**Unpacking SafeDisk 4.6 on example**

**Launch Sid Meier's Civilization 4**

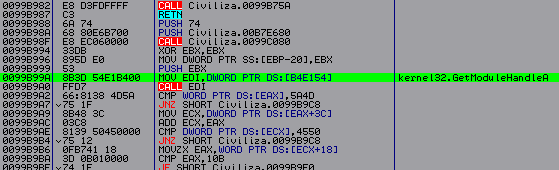
**TOOLS :** SoftIce v4.3.2+IceExt v0.67, OllyDbg+plugun Olly advanced v1.26 beta 9, PETools v1.5, ImpREC v1.6.

**VaZeR**

**vazerdim@mail.ru**

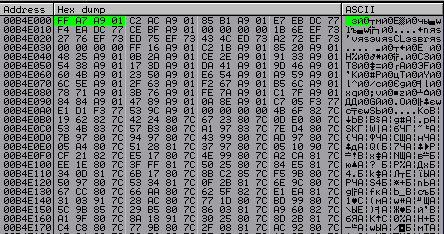
**PART III. Import recovery and emulated opcodes**

Now we need to restore the file import. There is basically nothing new here compared to the 2.XX version . If you look at our dump, you can see that there are normal calls, as well as those that go through the SafeDisk adapter . To restore the import, it will be enough to use only OllyDbg . First, let's find the dimensions of the import table. By launching our protected file in OllyDbg . And waiting for the game to go into an endless loop. Let's move on to the OEP and look at some API function calls:



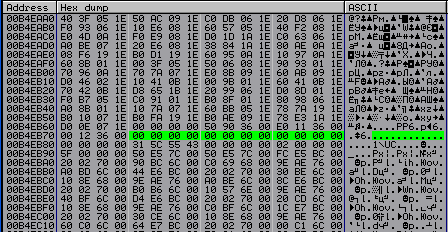
The very first API is GetModuleHandleA . Let's pay attention to the address B 4 E 154. Let's go to it in the dump window and start looking at where this table begins and ends.

Start:



Addresses starting with 01 A 9 XXXX lead directly to these adapters, so we also need to take them into account when determining the size. You can find out that this is a worn-out import by simply taking some address and going to it to see where we ended up in the application library or in the selected section. By looking at the memory map in OllyDbg .

End:

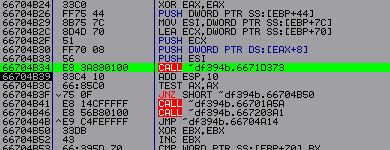


The end of the table can be easily found, since there are more than one dword of zeros there. It turns out that:

Start – B 4 E 000

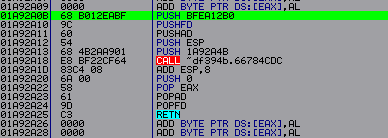
End – B 4 EB 74

Now, to start learning the import, we need to stop in OllyDbg before it freezes, but also after the code section is decrypted. To find such a place, I started tracing the program, starting from EP and looking at the dump ( d 401000) when it was decrypted. If at the beginning we, while going through some kind of CALL , decrypted a section and at the same time froze, then we put a breakpoint at this address - Hardware , on execution . And next time we go into it. There's not much to trace there. I chose the following location:

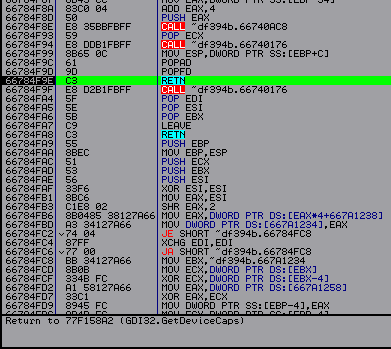


After passing CALL – 66704 B 34. The entire section of the code is decrypted. This is where we’ll put our breakpoint – 66704 B 39.

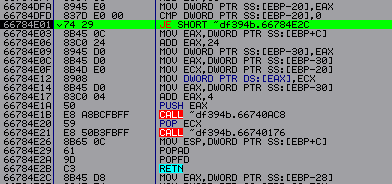
Now let's look at our import table and select some value that does not lead to more than one library. Next, let's go to this address (I took 1 A92A0B ):



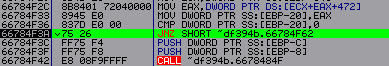
Here we have two parameters transmitted to CALL - 66784C DC : BFEA 12 B 0 (always starts with BF ) and 1 A 92 A 4 B (address 1 A 92 A 0 B + 40 h ). Let's put eip at the address 01 A 92 A 0 B ( New origin here ). And let's start tracing the code by going to CALL , the code starting from address 1 A 92 A 1 D will not receive execution. We trace to the exit from this CALL :



Here this RET leads to the GDI 32 API function - GetDeviceCaps . From this we can conclude that we need to write a code that would find all the incorrect API addresses , go to it and reach the exit, as we have done now, and replace the incorrect address in the table with the correct one. But since the import pointers in the code section are also incorrect, we need to create a temporary import table. We will place it in section stxt 371. But we will not touch section stxt 774, since LongJMP leads there (we will find out this a little later). Our code can be placed in the header section of the file. I wrote the code itself on FASM , and then after compilation I inserted it into the game itself, simply by opening our compiled file in another OllyDbg and copying all the code into the header section of our protected application, after which we save our changes. Also tracing the code of this CALL , you can notice two checks for already restored API addresses :



This is always a comparison with zero. Here we need to change JE to JMP . There is also another similar check:



Here we have to change JNZ to NOP . This needs to be done for two reasons: so that we always end up at the same place 66784 F 9 E at the output, and also so that the API address is recalculated each time .

First, let's write the code that would create a temporary import table:

|  |
| --- |
| . code  start :  mov eax ,0 B 4 E 000 h ;Place the beginning of the import table in EAX  mov byte [66784 E 01],0 EBh ;  mov word [66784 F 3 A ],9090 h ;Patching our checks  mov byte[66784F99], 0E9h  mov dword [66784 F 9 A ], 99 C 7 B 4 DAh ; Let's switch to our code  main:  cmp eax , 0b4EB74h; Let's check if the import table has ended?  je exit ; If so, let's go out  mov ecx , dword [ eax ] ; Put the API address in ECX  cmp ecx ,1 A 90000 h ;Check whether the import address is erased  jbe add \_ API ; Let's move on if not  cmp ecx,1AB9FFFh  ja add\_API  jmp ecx ; In ECX there is a worn-out import address to which we go  @mov​ esp , dword [ ebp + C ]; Here we restore worn-out mnemonics  popad  popfd  mov ecx, dword[esp] ; Let's put to ECX API address  add esp, 4 ; Stack correction  jmp add\_API​​  add \_ API : ;We proceed to the ship if the address is normal  add eax ,29 C 000 h ;Add correction to EAX  mov dword [ eax ], ecx ; We put the address in the temporary import table  sub eax ,29 C 000 h ;We put EAX in order  add eax,4 ;  jmp main  exit:  int3 ; After finishing the code we will stop here  .end start |

*Note: First we patch all the checks, and also make a transition to the address that is marked @, we will recognize it already when we insert the code into the header section, as well as bytes 99 C 7 B 4 DA . In my case it looks like this:*

**

*This is the same end only instead of mov commands esp , dword [ ebp + C ], popad , popfd , we inserted JMP to address 400478 (in my case). Of course, we can insert it manually, but we will still have to change its address, and this will not be very convenient later.*

*The addresses 1 A 90000 and 1 AB 9 FFF are the section sizes where our incorrect import values point.*

*Since I decided to place the temporary import table in the section* *stxt 371, starting from the address DEA 000. Then I got the following amendment: DEA 000 – B 4 E 000 = 29 C 000.*

Now we need to fix the import links (let's start with the most common ones, which start with FF 15) since some of them are incorrect. For example, let's take:

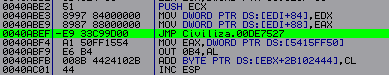


By calling the API from the address 4088 F 3, as well as from some other address the same pointer - 01 A 9 FEBB , we will receive different API addresses , this is due to the fact that there are only three parameters that we pass to the protector, and just the third is the return address. When we created the temporary import table, we did not use it at all. But in order to restore all import links, we need to go to our worn-out import addresses with the return address we need on the stack. After we find the API address again , we must find it in the temporary import table and, if necessary, change the API link . Just not on our temporary table, but taking into account the amendment to the real one. Now let's write the corresponding code:

|  |
| --- |
| start:  mov dword[66784F9Ah],99C7B453h ; Let's change JMP address at @  mov eax,401000h  main:  cmp word [ eax ],15 FFh ; We look for opcode 15 FF in the code section  je proverka\_FF15  inc eax  cmp eax, 0B4DFFFh  je exit  jmp main  proverka\_FF15:  add eax,2  mov ecx, dword[eax]  cmp ecx ,0 B 4 E 000 h ; Filtering out values we don’t need  jl main  cmp ecx,0B4EB74h  ja main  mov edx,[ecx]  cmp edx,1A90000h  jl main  cmp edx,1AB9FFFh  ja main  sub eax,2  jmp eax  @ mov esp, dword[ebp+0Ch]  popad  popfd  mov ecx,0DEA000h  search :  mov edx , dword [ ecx ]; We look for the received API address in the temporary import table  cmp dword[esp],edx  je write  add ecx,4  cmp ecx,0DEAB74h  je errol  jmp search  write:  sub ecx ,29 C 000 h ; In this block we write down the correct link  add eax,2  mov dword[eax],ecx  add eax,4  jmp main  errol:  int3 ; In principle, we should not get into the ships  exit:  int3  .end start |

*Note: The @ sign is just a mark, it was entered by me to understand which address we should go to.*

Having fixed all CALL [ API Link ], let's start fixing LongJMP . They are the following:



This is the JMP that leads into the tread section of the stxt 774. If you look below this JMP you can see a strange command: *mov EAX , dword [5415 FF 50]* . This is due to one garbage byte A 1. If we enter NOP instead , we will see the following:



From this we can say that this JMP should eventually be converted to the following:



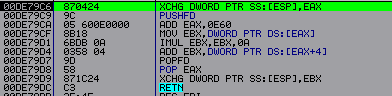
But we also need to take into account that JMP may take 5 bytes, not 6. Then it must be converted to the form:

|  |
| --- |
| CALL address  ...  JMP Address [ API ] |

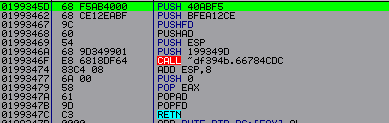
In this case, having learned the API address , we must find the corresponding JMP [ API ] in the code section. And put a CALL with the address on it. But there was no JMP in my program with 5 bytes. We will determine the number of bytes by the return address. Now let's see what is located at DE 7527:



We go to CALL at DE 7527:



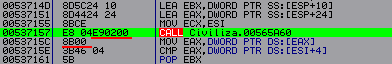
We exit through RET and get here:



First, we put the return address - 40 ABF 5. And then we get to the familiar procedure for obtaining an API address . It turns out that we have to write code that would look for all JMPs in the tread section. Next, we would find the API address , and also compare the return address with 5 and 6. If it does not correspond to these numbers, then we stop and figure it out manually. Well, when everything is fine, then we change our LongJMP accordingly. But there is one difficulty with finding these same JMPs . We must first find byte E 9. Next, check where the address points. The address itself is calculated as follows:

|  |
| --- |
| ***JMP address itself + mixing (the number after byte E 9) + 5***  *Example: 40 ABEF + 9DC933 + 5 = DE7527* |

After we check the address if it points to our tread section. We still can't be sure that we have a valid LongJMP . For example, you can take this piece of code:



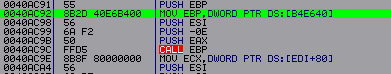
If we count the allocated bytes, it turns out that we have JMP DE 7160. And this address exactly fits our description. Therefore, in order not to freeze the program again, we conduct another check of the first two bytes at our address ( E 8 53). And only if this check was successful, then we will begin to find the API .

Now let’s implement all of the above in code:

|  |
| --- |
| .code  start:  mov dword[66784F9Ah],99С7B54Dh ; Корректируем наш JMP на @  mov eax,401000h  main:  cmp byte[eax],0E9h  je proverka  inc eax  cmp eax,0B4DFFFh  je exit  jmp main  proverka:  inc eax  mov ecx, dword[eax]  add ecx, eax  add ecx , 4 ;Increase by 4, not 5. Because already increased EAX by 1 before  cmp ecx,0DE7000h  jl main  cmp ecx,0DEA000h  ja main  cmp word [ ecx ],0 E 853 h ; Our additional check  jnz main  dec eax  jmp eax  @ mov esp, dword[ebp+0Ch]  popad  popfd  mov edx , dword [ esp +4]; This address is the return address  sub edx, eax  cmp edx.5  je JMP\_5 ; Let's move on if we have 5 bytes  cmp edx,6  jnz errol ; if the number of bytes is different then go to the output  mov word[eax],15FFh ; enter CALL  add eax,2  mov edx,0DEA000h  mov ebx,dword[esp]  poick\_iat:  cmp dword [ edx ], ebx ; We look for a suitable address in the temporary import table  je write  add edx,4  cmp edx,0DEAB74h  je errol ;  jmp poick\_iat  write:  sub edx,29C000h  mov dword [ eax ], edx ; Change to the correct import link  add esp,8 ; Stack correction  jmp main  errol:  int3  JMP\_5:  int3 ; If you got here, then the code contains five-byte LongJMPs and you will have to add additional code  exit:  int3  .end start |

*Note: I did not give the code that converts five byte LongJMPs , since I don’t have them, which means I can omit something in the implementation.*

But we haven’t looked at all API calls yet ; in my case, a call through registers is also used:

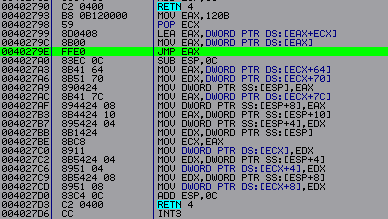
**

In EBP, the API address is first entered , and then with the CALL command EBP is what it is called. Not only the EBP register can be used , but also others. First, we need to write code that would find these places. First, we will look for the address of the erased API in the code section. When we find it, we'll check whether it fits into the register. If yes, then we start looking below *the CALL register* . Well, then we again find the API address , as we have already done. And we adjust the team - *mov register, dword [address].*

Here's the relevant code:

|  |
| --- |
| .code  start:  mov dword [66784 F 9 Ah ],99 C 7 B 67 Ah ; Change the address to a new @  mov eax,401000h  poick\_1:  inc eax  mov edx, dword[eax]  cmp eax,0B4DFFFh  je exit  cmp edx,0B4E000h ; Filtering values  jl poick\_1  cmp edx,0B4EB74h  ja poick\_1  mov ecx,0B4DFFCh  poick\_2:  add ecx,4  cmp ecx,0B4EB74h  je poick\_1  mov ebx, dword[ecx]  cmp ebx ,1 A 90000 h ; We find the mangled import  jl poick\_2  cmp ebx,1AB9FFFh  ja poick\_2  cmp edx , ebx ; We compare our value with imports  je poick\_3  jmp poick\_2  poick\_3:  sub eax,2  cmp word [ eax ],15 FFh ; Let's check if this is a CALL ?  jnz poick\_4  add eax,6  jmp poick\_1  poick\_4 :  cmp word [ eax ],2 D 8 Bh ; Checking which register is used - EBP  je EBP\_reg  cmp word[eax],0D8Bh ; ECX  je ECX\_reg  cmp word[eax],158Bh ; EDX  je EDX\_reg  cmp word[eax],1D8Bh ; EBX  je EBX\_reg  cmp word[eax],358Bh ; ESI  je ESI\_reg  cmp word[eax],3D8Bh ; EDI  je EDI\_reg  inc eax  cmp byte[eax],0A1h ; EAX  je EAX\_reg  add eax,4  jmp poick \_1 ; If there is something else there, let's look further.  EBP\_reg:  mov edx ,0 D 5 FFh ; Set the corresponding opcode bytes  jmp poick\_6  ECX\_reg:  mov edx,0D1FFh  jmp poick\_6  EDX\_reg:  mov edx,0D2FFh  jmp poick\_6  EBX\_reg:  mov edx,0D3FFh  jmp poick\_6  ESI\_reg:  mov edx,0D6FFh  jmp poick\_6  EDI\_reg:  mov edx,0D7FFh  jmp poick\_6  EAX\_reg:  mov edx,0D0FFh  dec eax  jmp poick\_6  poick\_6:  mov esi,eax  add esi,6  poick\_7:  cmp word [ esi ], DX ; I'm looking for the CALL register command  is opr\_API  inc you are  cmp esi,0B4DFFFh  is errol  jmp poick\_7  opr\_API:  add esi,2  push you  jmp ebx  @ mov esp, dword[ebp+0Ch]  attack  popfd  mov edx,0DEA000h  poick\_iat:  mov esi , dword [ esp ]; We look for the address in the temporary import table  cmp dword[edx],esi  je write  cmp edx,0DEAB74h  je errol  add edx,4  jmp poick\_iat  write:  add eax,2  sub edx,29C000h  mov dword[eax],edx ; we adjust API call  add eax,4  add esp,8  jmp poick\_1  errol:  int3  exit:  int3  .end start |

Now all we have to do is fix the emulated opcodes. For example, here is one of the calls:



By JMP command EAX we'll move on to the tread section. The code from address 4027 A 0 will never be executed. To get to such a place, it is better to use a binary search by signature ( FF E 0). Now, to understand how this defense mechanism works, we need to investigate it. We can't just put eip on address 402793 and start tracing. We need to find the CALL with which we get here. We put the pointer to 402793 and Find references to => Selected command .

There is only one CALL:



Let's move on to it:

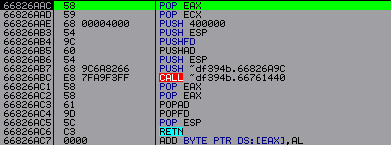


Next, we place a pointer to address 4010 A 9 and again look for CALL , which refers to this address. And here we will get about a hundred links. And they all lead to the same JMP EAX .

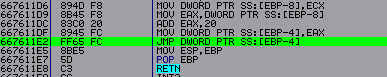
Let's go to one of these addresses:



Here we see a regular CALL XXXXXXXX . We put eip on it and start going deeper into the tread, through JMP EAX . First we get to the following code, which is similar to the one that appears before the import restoration code:



Let's start tracing the tread code. Until we reach the exit ( *Note: It is best to trace the code through F 7, except for the API calls themselves* ). By the way, we also exit via the stack in a unique way:



...



From where we get to the code section:



It turns out that we ended up on JMP [ API ]. Now if we look back at the address 4011 F 8 we will see the following:



Instead of the address 4010 A 9, it changed to the correct one. This means that we need to write code that would find all calls CALL 4010 A 9, entering each one, and inside the protector procedures we need to find a place where we would put *the JMP address on our patch* . But before that, we must also remember the current value of the ESP register , this must be done while we are still standing at address 99 B 850, for subsequent adjustments. I have it there - 12AB0C.

I decided to put it here:



If we stand on this CALL and look at the CALL from where we entered, we will see that it has already been restored. But there can be many different options here. I also need to say right away that I have only one JMP command EAX in the program. And this may not always be the case. Therefore, it is better to check the entire section carefully. Well, you also cannot be 100% sure that the call occurs only through the EAX register , and not through some other one. This means you need to check calls through other registers. We can also say that CALL does not always simply change only the address to JMP [ API ], there may be such a case:

Before:



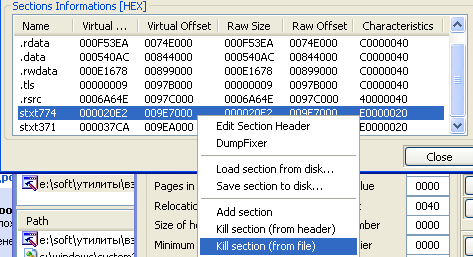
After:



Having looked at how this code works, let’s now write ours:

|  |
| --- |
| .code  start:  mov byte [6676142 Dh ],0 E 9 h ; We insert JMP into the protector procedure at address @  mov dword[6676142E],99C9F265  mov eax,401000h  poick\_call:  cmp byte[eax],0E8h  je poick\_adr  inc eax  cmp eax,0B4DFFFh  je exit  jmp poick\_call  poick\_adr:  inc eax  mov ecx, dword[eax]  add ecx,eax  add eax,4 ; Прибавляем 4 потому что уже увеличили EAX на 1  cmp ecx,4010A9h  je main  jmp poick\_call  main:  push eax  jmp eax  @ mov esp,12AB0Ch  pop eax  add eax,5  jmp poick\_call  exit:  int3  .end start |

Having assembled all this code and all the others, and placed them in the header section, you still need to configure the addresses and bytes for patching. You can also combine all these pieces of code into one, i.e. the end of the first code - JMP - to the beginning of the next, etc. Also, before starting, you must copy into our file, with the codes for restoring imports, the section of code with the restored nanomites. Simply open the dump obtained in the second part in another OllyDbg and do the following: Copy – Select all => Binary – Binary copy , answer “YES” to the question about the size being too large. AND insert V our file Copy – Select all => Binary – Binary paste. Having done all this, we put eip at the very beginning of our code and run it, all that remains is to wait for the code to finish running, insert a temporary import table in place of the real one and dump it into PETools . Also open ImpREC , select process, enter OEP , press IAT AutoSearch and be sure to check RVA and import size, if everything matches then Get Imports . Let's see if everything is determined correctly by clicking on Show Invalid and C ut thunk ( s ). We won’t edit the dump for now, but just save Save Tree to some file. Now we will need to further correct our dump, namely, remove the last two sections of the tread:



Then we fix our dump in ImpREC . Well, at the end you can also rebuild the file. This is where PETools couldn't help me . The working file was released only in LordPE . Also, before rebuilding, you can remove the line with the SafeDisk version from the file header section, this will reduce its size, since this line prevents us from doing this.

Having done all this work, we restored the import of the file, pointers to it, as well as emulated opcodes, but this still turns out to be not enough, there are also protective mechanisms that we will consider in the next part.

I would also like to say that emulated opcodes may not always recover themselves. In my case, apparently a lighter version of this protective mechanism was used. Which may be called SDAPI v1 .