Study of **SDAPI protective functions**

**second version**

**on example Need For Speed Most Wanted 1.2**

**Posted by DillerInc**

**Date: October 16, 2006**

|  |
| --- |
| **Content** |
| Introduction |
| Portrait of a hero |
| Stage two - vestibule of the tread |
| Data section of a protected file, protector structure |
| Stage three - library ~df394b.tmp |
| Let's lift the curtain... |
| Section data 'revisited' — Trigger Tables |
| A few final thoughts |
| Acknowledgments |

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

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

SafeDisc Advanced is essentially nothing more than the fourth version of the well-known SafeDisc protector , with the only difference being that now the developer has at his disposal another protective mechanism, which will be discussed in this article.

As an example, we will look at the game " Need For Speed: Most Wanted" version 1.2 from EA GAMES. This choice is due not only to the wide popularity of this game, but also to some other circumstances, which will become clearer as the story progresses.

By the way, it is worth noting that EA GAMES is one of the few large companies that is now actively using this “enhanced” variation of the SafeDisc protector , protecting its game releases with it. A similar circumstance also determines the fact that the version of the protector in these releases cannot be analyzed by standard methods, so now, as a small digression from the main topic, I will allow myself to clarify this situation:

|  |
| --- |
| ***Methods for obtaining version information***  ...speaking of “standard methods” for determining the protector version , I mean searching in the header of the PE file for the line “BoG\_ \*90.0&!! Yy>”, followed by values of type Unsigned Long, which represent the version. However, as I already mentioned, this line can simply be overwritten. It is in such cases that you can use the following method:  **offset stxt774 + vsize stxt774 + 38h**  ... Where:  offset stxt774 — offset of the section in the file  vsize stxt774 — virtual section size  Using the resulting offset, we read three values of the same Unsigned Long type. Using the above methods we get the standard view of the version :  \* for example, 4.60.00  For special gourmets, we can also offer the opportunity to find out the full version of the protector :  \* for example, 4.60.00.1702 (2005/08/30)  ... reading it as a resource from one of the files that are stored in a temporary directory while the protected application is running. The required file may have different names in different cases, but its so-called. InternalName ( see resource help on some MSDN ) will always be " AuthServ ".  The disadvantages of this method are the need to launch a protected application in order for the necessary files to be decrypted and placed in a temporary directory, as well as the fact that such version information may not always be available at all, i.e. may simply be missing. |

|  |
| --- |
| **Portrait of a hero** |

In my opinion, SDAPI of the second version as a protective mechanism stands out quite sharply against the background of everything else that the SafeDisc protector ( hereinafter referred to as SD) offers . By “everything else” I mean here the attached essence of most protector mechanisms (import adapters, nanomites, etc. ).

|  |
| --- |
| **Quote :** Relayer, author of ExeCryptor  The fact that mounted protectors are sooner or later removed once and for all was also known a long time ago. It was necessary to find some universal method that would allow one to reliably protect the code from investigation and achieve a fairly strong “fusion” of the protector code with the application code. |

It is with the help of the second version SDAPI, Macrovision, a company engaged in SD development , has taken a fairly serious step towards that very “merging”.

Previously, there was also a first version of the security mechanism, the principle of which, however, was completely different - something related to the encryption of individual functions.

SDAPI of the second version are, in most cases, a kind of computational functions that are firmly embedded in the main code of the protected application and work approximately according to the following principle: using certain parameters and tables of values, the protector, using these functions, calculates the necessary values and returns them to protected application. Among the returned calculation results there can be both protector values used for subsequent calculations and variables/constants necessary for the application itself to function.

SDAPI , introduced by the programmer, most likely, into the source code of the protected application, thus forming a single whole with the main code.

Here is a small example of using these functions in NFSMW:

|  |
| --- |
| ***Code – SDAPI call option***  .text:006E7751 push ebx ; bMenu  **. text :006 E 7752 push 54052 D 06 h ; tread values**  **.text:006E7757 push 54052D06h**  **.text:006E775C call sub\_6E6F90 ; вызов SDAPI**  .text:006E7761 add esp, 8  **.text:006E7764 push eax ; dwStyle**  .text:006E7765 lea eax, [esp+4+X]  .text:006E7769 push eax ; lpRect  .text:006E776A call ds:AdjustWindowRect |

...after calling SDAPI in this case, the EAX register contains the value 80000000h , which corresponds to the WS\_POPUP window style. A little higher in the code there are also calls to the protector functions, which, so to speak, prepare the ground for obtaining the “useful” value that the game itself needs.

This example in some way illustrates that such an approach as patching ( NOP them what belongs to the protector , and insert your useful command) is not entirely appropriate here.

The explanation for this is as follows :

\* firstly, there is no firm guarantee that the returned result of a single SDAPI will be the same under any conditions.

\* secondly, by going the route of such a patch, you will have to - believe me - destroy a good part of the code section with NOPs and the like.

The body of protective functions can be decomposed into several parts - stages:

* Stage one is a direct call to the protector API from the code section of the protected application. Before this, as a rule, two parameters are placed on the stack , which determine to some extent output result. Next, a call occurs, carried out by the usual CALL command.
* Stage two - the so-called the vestibule of the tread, also located in the code section.
* Stage three is the tread code itself, located in a separate module.

The last two require stopping and taking a closer look.

|  |
| --- |
| **Stage two - vestibule of the tread** |

Our starting point will be the code section of the protected file.

By pushing two parameters onto the stack and following the CALL command, we end up in the following location:

I propose to immediately agree that we We will consider the very first SDAPI call from the moment the protected application is launched. This call occurs at address 0088 5F60.

|  |
| --- |
| ***Code – Tread vestibule***  .text:00666E40 sub esp, 20h  .text:00666E43 push esi  .text:00666E44 mov esi, dword ptr [.data] section  .text:00666E4A test esi, esi  .text:00666E4C jnz short loc\_666E57  .text:00666E4E mov eax, dword ptr [.data] section  ...  .text:00666E57 push ebx  .text:00666E58 mov ebx, dword ptr [.data] section  .text:00666E5E push edi  .text:00666E5F mov edi, dword ptr [.data] section  ...  **.text:00666E73 call CryptParamsArray**  **.text:00666E78 lea eax, [esp+2Ch+var\_20]**  .text:00666E7C cdq  .text:00666E7D push edx  .text:00666E7E push eax  .text:00666E7F push ebx  .text:00666E80 push edi  .text:00666E81 call esi  …  .text:00666E8C retn |

... I deliberately omitted some commands in this listing, leaving only the most important ones at the moment. In any case, in the code section you can easily find one of these thresholds (there are quite a lot of them) by signature:

83h ECh 20h 56h 8Bh 35h

So, first you need to concentrate your attention to the CryptParamsArray function ( my name), where you can see the following code:

|  |
| --- |
| ***Code – CryptParamsArray function***  .text:0065FC40 push ecx  .text:0065FC41 push esi  ...  .text:0065FC50 mov edx, [esp+4]  .text:0065FC54 and edx, 3  **.text:0065FC57 mov eax, dword\_8F7648[edx\*4] ; 1. DWORD пары**  **.text:0065FC5E mov edx, dword\_8F7658[edx\*4] ; 2. DWORD пары**  ...  .text:0065FC6B mov dword ptr [ecx+4], **10804201h**  .text:0065FC72 mov dword ptr [ecx+8], **10804205h**  .text:0065FC79 mov dword ptr [ecx+0Ch], **10804201h**  ...  ...много команд XOR, IMUL  ...  .text:006 5FCEA retn 8 |

The main idea of this procedure is to encrypt the data transmitted further to the protector. This is done using certain values from the data section of the protected file. The fact is that in this section at addresses 008F76xx and 00900Axx there are two arrays from which, with some degree of probability, they are selected two DWORDs involved in encrypting input data using the XOR and IMUL commands.

For example, for those two DWORDs we’ll take a couple of values – **F6B234CFh** and **A216268Bh** – As a result, we will get an encrypted array, which I called ParamsArray and which will look something like this :



We place a pointer to this array in the EAX register immediately after calling the CryptParamsArray function. Note that the first value of the pair at the end of the procedure, it is overwritten while the second value from the pair remains unchanged and takes first place in the array - both will play a key role in subsequent decryption.

If we trace which values are encrypted, then we should have some idea of the transmitted data. These certainly include two protector parameters that are placed on the stack when SDAPI is called, in addition we observe two values that were at that moment randomly above the top of the stack, as well as three constants (I marked them in blue in the listing). The latter will be discussed later will determine the overall progress of calculations in the body of the protector, therefore they are quite important milestones.

Looking again at the listing of the Vestibule..., you can see that this code is actively taking something from the .data section - well... we'll figure it out.

|  |
| --- |
| **Data section of a protected file, protector structure** |

In the case of SDAPI, the data section occupies a rather significant position, and now we have to make sure of this.

Following at the address where the first DWORD for the ESI register comes from , we will find ourselves in the .data section and see the value 6672F2F7h.

If you take a closer look and scroll up/down the hexadecimal “columns” (for example, in WinHex), you will notice a certain order in the arrangement of nearby data.

In order to immediately clarify the situation, I will say that here we are dealing with a table, the elements of which are instances of the SDArgsStruc structure (see below), containing certain information necessary for the functioning of those same SDAPI.

There are usually 40 elements in the table (instances of structure) things.

Size structure is 184 h byte.

I defined this structure as follows:

SDArgsStruc struc

LeadIn dd ? ; **67231341h** – start marker

SDArgsStrucSize dd ? ; **184h**

LeadIn\_ dd? ; **00000001h**

MagicAreaOrdinal dd ?

TrashDwordA dd ? ; 00

HeapHead dd ?

HeapArray dd 8 dup(?)

SDRoutineAddr dd ?

TrashDwordB dd ? ; 00

SDRoutineAddrDouble dd ?

TrashBytesArrayA124 db 124 dup(?) ; 00

MagicAreaPtr dd ?

TrashDwordC dd ? ; 00

TrashBytesArrayB112 db 112 dup(?) ; 00

HeapArrayAux dd 8 dup(?)

TrashBytesArrayC32 db 32 dup(?) ; 00

PtrToCurrentStruc dd ? ; Pointer to the beginning of the structure

DataSectionOrdinal dd ?

LeadOut dd ? ; **78822408h** – end marker

SDArgsStruc ends

Let me note right away, just in case, that fields like Trash\* are just zero garbage bytes.

When examining this structure , it is worth noting that the highlight of each is the double word HeapHead and the HeapArray , consisting of eight DWORDs . Now, however, we will not dwell on them, because It will make more sense to do this when we dive into the SD security code.

You can find the table itself in the data section by the first member:

41h 13h 23h 67h

... each structure, a kind of marker, begins with these bytes.

Now we will look at other important fields of the structure using a suitable example.

Now taking into account the features of the .data section , you can process the required piece of data in IDA . As a result, we will get something like this listing:

|  |
| --- |
| ***Code – Tread vestibule***  .text:0066 6E40 sub esp, 20h  .text:00666E43 push esi  **.text:00666E44 mov esi, stru\_8FACC8.SDRoutineAddr**  ...  .text:00666E4E mov eax, stru\_8FACC8.TrashDwordB  ...  .text:00666E58 mov ebx, stru\_8FACC8.TrashDwordC  ...  **.text:00666E5F mov edi, stru\_8FACC8.MagicAreaPtr**  ...  .text:00666E73 call CryptParamsArray  .text:00666E78 lea eax, [esp+2Ch+var\_20]  ...  **.text:00666E81 call esi ; в модуль ПРОТЕКТОРА !**  .text:00666E83 add esp, 10h  …  .text:00666E8C retn |

Particularly interesting here are the SDRoutineAddr and MagicAreaPtr fields:

* SDRoutineAddr is the address of the protector procedure, where the dog is buried. In this case, it is 6672F2F7h - this is where we go with the CALL ESI command .
* MagicAreaPtr is a pointer to an area in the heap of the process under study, where very important information is stored.

If everything should be clear with the SDRoutineAddr field , then in the second case it’s worth looking into the dump window, which shows us that memory area:



The beginning looks familiar, doesn't it??

The casket actually opens simply - in the heap there is the same structure SDArgsStruc only with slightly changed components (fields). Such similarities will help us in the future.

The double word following the unit - MagicAreaOrdinal - is identical to the DataSectionOrdinal field of the corresponding structures in the data section . Based on the name, this value is a kind of ordinal number of the structure, so you can, for example, conclude that the DataSectionOrdinal field of the last structure in the data section will be equal to 0000003Fh.

Further in memory there are nine DWORDs ... more precisely one + eight DWORDs , which are specially associated with the HeapHead and HeapArray fields, but to which we will return a little later.

So.

CALL ESI

Deep breath...we are in the protector's body.

|  |
| --- |
| **Stage three - library ~df394b.tmp** |

the SD protector is the ~df394b.tmp library. When a protected application is launched, it is decrypted from the overlay attached to the protected file and placed in a temporary directory, from where it is then loaded into the address space of the main process using the LoadLibraryA function. After this, the address is obtained and the protector initialization function - Ox12121212 - located in this library is called (for reference: the first character of the function name is a capital “ O ”, not zero). During this initialization, some values are generated that we will soon have to deal with collide.

Having finally found ourselves in the protector at address 6672F2F7, we will be interested in procedure 6672F045 , which is the main branch of code from which everything else originates.

The first thing that strikes you when you scroll through the code window in the debugger is certainly the size of the protector procedures. The aphorism of Kozma Prutkov, which says:

*“Where is the beginning of the end with which the beginning ends??”*

This is mainly due to the fact that the security library code in this case (in this version of the protector) contains a fair amount of metamorph.

|  |
| --- |
| **Quote:** Ms-Rem  ... the metamorph tries to completely change the appearance of the code, while preserving the original algorithm of its work, for which it replaces instructions with their synonyms, which in turn consist of one or more other instructions. Most of the new code produced by a metamorph is usually needed for the program to work; the proportion of garbage code is very small. |

Examples of such obfuscation will certainly be considered, but first we need to create a kind of map of the area - a logical diagram that would illustrate the general course of actions performed by the protector code.

1. Creating a HeapArray
2. Obtaining the value required for decrypting the ParamsArray array
3. the ParamsArray array itself
4. Getting the CASE value for the first switch..case statement
5. First switch..case statement
6. Getting the CASE value for the second switch..case statement
7. Second switch..case statement
8. Final value calculation

So, let's take a look at the initial code of procedure 6672F045:

|  |
| --- |
| ***Code - procedure 6672F045***  .text:6672F045 mov eax, offset loc\_6677E55E  .text:6672F04A call \_\_EH\_prolog  .text:6672F04F sub esp, 0CCh  .text:6672F055 push ebx  .text:6672F056 push esi  **.text:6672F057 mov esi, [ebp+8]** |

A pointer to the encrypted ParamsArray is passed to the ESI register . To decrypt the array, we need the first value from the pair with which we encrypted the array, but it turns out to be inaccessible, because at the end of the CryptParamsArray procedure , it is overwritten. The second value from the pair is located unchanged at the very beginning of the array. It is with the help of it that we obtain/restore the first value of the pair. However, for these calculations we will need additional data, namely the HeapArray array.

|  |
| --- |
| ***Код – процедура 6672F045***  .text:6672F069 call sub\_66716F10  .text:6672F06E and dword ptr [ebp-4], 0  .text:6672F072 lea eax, [ebp-84h]  .text:6672F078 push eax  .text:6672F079 lea eax, [ebp-0BCh]  .text:6672F07F push eax  .text:6672F080 lea ecx, [ebp-14h]  **.text:6672F083 call sub\_6672620A**  .text:6672F088 mov ecx, eax |

Actually, our goal at the moment is procedure 6672620A, but before going into it, I would like to explain one point regarding the representation of various kinds of values /parameters in the protector code.

**The** fact is that important values - the results of calculations - do not just wander through the code. They are encrypted. There are at least two functions responsible for this - 66716F10 ( which can be seen in the listing above) and 66716F53.

They work on approximately the following principle: before using any value in a calculation, the latter is decrypted. After receiving the result, the latter is encrypted again and passed on through the code.

|  |
| --- |
| ***Code - procedure 66716F10***  **.text:66715658 call sub\_667146F8**  .text:6671565D mov ecx, eax  **.text:6671565F xor ecx, @@Value**  .text:66715662 mov eax, ecx  .text:66715664 not eax  .text:66715666 xor eax, ecx  .text:66715668 and eax, 32884FD1h  .text:6671566D not ecx  .text:6671566F xor eax, ecx |

...encryption or decryption of the @@ Value occurs using a special value - a mask - which in this version of the protector is dynamic, i.e. each time a protected application is launched, a new mask is calculated. The purpose of such a trick is to confuse the researcher as much as possible, because with each new launch he will be faced with completely new and unusual encrypted values.

The mask appears in the EAX register after calling procedure 667146F8, where you can see the following:

|  |
| --- |
| ***Code – procedure 667146F8***  ...  .text:667146FE cmp dword\_667A03F8, 0  **.text:66714705 mov ecx, dword\_667ADB0C**  .text:6671470B jnz short loc\_6671471C ; let's jump  .text:6671470D imul ecx, ecx  .text:66714710 imul ecx, 0CB495FD8h  .text:66714716 mov dword\_667ADB0C, ecx  .text:6671471C test ecx, ecx  .text:6671471E jnz short loc\_66714779 ; let's jump  ... |

If the initialization of the protector was successful, then the necessary values were generated and placed in their cells, therefore we make a jump in both cases of conditional transitions in the above listing. Specifically, we are interested in the memory cell at address 667 ADB0C, where that same mask is located.

We can track the write to this address by setting a hardware read/write breakpoint there.

What will it give?

A little convenience during research. We can then replace the new value each time with an already known and familiar one (once generated), so as not to get so confused among the encrypted data.

So, let's go inside procedure 6672620 A :

|  |
| --- |
| ***Code – procedure 6672620 A***  .text:6672620A mov eax, offset loc\_6677D175  .text:6672620F call \_\_EH\_prolog  .text:66726214 sub esp, 1FCh  ...  **.text:66726235 call sub\_66725B7D**  ... |

Generally speaking, this procedure is called three times during one session ( SDAPI call ) , calculating and returning each time the special values necessary for the operation of security code in general.

In the first case, the procedure receives/restores the first value from the pair, which is then used to decrypt the ParamsArray array. But as I already mentioned, for this you need to create a HeapArray. That is why, at the very beginning of procedure 6672620A , procedure 66725B7D was quietly hidden , which does exactly this .

**HeapArrays** are extremely important elements of the SDAPI security mechanism because ... act as a kind of intermediary in the most significant calculations. Being an integral part of each instance of the SDArgsStruc structure , any given HeapArray array is logically associated with the \*Ordinal field of its structure. From this we can again conclude that the maximum number of possible arrays is equal to the number 40h.

Eight DWORD values are generated for each array. The key element involved in the calculation of each member of the array is

the corresponding array of eight DWORDs , which is located on the heap, in the area pointed to by MagicAreaPtr.

I called the array located in the “magic area” MagicArray .

The responsible protector code, having generated an array once for a certain Ordinal , sets a special mark that prevents the same array from being generated again.

For this case there are two similar checks at the beginning of the procedure:

|  |
| --- |
| ***Code – procedure 66725B7D***  ...  .text:66725BA3 call sub\_66720075  .text:66725BA8 or dword ptr [ebp-4], 0FFFFFFFFh  **.text:66725BAC test al, al ; If the array has already been received,**  **.text:66725BAE jz loc\_66725CDF ; then let's jump to the exit**  ... |

If you remember, in the heap, in addition to the eight DWORDs ( MagicArray), there was one more DWORD. So this is it the value ( for example, 9 BE 6 BF 89 h ) corresponds to the HeapHead field of the SDArgsStruc structure and also goes through the evaluation process. It happens here:

|  |
| --- |
| ***Код – процедура 66725B7D***  ...  **.text:66725C04 call dword ptr [ebx+8] ; [ebx+8] = 667286ED**  .text:66725C07 push eax  .text:66725C08 mov ecx, edi  .text:66725C0A mov byte ptr [ebp-4], 3  .text:66725C0E call sub\_66725199 |

But in fact, the value of the HeapHead field for any HeapArray will always be the same - **5 CAC5AC5h -** so I didn’t go into much detail about calculating this value. However, the same cannot be said about the HeapArray array itself , the code for obtaining which must not only be parsed, but then implement it yourself. In general, closer to the point.

All eight array values are calculated in a loop:

|  |
| --- |
| ***Code – procedure 66725B7D***  ...  .text:66725C32 **loc\_66725C32** :  ...  .text:66725C4E call sub\_66725882  .text:66725C53 test al, al  ...  . text :66725 C 5 F jz short loc \_66725 CC 6 ; exit the loop  ...  .text:66725CC1 jmp **loc\_66725C32** |

For the received values, cells are prepared in a buffer that is also on the heap (that’s why I called this array *Heap* Array):

|  |
| --- |
| ***Code - procedure 66725B7D***  .text:66725C64 call sub\_66716F53 ; get the index of the array element  .text:66725C69 imul eax, 1Ch  .text:66725C6C mov ecx, [esi+94h] ; in ECX we put a pointer to the buffer  .text:66725C72 add eax, ecx ; calculate the offset in the buffer |

The main calculations occur again in procedure 667286ED, which this time is called using the command:

.text:66725C88 call dword ptr [edx+8]

Once inside, you must immediately proceed to procedure 6672619 F , because it is there that what we need happens. Let’s look there and highlight the main points of interest.

|  |
| --- |
| ***Code - procedure 6672619F***  ...  **.text:667261C0 call dword ptr [eax+0Ch] ; 66725EA0**  ...  **.text:667261D4 call dword ptr [eax+0Ch] ; 66725EA0**  ...  **.text:667261E2 call sub\_66725175**  ...  **.text:667261F2 call sub\_667258E3**  ... |

let 's remember what I said about metamorph. As an example, let's take function 66725175 , which I called GetDerivative . The core of this function is the following code:

|  |
| --- |
| ***Code - function GetDerivative***  **.text:66720CE2 call ds:off\_66789EA8** ; MFC 3.1/4.0/4.2/7.1 32bit  .text:66720CE8 push 4B495FD8h  .text:66720CED push 109h  .text:66720CF2 push 2  .text:66720CF4 lea ecx, [ebp-24h]  **.text:66720CF7 call sub\_667388E2** |

Both calls use some C library functions , in which simply incredible calculations take place.

In order not to get lost in thought, and also due to the fact that presenting the obfuscated version of the code here will take up a lot of space, I immediately want to present the code cleared of the metamorph:

I note that the GetDerivative function appears in the protector code in two forms, slightly different in the commands executed. I conveyed this point using the WAY parameter, which determines what calculation needs to be performed in a particular case.

|  |
| --- |
| ***Then – you can get GetDerivative***  GetDerivative proc  arg @@ValueA:DWORD, @@ValueB:DWORD, @@WAY:DWORD  uses editing  move edit , @@ValueB  mov eax , @ @ ValueA  cmp dword ptr @@WAY,  jne @@m1  **imul eax, edit**  jmp @@ret  @@m1:  **add eax, edit**  @@ret:  ret  GetDerivative ends |

As a result, the whole fabulous code comes down to either multiplying two parameters,

or - to addition. To feel the difference, I advise you to slightly scratch the local tread code.

You, readers, will probably have a question: “How could he do this?”

In fact, everything is very simple - everything is learned by comparison.

The fact is that not all versions of the protector (...or specific cases) use such obfuscation. Therefore, studying many examples can be very effective in terms of a general understanding of the material.

Function 667258E3 is also worth a closer look. Here you need to pay attention to the following piece of code:

|  |
| --- |
| ***Code – procedure 667258E3***  ...  .text:66725921 mov dword ptr [ebp-10h], **24C3E94Dh**  .text:66725928 call sub\_667173DF  ... |

The essence of the function is that it takes two parameters and can perform one of three operations on them: XOR, AND or OR.

Which exact operation will be performed, or, in other words, which branch of code in procedure 667173 DF will be performed, is determined by the DWORD that is highlighted in bold in the listing. In this case, this label corresponds to the XOR operation.

It follows that we can replace the call to the entire procedure with the following code:

|  |
| --- |
| ***The code is one of the alternatives to procedure 667258E3***  mov eax, @@ValueA  mov ebx, @@ValueB  xor eax, ebx |

At this stage, the matter remains with procedure 66725EA0.

It is called twice, in the first of which the key role is played by the value of the current Ordinal , in the second - the current index of the array element.

The calculations again involve procedures 667173 DF, where the XOR operation is performed, as well as procedures in which linear commands are used logical shift:

|  |
| --- |
| ***Code – procedures 66726055 and 667260С0***  .text:667260A3 shl esi, cl  ... either  .text:6672610E shr esi, cl |

Constants act as a shift counter, which throughout the entire procedure 66725 EA0 are pushed onto the stack using the PUSH command:

|  |
| --- |
| ***Code – procedure 66725EA0***  ...  .text:66725EE4 push 1  ...  .text:66725F24 push 2  ...  .text:66725F54 push 3  ... and so on |

In the end, everything works according to the following scenario:

1. We obtain the derivative from the current Ordinal in procedure 66725EA0
2. We obtain the derivative of the current array element index in procedure 66725EA0
3. We cross the resulting values with each other using the GetDerivative function
4. Result of XOR with the corresponding DWORD from the MagicArray

Next, the final result is written to a specific cell in memory, and everything is repeated again to obtain the next element of the array.

For a more complete and visual example of the implementation of both this and subsequent procedures, please refer to the “ SD Procs examples.inc ” file attached to the article.

**Returning** again to procedure 6672620A, which should return the first value of the pair necessary to decrypt the ParamsArray array, I would like to note that here, too, the main workhorses (meaning functions) are those that have already been discussed above. As a parameter, as I already mentioned, the second value from the pair is transmitted, which is in unmodified in ParamsArray . This is then processed using the HeapHead value as well as the first two values from the HeapArray.

Again, look at the attached file.

Having received the required value, we begin to decrypt the array. If you remember, the array was encrypted using the XOR and IMUL commands.

So, the function 6672512A, also known as GetDerivative, which in a simplified version clearly demonstrates this possibility, acts as a signed multiplication operation . And the XOR operation is openly on the surface:

|  |
| --- |
| ***Code – procedure 6672 F045***  .text:6672F0D9 xor [esi+1Ch], eax  ... and in the same way further |

As a result, the ParamsArray array in decrypted form will look something like this:



The two null DWORDs are the two protector parameters that were pushed onto the stack when calling SDAPI. But now we will be more interested in those same three (or rather two) protector constants that we encountered in the CryptParamsArray function and which I marked there in blue.

If we remember our map of the area, now we should have the first switch..case statement on our nose , for which we must first get the CASE value. Here we again encounter procedure 6672620A , in which this value is calculated. The key parameter passed to the function in this case is one of the protector constants, namely:

**[ ESI +08] = 10804205** ; don't forget that the pointer to the ParamsArray array was

; placed exactly to the ESI register

The rest of the calculations in procedure 6672620A are similar to the option already discussed above. After we get our result - the CASE value - we come to the operator itself.

|  |
| --- |
| ***Code – first SWITCH..CASE statement***  .text:6672F208 cmp eax, 7 ; switch 8 cases  .text:6672F20B ja short loc\_6672F280 ; default  .text:6672F20D jmp ds:off\_6672F2B0[eax\*4] ; switch jump |

**purpose** is approximately the following: determining the type/size of the operands with which the necessary calculations will be performed. This point can be demonstrated much more clearly by when we get to these calculations themselves, for now I’ll just try to give a diagram of this operator.

So, there are eight possible options in total:

* CASE 0 and 1 — operand size: byte
* CASE 2 and 3 - operand dimension: word
* CASE 4 and 5 - operand size: double word
* CASE 6 and 7 - operand size: quad word

Taking this for granted for now, let's move on.

If we are currently debugging the very first SDAPI call, which I mentioned above, then the first CASE value will be the number 5, which after the decrement command becomes equal to four. Now we go to the desired code branch and almost immediately notice the call to procedure 6672620A - already the third count and this time the last one. It calculates the CASE value for the second statement using the protector constant:

**[ ESI +0C] = 10804201**

**Before** moving further to the second operator, it is necessary to analyze one important detail - the formation of data/parameters involved in the final calculations, the result of which will be the output value.

To do this, you need to take another look at the decrypted ParamsArray array, the pointer to which at the right moment appears again in the ESI register .

The idea here is the following - two pairs of values are formed, each of which contains, so to speak, primary and secondary values. The primary ones are taken from those primary parameters placed on the stack when calling SDAPI ( in this case, these are two zero DWORDs ), the secondary ones - from the last two DWORDs of the ParamsArray array , which I spoke of as being randomly located above the top of the stack. In general, to be frank, the decisive values here are only the primary ones. The secondary ones were apparently added simply “to the heap” (to give “volume” calculations), and their presence does not in any way affect the final result.

Here is one version of such formative code:

|  |
| --- |
| ***Code – procedure 6672EA14***  . text :6672 EA 26 mov esi , [ ebp +8]; ESI​ pointer to array  ...we get minor terms for pairs from the array  .text:6672EA58 mov edi, [esi+18h]  .text:6672EA5B mov edx, [esi+14h]  ... forming pairs  .text:6672EA69 mov eax, [esi+10h]  .text:6672EA6C xor ebx, ebx  .text:6672EA6E or ebx, eax  .text:6672EA70 mov eax, [esi+1Ch]  .text:6672EA73 xor ecx, ecx  .text:6672EA75 xor esi, esi  .text:6672EA77 or edi, ecx  .text:6672EA79 or eax, esi  .text:6672EA7B or ecx, edx  .text:6672EA7D push eax ; Сохраняем в стеке  .text:6672EA7E push ecx ; первую пару  .text:6672EA7F lea ecx, [ebp-0F8h]  .text:6672EA85 call sub\_667149DB  .text:6672EA8A push edi ; Save V stack  .text:6672EA8B push ebx ; second a couple  .text:6672EA8C lea ecx, [ebp-88h]  .text:6672EA92 mov dword ptr [ebp-4], 2  .text:6672EA99 call sub\_667149DB |

Having formed pairs of parameters, and also already having the next calculated CASE value in our pocket , we move on to the second switch..case statement .

**In** the second, the operator, unlike the first, is more extensive:

|  |
| --- |
| ***Code – procedure 6672 D83E – second SWITCH..CASE statement***  .text:6672D877 cmp eax, 16h  .text:6672D87A ja loc\_6672DAC5 ; default  .text:6672D880 cmp eax, 16h ; switch 23 cases  .text:6672D883 ja loc\_6672DAC5 ; default  .text:6672D889 jmp ds:off\_6672DB04[eax\*4] ; switch jump |

Here, the selection options determine the operations/commands that will be performed on the intermediate results of calculations.

Take, for example, CASE with value 2:

|  |
| --- |
| ***Code – procedure 6672A883***  . text :6672 A 898 push dword ptr [ ebp +0 Ch ] ; We put the second pair on the stack  ...  .text:6672A8A1 call sub\_667284C4 ; We produce calculations  ...  .text:6672A8A6 push dword ptr [ebp+10h] ; Let's put V stack first a couple  ...  . text :6672 A 8 B 3 call sub \_667284 C 4 ; We make calculations  ...  .text:6672A8C7 call sub\_66723219 ; We perform CASE calculations  ...  .text:6672A8DA call sub\_66728508 ; We make final calculations |

Based on this, we can highlight the following step-by-step diagram options for the second operator:

1. We calculate the derivative of the second pair using function 667264A2
2. We calculate the derivative of the first pair using function 667264A2
3. We use both results in a CASE calculation, which depends on the specific CASE
4. We use the obtained result in function 66727342, where the final value is calculated

CASE calculations are arithmetic or logical operations on operands. Each CASE is characterized by its own specific operation. For example, the second CASE is characterized by the operation

addition - ADD:

|  |
| --- |
| ***Code – procedure 667212 DA***  .text:667212DF call sub\_6671F6E6 ; We get the second result  .text:667212E4 mov ecx, [esp+4+arg\_4]  .text:667212E8 mov esi, eax  .text:667212EA call sub\_6671F6E6 ; We get the first result  .text:667212EF add eax, esi ; Let's add them up |

The above step-by-step scheme is present in all CASEs except numbers 1 and 0Dh - they use only one strictly defined pair and one function:

* in CASE1 - the first pair and function 66727342
* in CASE0D - second pair and function 667264A2

... and no CASE calculations.

I will not now give the remaining operations characteristic of other CASEs , because You, dear readers, can easily do this yourself by looking at the necessary pieces of code in a disassembler , guided by the above diagram.

**Both** selection operators are related to each other using the same principle of dimension. The operands, the size of which is determined at the stage of the first operator, are then used in the calculations determined by the second operator.

Most clearly, in my opinion, this use of dimension can be seen in the following place:

|  |
| --- |
| ***Code – Step 4 of the above diagram***  . text :667282 E 6 call dword ptr [ eax ]; We get the result of Step 3  **. text :667282 E 8 movsx eax , al**  . text :667282 EB cdq ; We define the “to heap” parameter in EDX  . text :667282 EC push edx ; Pushing parameters onto the stack  .text:667282ED push eax  ...  . text :66728300 call sub \_66727342 ; Final calculation |

As you can easily guess from the command in bold, here we are dealing with bytes, i.e. with CASE 's 0 or 1 of the first statement.

Moreover, this forwarding command with extension varies slightly between CASEs of the same dimension.

For example, CASE 's 2 and 3 belong to “word”, so in the first case we will see something like this:

|  |
| --- |
| ***Code – Step 4 of the above diagram***  . text :66728401 call sub \_66721182 ; We get the result of Step 3  **. text :66728406 movsx eax , ax ; forwarding with signed extension**  .text:66728409 cdq  .text:6672840A push edx  .text:6672840B push eax |

...in the second case it will be:

|  |
| --- |
| ***Code – Step 4 of the above diagram***  . text :66728490 call sub \_66721210 ; We get the result of Step 3  **. text :66728495 movzx eax , ax ; forwarding with zero extension**  .text:66728498 cdq  .text:66728499 push edx  .text:6672849A push eax |

**finally** got to the final calculations. It was already mentioned above that there are two responsible functions here: 66727342 and 667264A2.

Despite the colossal size of the code (obfuscation), the principle of the latter is quite simple. Therefore, I will be brief.

One of the pairs of generated parameters is passed to the function, which is processed with values from the HeapArray. One of the key points of each of the functions is obtaining a special value, which I called the base - it is with its participation that the final calculations are constructed, as a result of which the final value is obtained. If you carefully analyze the code of the functions, it turns out that this base is calculated as many as four times per throughout the entire algorithm, i.e. The same code is duplicated many times - due to this, apparently, they tried to add volume.

In fact, all calculations fit “in three registers,” as you can see by looking again at the attached file containing the implementations of these functions.

|  |
| --- |
| **Let's lift the curtain...** |

Having at this point gained some insight into the activity of the protector, we are ready to finally find out the real truth about the new protective mechanism - SDAPI version 2. A truth that will make many who read this material involuntarily smile in surprise.

The fact is that all routine work related to the functioning of SDAPI can be successfully performed outside (!) the presence of the main protector module - the ~df394b.tmp library. In other words, all code capable of performing the required calculations and returning the required results is is available in the most secure application - in its code section. Just look again at one of the Vestibules:

|  |
| --- |
| ***Code – Tread vestibule***  .text:00666E40 sub esp, 20h  .text:00666E43 push esi  .text:00666E44 mov esi, stru\_8FACC8.SDRoutineAddr  **.text:00666E4A test esi, esi**  **.text:00666E4C jnz short loc\_666E57**  .text:00666E4E mov eax, stru\_8FACC8.TrashDwordA  **.text:00666E53 test eax, eax**  **.text:00666E55 jz short loc\_666E8D**  ...  **loc\_666E8D:**  .text:00666E8D mov edx, [esp+24h+arg\_4] ; второй параметр  .text:00666E91 mov eax, [esp+24h+arg\_0] ; first parameter  .text:00666E95 push edx  .text:00666E96 push eax  **. text :00666 E 97 call sub \_6665 B 0 ; We carry out all the calculations**  . text :00666 E 9 C add esp , 8; Restoring the stack  .text:00666E9F pop esi  .text:00666EA0 add esp, 20h  . text :00666 EA 3 retn ; And we leave the vestibule |

That is, if the address of the protector procedure - 6672 F 2 F 7 - is equal to zero, then we get to the treasured code.

However, in order to operate properly, this code, like the protector code, needs data. It is not difficult to guess that this data is HeapArray arrays, which now must be located somewhere at hand... namely, in the data section protected file, in SDArgsStruc structures.

If we look into the main procedure of this peculiar loophole, the very first call will lead us to the following code:

|  |
| --- |
| ***Code – procedure 00656740***  .text:00656740 mov eax, offset stru\_8FACC8  .text:00656745 retn |

A pointer to the actual SDArgsStruc structure is placed in the EAX register, which should already have the HeapHead and HeapArray fields correctly filled .

If we remember the similarity of the structures located in the data section and in the memory area in the heap, then we can confidently assume that the HeapArray array will take a position in the SDArgsStruc structure of the data section corresponding to the position of the MagicArray array in the structure located in the heap. The same is with the HeapHead field . And if the value of the HeapHead field is always equal to 5CAC5AC5h,

then the values of the HeapArray arrays must (best) be calculated yourself by writing the necessary code for this.

Thus, by correctly setting up the structures in the data section, we can free the protected application from the chains and shackles of the protector.

Looking at this outcome, a completely reasonable question arises:

“Why did Macrovision leave such a potential hole?” .

There is an opinion that this code is created at the development stage of the protected application for testing purposes and after this stage should be destroyed. However, as practice shows, only HeapArrays from the data section are destroyed, while the code in the .text section often remains untouched.

Based on the purposes of application, this code was dubbed “ debugging handlers ” (or debug handlers).

The game NFSMW versions 1.2 and 1.3 is exactly the case when the debugging handlers are intact, and the only difficulty is restoring the structures in the data section.

All this can be called good news, but there is also bad news. It lies in the fact that in some cases debugging handlers are destroyed, overwritten by 0CCh instructions. Moreover, the latter are not nanomites at all, but ordinary commands of the third interruption. Obviously, in this situation, removing the protector becomes an order of magnitude more difficult, but nevertheless remains possible. And here the researcher himself, based on his knowledge of the material, decides exactly how is going to do it.

But wait, let’s come down to earth for now... have we managed to restore everything so far? Is there any instability in the operation of the unpacked application?

|  |
| --- |
| **Section data ' revisited' — Trigger Tables** |

The SDAPI defense mechanism contains another trick that you need to understand in order to to allow the unpacked application to function properly. We will again have to deal with the data section of the protected file, which contains the next type of structures that require restoration (some kind of guidance...).

According to some rumors, these structures are called Trigger tables .

Accesses to this data can be tracked in the code section using the following commands:

**mov dword ptr [esp+XX], offset data-section  
 mov dword ptr [esp+XX], offset data-section**

...those. The following opcode should be searched:

C7h 44h 24h byte dword

… Where :

byte is an unknown value, which is the so-called. *team bias*

dword – offset in data section

In NFSMW there are four of these structures, an appeal to one of which can be seen here:

|  |
| --- |
| ***Code - example of accessing Trigger table***  **.text:006E772E mov dword ptr [esp+28h], offset dword\_8F8290**  .text:006E7736 mov dword ptr [esp+2Ch], offset off\_8F7B20  .text:006E773E call sub\_6E6E40 ; call SDAPI  .text:006E7743 push eax  .text:006E7744 push 899CC300h  .text:006E7749 call sub\_6E6DD0 ; call SDAPI |

The location of the structure is determined using the command marked in bold in the listing. In this case, the structure is located in the data section at address 008F8290.

The memory dump at this address doesn't look promising:



... but nevertheless, we get the information we need - this is the fourth double word, or rather a byte (in this case 0 Ch) - the so-called “known byte”. With the help of it we will then determine the desired structure.. .again in memory allocated for the heap.

Following the two MOV commands in the listing There is an SDAPI call , which we skip. We are interested in the next call to the protector function at address 006E7749. The fact is that these structures are processed in CASE with number 16 h of the second operator (possibly also in the fifteenth version). The CASE number of the first operator apparently does not play a role here. The code responsible for processing structural data ,is like an add-on for rooms 16h and 15h .

|  |
| --- |
| ***Code – calling the required procedure in CASE16***  .text:66725E54 call dword ptr [eax+4] ; [EAX+04] = 66724396 |

Actually, the only place among all the calculations there that might really interest us looks like this:

|  |
| --- |
| ***Code – procedure 6671 A 297***  **.text:6671A334 mov eax, [esi+0Ch]**  **.text:6671A337 mov esi, [ebp-0A0h]**  .text:6671A33D cmp esi, eax  .text:6671A33F jz loc\_6671ABA7 |

EAX register contains a pointer to a memory area that contains pointers to the three structures being sought:



One of these pointers is placed in the ESI register, for example:



When analyzing the contents of the ESI register , what is important for us so far is that the fourth DWORD (00000001h) coincides with the seventh, and after that comes the same “known byte”, which defines the desired structure for us.

It is worth immediately noting that the representation of the structure in the heap is far from the ideal that should ultimately end up in the data section. Comparing both areas, we can come to the conclusion that the first four DWORDs in the allocated memory are simply cut off. Further reduction of the structure in order, drawn from the works of the well-known ReLOADeD team, for which special thanks to them.

The structure itself consists of twenty-three double words , the first and last of which are markers of the beginning and end, respectively. The first half of the structure, which is its defining part, appears to us in an almost ideal form in a heap, while the second half must be filled in in a unique way :



So. The start marker opens the structure - 69241641h . In the first half, only the four-byte value FFFFFFFFh was replaced with a zero DWORD. The second half, starting in the allocated memory with the double word 00000027h, is simply replaced by double words with values from 00000001 h to 00000008 h inclusive. The end marker - 73872468h - completes the structure.

Similarly for other structures.

However, as I said, there are only four structures (four genuine calls from the code section), but there are only three pointers in the EAX register . Take another look at the contents of the ESI register - the second DWORD is a pointer to the second structure. So, in one of the three structures in the heap must also have a pointer to the missing fourth one. At worst, you can, in principle, simply look in nearby memory for a “known byte” of the missing structure.

|  |
| --- |
| **A few final thoughts** |

At the beginning of this article, I noted that the game “ NFSMW” was not chosen by chance. The reason here lies in the implementation features of the protective mechanism in question.

In this secure application, those links that facilitate the final removal of the tread are preserved - debugging handlers. However, such a choice should not be considered unworthy. In my work, I tried to create an approximate diagram, to emphasize the most important points, which should, on the one hand, help to navigate, and on the other - leave the necessary space for independent maneuver of everyone who reads these lines. I do not claim the impeccable truth of my reasoning given in this article. This is just some attempt to present to the public what has accumulated in my head since I began to study this protective mechanism. If someone suddenly finds that I am fundamentally mistaken about something, and at the same time is able to argue their arguments in the necessary way, then that will be really great. I hope that this article can put those who decided to read it on the right path. However, you should always keep in mind that only personal initiative and perseverance can help you learn all the necessary material and ultimately achieve positive results.

*“What is unclear should be clarified.*

*What is difficult to create should be done with great perseverance.”*

*Confucius*

|  |
| --- |
| **Acknowledgments** |

First of all, I would like to express my gratitude to the German comrade mr\_magico for his articles on the SafeDisc protector , which put me on the right path.

Special thanks to Tim , who once prompted me to these articles.

Thanks to those people who in one way or another contributed to the appearance of this article.

Thanks to Bad\_guy for the fact that in the vastness of the Internet there is such a research portal as CrackL @ B.

Well, and of course, special thanks go to Bitfry for its help with the article and more.