**Unpacking SafeDisk 4.6 on example**

**Launch Sid Meier's Civilization 4**



**VaZeR**

**vazerdim@mail.ru**

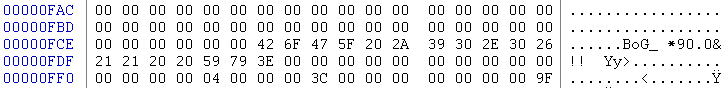
**TOOLS :** SoftIce v4.3.2+IceExt v0.67, OllyDbg+plugun Olly advanced v1.26 beta 9, PETools v1.5, ImpREC v1.6.

**PART I.​ AntiDebug , finding OEP**

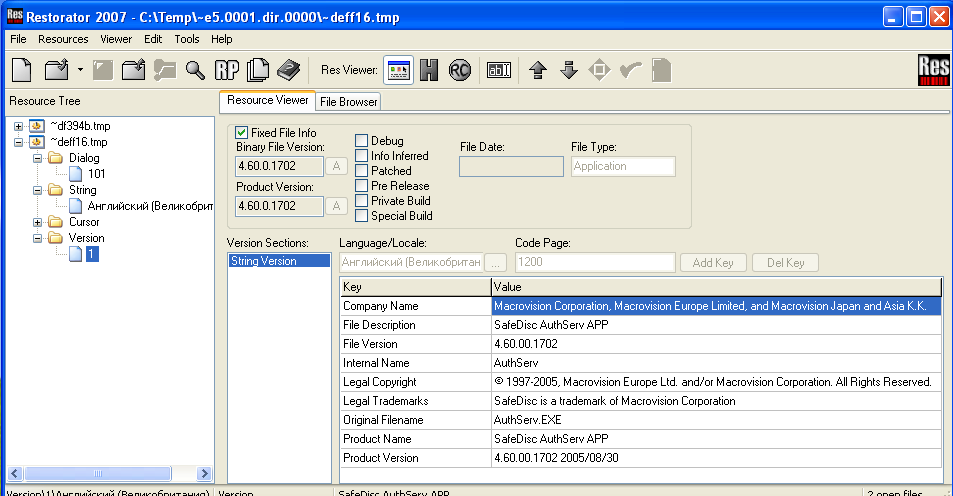
SafeDisk is a fairly common protector for games. It provides sufficient reliability, and most importantly, good application performance. For an unpacking example, I took the game CIVILIZATION IV , because it was there that I found all SafeDisk protections in one place: nanomites, corruption of imports and pointers to them, emulated opcodes, SDAPI version 2. Until this time, I only unpacked the SafeCast 2.4 version from Macrovision products, and in general we can say that protection has increased by an order of magnitude. AntiDebug is not very complicated for this protector and can be bypassed by patching a few bytes in the code, which are quite simple to find. Since the developers did not protect the verification procedures very well and did it all in one place in the program. Only a check for the presence of SoftIce is available in another place in the program, where all the code is obfuscated. But the lines that appear on the stack speak for themselves.

When running a protected application, several files are unpacked into the temporary directory ( Temp ), such as ~e5.0001 (debugger) and a folder with the corresponding name ~e5.0001.dir.0000, where there are four more files ~deff16.tmp (library involved in checking disk and decrypting the code section), ~df394b.tmp (the library in which all file protection is located), ~efe2.tmp, PfdRun.pfd.

To determine the version of the protector, you can look in the hex editor for the line in the file header section that begins with BoG , where after Yy > the version number, here I have 04 3C (4.60):

****

Sometimes this line is erased (there is no CRC integrity check ) and then you can look at the files that are in the temporary directory using Restorator. In the section about the file version you can get comprehensive information:



The file ~e5.0001 is a debugger for a protected application. Apparently, this is why OllyDbg goes into an endless loop when running a file, since the program waits for a response from its debugger. I did not study whether my assumption was correct or not. I just decided to use SoftIce , and for this I will have to deal with antidebug . And here OllyDbg can help us, as with the Olly plugin advanced 1.26 beta 9, it passes all checks and unpacks the code section. First, I found out exactly what checks SafeDisk uses ; for this, I unchecked all the checkboxes in the Olly plugin settings advanced and launched the application, a window appeared indicating that a debugger was detected:

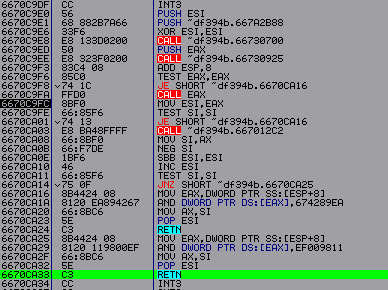


Next, we will check one box at a time and see when we pass the checks. The IsDebuggerPresent checkbox must be checked immediately, since in most cases this check is present in all protectors. After experimenting with the settings, you can understand that the application starts if the ZwQueryInformationProcess and ZwQuerySystemInformation checks are bypassed . In principle, you may not even find out what checks exist. And it will be limited only to IsDebuggerPresent . But then it will not be entirely clear why we are being discovered.

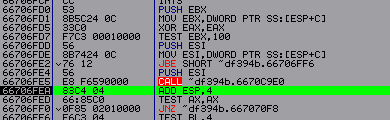
our application into OllyDbg, we need to put a breakpoint on the IsDebuggerPresent function , but we need to put the breakpoint not at the beginning, since the first byte is checked for the presence of a breakpoint. And somewhere below. Although in my case this check occurs after calling IsDebuggerPresent , this may not always be the case. Having launched the program, let’s stop at our breakpoint:



We exit this function and get to the code for checking the received data:



From this code it is clear that if there is a debugger, then the return value is AX 1, if not then 0. Again we exit this procedure and end up here:



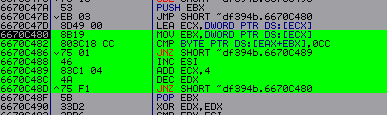
This is the main branch of the program responsible for detecting the debugger. In general we can say that it consists of CALL ... TEST ... JNZ ( JBE ). Where in CALL is there any checking. And it is these conditional transitions that determine whether we pass antidebug or not. If the debugger is noticed, then we go to address 667070 F 8.

You can explore these CALLs here . For example, at address 66707013 there is a CreateFileA check to see if [\\.\NT I CE is running](file:///\\.\NTICE) . If it is launched, then in my case 84 is returned to EAX . If not, then FFFFFFFF . At 6670702 D there is the following check:



API reference you can find out that if InfoClass = 7, then we get information about the presence of a debugger. After execution, a number will be located at address ESP + C , if it is 0 then there is no debugger, if FFFFFFFF then it is present.

At address 6670708 B , the API ( kernel 32. dll ) is checked for the presence of a breakpoint:



At address 667070 F 0 there is also a check for the presence of a debugger:

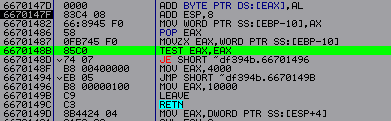


While examining these CALLs, you can also come across a code execution time check:

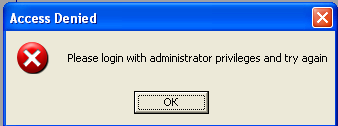


If the number in EAX is greater than 190, then our presence has been detected. Therefore, it is necessary to monitor such checks. Next, we’ll install a breakpoint at the very beginning of our branch (66706 FD 0) – Hardware , on execution .

Now let's try to launch SoftIce c IceExt (! protect ON ). And bypass all these checks. Again, open our program in OllyDbg and stop at our breakpoint. Next, we begin to trace the code, simultaneously looking at which conditional transition we go to address 667070 F 8, if we go, we go back “-” and New Origin here . We remember the address of the conditional jump, change the zero flag ( Z ) to the opposite one and move on. You can first trace this code in OllyDbg without SoftIce and with all the options in the Olly plugin advanced and remember the correct order. And understand why we are being discovered. But there is an even easier way to bypass the first check. If we exit via RET at the end of this program branch, we will end up at the following location:



By tracing this code, you can see that if we exit our procedure with zero in EAX, then we pass the first check, if there is something else there, then it’s not. Also, if we have a zero in EAX , then later we put 10000 there, and when 4000 is found, also in some versions of the protector there is also a third number 2000. You can try to put 2000 for us and look at the following message:



It turns out that the developers did not touch the local code at all, and it simply wanders unchanged from version to version. This means that in order to pass the first check you just need to put a zero at the end of the procedure and that’s it.

So we examined the first check for the presence of a debugger. Now, when launched, the window with a message about the debugger will disappear, but the application will also close, either in SoftIce or in OllyDbg (provided that SoftIce is running without it, everything is fine). This all happens due to the fact that there is another check for the presence of SoftIce . Which is carried out from ring 0. Using the SafeDisk driver SecDrv . sys (… system 32\ drivers ). This check does not display any windows, but simply quietly shuts down the application if a debugger is detected. It's harder to find her. I decided to look for it using a fairly simple method, just trace the code in OllyDbg , starting at address 6670149C. First, without entering into any procedures ( F 8). And when, having passed some CALL in which the application is terminated, remember its address and next time enter it ( F 7) and so on. Moving as if through layers, you can eventually come to a check. But you will have to trace quite a lot. Therefore, it is better to use self-tracing - Animate over ( Ctrl + F 8). The initial splash screen of the game appears and the program closes by going through CALL at 66704 A 55. We put a breakpoint on it ( Hardware , on execution ) and restart the program, when we stop, enter it. And we begin to trace the code further. Now we close at the address - 667018 F 8 => 6676142 D ( *we get to this address 18 times before the program closes* ) => after which we go inside and begin to study the code, there is a lot of code here, but most of it is just garbage . Therefore, let's clear this code of it and try to understand how it works. After examining the code, we can say that this is a loop that iterates through the following values ( *full listing APPENDIX 1 at the end of the article* ):

"SoftICE at address";

"Hooking IOAPIC";

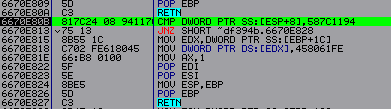
"I3HERE [ON";

"BPM, BPMB, BPMW";

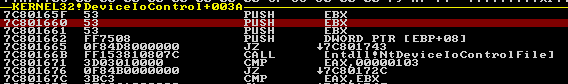
"WINICE: Int41 Get".

Moreover, it goes through several times, first 4 times each, then 2 times each, and at the end 1 time each. And it is when it iterates 1 time each after " I 3 HERE [ ON " that we close. Looking at this code ( *APPENDIX 1* ) you can understand that the check is carried out at address 6670 AEC 9. I found this address by comparing work with SoftIce and without it.

Now let's go into this CALL and see how conditional jumps work under certain conditions. And then it will be clear where we are going down the wrong path:



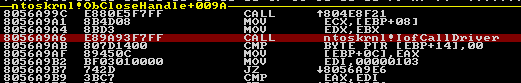
This is the same test, if the number in ESP +8 is 587С1194 then we exit. Moreover, SafeDisk terminates the application only the second time, i.e. if we have a value in ESP +8 equal to 587 C 1194 twice. This is apparently done to reduce the likelihood of false triggering of the protection. Also, if there is no debugger, then at address ESP +8 in most cases the number 587C1284 is located. Now we need to understand where this number comes from. This can be found out by tracing this loop with the values. Based on the trace, we can say that the recording occurs after the API function DeviceIoControl is executed . And this API connects our application with its driver. Now we need to examine the driver code. To understand where the value 587C1194 comes from. Here you can only use SoftIce or some other ring 0 debugger. We run our program in SoftIce , first we bypass the first check ( bpx 6670147 F , after it works we reset EAX ) and set a breakpoint at address 6670 E 80 B. When it works, let's move on to the API DeviceIoControl ( u DeviceIoControl , place a pointer to the beginning of the function and F 7). Now we are at the very beginning of the function and from here we need to start tracing the code all the way to the driver. The best way to trace the code is to go into all the procedures. So as not to miss anything interesting. First we must get into the Windows kernel , through the native call API NtDeviceIoControlFile :



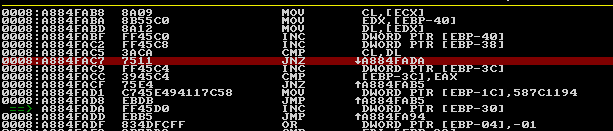
ring 0 itself looks like this:



Next, we need to find a transition to our SecDrv driver in the Windows kernel . sys :



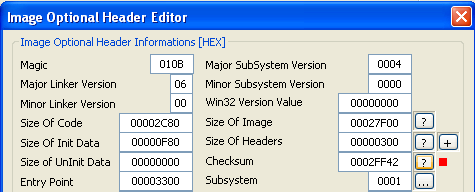
By going to this CALL , we will be taken to the code of our driver. The driver code is not that big, so we just need to trace it carefully and watch when we get the value 587С1194 or 587С1284. This way you can get to the following code:



From this we can see that the value 587С1194 is a constant in the driver.

*Note: In principle, it was possible to use a disassembler instead of a debugger and find this place using a constant, but this is not always possible; there are cases when other constants can be used, for example 58803194. They can be found in the driver in a similar way.*

And so that it is not used, the conditional jump to address A 884 FAC 7 responds. Therefore, we need to correct byte 75 to EB ( JMP ) in any hex editor . After which you must recalculate the driver checksum, otherwise Windows will not load it. This can be done using PETools :



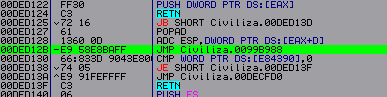
By doing this, we bypassed the check for the presence of a debugger from ring 0.

*Note: In order for the SoftIce window to look normal with the game running, without distortion, you need to set the game resolution to the same as in Windows .*

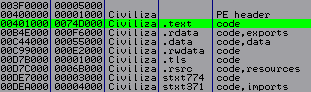
Now we can run the game freely with SoftIce even without IceExt (! Protect ON ). Which is very convenient in research.

**OER**

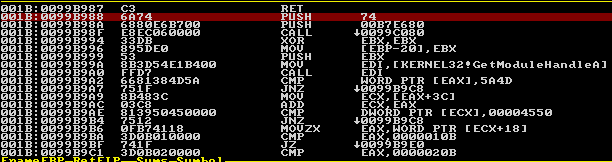
SafeDisk anti-debugging and now we can start studying the code further. Let's try to find OEP . Let's use the old search method. Open the program in the debugger and look at the transition to the program code section ( *Note: it is located below EP* ). And there is one here:



He is the only one here that leads to the code section of the program; there are no others. I found out that it leads to a code section by looking in OllyDbg ( ALT + M ) at the file sections (their addresses):



But we need to check whether we are right or wrong. Let's put a breakpoint on it in SoftIce and when we stop, we'll go to the address 99 B 988, where we see the following:



This is an OEP program written by the seventh version of C ++. Now you can dump the program; to do this you need to loop the program, i.e. instead of the first bytes 6 A 74 enter EBFE . After which they need to be restored back in the dump.

**APPLICATION 1**

|  |
| --- |
| ...  6670AE2B PUSH EDI  6670AE2C CALL ~df394b.66710920  6670AE31 ADD ESP,4  6670AE34 MOV EBX,EAX  6670AE52 TEST EBX,EBX  6670AE54 JE ~df394b.6670B008  6670AE5A MOV EAX,DWORD PTR SS:[ESP+14]  6670AE5E TEST EAX,EAX  6670AE60 MOV DWORD PTR SS:[ESP+10],0  6670AE68 JBE ~df394b.6670AF46  => 6670AF0D  6670AEB8 MOV EAX,DWORD PTR DS:[EDI]  6670AEBA MOV ECX,DWORD PTR DS:[ESI+4]  6670AEBD LEA EDX,DWORD PTR SS:[ESP+28]  6670AEC1 PUSH EDX  6670AEC2 MOV EDX,DWORD PTR DS:[ESI]  6670AEC4 PUSH EAX  6670AEC5 PUSH EBX  6670AEC6 PUSH ECX  6670AEC7 PUSH EDX  6670AEC8 PUSH EBP  6670AEC9 CALL DWORD PTR SS:[ESP+38]  6670AECD ADD ESP,18  6670AED0 TEST AX,AX  6670AED3 JE ~df394b.6670B05C  6670AEF1 CMP DWORD PTR SS:[ESP+28],458061FE  6670AEF9 JE SHORT ~df394b.6670AF15 ;прыгаем если есть отладчик  …  6670AF15 MOV EAX,DWORD PTR SS:[ESP+18]  6670AF19 INC EAX  6670AF1A MOV DWORD PTR SS:[ESP+18],EAX  6670AF32 MOV ECX,DWORD PTR SS:[ESP+2C]  6670AF36 CMP EAX,DWORD PTR DS:[ECX+C68]  6670AF3C JNB ~df394b.6670B0C6 ;завершение приложения  => 6670AF13  6670AF42 MOV ESI,DWORD PTR SS:[ESP+2C]  6670AF46 MOV EDX,DWORD PTR DS:[EDI]  6670AF48 INC EDX  6670AF49 PUSH EDX  6670AF4A PUSH EBX  6670AF4B CALL ~df394b.66710980  6670AF50 MOV EAX,DWORD PTR SS:[ESP+24]  6670AF54 MOV ECX,DWORD PTR DS:[ESI+7A8]  6670AF5A ADD ESP,8  6670AF5D INC EAX  6670AF5E ADD EDI,65  6670AF61 CMP EAX,ECX  6670AF63 MOV DWORD PTR SS:[ESP+1C],EAX  6670AF67 JB ~df394b.6670ADC3; повторяем значение  6670AF6D JBE SHORT ~df394b.6670AF76 ; на ret  {  ; We get there ships If no SoftIce  6670AEFB MOV EAX,DWORD PTR SS:[ESP+10]  6670AEFF MOV ECX,DWORD PTR SS:[ESP+14]  6670AF03 INC EAX  6670AF04 ADD ESI,8  6670AF07 CMP EAX,ECX  6670AF09 MOV DWORD PTR SS:[ESP+10],EAX  6670 AF 0 D JB ~ df 394 b .6670 AE 74 ; if repeat  6670 AF 13 JMP SHORT ~ df 394 b .6670 AF 42 ;new value  } |