**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,

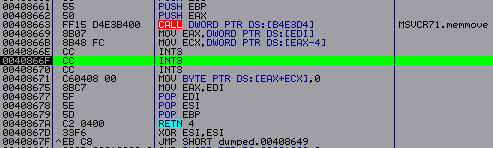
PETools v1.5 , ImpREC v1.6

**VaZeR**

**vazerdim@mail.ru**

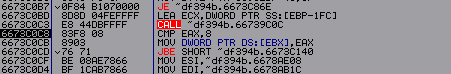
**PART II. Nanomites**

In the first part we received a dump of the game. By correcting the first two bytes to OEP . You also need to write this address in the file OEP 99 B 988-400000=59 B 988. Now let's try to run it in OllyDbg , let's stop here:



This design three INT 3 is nanomites. When executed in a protected program, the debugger ~e5.0001 receives an exception, processes it and inserts the correct mnemonics instead of INT 3. Moreover, the number of INT 3 can be not only three, but an arbitrary number.

*Note: In my case, no more than 8, here is the corresponding code (in EAX the number of INTs is 3):*

**

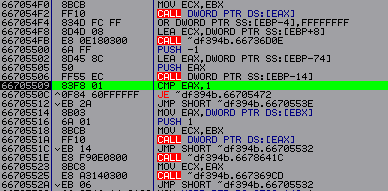
*But I'm not sure that this is always the case. Although I have never seen 8 in the code, mostly the large mass is 2 and 3.*

The very first problem that nanomites hide within themselves is their search in the code. For example, these are no longer nanomites:



This code will never receive execution - it is empty space between procedures. It should also be said that when executing nanomites, the correct mnemonics are restored, which are no longer overwritten, at least those that are processed frequently (after the second execution), since otherwise this would have a bad effect on the speed of the game.

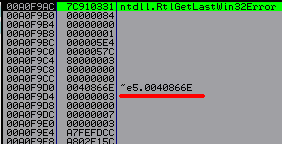
To restore nanomites, it is very convenient to use two debuggers. Now, in order for us to start studying the code responsible for the operation of nanomites, we must go to the OEP in SoftIce and loop it ( EB FE ). Minimize SoftIce ( Ctrl + D ) launch OllyDbg and access the SafeDisk debugger - ~e5.0001. The API function WaitForDebugEvent is responsible for handling interrupts ( INT 3) . (Let's go to her in OllyDbg Ctrl + G and enter the name). Let's put the breakpoint not at the beginning, but at RET and launch the application ( F 9). Now let's go back to SoftIce and restore our EB bytes FE on 6 A 74. We did this so that the program would not execute its code while we went to OllyDbg . We minimize SoftIce and we will immediately stop at our breakpoint. We exit this function and get to the following code:



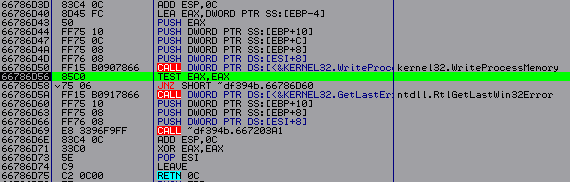
The conditional jump to address 66705509 depends on the value of EAX . If the function succeeds, the return value is 1. This is one of our breakpoints.

Now we need to talk about the method of restoring nanomites itself. In my opinion, there are only three recovery methods. The most difficult one is to find encrypted tables and decrypt them. But SafeDisk uses a rather confusing and complex algorithm. Therefore, I rejected this method immediately. The second method is to search for a conditional jump in the debugger, which is responsible for writing nanomites into the program. And if you set it up, the nanomites will be recorded immediately, and not after the second execution. Next, you need to play the game very carefully, i.e. try to use everything possible. And finally dump it. This will ensure that most of the nanomites are restored. But I note that only a part. There is a potential percentage of unrecovered nanomites, and if we add to this the difficulty of finding them, then it is better not to use this method. I chose the third method, it can be said to be a synthesis of the previous two. We examine the debugger, find all the checkpoints and redesign it so that it deciphers everything itself and writes it into the program. It turned out to be not as difficult as it seemed to me at first studying it. But, having examined the debugger, I had the idea that if, instead of WaitForDebugEvent, I entered my own code that would supply the addresses of the entire section, i.e. from 401000 to B 4 DFFF (in my case). And then I wondered if there is a conditional transition that is responsible for the “correctness of the nanomite.” And one was found. Which helps us a lot. Although if you just think logically, there should be one that, if the address does not match the base, would lead to an exit or window with an error. Now we need to implement this whole idea.

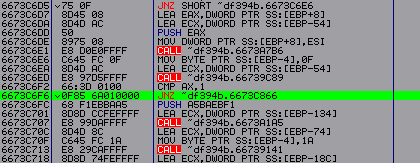
Standing at the same address and looking at the stack, we see that the nanomite address is written at ESP +24 (in my case):



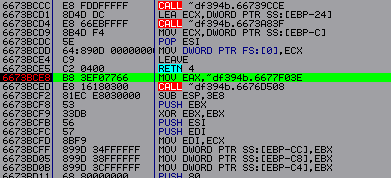
Now let’s put a breakpoint on the WriteProcessMemory API function ; it’s not far from it that the checks we’re interested in take place. But before that, let’s put another breakpoint on our comparison at the address 66705509, it will still be useful to us, since in order to understand all this work we will have to submit different values quite a few times, both correct and not. Having launched the program and stopped at our breakpoint, we exit the function ( Ctrl + F 9). Next we get to the code that checks whether the function completed successfully:



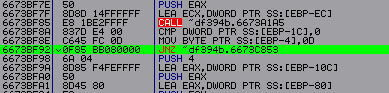
Let's leave here again and get to the program branch, where all the interesting things happen. The transition we are interested in, which is responsible for recording the nanomite from the second time, should be located somewhere above the place from where we left. We start looking and find it:



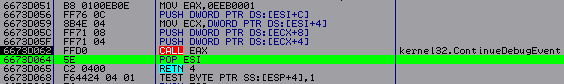
To check whether we are right or wrong, we can put a breakpoint on it and look at its work. If the nanomite was executed once, then AX is 0, the second is 1. Here is another checkpoint. It would also be a good idea to remember the addresses of the real nanomites that appear after calling WaitForDebugEvent , since we will still need them to test our code. By exploring this rather large procedure, you can find another rather interesting transition that determines whether the nanomite is “correct” or not. It is located even higher than the comparison 6673С6 F 6. It is not difficult to find, you need to put a breakpoint at the very beginning of the procedure:



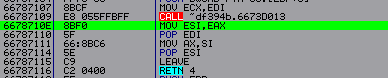
But before that, you need to change the value in ESP +24 ( *Note: this must be done immediately after calling WaitForDebugEvent* ) to the wrong one, i.e. to the address where there is no nanomite in the program. And after we have entered the “left” address, we begin to trace the code in the same procedure from the very beginning and depending on where we go through the nanomite record into the program. For me this is a transition at address 6673 BF 92. It is this that determines the “correctness of the nanomite”:



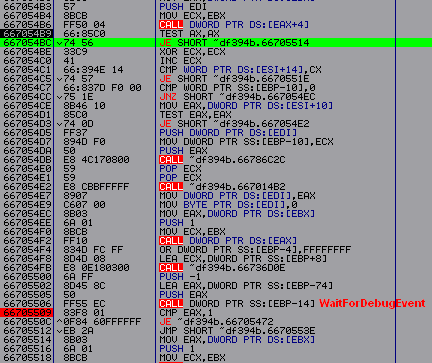
There is also one more conditional transition that terminates our application. This is due to the API call to the ContinueDebugEvent function ( *Note: ContinueDebugEvent continues the work of the main program. And since we will no longer use WaitForDebugEvent , we have nothing to continue* ). If we process it, the return value is 0, but should be 1 if successful. This is exactly why we are going out. Therefore, we’ll put a breakpoint on it and when it works, we exit the function and get to the following code:



There is no verification here, but it will be a little further. We exit through RET and get here:



Again we exit via RET and here we have a check:

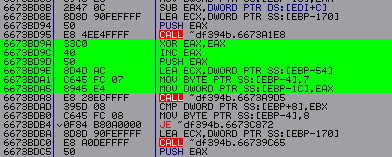


If this conditional jump fires (667054 BC ), then we jump through WaitForDebugEvent , which shouldn't happen. Therefore, we need to implement this conditional transition or the ContinueDebugEvent call itself with all its PUSH , not forgetting to set EAX to one, which is best. Now let's look at the location of the API call to the WaitForDebugEvent function . If the code at the address 66705514 does not receive execution, then we can directly enter our patch without fear of lack of space.

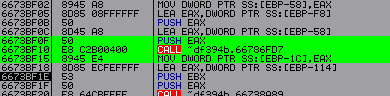
Let's write a fairly simple and understandable code :

|  |
| --- |
| 66705500 MOV EAX ,1  66705505 CMP DWORD PTR SS :[ ESP +24],0 B 4 DFFF ;comparing whether the section has ended  6670550 D JE SHORT ~ df 394 b .66705518; if yes then move on  6670550 F INC DWORD PTR SS :[ ESP +24];if not, then increase the address by one  66705513 JMP ~df394b.66705472 ; let's move on to executing the code  66705518 NOP ;we move on if the section is over. Therefore, the vessels need to set a breakpoint. |

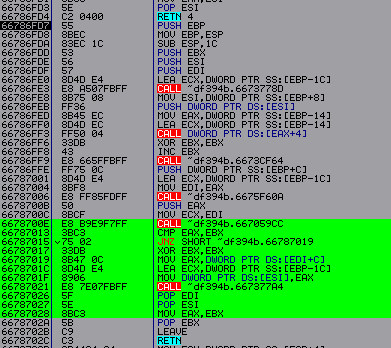
Before running our code, we need to change the value in ESP +24 to 401000, i.e. the very beginning. At first I simply traced manually ( F 8) and watched how the nanomites were restored, everything was fine. But when I ran the program ( F 9) removing all breakpoints except the address 66705518. And after our code ran through the entire section in search of nanomites. I switched to SoftIce ( addr civilization 4) and looked at the code section and saw that only a few nanomites that were at the very beginning were restored. But the rest don't. First of all, I thought that there was some kind of check for the number of nanomites, but everything turned out to be much simpler. Based on the fact that there is a conditional transition that is responsible for writing nanomites the second time. All these addresses must be stored somewhere, but as I later found out, all the values, even the wrong ones, are written there. And it is precisely because we start submitting 1000 addresses that some kind of inconsistency occurs. Because even if you now enter the correct address of the nanomite, it identifies it as incorrect. The previously found transition at address 6673 BF 92 is responsible for this . Let's restart the game and look at the comparison 6673 BF 8 A. If EBP -1 C - 00A0F498 is zero, then we go to restore the nanomite; if it is one, then no. We again set the breakpoint at the very beginning of this procedure - 6673 BCE 8. And when it works, look where the entry is at address 00A0F498. The unit is always written first :

**

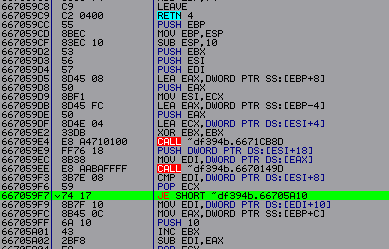
My nanomite address was correct, so we trace further and look at the dump, where a zero is recorded:



After passing CALL 6673 BF 10 to EAX zero, where it is further recorded. Now we need to study the procedure 6673 BF 10. Where at the very end is the decision about writing one or zero:



If the conditional jump to address 66787015 fails, then the EAX return value is zero. If the nanomite is correct, then in EAX and EBX there is a unit at 66787013. We now need to investigate CALL - 6678700 E. Having entered it, let's look at the work of conditional transitions :



Here you can see that if the conditional transition 667059 F 7 does not work, then the nanomite is correct. First, EBX is increased by one - 66705 A 01, and then EBX is added to EAX . Now let's look at the transition condition - 667059 F 3. If EDI is not equal to ESI +8, then the transition does not fire. It is very important to look at the values themselves:



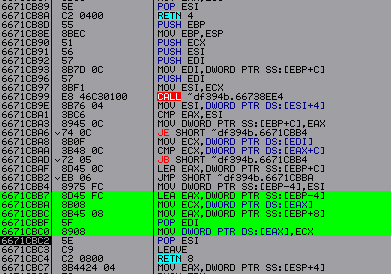
A 2 XXXX here . By going to the dump on any of these two, you can see that there are some tables there. This is the place where we write down our addresses, we can also say that it is there that the encrypted nanomites are located, but we will find out this a little later. Now putting a breakpoint at the address 667059 F 3. I changed the address of the real nanomite to any other incorrect one. And looked at the new values :



This means that the value in ESI +8 is constant, and only EDI changes . And in EDI the value is located at 667059 EC . Let's put a breakpoint on it and see what kind of address we have in EAX :



Now let’s find the place where the recording occurs at the address A 0 F 068. And it occurs higher in CALL - 667059 E 4. Now let’s try to study it too :



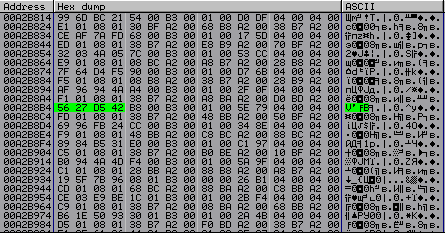
A write to address A 0 F 068 occurs at 6671 CBC 0 from address EAX – 6671 CBBA . Let's see what this address is, by setting the breakpoint to 6671 CBBA and running the program, stopping, we will see the following:



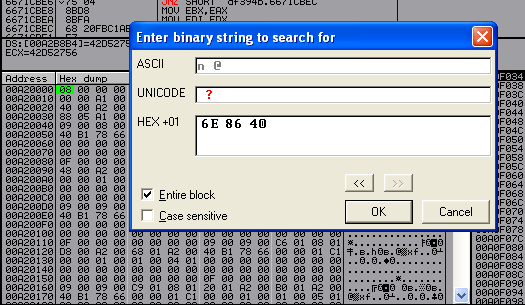
The recording comes from the address A 0 F 03 C , let's see where the data is written to A 0 F 03 C from . And they are written there at address 6671 CBB 4. Having looked at the operation of this code, we can say that if the conditional jump to address 6671 CBAD is triggered, then we have the wrong nanomite address, if not, then the correct address. Let's look at the condition - 6671C BAA:



This is provided that the nanomite address is correct, this is where our encrypted nanomite address is compared with the base. Let's go to the dump address A 2 B 8 B 4 ( d 0 A 2 B 8 B 4). And we will see the following:



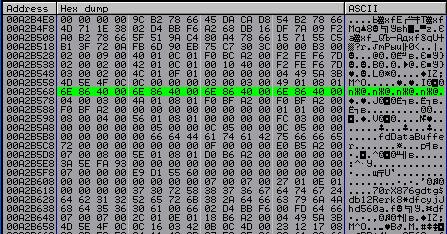
Here we have a table where 32 bytes of encrypted data are allocated for each nanomite. Now, to find out where our addresses are cached, let’s do a binary search. By placing a pointer at the very beginning of the section and Ctrl + B :



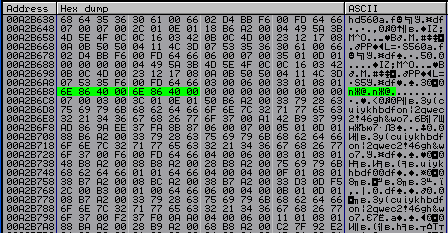
Where we will enter any address that we submitted while standing on WaitForDebugEvent , for example, I took 40866 E. First we find this place:



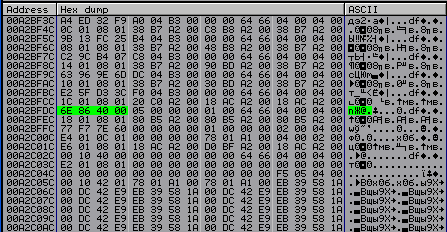
There is nothing interesting here for us, here we simply write down the current address of the nanomite. We search further ( Ctrl + L ):



This is already interesting, from here you can see that I ran the code four times with the same address 40866 E . We continue to search:



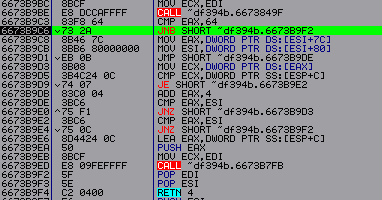
For some reason there are only two addresses here. Let's search further:



We write here only when we submit to different addresses. This is exactly what we need to fix. Since, by correcting this, we will correct all the others before writing to the section. Now we just have to find the place in the program where these values are recorded. To do this, we again set the breakpoint at the very beginning of the procedure at the address - 6673 BCE 8. We have used it more than once during the study process.

*Note: I did not use a breakpoint for recording because there were too many unnecessary triggers. It will be faster to find by tracing.*

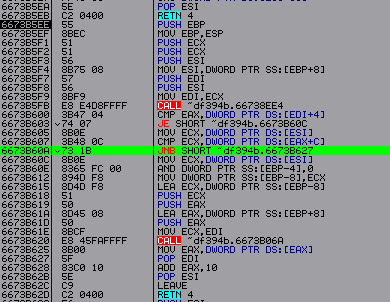
I decided to first find the place where the recording comes from at A 2 BFDC . To do this, you need to specify a different address in WaitForDebugEvent . And when we stop at our breakpoint. Start tracing the code, simultaneously looking at the dump to see if a new value is written. If this happens, then set the breakpoint to the corresponding CALL and enter it next time, not forgetting to change the nanomite address beforehand. My first address is 6673 C 858; when you go to it, you will see the following:



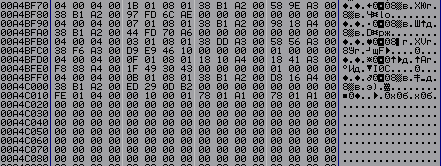
If you look at how this code works, you can conclude that if you replace JNB at the address with JMP, then there will be no entry in the section. We exit this procedure and begin to trace the code further, looking again at the dump entry at address A 2 BFDC .

*Note: The recording does not go directly to this address, but each time a little lower. But it's easy to notice.*

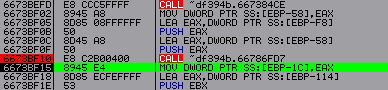
The next address is 6673С894. Having entered which we will see the following:



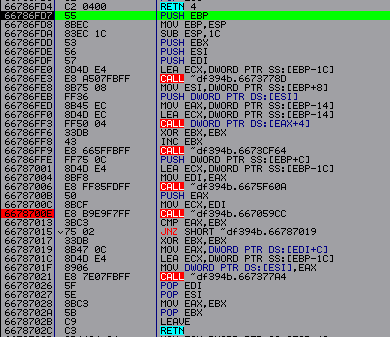
We see a similar situation if we replace the conditional jump at address 6673 B 603 with NOP , and at address 6673 B 60 A with JMP . Then we stop writing our addresses in the section. Now let's try to completely patch the debugger, and run it and check if we did everything correctly. It turns out that this is not all, since the nanomites have not been restored. Let's go to the dump to our encrypted tables and see what's wrong again. Scroll to the very end and see the following:



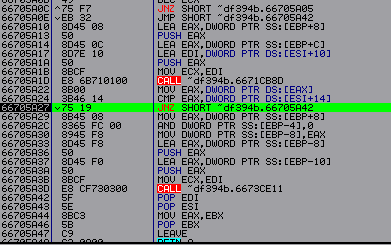
These values are not in the initial table. After examining the procedure again at 6673 BCE 8. I found the following place in the program where the recording comes from:



After passing CALL – 6673 BF 10, where the program writes the code that is not needed for us. Let's go into it:



There is nothing interesting here, so we move on to CALL at 6678700 E :



Here the entry occurs in CALL 66705 A 3 D and if we forward the conditional jump 66705 A 27 c JNZ to JMP, then everything should be fine. We restart the program, instead of restoring the original bytes on the OEP , enter CC CC . To subsequently take a dump on OEP . How to stop in OllyDbg at the output of WaitForDebugEvent . We patch the debugger and run it.

**CONCLUSION – To restore nanomites we must do the following:**

1. Instead of WaitForDebugEvent, enter our patch.
2. Call ContinueDebugEvent with all PUSH and enter MOV instead EAX , 1.
3. Fix the transition responsible for recording the nanomite from the second time - 6673С6 F 6.
4. Forward conditional transitions responsible for caching addresses - 6673 B 603, 6673 B 60 A , 66705 A 27, 6673 B 9 C 6.

After the code is completed, we check in SoftIce and see that all nanomites have been restored. Recovering the first two bytes on the OEP CC CC – 6 A 74. And dump it into PETools or any other program.