Hacking SafeDisc Advanced and investigating SDAPI version 2

Study of the protective functions of SDAPI version 2 using the example of Need For Speed Most Wanted 1.2

Author : DillerInc

Content

Introduction

All of the following is for educational purposes only‼

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

As an example, we will consider 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 circum will become clear in the course of the story.

By the way, it is worth noting that EA GAMES is one of the few large companies that are now actively using this "reinforced" variation of the SafeDisc protector, protecting their game releases with it 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 mys this situation:

Methods for Obtaining Version Information

...speaking of "standard methods" for detecting the protector version, I mean searching the header of the PE file for the line "BoG_ *90.0&!! Yy>", followed by values of type Unsigned Long, which are the version. However, as I already mentioned, this line can simply be overwritten. In such cases, you can use the following method:

offset stxt774 + vsize stxt774 + 38h

...Where:

offset stxt774 - section offset in the file

vsize stxt774 - virtual section size

Based on the resulting offset, we read three values of the same Unsigned Long type. In the above ways, we get the standard view of the version:

* e.g. 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)

...by reading it as a resource from one of the files that are stored in the 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.

Hero portrait

In my opinion, the 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 By "everything else" I mean here the hinged essence of most protector mechanisms (import adapters, nanomites, etc.).

> **Quote:** Relayer, author of ExeCryptor
>
> The fact that hinged protectors are removed once and for all sooner or later 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 research and achieve a sufficiently strong "splicing" of the protector code with the application code.

It was with the help of the second version of SDAPI that Macrovision, which develops SD, took a rather serious step towards that very "fusion".

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

SDAPI of the second version is, 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 follo principle: using certain parameters and tables of values, the protector calculates the necessary values using these functions and returns them to the protected application. Among the returned result calculations, 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 to the source code of the protected application, thus form a single whole with the main code.

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

Code - SDAPI call variant

```
.text:006E7751 push ebx ; bMenu
 .text:006E7752 push 54052D06h ; protector values
.text:006E7757 push 54052D06h
.text:006E775C call sub_6E6F90 ; SDAPI call
.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 the SDAPI call, 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 more calls to the protecto which, so to speak, pave the way for obtaining a "useful" value that is necessary for the game itself .

This example in some way illustrates that such an approach as patching (NOP we name what belongs to the protector and insert our 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, following the path of such a patch, you will have - trust me - to spoil a good part of the code section with NOPs and the like.

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

- The first stage is a direct call to the protector API from the code section of the protected application. Before that, as a rule, two parameters are placed on the stack , which determine to some output result. Next comes the call, carried out by the usual CALL instruction.
- Stage two - the so-called. protector threshold, also in the code section.
- Stage three - the protector code itself, located in a separate module.

The last two require a pause and more detailed consideration.

Stage two - the threshold of the protector

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

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

I propose to immediately agree that we will consider the very first SDAPI call since the launch of the protected application. This call occurs at address 00885F60.

Code - Protector threshold

```
.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 significant ones at the moment. In any case, in the code section, you can easily find one of these vestibules (there are c
them) by the signature:

```
83h ECh 20h 56h 8Bh 35h
```

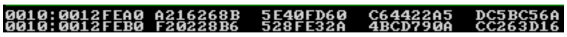
So, for starters, it’s worth focusing on the CryptParamsArray function (my name), where you can see the following code:

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 pair
.text:0065FC5E mov edx, dword_8F7658[edx*4] ; 2. DWORD pairs
....text:0065FC6B mov
dword ptr [ecx+4],                       10804201h
.text:0065FC72 mov
dword ptr [ecx+8],                       10804205h
...
...many XOR, IMUL commands
...
.text:0065FCEA retn 8
```

The main idea of this procedure is the encryption of 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 secti
addresses 008F76xx and 00900Axx there are two arrays, from which two DWORDs are selected with a certain degree of probability, participating in encoding the input data using the XOR and IMUL

For example, as those two DWORDs, let's take a pair of values - **F6B234CFh** and **A216268Bh** - as a result, we will get an encrypted array, which I called ParamsArray and which will look somethin



```
0010:0012FEA0 A216268B  5E40FD60  C64422A5  DC5BC56A
0010:0012FEB0 F20228B6  528FE32A  4BCD790A  CC263D16
```

We place the 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 is overwritten, while th
value of the pair remains unchanged and takes first place in the array - both will play a key role in the 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 pushed onto the stack when calling SDAPI,
we observe two values that were randomly above the top of the stack at that moment, as well as three constants (I marked them in blue in the listing). in the future they will determine the general
calculations in the body of the protector, therefore they are quite important milestones.

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

Protected file data section, protector structures

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

Following 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 can 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 informa
necessary for the functioning of those same SDAPI.

Elements in the table (structure instances), as a rule, are 40h pieces.

The size of the structure is 184h bytes.

I defined this structure as follows:

```
SDArgsStruc struc
      LeadIndd ? ; 67231341h - start marker
      SDArgsStrucSize dd ? ; 184h
      LeadIn_ dd ? ; 00000001h
      MagicArea Ordinal dd ?
      TrashDwordA dd ? ; 00
      HeapHead dd ?
      HeapArray dd 8 dup(?)
      SDRoutineAddrdd ?
      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 ?
      LeadOutdd ? ; 78822408h - end marker
 SDArgsStruc ends
```

I'll note right away, just in case, that fields of the Trash * type are just zero garbage bytes.

Analyzing this structure, it is worth noting that the highlight of each is the HeapHead double word and the HeapArray array, consisting of eight DWORDs. Now, however, we will not dwell on them, be make more sense to do so 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 begins with these bytes, a kind of marker.

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

Keeping in mind the peculiarities of the .data section, you can now process the necessary piece of data in IDA. As a result, we get something like this listing:

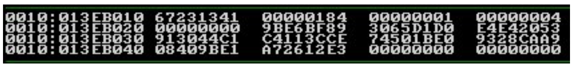
Code - Protector threshold

```
.text:00666E40 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 ; to the PROTECTOR module!
.text:00666E83 add esp, 10h
…
.text:00666E8C retn
```

The SDRoutineAddr and MagicAreaPtr fields are especially interesting here:

- 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 investigation, where very important information is stored.

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

  ```
  0010:013EB010 67231341  00000184  00000001  00000004
  0010:013EB020 00000000  9BE6BF89  3065D1D0  E4E42053
  0010:013EB030 913044C1  C4113CCE  74501BE0  9328CAA9
  0010:013EB040 08409BE1  A72612E3  00000000  00000000
  ```

  The beginning looks familiar, doesn't it?

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

  The double word following the 1 - MagicAreaOrdinal - is identical to the DataSectionOrdinal field of the corresponding structure in the data section. Based on the name, this value is a kind of ordinal, so you can, for example, conclude that the DataSectionOrdinal field of the last structure in the data section will be equal to 0000003Fh.

  Next in memory are nine DWORDs... more precisely, one + eight DWORDs, which are specifically related to the HeapHead and HeapArray fields, but which, however, we will return to a little l

  So.

  CALL ESI

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

  Stage three - library ~df394b.tmp

  The heart of 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 d from where it is then loaded into the address space of the main process using the LoadLibraryA function. After that, the address is obtained and calling the protector initialization function - O located in this library (for reference: the first character of the function name is a capital "O", not zero). During this initialization, some quantities are generated that we will soon have to deal

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

  The first thing that strikes you when scrolling through the code window in the debugger is definitely the size of the protector routines. Here, 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 code of the protective library in this case (in this version of the protector) contains a decent portion of the metamorph.

  > **Quote from** Ms-Rem
  >
  > ...metamorph tries to completely change the look of the code, while retaining the original algorithm of its work, for which he replaces the instructions with their synonyms, which in turn consist of one or more other instructions. Most of the new code produced by the metamorph is usually needed for the program to work, the proportion of junk code is very small.

  Examples of such obfuscation will certainly be c but first we need to draw up a kind of map of tl logical diagram that would illustrate the genera actions performed by the protector code.

  1. Creating a HeapArray
  2. Getting the value needed to decrypt the Params
  3. The decoding of the array ParamsArray 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. The second switch..case statement
8. End 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 array ParamsArray is passed to the ESI register. at the end of the CryptParamsArray procedure, it is overwritten. The second value from the pair is unchanged at th beginning of the array. It is with the help of it that we get / restore the first value of the pair. However, for these calculations, we need additional data, namely the HeapArray array.

Code - procedure 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 entering it, I would like to explain one point regarding the representation of various kinds of values / parameters in the p code.

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

They work according to the following principle - before using any value in the calculation, the latter is decrypted. After receiving the result, the latter is again encrypted and transmitted furth the code.

Code - procedure 66716F10

.text:66715658 call sub_667146F8 .text:6671565D mov ecx, eax .text:6671565F xor ecx, @@ **Value** and eax, 32884FD1h .text:6671566D not ecx .text:6671566F xor eax, ecx

...encryption or decryption of the @@Value value is performed using a special value - a mask - which is dynamic in this version of the protector, i.e. every time the protected application start mask is calculated. The purpose of such a trick is to confuse the researcher as much as possible, because with each new start he will encounter 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 ; 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, so we jump in both cases of conditional jumps in the above listing. Spe are interested in the memory cell at 667ADB0C, where the mask is located.

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

What will it give?

A bit of convenience when exploring. We can then each time replace the new value with the already known and familiar (once generated), so as not to be so confused among the encrypted d

So, let's go inside procedure 6672620A:

Code - procedure 6672620A

.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), each time calculating and returning the special values necessary for the operation of the security code whole.

In the first case, the procedure gets / restores the first value from the pair, with the help of which the ParamsArray array is then decrypted. But, as I already mentioned, for this it is necessar an array HeapArray. That is why, at the very beginning of procedure 6672620A, procedure 66725B7D quietly lurked, which is precisely this is engaged.

HeapArrays are extremely important elements of the SDAPI defense mechanism, as they act as a kind of intermediary in the most significant calculations. As part of every instance of the SD/ structure, any particular HeapArray 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 40h.

For each array, eight DWORD values are generated. The key element that takes part in the calculation of each member of the array is

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

The array in the "magic area" I named MagicArray .

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

In 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 obtained, .text:66725BAE jz loc_66725CDF ; then let's go out ...
```

If you remember, in addition to eight DWORDs (MagicArray array) tl one more DWORD in the heap. So this value (for example, 9BE6BF8 corresponds to the HeapHead field of the SDArgsStruc structure anc through the calculation process. It happens here:

Code - Procedure 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 - **5CAC5AC5h -** so I didn't really go into the details of calculating this value. What, however, cannot be the HeapArray array itself, the code for obtaining which must not only be parsed, but also then implemented independently. In general, get 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:66725C5F jz short loc_66725CC6 ; exit loop ...
.text:66725CC1 jmp loc_66725C32
```

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

Code - Procedure 66725B7D

```
.text:66725C64 call sub_66716F53 ; get the index of an 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 buffer offset
```

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

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

Going inside, you must immediately go to procedure 6672619F, because it does exactly what we need. 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
...
```

Now let's remember what I said about the metamorph. As an example, let's take the function 66725175, which I called GetDerivative (from the English derivative - derivative). The core of this funct following code:

Code - GetDerivative function

```
.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 library functions of the C language, in which simply incredible calculations occur.

In order not to spread my thoughts along the tree, and also due to the fact that bringing the obfuscated version of the code here will take a lot of space, I immediately want to give the code cleared metamorph:

I note that the GetDerivative function is found in the protector code in two forms, slightly different in the commands executed. I passed this moment using the WAY parameter, which determines wh calculation needs to be done in a particular case.

Code - GetDerivative function without metamorph

```
GetDerivative proc
arg @@ValueA:DWORD, @@ValueB:DWORD, @@WAY:DWORD
uses edi
                                 mov edi, @@ValueB
                                 mov eax, @@ValueA
                                 cmp dword ptr @@WAY, 00
                                 jne @@m1
        imul eax, edi
                                 jmp @@ret
@@m1:
        add eax, edi
@@ret:
                                 ret
GetDerivative endp
```

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 rub the tread code there a little.

You, the reader, will probably have a question: "How could he do this?"

In fact, everything is very simple - everything is known in comparison.

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

Function 667258E3 is also worth considering in detail. 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 operation will be performed, or, in other words, which code branch in procedure 667173DF will be performed, is determined by the DWORD, which is highlighted in bold in the listing. In this ca label corresponds to the XOR operation.

It follows that we can replace the entire procedure call 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 value of the actual Ordinal plays a the second - the actual index of the array element.

The calculations again involve procedures 667173DF, where the XOR operat performed, as well as procedures that use linear logical shift commands:

Code - procedures 66726055 and 667260C0

```
.text:667260A3 shl esi, cl
...or
.text:6672610E shr esi, cl
```

The shift counter is the constants that are sequentially placed on the stack by the PUSH command throughout the entire procedure 66725EA0:

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 get a derivative of the actual Ordinal in procedure 66725EA0
2. We get the derivative of the actual index of the array element in the procedure 66725EA0
3. We cross the resulting values with each other using the GetDerivative function
4. XOR result with corresponding DWORD from MagicArray

Next, the final result is written to a certain cell in memory, and everything is repeated anew 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 file “SD Procs examples.inc”, attached to the article.

Returning again to procedure 6672620A, which should return the first value from the pair required to decrypt the ParamsArray array, I want to note that here the main workhorses (meaning functior that have already been discussed above. As a parameter, as I already mentioned, the second value of the pair is passed, which is unchanged in the ParamsArray. It is then processed using the Heapl as well as the first two values from the HeapArray.

Again, take a look at the attached file.

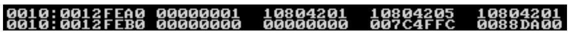
Having received the required value, we proceed to decrypt the array. If you recall, the array was encrypted using the XOR and IMUL commands.

So, the function 6672512A, aka GetDerivative, acts here as a multiplication operation with a sign, which in a simplified version clearly demonstrates this possibility. And the XOR operation is openly surface:

Code - procedure 6672F045

```
.text:6672F0D9 xor [esi+1Ch], eax
...and similarly further
```

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



```
0010:0012FEA0 00000001  10804201  10804205  10804201
0010:0012FEB0 00000000  00000000  007C4FFC  0088DA00
```

The two null DWORDs are the two protector parameters that were pushed onto the stack when the SDAPI was called. But now we will be more interested in the same three (more precisely, two) pro constants that we met in the CryptParamsArray function and which I marked in blue there.

If we recall our map of the area, now we should have the first switch..case statement on our nose, for which we first need to get the CASE value. Here we again encounter procedure 6672620A, in w 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 ParamsArray pointer was

; placed in the ESI register

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

Code - first statement SWITCH..CASE

```
.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
```

Its purpose is approximately the following - determining the type / dimension of the operands with which the necessary calculations will be performed. This moment can be demonstrated much more when we get to these calculations themselves, but for now I will just try to give a diagram of this operator.

So, there are eight possible options:

- CASE 0 and 1 - operand dimension: byte
- CASE 2 and 3 - operand dimension: word
- CASE 4 and 5 - dimension of the operand: double word
- CASE 6 and 7 - operand dimension: quadruple word

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

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

[ESI+0C] = 10804201

Before moving on 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 already be the value.

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

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 parameters are taken from those main parameters tl placed on the stack when calling SDAPI (in this case, these are two null DWORDs), the secondary ones are taken from the last two DWORDs of the ParamsArray array, which I mentioned as being ra above the top of the stack. In general, frankly, only the paramount ones are decisive here. Secondary ones are added apparently just “to the heap” (to give “volume” to calculations), and their prese not affect the final result in any way.

Here is one of the variants of such a forming code:

Code - procedure 6672EA14

```
.text:6672EA26 mov esi, [ebp+8] ; ESI - array pointer
```

```
...we get from the array secondary members for pairs
.text:6672EA58 mov edi, [esi+18h]
.text:6672EA5B mov edx, [esi+14h]
...we form 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 ; Save on the stack
.text:6672EA7E push ecx ; first couple
.text:6672EA7F lea ecx, [ebp-0F8h]
.text:6672EA85 call sub_667149DB
.text:6672EA8A push edi ; Save on the stack
.text:6672EA8B push ebx ; second pair
.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 having the next calculated CASE value in our pocket, we proceed to the second switch..case statement.

The second operator, unlike the first, is more extensive:

Code - procedure 6672D83E - 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 those operations/commands that will be performed on the intermediate results of calculations.

Take, for example, CASE with value 2:

Code - procedure 6672A883

```
.text:6672A898 push dword ptr [ebp+0Ch] ; Put the second pair on the stack
...
.text:6672A8A1 call sub_667284C4 ; We make calculations
...
.text:6672A8A6 push dword ptr [ebp+10h] ; Put the first pair on the stack
...
.text:6672A8B3 call sub_667284C4 ; We make calculations
...
.text:6672A8C7 call sub_66723219 ; We perform CASE calculations
...
.text:6672A8DA call sub_66728508 ; Making final calculations
```

Based on this, we can distinguish the following step-by-step scheme of variants of the second operator:

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

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

Code - procedure 667212DA

```
.text:667212DF call sub_6671F6E6 ; Getting the second result
.text:667212E4 mov ecx, [esp+4+arg_4]
.text:667212E8 mov esi, eax
.text:667212EA call sub_6671F6E6 ; Getting the first result
.text:667212EF add eax, esi ; Add them up
```

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

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

...and no CASE calculations.

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

Both selection operators are interconnected using the same dimension principle. Operands whose size is determined at the stage of the first operator are used further in the calculations determined l second operator.

This use of dimension can be most clearly seen, in my opinion, in the following passage:

Code - Step 4 of the above diagram

```
.text:667282E6 call dword ptr [eax] ; Getting the result of Step 3
.text:667282E8 movsx eax, al
.text:667282EB cdq ; We define the parameter “to the heap” in EDX
.text:667282EC push edx ; Pushing parameters on 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 CASEs 0 or 1 of the first statement.

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

For example, CASEs 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 ; Getting the result of Step 3
.text:66728406 movsx eax, ax ; forwarding with a 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 ; Getting 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
```

So we finally got to the final calculations. It was already mentioned above that there are two responsible functions: 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 array. One of the key points of each of the functions is getting a special value, called base, - it is with its participation that the last calculations are built, as a result of which the final value is obtained. If you carefully analyze the function code, it turns out that this base is calcu many as four times 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”, which can be seen by looking again at the attached file containing the implementations of these functions.

Opening the curtain...

Having received by this moment some idea of the protector's activities, we are finally ready to find out the real truth about the new protective mechanism - SDAPI second version. The truth, which w many who read this material, involuntarily smile in surprise.

The fact is that all the 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 word code capable of performing the necessary calculations and returning the required results is available in the most secure application, in its code section. It is enough to look again at one of the Thresh

Code - Protector threshold

```
.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] ; second parameter
.text:00666E91 mov eax, [esp+24h+arg_0] ; first parameter
.text:00666E95 push edx
.text:00666E96 push eax
.text:00666E97 call sub_6665B0 ; We do all the calculations
.text:00666E9C add esp, 8 ; Restoring the stack
.text:00666E9F pop esi
```

```
.text:00666EA0 add esp, 20h
.text:00666EA3 retn ; And we come out of the doorway
```

That is, if the address of the protector procedure - 6672F2F7 - is equal to zero, then we get to the coveted code.

However, in order to operate properly, this code, like the protector code, needs data. It is easy to guess that these data are HeapArray arrays, which should now be located somewhere at hand ... ex the data section of the protected file, in the SDArgsStruc structures.

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

Code - procedure 00656740

```
.text:00656740 mov eax, offset stru_8FACC8
.text:00656745 retn
```

The 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 recall the similarity of structures located in the data section and in the memory area on the heap, then we can safely assume that the HeapArray array will occupy a position in the SDArgsStruc of the data section corresponding to the position of the MagicArray array in the structure located on the heap. The same with the HeapHead field. And if the value of the HeapHead field will always b 5CAC5AC5h,

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

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

Looking at such a denouement, a quite 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 it should be destroyed. However, as practice shows, only H arrays from the data section are destroyed, while the code in the .text section often remains untouched.

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

Playing NFSMW versions 1.2 and 1.3 is exactly the case when the debug 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 debug handlers are destroyed, overwritten by 0CCh instructions. Moreover, the latter are not nanom but the usual commands of the third interruption. Obviously, in this situation, the removal of the protector becomes an order of magnitude more difficult, but nevertheless remains possible. And ther researcher himself, based on his knowledge of the material, decides exactly how he is going to do it.

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

'revisited' data section - Trigger Tables

The SDAPI security mechanism hides another trick that needs to be dealt with in order to allow an unpacked application to function properly. We will again have to deal with the data section of the p which contains the next type of structure that needs to be restored (some kind of guidance ...).

These structures are rumored to be referred to as Trigger tables .

Access 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 - an unknown value, which is the so-called. team shift
    dword - offset in the data section
```

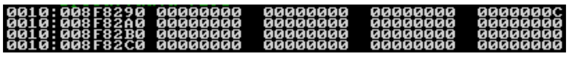
There are four of these structures in NFSMW, an appeal to one of which can be seen here:

Code - an example of calling 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 ; SDAPI call
.text:006E7743 push eax
.text:006E7744 push 899CC300h
.text:006E7749 call sub_6E6DD0 ; SDAPI call
```

The location of the structure is found 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:



```
0010:008F8290 00000000  00000000  00000000  0000000C
0010:008F82A0 00000000  00000000  00000000  00000000
0010:008F82B0 00000000  00000000  00000000  00000000
0010:008F82C0 00000000  00000000  00000000  00000000
```

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

Following the two MOV commands in the listing is the SDAPI call, which we skip. We are interested in the following protector function call at address 006E7749. The matter is that these structures ar in CASE with number 16h of the second operator (it is possible that also in the fifteenth variant). The CASE number of the first operator does not seem to play any role here. The code responsible fo structural data is, as it were, an add-on for numbers 16h and 15h.

Code - calling the required procedure in CASE16

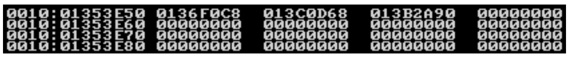
```
.text:66725E54 call dword ptr [eax+4] ; [EAX+04] = 66724396
```

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

Code - procedure 6671A297

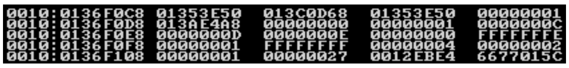
```
 .text:6671A334 mov eax, [esi+0Ch]
.text:6671A337 mov esi, [ebp-0A0h]
.text:6671A33D cmp esi, eax
.text:6671A33F jz loc_6671ABA7
```

A pointer to a memory area is placed in the EAX register, which contains pointers to the three desired structures:



```
0010:01353E50 0136F0C8  013C0D68  013B2A90  00000000
0010:01353E60 00000000  00000000  00000000  00000000
0010:01353E70 00000000  00000000  00000000  00000000
0010:01353E80 00000000  00000000  00000000  00000000
```

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

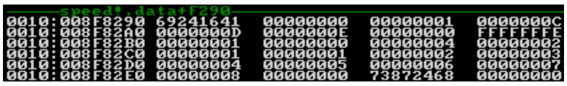


```
0010:0136F0C8 01353E50  013C0D68  01353E50  00000001
0010:0136F0D8 013AE4A8  00000000  00000001  0000000C
0010:0136F0E8 0000000D  0000000E  00000000  FFFFFFFE
0010:0136F0F8 00000001  FFFFFFFF  00000004  00000002
0010:0136F108 00000001  00000027  0012EBE4  6677015C
```

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 the same "known byte" follows, whi
determines the desired structure for us.

It should be noted right away that the representation of the structure on the heap is far from the ideal that should end up in the data section. Comparing both areas, we can conclude that the first fo
in the allocated memory are simply cut off. Further tidying up the structure was taken from the works of the notorious ReLOADeD team, for which special thanks to them.

The structure itself consists of twenty-three double words , the first and last of which are start and end markers, respectively. The first half of the structure, which is its defining part, appears before
almost ideal form in a heap, while the second half must be filled in by a peculiar method:



```
      speed!.data+F290
0010:008F8290 69241641  00000000  00000001  0000000C
0010:008F82A0 0000000D  0000000E  00000000  FFFFFFFE
0010:008F82B0 00000001  00000000  00000004  00000002
0010:008F82C0 00000001  00000001  00000002  00000003
0010:008F82D0 00000004  00000005  00000006  00000007
0010:008F82E0 00000008  00000000  73872468  00000000
```

So. Opens the structure start marker - 69241641h. In the first half, only the four-byte value FFFFFFFFh was replaced with a null DWORD. The second half, which starts at dword 00000027h in allocat
is simply replaced by dwords with values from 00000001h to 00000008h inclusive. The end marker ends the structure - 73872468h.

Likewise for other structures.

However, as I said, there are only four structures (four genuine accesses from the code section), but only three pointers in the EAX register. Take another look at the contents of the ESI register - th
DWORD is a pointer to the second structure. So, in one of the three structures in the heap, in the same way, there must be a pointer to the missing fourth one. At worst, you can, in principle, just lo
nearby memory for a "known byte" of the missing structure.

A couple of 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 peculiarities of the implementation of the considered protective mechanism.

In this protected application, those links that facilitate the final removal of the protector are preserved - debug handlers. However, this choice should not be considered unworthy. In my work, I tried
approximate scheme, to emphasize the most important points, which, on the one hand, should help you orient yourself, and on the other hand, leave the necessary space for independent maneuver
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
study this defense mechanism. If someone suddenly finds that I am fundamentally mistaken in something, and at the same time can argue their arguments in the right way, then it will be really gre
that this article is able to put on the true path those who undertook to read it. However, it should always be borne in mind that only personal, faithful initiative and perseverance can help to assimilat
necessary material and ultimately achieve positive results.

" *What is unclear should be clarified.*

That which is difficult to do must be done with great perseverance."

Confucius

Thanks

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 prompted me to read these articles.

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

Thanks to Bad_guy for having a research portal like CrackL@B on the web.

Well, and of course, special thanks are expressed to Bitfry for his help on the article and not only.

Download article "Hacking SafeDisc Advanced and investigating SDAPI version 2" in author's design + files.