**Unpacking SafeDisk 4.6 on example**

**Launch Sid Meier's Civilization 4**

**TOOLS :** SoftIce v4.3.2+IceExt v0.67, OllyDbg+plugun Olly advanced v1.26 beta 9,

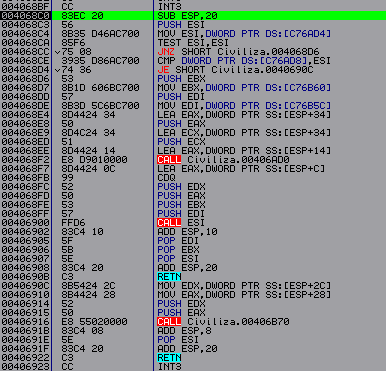
PETools v1.5, ImpREC v1.6

**VaZeR**

**vazerdim@mail.ru**

**PART IV. SDAPI v2**

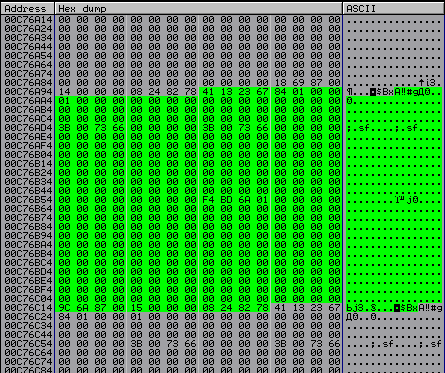
Now we're left with the hard part, SDAPI . Which are not links to some API , but independent computational functions. Which are firmly built into the program. And their correct operation is necessary for the application, since the returned values are used later in the code in other calculations. It follows from this that they can be restored only if you understand the mechanism of their operation. This means that we will need to examine a lot of code, which still contains a lot of garbage and various unnecessary calculations. But first, let's look at the SDAPI call itself :



But before we get to this code, we put two parameters on the stack. Because of which the return value depends. For example, if I put two zeros on the stack, I get the result - 1С592 D 76, then I put two ones and get a completely different result - 9 F 0 BAC 47. Only this alone can tell us that we cannot simply take and enter a strict number. And there are simply a huge number of these calls themselves in the code section. Now let's start looking at this code. At the beginning (4068C4) we put in the ESI the address where we get through CALL ESI . I always have the same number there - 6673003B (in the tread section), which means that there is only one function for generating values. You can also immediately say that in this game the debugging handlers remain intact, this is the code that begins with the address - 40690C.

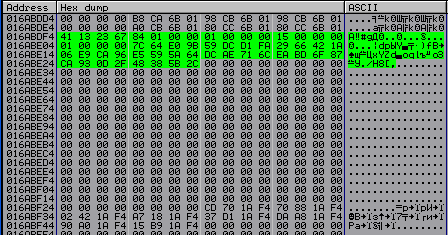
*Note: Why the protector developers even made these debugging handlers is still not clear to me; they are probably needed at the stage of implementing SDAPI into the program, although it is also not clear why. There are some versions of the protector that do not have debugging handlers. And in this case, you will have to completely rewrite the function of the protector. But it seems that new versions of the tread always have them.*

We don’t get to this address because our ESI value is 6673003 B and not zero. But forwarding conditional jumps will not help us, since the arrays necessary for the calculations have been erased. This means that we need to restore these arrays, which, by the way, are also calculated in the protector code because they play a key role in calculating the final values. Here we need to look in the dump at the address C76 AD 4, where we will see a certain structure that is located in the data section of the file ( SD\_struct ) :



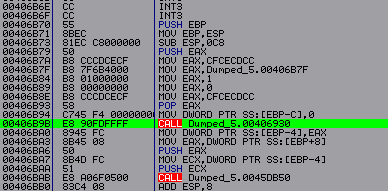
If you look at the nearest data, you can say that this is a table that starts with bytes 67231341 and ends with bytes 78822408. If you look higher or lower, we will see more of the same tables. In total I have 40 h of them . The next number that follows the beginning is 184, this is the size of this table, the third number is always 1. But the penultimate 15 is the table number, they are numbered starting from zero. It is this table that we will need to restore later.

Now let's start moving further through the code, at address 4068 DE we put another interesting number 16 ABDF 4 into EDI . When we go to the dump we will see the following:

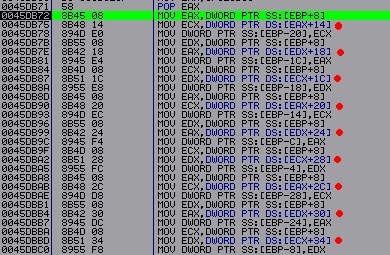


This is a structure (let's call it Initial table ) coincides with ours, also the beginning - 67231341, the same number - 15. But these are not the values that should be in our SD \_ struct . Here are the values, thanks to which we will subsequently obtain the values we need.

comparing the code in the debug handler with ours in the protector is very helpful in learning the SDAPI mechanism. For example, to understand what values the debug handler needs, you need to examine its beginning:

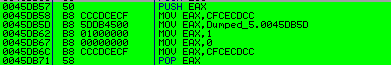


In CALL at address 406 B 9 B , we put a pointer to our SD \_ struct and in CALL – 406 BAB we read the required values from the table:



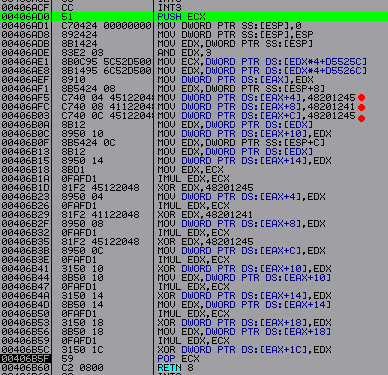
Here we are reading 9 dword from our table and this number is the same as what we have in Initial table (data after 01 15 01).

*Note: Also looking through these debugging handlers, you can come across another paradox:*

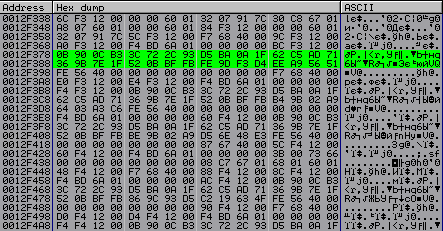


*This is just garbage, and not typical for SafeDisk (too simple). I just want to ask why he is needed here. There’s definitely no need for it in debugging these very SDAPIs .*

Now that we've sorted out our tables, let's go back to the code that prepares the data for the protector function. We still have one more interesting point here : CALL 406 AD 0. Having entered it, we will see a procedure that encrypts the data transmitted to the protector, i.e. two of our parameters that we put on the stack before entering the SDAPI procedure , two more values are also taken from the stack, those that were above the stack at the time of entering the SDAPI procedure, but they do not affect the result of the calculation, and are added only for weight:

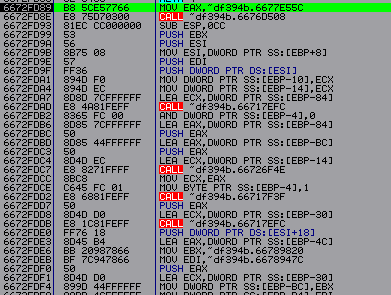


During the encryption process, two random variables are used - addresses 406 AE 1 and 406 AE 8, which depend on EDX , for example , I take the values from there - EF 00900 B , B 30 C 900 B. Below there are three more constants - 48201245, 48201241, 48201245. Thanks to all these numbers, in the end, we get another array ( CryptData ), which is located at the address EAX , the number is placed there even before entering this procedure:

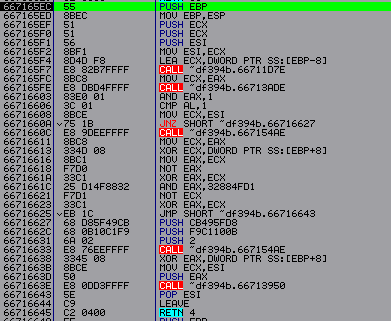


The first value in this array corresponds to our second number, which we took randomly. But the first one is overwritten. Later we will decrypt this array back.

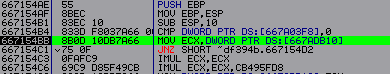
Now we need to start studying the code for the protector itself. First we have to find the main branch of this feature. It’s not difficult to find it, the first long procedure is where it is. It starts like this:



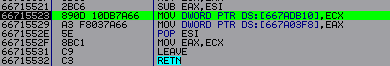
Also, before studying this procedure, it is necessary to say something about the presentation of data in the protector. Namely, that all important data is not in the desired form, but is encrypted and decrypted immediately before calculation, where the result is again encrypted. Moreover, the encryption key is generated every time. This means that all the data in the code is constantly changing, which is not very convenient for us. So let's solve this problem first. The encryption procedure is not difficult to find, since it is probably the most common one here. For example, let's enter CALL 66717 EFC , in which we will find the following procedure:



It is easily recognized by its many XOR commands . After passing CALL 667154 AE, we will have the key to encrypt the data in EAX . So let's go into it:

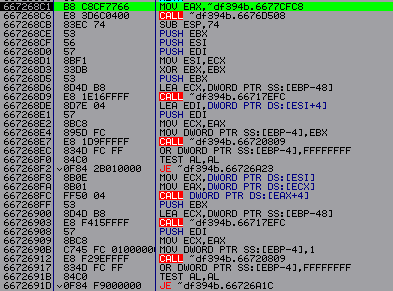


We put this very key in ECX from address 667 ADB 10. Let's try to track where the recording occurs at this address, put a hardware breakpoint at this address for writing a double word and restart the program. Let's stop here:



This is where we write down our key, so every time we start we will change the value in ECX to the same one; the number itself can be taken to be any number except zero.

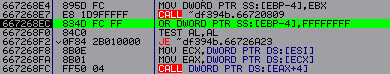
Now let's go back to studying our code. Examining the debug handler code, we can come to the conclusion that the main elements in this code are the missing SD \_ struct array . Based on this, we can assume that at the beginning of the protector code we should get this array from our Initial table because We don't have anything else that looks like the SD\_struct table . Therefore, to find such a place you just need to trace the code and see when we use the values from Initial table . Well, when we find such a place, we begin to study this code more thoroughly. So I found this place:

**

This procedure calculates the array we need. This procedure itself is located, as expected, in the second CALL in the main branch. Through which we get to another procedure, which also plays a fairly large role in the calculations and which we get to several times during the operation of SDAPI . So again, at the beginning of this procedure (third CALL ) we find ourselves in the right place. We need to analyze this procedure in more detail.

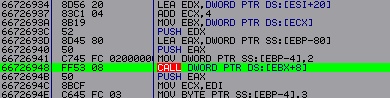
*Note: There is so much code in all of these SDAPI procedures that it can take forever to go through them all. Therefore, when studying, it is best to be guided by the principle that all the most important things are at the end of the procedures. And literally this is the last or in most cases the penultimate CALL .*

If you look at how this procedure works, you can say that the array we need is created only once. For example, if you create an array for SD \_ struct 15 once , then during subsequent calls it will not be created anew, but the already prepared one will be used. There is a corresponding check for this:

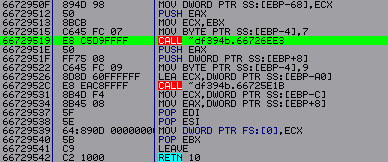


If the array has been created, then we put the number FFFFFFFF at address EBP -4 , which acts as a label. There are two such checks here.

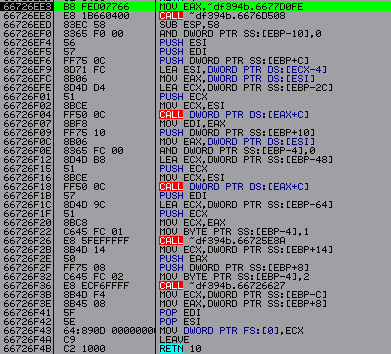
Next comes one procedure (66729431), which gives us information that in those nine double words in the array, there are actually eight values and one header, it is the very first one. And for all arrays it will be the same - 5 CAC 5 AC 5. This can be checked if you go into this procedure. The call itself, which is not strict, but using a register value:



Having entered which we will move to the end of the procedure:



Now let's go to the penultimate CALL :



This procedure is used to get each element of an array. You can look at the work of each procedure available here. But the final result is calculated in CALL 66726627. Going there, we get into the procedure, where the calculation result is obtained in the procedure - 667183 D 2:



This procedure itself can perform several different XOR , AND , OR calculations . Where the type of operation depends on the parameter being passed, in this case 24C3 E 94 D , which corresponds to XOR . By going into this procedure, we will see our three comparisons:







The end result is again obtained in the penultimate CALL . After execution, which in EAX will be the array header SD \_ struct – 5 CAC 5 AC 5.

Next, let's return to the main branch of calculating the eight values. Where there is a loop that calculates one array value in one iteration. The calculation procedures are the same as for calculating the array header. The input to the calculations itself looks like this:



This CALL leads to the same procedure 66729431, where we calculated the header values. The result turns out to be at the same address.

That's all we need to know about SDAPI . Since there are debugging handlers, we can stop there. Now we need to restore all the arrays and configure the SD\_struct correctly . To restore the arrays, I decided to write a script for OllyDbg :

|  |
| --- |
| var startscan  var startdata  var addr\_SDAPI  var order  var struct  var array  mov startscan, 401000  mov startdata, C74AC8 ; Start tables SD\_struct  find startscan,#83EC20568B35# ; Are looking for SDAPI call  mov addr\_SDAPI, $RESULT ; Let's put V variable SDAPI address  VIZOV:  cmp struct , 40; Checking if SD \_ struct is finished  je END  inc struct  add addr\_SDAPI , 20 ; Let's move on to the command that puts the Initial address in the EDI table  add startdata, C0 ; Let's move on in SD\_struct to address Initial table  mov [addr\_SDAPI],startdata ; We replace V our calling SDAPI address on Initial table  sub addr\_SDAPI, 20  sub startdata, C0  mov eip, addr\_SDAPI  add startdata,18  CICL:  bp 667269CC  run  cmp eip, 667269CC  jne ERROL ; These checks are very helpful when debugging the script.  inc order  bp 66729519  run  cmp eip, 66729519  jne ERROL  bp 66726F36  bc 66729519  run  cmp eip, 66726F36  jne ERROL  bp 6672666C  bc 66726F36  run  cmp eip, 6672666C  jne ERROL  bp 667184DE  bc 6672666C  run  cmp eip, 667184DE  jne ERROL  mov [ startdata ], eax ; How to find the correct address, put it in SD \_ struct  bc 66729519  bc 667184DE  add startdata,4  cmp order ,8 ; Checking if our values in the array have run out  je KORECTIROVKA  jmp CIKL  KORECTIROVKA:  bc 667269CC  sub startdata , 38; Return to the beginning SD \_ struct  add startdata , 184; Let's move on to the next structure  mov order,0  jmp VIZOV  ERROL:  mov eax , FFFFFFFF ; In case of an error, we will put a mark in EAX indicating that we have an error  END:  ret |

*Note: Since many calculation procedures are used quite a lot in other calculations, I therefore used the same path in the script as when manually searching for the desired value. By installing breakpoints.*

After we have restored all the values, i.e. eight double words in each structure. We still need to correctly adjust these same structures. First, we must add those first values that are missing at the beginning of the table; they can be taken from Initial table . Also add the array header - 5 CAC 5 AC 5. And finally, remove all values in the middle of the array that are no longer needed, such as the Initial address table , address of the protector procedure. To do this, I decided to write another script:

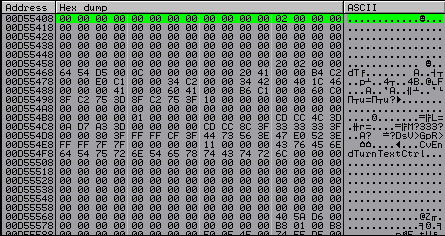
|  |
| --- |
| var startdata  var temp  var order  var array  var number  mov startdata, C74AC8  COPY:  add startdata,C0  mov array, [startdata] ; Let's put Initial table address  sub startdata,C0  CICL:  mov temp , [ massive ]; This loop copies the first values up to the header with Initial table  mov [startdata],temp  add startdata,4  add array,4  inc order  cmp order,5  je HEAD  jmp CIKL  HEAD:  mov [startdata],5CAC5AC5 ; Let's put title  mov order,0  add startdata ,24; Go to the address located after 8 dword  CLEAN:  mov [ startdata ],0; Clearing all unnecessary values in the structure  inc order  cmp order ,2 C ; I clear a block of size 2 C at once  je DALEE  add startdata,4  jmp CLEAN  DALEE:  sub startdata , E 4; Return the address to the beginning of SD\_struct  add startdata ,184; Let's move on to the next structure  inc number  cmp number,40  je END  mov order,0  jmp COPY  END:  ret |

After running these two scripts, let's copy all 64 tables to our file with the restored import. After which the game works without any problems.

But that's not all, all that remains is to find Trigger table . Which are also erased in the data section. You can find these tables using the following command:

*mov dword ptr [ esp + XX ], address in data section*

You can search through binary search (C74424) or you can write a script. After looking through the entire section, I found only one such command. Where are the zeros in the data section? There were, of course, other commands where there are also zeros in the data section, but we only need the case where there is a “known byte” nearby:



In my case, the “known byte” is 2, but it can be anything. Also below this command there should be SDAPI calls :

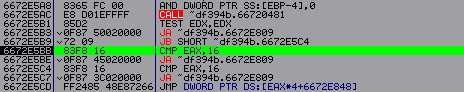


We first call SDAPI at address 410 D 70 and get the value in EAX that will be used in the next call 410 AE 0.

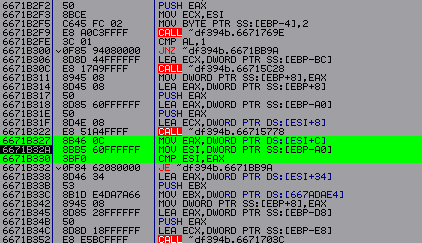
Let's now start tracing the code of this SDAPI , for now we will go only along the main branch of the function until the first switch ... case statement , which looks like this:



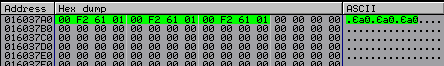
This operator determines the course of further calculations of the final values. There are two such operators here, this one is of the first order, and then there will be another of the second order. We are interested in the second option, where structures are processed. Let's move on and find the second switch... case statement:



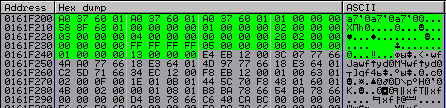
CASE number 16 is responsible for processing structural data. And we have the number 16 in EAX . This means that everything is going correctly. Now you need to find the code that processes structural data:



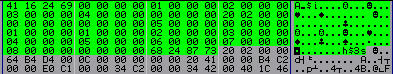
At this point in the code, we have a pointer in EAX to the addresses of structures in allocated memory:



Since I only have one structure, all three addresses are the same. Now let's go to this address and see the following:



Here the eighth double word is our “known byte”. Now our task is to convert this table to a normal form. The structure consists of a total of twenty-three double words. We skip the first four double words, instead of the number 01688 F 58 we put the beginning marker 41162469. Then we leave everything as it is, only replacing FFFFFFFF with 0. Starting from the number 13, in my case we must number eight double words, starting from one. And at the end put the end marker 73872468. In general, it should look like this:



In conclusion, we can say that SafeDisk is a very interesting protector, which is worth the time and effort to study. I would also like to especially thank **DillerInc** for its help. With its help, a fairly large number of errors and shortcomings were corrected.

