuild IAT :**  **jmp eip ;** Let's loop... |

Once this code is running, we'll have a great-looking temporary IAT that we can use to fix our Call [ Ref ]s.

Let us now pay attention to the appearance of Call [ Ref ] itself, for example:

0076 F 661 **FF15** 50937C00 CALL DWORD PTR DS: [7C9350]

* The first two bytes are always **FF15** ( opcode for Call [ Ref ])
* The rest is the offset in the import address table in Little Endian byte order , i.e. must be read backwards

.text section for FF15 opcodes.

If you find such, you will need to check whether they are talking about overwritten imports.

If so, we intend to modify this import so that the particular imported function can read the correct address in our temporary IAT and therefore then end up in the correct place in the original IAT.

|  |
| --- |
| mov eax, 00401000 ; Start sections . text  \_SearchForCall[Ref]'s:  cmp word ptr [ eax ], 15 FF ; The first two bytes of the actual function FF 15 ?  je\_Call [ Ref ] Found ; \_ \_ If yes, then check Call [ R ef ]  \_ SearchForCall [ Ref ]' sLoop :  inc eax ; Otherwise, let's keep looking...  cmp eax , 007 C 8 FFF ; Have we reached the end of the section? text ?  jne\_SearchForCall [ Ref ] ' s \_  jmp eip ; We loop to give us control  \_Call[Ref]Found:  cmp dword ptr [ eax +02], 007 C 9000 ; Is the current offset  jl\_SearchForCall [ Ref ] ' sLoop \_  cmp dword ptr [ eax +02], 007 CA 000 ; in the field of the original IAT ?  ja\_SucheNachCallRef'sLoop  \_CheckIfMangledCall[Ref]:  mov ecx , dword ptr [ eax +02]; In ECX we enter the import offset in the address table  mov ecx , dword ptr [ ecx ]; In ECX we enter the RVA of the current imported function  cmp ecx , 01900000; Does this RVA indicate  jl \_SearchForCall[Ref]'sLoop  cmp ecx , 01 A 00000; tread area?  ja\_SearchCall[Ref]'sLoop  Push \_ FixCall [ Ref ] ReturnPoint ; If yes, set our desired return point  add eax , 6; Calculate the third parameter ( Call [ R ef ] offset + 6 )  push eax ; We push the resulting value onto the stack  sub eax , 6; In ECX again the address of the current Call [ R ef ]  jmp ecx ; And call the SD procedure to get the correct RVA  \_FixCall[Ref]ReturnPoint:  mov ebx , 008 CB 000; In EBX we place the beginning of the temporary IAT  \_FixCall[Ref] Loop:  cmp dword ptr [ ebx ], edx ; Looking for the current RVA in the temporary IAT  je\_FixCall [ Ref ] \_ \_  add ebx , 4; Checking the following import in temporary I AT  cmp ebx , 008 CC 000; Have we reached the end of the temporary IAT ?  jl\_FixCall [ Ref ] Loop; \_ \_ If not, then look further...  jmp eip ; Otherwise, by looping, we transfer control to find the error  \_FixCall[Ref]:  sub ebx , 008 CB 000; In EBX we enter the offset of the found RVA relative to the beginning of the temporary IA T  add ebx , 007 C 9000; and then calculate the correct offset in the original IAT  mov dword ptr [ eax +02], ebx ; Finally, we update Call [ R ef ] with the new offset of the current import |

At the end of this whole procedure, we will have a corrected section in our hands. text , where all Call [ Ref ]s cause the correct import.

|  |
| --- |
| **Import fix, part two (digression from the main topic)** |

Unfortunately, the so-called Call [ Ref ]s are not the only defense mechanisms that ruin our lives. There are two more “ flaws ” , of which, fortunately, we only need to correct one.

1. Emulation of calls through general purpose registers is used
2. There are so-called “ unconditional jump tables ” ( Jumptables ), which, strictly speaking, function almost similarly to Call [ Ref ]s, i.e. we can say that there are so-called *Jmp [ Ref ]'s* , which are called by the command *Call offset\_ Jmp [ Ref ]*

Although we do not yet need to correct the first option in this version of the protector, nevertheless, the author provides the corresponding corrective code for general development:

|  |
| --- |
| **mov ecx, $ Beginning\_Original\_IAT - 4**  **\_SearchIAT:**  **add ecx, 4 ; Are looking for Further**  **cmp dword ptr [ ecx ], 01900000; Does RVA indicate**  **jl\_SearchIAT \_ \_**  **cmp dword ptr [ ecx ], 01 A 00000; on the tread area?**  **ja\_SearchIAT:**  **mov eax, $ Beginning\_Text\_Section**  **\_SearchForReference:**  **cmp dword ptr [ eax ], ecx ; Does the current function match the current RVA ?**  **je \_ CheckIfCallRegister ; If yes, check whether the call occurs through the register**  **\_SearchFor ReferenceLoop :**  **inc eax ; Let's move on to the .text section**  **cmp eax , 007 C 8 FFF ; Have we reached the end of the section? text ?**  **jl\_SearchForReference ; \_ \_ ...if not, look further...**  **jmp eip**  **\_CheckIfCallRegister:**  **cmp word ptr [eax-2], 1D8B ; Used whether register EBX at call current RVA?**  **je\_ebxHandle**  **cmp word ptr [ eax -2], 358 B ; Is the ESI register used when calling the actual RVA ?**  **je\_esiHandle**  **cmp word ptr [ eax -2], 3 D 8 B ; Is the EDI register used when calling the actual RVA ?**  **je\_ediHandle \_ \_**  **cmp word ptr [ eax -2], 2 D 8 B ; Is the EBP register used when calling the actual RVA ?**  **je\_ebpHandl e**  **jmp \_S searchFor ReferenceLoop**  **\_ebxHandle:**  **mov edx , D 3 FF ; Set the appropriate search tag for " Call edx "**  **jmp\_FindCallRegister**  **\_esihandle:**  **mov edx , D 6 FF ; Set the appropriate search tag for " Call esi "**  **jmp\_FindCallRegister**  **\_ediHandle:**  **mov edx , D 7 FF ; Set the appropriate search tag for " Call edx "**  **jmp\_FindCallRegister**  **\_ebpHandle:**  **mov edx , D 5 FF ; Set the appropriate search tag for " Call ebp "**  **jmp\_FindCallRegister** |

...continuation:

|  |
| --- |
| **\_FindCallRegister:**  **mov esi, eax**  **add esi , 4; We place the following function in ESI after “mov Register, [007 C 9 XXX ] ”**  **\_FindCallRegisterLoop:**  **cmp word ptr [ esi ], dx ; We are looking for the corresponding two bytes**  **je\_CallRegisterFound**  **\_SearchFurther CallRegister :**  **inc esi**  **jmp\_FindCallRegisterLoop**  **\_CallRegister Found :**  **push \_ CallRegisterReturnPoint ; Setting the desired return point**  **add esi , 2; We calculate our third parameter**  **push esi ; and put it on the stack**  **jmp dword ptr [ ecx ]; Calling the SD procedure to get the correct RVA**  **\_CallRegisterReturnPoint : \_**  **mov ebx , 008 CB 000; In EBX we place the beginning of the temporary IAT**  **\_FixCallRegisterLoop:**  **cmp dword ptr [ ebx ], edx ; Finding the current RVA in the temporary IAT**  **je\_FixCallRegister**  **add ebx , 4; Checking the following import in temporary IAT**  **cmp ebx , 008 CC 000; Have we reached the end of the temporary IAT ?**  **jl\_FixCallRegisterLoop ; \_ \_ If not, then look further...**  **jmp eip ; Otherwise, by looping, we transfer control**  **\_FixCallRegister : \_**  **sub ebx , 008 CB 000;** In EBX we enter the offset of the found RVA relative to the beginning of the temporary IA T  **add ebx , 007 C 9000;** and then calculate the correct offset in the original IAT  **mov dword ptr [ eax ], ebx ;** Finally, we update “mov Register, [ 007C9XXX ] ” with the new offset  ; current import |

Okay, we’ve looked and that’s enough... it’s time to finish the main work...

|  |
| --- |
| **Import fix, part three** |

So, let's return to option number two, namely to the tables of unconditional transitions.

As noted, these Jmp [ Ref ]s function similarly to Call [ Ref ]s and are invoked with the *Call offset\_Jmp [ Ref ] command* .

Let's illustrate this matter:

|  |
| --- |
| ; If we trace a little further from the OEP, we will see the following...  **0076 F 69 E E872010000 CALL 0076F815 ; This is the type Call offset\_ Jmp [ Ref ]**  ; Having gone inside and traced a little, we will find ourselves in such a place...  **0076 F816 FF25E0937C00 JMP [007 C 93 E 0] ; Here it is - the same Jmp [ Ref ]**  ; Jumping on it, we will find ourselves in normal SD procedure |

In general, everything is painfully familiar...these same Jmp [ Ref ]s are eventually restored just like Call [ Ref ]'s. However, not everything is as simple as it seems. Tracing in the vast expanses of the .text section, we According to the author, we can stumble upon the following “ jumping ” option , for example :

|  |
| --- |
| **00408E6C E9B61D4C00 JMP 008CAC27 ; Let's jump...**  ...  **008CAC27 53 PUSH EBX ; Оказываемся в пространстве секции .stxt774**  **008CAC28 E800000000 CALL 008CAC2D**  **008CAC2D 870424 XCHG DWORD PTR SS:[ESP],EAX**  **008CAC30 9C PUSHFD**  **008CAC31 05D3F3FFFF ADD EAX,-0C2D**  **008CAC36 8B18 MOV EBX,DWORD PTR DS:[EAX]**  **008CAC38 6BDB6B IMUL EBX,EBX,6B**  **008CAC3B 035804 ADD EBX,DWORD PTR DS:[EAX+4]**  **008CAC3E 9D POPFD**  **008CAC3F 58 POP EAX**  **008CAC40 871C24 X C HG DWORD PTR SS:[ESP],EBX**  **008CAC43 C3 RET ; We leave via RET...**  **...**  **019863 A 0 68718 E 4000 PUSH 408 E 71 ; Manually push the third parameter onto the stack**  **019863 A 5 685711 EABF PUSH BFEA 1157; Second parameter**  **019863AA 9C PUSHFD**  **019863AB 60 PUSHAD**  **019863AC 54 PUSH ESP**  **019863 AD 68 E 063 C 801 PUSH 19863 E 0 ; First parameter**  **019863 B 2 E 8 D 9 C 848 FF CALL 0 0F2 2 C 90 ; Calling the SD procedure**  **019863B7 83C4 08 ADD ESP,8**  **019863BA 6A 00 PUSH 0**  **019863BC 58 POP EAX**  **019863BD 61 POPAD**  **019863BE 9D POPFD**  **019863BF C3 RET** |

Again, for brevity, let's call this option *LongJump* .

Ultimately, these LongJumps are also restored by the SD procedure , just like Jmp [ Ref ]s (and, accordingly, Call [ Ref ]s), with the only difference that that scandalous third parameter is pushed onto the stack as if manually, which is what we observed above.

The fact that LongJump 's are tagged into the .stxt774 section gives us the opportunity to focus our attention within the confines of that section, namely the space 008 CA000 - 008CB000.

Let's take a closer look at these LongJumps :

**00408E6C E9B61D4C00 JMP 008CAC27**

**As you can see, they consist of byte E9 and a DWORD value , which denotes the number of bytes from the offset of the current LongJump to the target offset.**

**It is also worth noting that not all LongJumps leading to the . stxt774, were previously calls ( CALL) to unconditional jump tables. This can be found out already because not all imported functions restored according to the following principle have Jmp [ Ref ]s.**

**In short, this means that some of these LongJumps were previously Call [ Ref ]s.**

**And here the decisive difference between Jmp [ Ref ] and Call [ Ref ] will be the different length , because *Call offset\_ Jmp [ Ref ]* (like a prelude to Jmp [ Ref ]) is five bytes long , and Call [ Ref ] is six bytes in length .**

**In order to now fix LongJump 's we need to decide on a possible replacement, i.e. either it is Call [ Ref ] or Jmp [ Ref ].**

**We will determine by the byte that would be overwritten by Call [ Ref ]:**

**If this byte is equal to 90 h, then we are dealing with Call [ Ref ]'om.**

**If there is something different there, then we look for a suitable Jmp [ Ref ] and if one is found, we replace LongJump with the link Call offset\_ Jmp [ Ref ].**

**If a suitable Jmp [ Ref ] is not found, then we will transfer control to us to deal with it manually.**

**In fact, such a “ manual ” situation means that this LongJump is 99.9999% still a Call [ Ref ], and that same byte being overwritten is most likely garbage code introduced by Macrovision programmers to complicate our work.**

**Let's assemble:**

|  |
| --- |
| **mov eax , 00401000; Beginning of the section. text**  **\_ SearchFor LongJumps : \_**  **cmp byte ptr [ eax ], E 9 ; Are we dealing with LongJump ?**  **je\_JumpFound \_ \_**  **\_SearchForLongJumpsLoop : \_**  **inc eax ; Checking the next byte...**  **cmp eax , 00 7C8 FFF ; Have we reached the end of the section? text ?**  **jne\_SearchForLongJumps**  **jmp eip ; Let's go full circle...!!!** |

**...continuation :**

|  |
| --- |
| **\_JumpFound:**  **mov ecx, dword ptr [eax+01] ; IN We add ECX DWORD value - distance**  **add ecx , eax ; We add it with the current offset**  **add ecx , 00000005; and with LongJump 's length == offset-target**  **cmp ecx , 008 CA 000; Is the offset-target in**  **jl\_SearchForLongJumpsLoop \_ \_**  **cmp ecx , 008 CB 000; section areas. stxt774 ?**  **jnb\_SearchForLongJumpsLoop \_ \_**  **push \_ LongJumpFixReturnPoint ; If yes, then set our desired return point**  **jmp ecx ; and using the SD procedure we calculate the correct RVA of import**  **\_ LongJumpFixReturnPoint :**  **mov ebx , 008 CB 000; In EBX we place the beginning of our temporary IAT**  **\_LongJumpFixLoop:**  **cmp dword ptr [ ebx ], edx ; and examine it in search of the current RVA**  **je \_ LongJumpFix ; Found it, then let's jump further...**  **add ebx , 00000004; Otherwise, we look further...**  **cmp ebx , 008 CC 000; Have we reached the end of the temporary IAT ?**  **jl\_LongJumpFixLoop ; \_ \_ If not, then we look, look and look again...**  **jmp eip ; Otherwise, we’re going in cycles - in theory, we shouldn’t go in cycles here!**  **\_LongJumpFix:**  **sub ebx , 008 CB 000;** In EBX we enter the offset of the found RVA relative to the beginning of the temporary IA T  **add ebx , 007 C 9000;** and then calculate the correct offset in the original IAT  **cmp byte ptr [ eax +05], 90; The byte after LongJump is NOP ?**  **jne\_SearchJumpTable**  **\_LongJumpIsCall[Ref]:**  **mov word ptr [ eax ], 15 FF ; If yes, then we turn LongJump into Call [ R ef ] With**  **mov dword ptr [ eax +02], ebx ; suitable offset from our temporary IAT**  **jmp\_SearchForLongJumpsLoop ; \_ \_ And looking for the next LongJump ...**  **\_SearchJumpTable:**  **mov ecx, 00401000 ; Start .text sections**  **\_SearchJumpTableLoop:**  **cmp word ptr [ecx], 25FF ; Current function Jmp[ Ref ] ?**  **je\_CheckJmp [ Ref ] ; \_ If yes, let's check it...**  **\_SearchFurtherJmpTable:**  **inc ecx ; Otherwise, check the following function**  **cmp ecx , 007 C 8 FFF ; Have we reached the end of the section? text ?**  **jl\_SearchJumpTableLoop ; \_ \_ If not, look further...**  **jmp eip ; Looping, manual correction. This is 99.9999 % \_LongJmpIsCall[Ref]**  **\_CheckJmp[Ref]:**  **cmp dword ptr [ ecx +02], ebx ; Does the actual Jmp[Ref ] find the correct RVA in the temporary IAT ?**  **jne\_SearchFurtherJmpTable ; \_ \_ If not, look further...**  **sub ecx , eax ; Otherwise, we calculate the length of the command CALL,**  **sub ecx , 00000005; calling Jmp [ Ref ] \_**  **mov byte ptr [ eax ], E 8 ; and turn LongJump**  **mov dword ptr [eax+01], ecx ; in Call offset\_Jmp[Ref]**  **jmp\_SearchForLongJumps** |

*Note: when manually correcting it is necessary, while looping in this place, right in the debugger to change the command* ***JMP EIP*** *to* ***JMP \_ LongJumpIsCall [ Ref ]*** *, jump to the right place, and then return the original command to its place. On the other hand, we could assemble from the very beginning so that you don’t get stuck in this place at all, but go straight to where you’re supposed to go.*

|  |
| --- |
| **Have you reached the final?** |

If we now embed all our bits of corrective code into the executable file and run the latter, all imports will finally be fixed!

We only have to do the final touch - in the debugger using the command

: m 008CB000 L 1000 007C9000

replace the old original IAT our new “ temporary ”.

Or, if you want to automate this process, you can assemble another small piece of code:

|  |
| --- |
| **\_Copy IAT : \_**  **mov ebx , 008 CB 000; Start of temporary IAT**  **mov ecx , 007 C 9000; Beginning of the original IAT**  **\_Copy IATLoop :**  **mov eax, dword ptr [ebx] ; IN EAX we place RVA correct import**  **mov dword ptr [ ecx ], eax ; and write it in the appropriate place in the original I AT**  **add ebx , 4; Let's move on to the temporary IAT**  **add ecx , 4; Let's move on to the original IAT**  **cmp ecx , 007 CA 000; Have we already copied everything?**  **jle\_CopyIATLoop ; \_ \_ Otherwise, copy further...**  **jmp eip ; As usual...looping** |

|  |
| --- |
| **Dumping and recompilation** |

Now we have come to that significant moment when a looped process, all imports of which have been restored, hangs in memory, peacefully chewing processor resources.

Armed with, for example, PE Tools , we take a complete dump from our looped process.

Next we open in Hex Workshop , the resulting dump, go to offset 004 CA000 ( without ImageBase), select everything that lies after (below) this offset and delete it.

Then open the dump in PE Editor and kill ( Kill Section (from Header)) the last two the last sections that belonged to SD. Thus, as you might guess, we cut off sections of the tread that have become unnecessary.

*During all these executions, the looped process must continue to hang in memory!!!*

Next, open ImpRec, select our looped MaxPayne.exe from the list of processes, enter OEP and click “IAT AutoSearch”. After the message that something has been found, click “Get Imports”. Now everything that is not detected among imports is subject to rough removal...we still took special care to ensure that everything was detected. Personally, I had to just delete the last two *Thunks* , while the rest of the import was detected with a bang! Well, finally click “Fix Dump”.

The only thing left to do now is to recompile the executable file. To do this, you can use the option of the same PE Tools “Rebuild PE”.

|  |
| --- |
| **Some conclusions** |

So, what can we say about the SafeDisc protector version 2 after the successful completion of our colossal work?

Is there some kind of pattern here?

Yes, there is a scheme. Due to the fact that the program (game) is supposed to work, we will always be able to find and completely restore damaged places in the code using appropriately adjusted protector procedures.

There are no invincible defenses!

|  |
| --- |
| **Personal wishes (it is advisable to read first)** |

First of all, I would like to wish all the best to the author of this

articles to comrade Mr. Magic - for the very interesting material, and also personally for the fact that this material helped me demonstrate my knowledge of the German language in this area, which cannot but touch me. God grant that the author, having already published material about unpacking the SafeDisc protector of the third version, did not stop there.

Further, addressing the readers, among whom there will probably be newcomers to unpacking, I would like to note that unpacking the SafeDisc protector is by no means a trivial matter, but requires certain knowledge.

That is why I urgently ask:

*just rewrite the correction code below!!!*

*Read the article first and try to figure it out!!!*

Because Simply rewriting the code is the surest way, in my opinion, to fail at this job.

Well, what else... if you suddenly, after reading the same place for a long time, cannot understand its essence, know that I myself do not fully understand some of the author’s reasoning,

That’s why I tried to formulate the thought as best I could.

debugging was, is and will be a good way to find a possible error or defect in the code - don’t forget about it.

Please don't criticize me too much if the article is too long.

And if anything happens, write letters... ☺.

[diller\_inc@mail.ru](mailto:diller_inc@mail.ru)

|  |
| --- |
| **Disassembled PE Header** |

It should look something like this (at least in my case):

|  |
| --- |
| **:00000500 C705472E F200 E97C0100 mov dword ptr [00 F2 2E47], 00017CE9 ; Patch**  **:0000050A 66C7059F31F2009090 mov word ptr [00F2319F], 9090 ; For**  **:00000513 C7055932F2008B650C61 mov dword ptr [00F23259], 610C658B ; procedures**  **:0000051D C7055D32F2009D5AC300 mov dword ptr [00F2325D], 00C35A9D ; SD**  **:00000527 EB13 jmp 0000053C**  **:00000529 90 nop**  **:0000052A 90 nop**  **:0000052B 90 nop**  **:0000052C 90 nop**  **:0000052D C7055F32F20083C404C3 mov dword ptr [00F2325F], C304C483 ; Patch corrections the stack**  **:00000537 EB39 jmp 00000572**  **:00000539 90 nop**  **:0000053A 90 nop**  **:0000053B 90 nop**  **:0000053C B900907C00 mov ecx, 007C9000**  **:00000541 BB00B08C00 mov ebx, 008CB000**  **:00000546 90 nop**  **:00000547 90 nop**  **:00000548 8B11 mov edx, dword ptr [ecx]**  **:0000054A 81FA00009001 cmp edx, 01900000**  **:00000550 7C0E jl 00000560**  **:00000552 81FA0000A001 cmp edx, 01A00000**  **:00000558 7706 ja 00000560**  **:0000055A 90 nop**  **:0000055B 90 nop**  **:0000055C FFD2 call edx**  **:0000055E 90 nop**  **:0000055F 90 nop**  **:00000560 8913 mov dword ptr [ebx], edx**  **:00000562 83C104 add ecx, 00000004**  **:00000565 83C304 add ebx, 00000004**  **:00000568 81F900A07C00 cmp ecx, 007CA000**  **:0000056E 7CD8 jl 00000548**  **:00000570 EBBB jmp 0000052D**  **:00000572 B800104000 mov eax, 00401000**  **:00000577 668138FF15 cmp word ptr [eax], 15FF**  **:0000057C 740C je 0000058A**  **:0000057E 40 inc eax**  **:0000057F 3DFF8F7C00 cmp eax, 007C8FFF**  **:00000584 75F1 jne 00000577**  **:00000586 EB72 jmp 000005FA**  **:00000588 90 nop**  **:00000589 90 nop**  **:0000058A 81780200907C00 cmp dword ptr [eax+02], 007C9000**  **:00000591 7CEB jl 0000057E**  **:00000593 81780200A07C00 cmp dword ptr [eax+02], 007CA000**  **:0000059A 77E2 ja 0000057E**  **:0000059C 90 nop**  **:0000059D 90 nop**  **:0000059E 8B4802 mov ecx, dword ptr [eax+02]**  **:000005A1 8B09 mov ecx, dword ptr [ecx]**  **:000005A3 81F900009001 cmp ecx, 01900000**  **:000005A9 7CD3 jl 0000057E**  **:000005AB 81F90000A001 cmp ecx, 01A00000**  **:000005B1 77CB ja 0000057E**  **:000005B3 90 nop**  **:000005B4 90 nop**  **:000005B5 68CB054000 push 004005CB**  **:000005BA 0506000000 add eax, 00000006**  **:000005BF 50 push eax**  **:000005C0 2D06000000 sub eax, 00000006**  **:000005C5 90 nop**  **:000005C6 90 nop**  **:000005C7 FFE1 jmp ecx**  **:000005C9 90 nop**  **:000005CA 90 nop** |

...продолжение :

|  |
| --- |
| **:000005CB BB00B08C00 mov ebx, 008CB000**  **:000005D0 3913 cmp dword ptr [ebx], edx**  **:000005D2 740F je 000005E3**  **:000005D4 83C304 add ebx, 00000004**  **:000005D7 81FB00C08C00 cmp ebx, 008CC000**  **:000005DD 7CF1 jl 000005D0**  **:000005DF EBFE jmp 000005DF**  **:000005E1 90 nop**  **:000005E2 90 nop**  **:000005E3 81EB00B08C00 sub ebx, 008CB000**  **:000005E9 81C300907C00 add ebx, 007C9000**  **:000005EF 895802 mov dword ptr [eax+02], ebx**  **:000005F2 90 nop**  **:000005F3 90 nop**  **:000005F4 EB88 jmp 0000057E**  **:000005F6 90 nop**  **:000005F7 90 nop**  **:000005F8 90 nop**  **:000005F9 90 nop**  **:000005FA B800104000 mov eax, 00401000**  **:000005FF 90 nop**  **:00000600 90 nop**  **:00000601 8038E9 cmp byte ptr [eax], E9**  **:00000604 740D je 00000613**  **:00000606 40 inc eax**  **:00000607 3DFF8F7C00 cmp eax, 007C8FFF**  **:0000060C 75F3 jne 00000601**  **:0000060E 90 nop**  **:0000060F EBFE jmp 0000060F**  **:00000611 90 nop**  **:00000612 90 nop**  **:00000613 8B4801 mov ecx, dword ptr [eax+01]**  **:00000616 03C8 add ecx, eax**  **:00000618 83C105 add ecx, 00000005**  **:0000061B 81F900A08C00 cmp ecx, 008CA000**  **:00000621 7CE3 jl 00000606**  **:00000623 81F900B08C00 cmp ecx, 008CB000**  **:00000629 73DB jnb 00000606**  **:0000062B 90 nop**  **:0000062C 6835064000 push 00400635**  **:00000631 FFE1 jmp ecx**  **:00000633 90 nop**  **:00000634 90 nop**  **:00000635 BB00B08C00 mov ebx, 008CB000**  **:0000063A 90 nop**  **:0000063B 90 nop**  **:0000063C 3913 cmp dword ptr [ebx], edx**  **:0000063E 740F je 0000064F**  **:00000640 83C304 add ebx, 00000004**  **:00000643 81FB00C08C00 cmp ebx, 008CC000**  **:00000649 7CF1 jl 0000063C**  **:0000064B EBFE jmp 0000064B**  **:0000064D 90 nop**  **:0000064E 90 nop**  **:0000064F 81EB00B08C00 sub ebx, 008CB000**  **:00000655 81C300907C00 add ebx, 007C9000**  **:0000065B 90 nop**  **:0000065C 90 nop**  **:0000065D 80780590 cmp byte ptr [eax+05], 90**  **:00000661 7511 jne 00000674**  **:00000663 90 nop**  **:00000664 90 nop**  **:00000665 66C700FF15 mov word ptr [eax], 15FF**  **:0000066A 895802 mov dword ptr [eax+02], ebx**  **:0000066D 90 nop**  **:0000066E 90 nop**  **:0000066F EB95 jmp 00000606**  **:00000671 90 nop**  **:00000672 90 nop**  **:00000673 90 nop** |

…окончание :

|  |
| --- |
| **:00000674 B900104000 mov ecx, 00401000**  **:00000679 668139FF25 cmp word ptr [ecx], 25FF**  **:0000067E 740D je 0000068D**  **:00000680 41 inc ecx**  **:00000681 81F9FF8F7C00 cmp ecx, 007C8FFF**  **:00000687 7CF0 jl 00000679**  **:00000689 EBFE jmp 00000689**  **:0000068B 90 nop**  **:0000068C 90 nop**  **:0000068D 395902 cmp dword ptr [ecx+02], ebx**  **:00000690 75EE jne 00000680**  **:00000692 90 nop**  **:00000693 90 nop**  **:00000694 2BC8 sub ecx, eax**  **:00000696 83E905 sub ecx, 00000005**  **:00000699 C600E8 mov byte ptr [eax], E8**  **:0000069C 894801 mov dword ptr [eax+01], ecx**  **:0000069F E962FFFFFF jmp 00000606** |