21/09/2012

Unpackme I am Famous

safedisc reverse nanomites unpackme

Introduction

Because it is simply obvious that I have not been posting on this blog for a while, here is a post about Safedisc v3.
Last week I was studying this protection in deep, each component under IDA, but I accidentally broke my external hard drive by giving a shot in. I lost a lot of .idb from different games, softwares or malware, my personal toolz, unpackers, ...
So to smile again I decided to write about how to unpack this protection.
For those familiar with safedisc, the only interesting part will be Nanomites, restoring Imports or emulated opcodes is a joke when you know how older versions work.

Extra data

During introduction I talked about different components, they are placed at the end of the file.
The size of the target game is 1 830 912 bytes, but if we look IMAGE_SECTION_HEADER closely :

```
 Name      VirtSize   VirtAddr   SizeRaw    PtrRaw     Flags      Pointing Directories
-------------------------------------------------------------------------------------
 .text     00131148h  00401000h  00132000h  00001000h  60000020h
 .rdata    0002F497h  00533000h  00030000h  00133000h  40000040h  Debug Data
 .data     012CDCE8h  00563000h  0000D000h  00163000h  C0000040h
 .rsrc     0003289Eh  01831000h  00033000h  00170000h  40000040h  Resource Table
 stxt774   00002059h  01864000h  00003000h  001A4000h  E0000020h
 stxt371   00003358h  01867000h  00004000h  001A7000h  E0000020h  Import Table
                                                                  Import Address Table
```

If we sum the last Real Offset and Real Size of stxt371 section :

```
>>> 0x1A7000 + 0x4000
1748992
>>> hex(0x1A7000 + 0x4000)
'0x1ab000'
```

1 748 992 bytes != 1 830 912 bytes.
Clearly there is some extra data at the end of the file.
By looking the main executable under IDA, I was able to find an interesting sub that retrieves and extracts those datas.
First, here is the structure used for extra data :

```
struct extra_data
{
        DWORD sig_1;
        DWORD sig_2;
        DWORD num_file;
        DWORD offset_1;
        DWORD offset_2;
        DWORD unknow_1;
        DWORD unknow_2;
        BYTE  name[0xD];
};
```

sig_1 must always be set to 0xA8726B03 and sig_2 to 0xEF01996C
And after deleting all there (weak?) obfuscation, we can retrieve the following "pseudo code" to extract additionnal data.

```
do
{
        SetFilePointer(hFile, actual_pos, NULL, FILE_BEGIN);
        ReadFile(hFile, buff, 0x121, &bread, 0);
        key = actual_pos;

        for (i = 0; i < bread; i++)
        {
                key = key * 0x13C6A5;
                key += 0x0D8430DED;
                buff[i] ^= (((((key >> 0x10) ^ (key >> 0x8)) ^ (key >> 0x18)) ^ (key & 0xF
        }
        memcpy(&data, buff, sizeof(struct extra_data));
        print_data_info(&data);

        actual_pos += data.offset_1 + data.offset_2;

} while (data.sig_1 == 0xA8726B03 && data.sig_2 == 0xEF01996C);
```

Result :

```
Name : ~def549.tmp
Num : 1
Name : clcd32.dll
Num : 1100
Name : clcd16.dll
Num : 1100
```

```
Name : mcp.dll
Num : 1101
Name : SECDRV.SYS
Num : 2
Name : DrvMgt.dll
Num : 2
Name : SecDrv04.VxD
Num : 11
Name : ~e5.0001
Num : 0
Name : PfdRun.pfd
Num : 0
Name : ~df394b.tmp
Num : 0
```

As you can see we can extract a lot of files, and here is the algorithm to decypher it :

```
ptr = (BYTE*)GlobalAlloc(GPTR, data.offset_1);
SetFilePointer(hFile, actual_pos - data.offset_1, NULL, FILE_BEGIN);
ReadFile(hFile, ptr, data.offset_1, &bread, NULL);
if (bread != data.offset_1)
{
        printf("[-] ReadFile() failed\n");
        exit(0);
}
key = 0x8142FEA1;
int init_key;
init_key = 0x8142FEA1;
for (i = 0; i < bread; i++)
{
        key = init_key ^ 0x7F6D09ED;
        ptr[i] = (((((key >> 0x18) ^ (key >> 0x10)) ^ (key >> 0x8))) & 0xFF) ^ ptr[i];
        ptr[i] ^= key & 0xFF;
        init_key = init_key << 0x8;
        init_key += ptr[i];
}
```

Each component will be extracted into %temp% path, they got their own goal, we will not study all of them there is no interest.

~def549.tmp, a DLL, whose goal is to call different anti-debug technics (not interesting), check files on CD-ROM, ...
~e5.0001, an executable, this process will debug the main executable, for managing Nanomites.
PfdRun.pfd, No type, This file will de decyphered for computing instruction table used for emulated opcodes.
~df394b.tmp, another DLL, Load and decyph section from other DLL, and manage debug event for ~e5.0001 process.

I will not discuss more about all this stuff, by loosing all my idb I am bored to reverse (rename sub) again and again with all this shitty C++ stuff, you can find some fun crypto when they decypher pfd file or code section, rijndael modified, different xor operation, anyway let's continue !

Find OEP

This is the easiest part :

```
stxt371:018670A2                mov     ebx, offset start
stxt371:018670A7                xor     ecx, ecx
stxt371:018670A9                mov     cl, ds:byte_186703D
stxt371:018670AF                test    ecx, ecx
stxt371:018670B1                jz      short loc_18670BF
stxt371:018670B3                mov     eax, offset loc_1867113
stxt371:018670B8                sub     eax, ebx
stxt371:018670BA                sub     eax, 5
stxt371:018670BD                jmp     short loc_18670CD
stxt371:018670BF ; ---------------------------------------------------------------------------
stxt371:018670BF
stxt371:018670BF loc_18670BF:                           ; CODE XREF: start+13j
stxt371:018670BF                push    ecx
stxt371:018670C0                mov     ecx, offset loc_1867159
stxt371:018670C5                mov     eax, ecx
stxt371:018670C7                sub     eax, ebx
stxt371:018670C9                add     eax, [ecx+1]
stxt371:018670CC                pop     ecx
stxt371:018670CD
stxt371:018670CD loc_18670CD:                           ; CODE XREF: start+1Fj
stxt371:018670CD                mov     byte ptr [ebx], 0E9h
stxt371:018670D0                mov     [ebx+1], eax
```

This code will replace Module Entrypoint by a jump to Real OEP, so if you like using OllyDbg execute first instructions and put a breakpoint on that jump.
But you will encounter a "dead lock" problem, before jumping to real OEP, it decyphers sections, loads dll AND CreateProcess "~e5.0001" giving the pid of the game process as argument.
This process will load ~df394b.tmp aka SecServ.dll, all strings inside this dll are encrypted, we can decrypt all of them :

```
int decrypt_func_01(char *mem_alloc, char *addr_to_decrypt)
{
    DWORD count;
    DWORD key;
    char  actual;
```

```
    if (mem_alloc && addr_to_decrypt)
    {
        count = 0;
        key = 0x522CFDD0;
        while (1)
        {
            actual = *addr_to_decrypt++;
            actual = actual ^ (char)key;
            *mem_alloc++ = actual;
            key = 0xA065432A - 0x22BC897F * key;
            if (!actual)
                break;
            if (count != 127)
            {
                count++;
                continue;
            }
            return 0;
        }
        return 1;
    }
    else
        return 0;
}
```

Here is the result of all strings decyphered :

```
Addr = 667A9240 : drvmgt.dll
Addr = 667A9264 : secdrv.sys
Addr = 667A9298 : SecDrv04.VxD
Addr = 667A92BC : ALT_
Addr = 667A9C78 : Kernel32
Addr = 667AA71C : \\.\NTICE
Addr = 667AA73C : \\.\SICE
Addr = 667AA75C : \\.\SIWVID
Addr = 667AAB80 : .text
Addr = 667A9928 : Ntdll.dll
Addr = 667A9948 : Kernel32
Addr = 667AA3F0 : GetVersionExA
Addr = 667AA6BC : ZwQuerySystemInformation
Addr = 667AA6EC : NtQueryInformationProcess
Addr = 667AA780 : IsDebuggerPresent
Addr = 667AAB50 : ZwQuerySystemInformation
Addr = 667AADBC : ExitProcess
Addr = 667A99F8 : DeviceIoControl
Addr = 667A9A40 : CreateFileA
Addr = 667A9A64 : ReadProcessMemory
Addr = 667A9A8C : WriteProcessMemory
Addr = 667A9AB8 : VirtualProtect
Addr = 667A9AE0 : CreateProcessA
Addr = 667A9B08 : CreateProcessW
Addr = 667A9B30 : GetStartupInfoA
Addr = 667A9B58 : GetStartupInfoW
Addr = 667A9B80 : GetSystemTime
Addr = 667A9BA4 : GetSystemTimeAsFileTime
Addr = 667A9BD4 : TerminateProcess
Addr = 667A9BFC : Sleep
Addr = 667AB8C0 : WriteProcessMemory
Addr = 667AB8EC : FlushInstructionCache
Addr = 667AB918 : VirtualProtect
Addr = 667ABB90 : SetThreadContext
Addr = 667ABBB8 : GetThreadContext
Addr = 667ABBE0 : SuspendThread
Addr = 667ABB64 : FlushInstructionCache
Addr = 667ABB38 : WriteProcessMemory
Addr = 667ABC84 : ContinueDebugEvent
Addr = 667ABB0C : DebugActiveProcess
Addr = 667ABAE4 : WaitForDebugEvent
Addr = 667A99F8 : DeviceIoControl
Addr = 667ACF00 : System\CurrentControlSet\Services\VxD
Addr = 667ACF5C : cmapieng.vxd
Addr = 667ACF3C : StaticVxD
```

The most interesting things are DebugActiveProcess, ContinueDebugEvent, WriteProcessMemory, FlushInstructionCache, SetThreadContext.
As I said earlier this dll will be in charge of debugging the game process, it prevents debugging it with Olly or any Ring3 debugger.
The game process after calling CreateProcess will wait (WaitForSingleObject) signal that temp executable will attach to it and give it signal and continue to debug it, but if you are already debugging game process, WaitForSingleObject will never catch this signal.
All the code below can be found inside ~df394.tmp aka SecServ.dll :

```
.text:667250C1
.text:667250C1 loc_667250C1:                           ; CODE XREF: sub_66724FDE+D5j
.text:667250C1                 push    0FFFFFFFFh      ; dwMilliseconds
.text:667250C3                 push    edi             ; hHandle
.text:667250C4                 call    ds:WaitForSingleObject
.text:667250CA                 push    [ebp+hObject]   ; hObject
.text:667250CD                 mov     [ebp+return_value], eax
.text:667250D0                 call    esi ; CloseHandle
```

```
.text:667250D2                 push    edi             ; hObject
.text:667250D3                 call    esi ; CloseHandle
.text:667250D5                 cmp     [ebp+return_value], 0 ; WAIT_OBJECT_0
.text:667250D9                 pop     edi
.text:667250DA                 pop     esi
.text:667250DB                 jz      short exit_func
.text:667250DD                 call    ebx ; GetLastError
.text:667250DF                 call    Exit_Process
.text:667250E4
.text:667250E4 exit_func:                              ; CODE XREF: sub_66724FDE+11j
.text:667250E4                                         ; sub_66724FDE+20j ...
.text:667250E4                 pop     ebx
.text:667250E5                 leave
.text:667250E6                 retn
.text:667250E6 sub_66724FDE    endp
.text:667250E6
```

If you want to use OllyDBG, put a breakpoint on WaitForSingleObject call, and modify argument TIMEOUT to something different than INFINITE, and change ZF flag during the test of the return value.

Nanomites

Now the fun stuff can start, if you followed what I said, you can continue to debug game process, but at a moment you will encounter problem like follow :

```
.text:004519FC                 call    ds:dword_5331B0 ; kernel32.IsBadReadPtr
.text:004519FC ; ---------------------------------------------------------------------------
.text:00451A02                 db 0CCh
.text:00451A03                 db 0CCh ;
.text:00451A04                 db 0CCh ;
.text:00451A05                 db 0CCh ;
```

What are doing these 0xCC (int 3) aka Trap to Debugger or software breakpoint after a call to a kernel32 API ?
It's a well known technique, instructions are replaced by this opcode and informations about the removed opcode is stored in a table. (Remember pfd file ?)
Then, by using self-debugging, when one of these breakpoints is hit, the debugging process will handle the debug exception, and will look up certain information about the debugging break.

> Is it a Nanomite ?
> Yes ! So I have to emulate the removed opcode
> And restore the context of the thread correctly

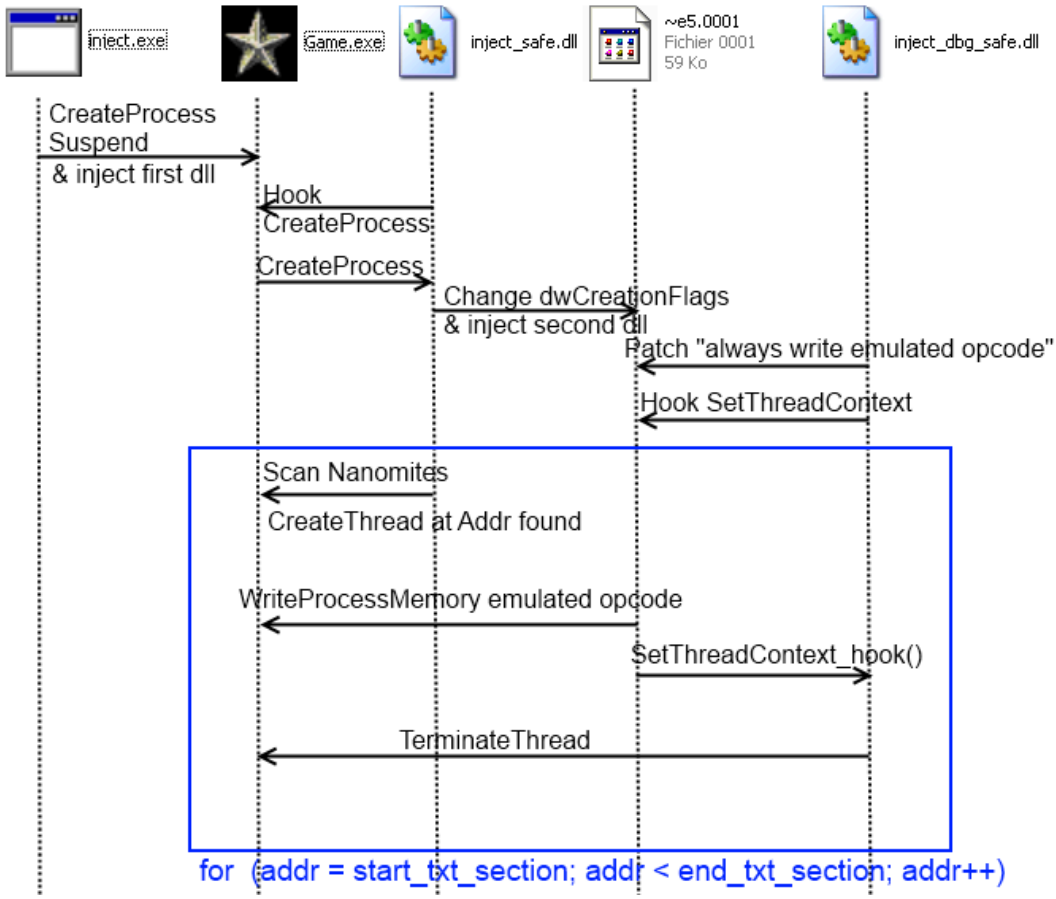
But the problem is, if Nanomites are called several times, it can impact a little the performance, right ? (Not anymore today), but Safedisc decided to count how much time a Nanomite is executed, and if this Nanomite is executed too much time, it will restore the replaced opcodes by writting it inside the debugged process.
So if we want to fix theses Nanomites, we just have to patch a branch instruction that say : "This nanomites has been executed too much time, restore opcode !", and scan txt section of game process to find all the nanomites, call them, and the debugger process will restore all the removed opcode :).

How To

When unpacking (real?) protection you need to write cool toolz, here are all the steps that I did :

> Create Game process in suspended state
> Inject a first (malicious?) dll into it and continue execution
> This first dll will setup an Hook on CreateProcessA, the goal of this task is when the debugger process (~e5.0001) will be created, it will change the dwCreationFlags to CREATE_SUSPENDED and inject a second dll in it.
> A second hook from the first dll will be setup on GetVersionExA to gain execution just after the jump to Real OEP.
> Once GetVersionExA is called, we scan txt section and look for 0xCC and for each one it create a thread at the address of the nanomites.
> The second dll will patch the branch condition for WriteProcessMemory the emulated opcode and hook SetThreadContext for terminating the thread in question and not continue his execution.

Need a diagram ?



I encountered a little problem during those operation, if we create a thread at an addr containing 0xCC followed by nop operation (0x90), Safedisc debugger crashes or emulates shit...
Visual Studio uses 0xCC, 0x90 and 0x00 opcode for padding, don't ask me why they don't just use only 0x00, I don't know.
Just so you know, if you don't provide the full path of these dll while you are injecting it, the first dll must be placed in the folder of the game process, and the second one in %temp% path, because debugger process is extracted and executed here.

You can find the branch instruction inside ~def394.tmp (SecServ.dll) at addr 0x6678F562 :

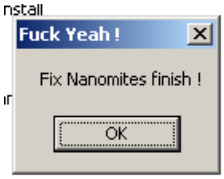
```
.txt5:6678F562                  cmp     ax, 1
.txt5:6678F566                  jnz     not_write_process_memory
```

Result

Just some debug information :

```
---
Process id : 894
EventCode : 1
Exception Code : 80000003
Exception Addr : 40170F
---
[+] GetThreadContext(0xB8, 0x635080); return_addr = 66733C55
lpContext->EIP = 7C91120F
[+] WriteProcessMemory(0x5C, 0x40170F, 0x61F58C, 0x2, 0x61F0F0); return_addr = 6672BA45
85 C0
[+] SetThreadContext(0xB8, 0x635080); return_addr = 66733C23
lpContext->EIP = 40170F
---
```

As you can see at address 0x40170F, an event occured 0x1 -> EXCEPTION_DEBUG_EVENT and his code 0x80000003 (EXCEPTION_BREAKPOINT), so the debugger process replaces the 0xCC 0xCC by 0x85 0xC0 -> "test eax, eax", and try to SetThreadContext but we hooked it to terminate the thread.



Restoring Imports

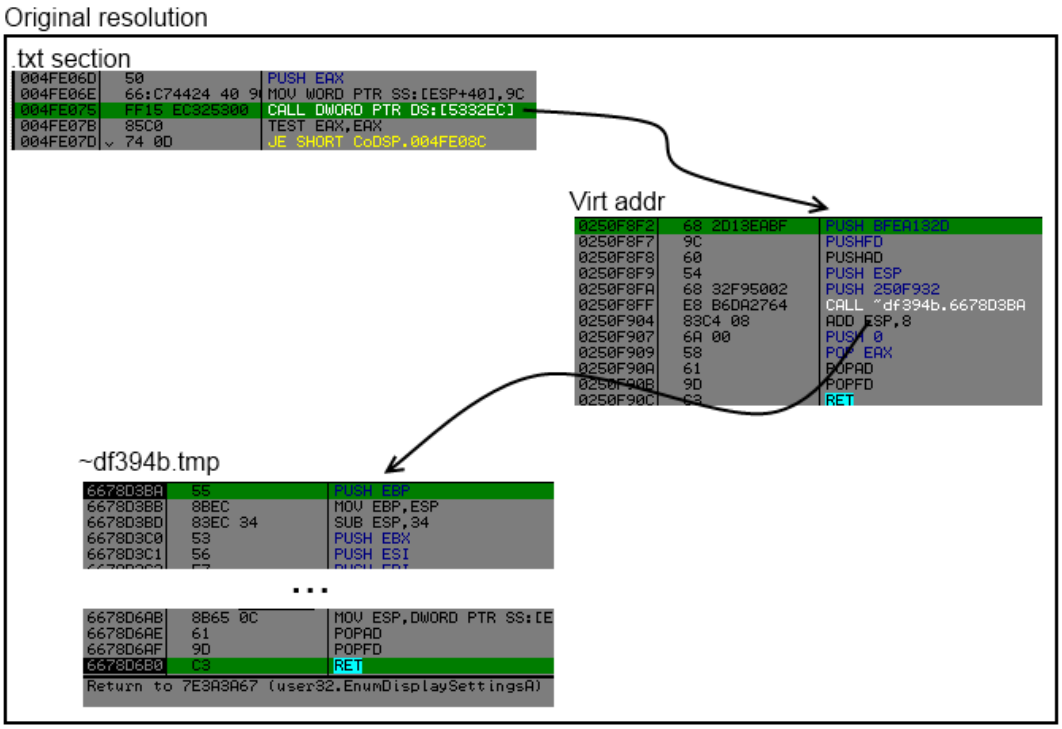
Like the previous version import points to some virtual address where the code calls routine to find the correct import.
By using algo against itself we can resolve all correct address of imports.
Inside txt section we can find different type of call to imports :

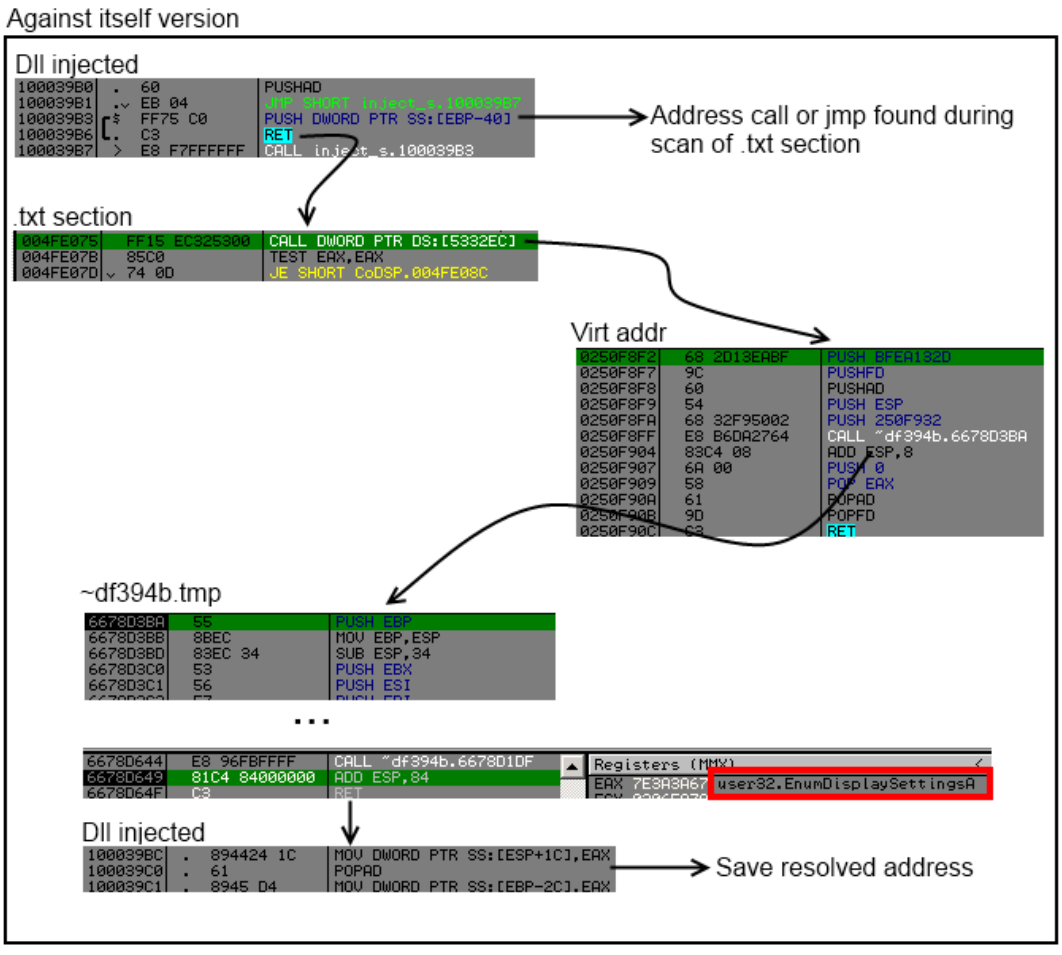
call dword ptr[virtual_addr]
jmp dword ptr[virtual_addr]
jmp section Stxt774

The idea is simple, scan .txt section look for call dword ptr or jmp dword ptr or jmp section Stxt774, hook the function that resolve the api and get the result and save into into a linked list.
This function in question is in ~df394b.tmp :

```
.txt:6678D644                   call    resolve_api
.txt:6678D649                   pop     ecx
.txt:6678D64A                   pop     ecx
```

Just replace the pop ecx, by "add esp, X; ret" and get the result into register eax.





BUT ! Sometimes by calling the same virtual_addr but from other location it don't resolve the same API address.

```
API (0x7E3AC17E) has rdata.0x53327C (txt.0x51A656)  rdata.0x53327C (txt.0x454509)  rdata.0
API (0x7E39869D) has rdata.0x53329C (txt.0x51A686)  rdata.0x53329C (txt.0x50B64E)  rdata.0
```

As you can see the address in rdata 0x53327C, can resolve different API when it is called from different locations (txt address).
To fix it, it's very simple we reorder the linked list according to the api address, and choose one rdata for each call, and we will change value of the call or jmp dword ptr at txt address for each entry of an api.

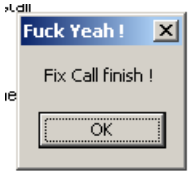
After reorder

Output after reordering :

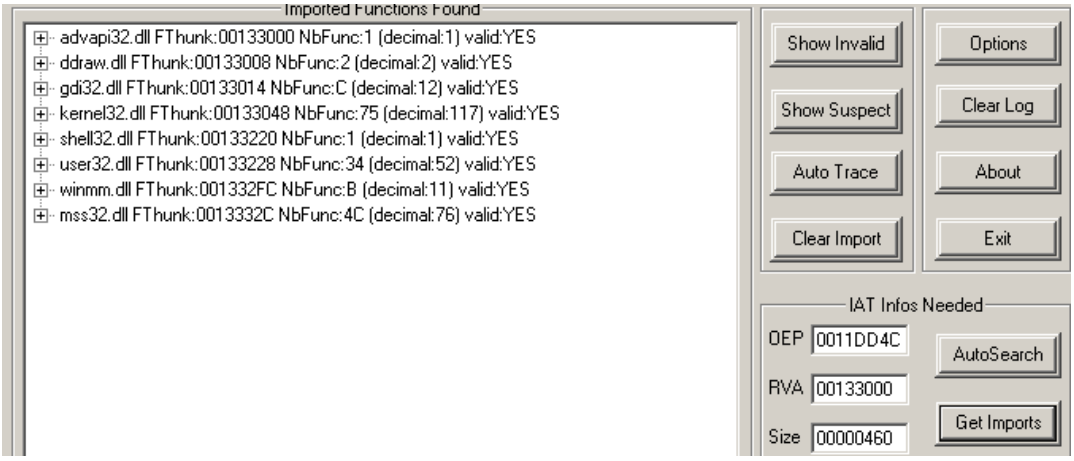
```
API (0x7E3AC17E) has rdata.0x53327C (txt.0x51A656)  rdata.0x53327C (txt.0x454509)  rdata.0
API (0x7E39869D) has rdata.0x53329C (txt.0x51A686)  rdata.0x53329C (txt.0x50B64E)  rdata.0
```

We can now write back into rdata addr the real adress of the api and fix the call or jmp at adress in txt section, to point to the good rdata address.
Now you can look with ImportRec and see that all imports are restored correctly :)

To fix jmp section Stxt774, we just have to replace the jmp by a call dword ptr[rdata], but wait jmp stxt774 is 5 bytes and we need 6 bytes to change it to call dword ptr, don't worry, after resolving the api and ret to it, the api will return at jmp stxt774 + 6, so there is enough place.



And Import Reconstructor is happy (Invalid imports 0) :



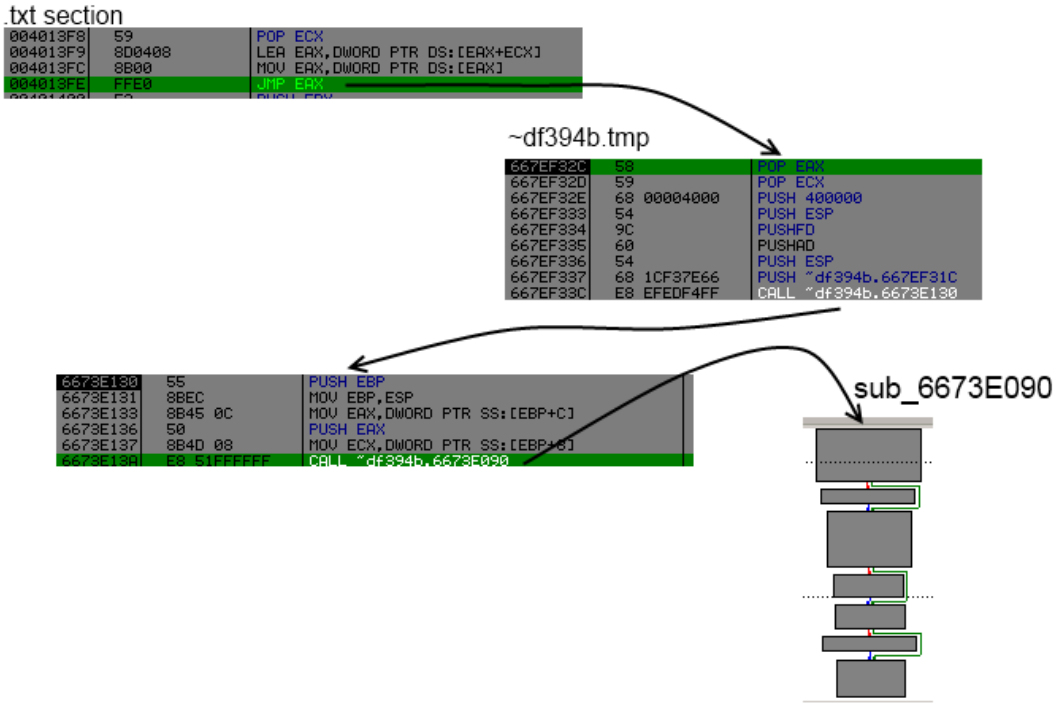
Emulated opcodes

After fixing Nanomites and restoring imports, I encounter a last problem.

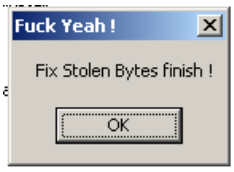
```
.text:00404909                push    ecx
.text:0040490A                push    eax
.text:0040490B                call    sub_4013F3
.text:0040490B sub_404909     endp ; sp-analysis failed
.text:0040490B
```

```
.text:004013F3                mov     eax, 1E1Bh
.text:004013F8                pop     ecx
.text:004013F9                lea     eax, [eax+ecx]
.text:004013FC                mov     eax, [eax]
.text:004013FE                jmp     eax
```

This code will just compute an address in txt section, get the value pointed by this address and jump to it. The jump destination is an address from ~df394b.tmp.



The goal of sub 0x6673E090 is simply to check from where it has been called, lookup in a table of emulated opcodes and restore it.
Here only one emulation is performed then it will write original opcode back.
Like for restoring imports, we find each reference to the sub 0x00404909, setup an hook at the end of the sub 0x6673E09, call each reference, and emulated opcodes will be restored automatically :)



Conclusion

Safedisc v3 is really not difficult, you can find the source of all my codes at the end of this post.
I will go back to school project, hopefully graduating this year :)

Sources

Injector

inject.asm

First dll

linklist.h
linklist.cpp
main.cpp

Second dll

main.cpp

15/09/2011

C and C

reverse red alert safedisc

Introduction

If you are bored of my tutorial on safedisc just go to Bonus section for laughing.
Today we are going to see safedisc version 2 (in this case i worked on v2.05.030).

Detection

No more *.icd file, the loader is now integrated into the main executable.
The signature is the same : "BoG_ *90.0&!! Yy>" followed by 3 unsigned integers : the version, subversion an revision number.

Anti Debug

In this case we are using ring 3 debugger, the tricks are the same than in safedisc version 1 :

IsDebuggerPresent()
PEB!IsDebugged
ZwQueryInformation (ProcessInformationClass = ProcessDebugPort)

You can use Phant0m plugin or follow what i did for safedisc 1.

Find OEP

Like I said in "detection" section, the loader is now integrated in the main executable, so we must find real oep after safedisc decyphering stuff.
I don't know if it's a good way to find it, but i set an hardware breakpoint on GetVersion, and look around for finding where am i. (GetVersion is one of the first called api).
But after watching some disas when i opened my ra2.exe into OllyDbg :

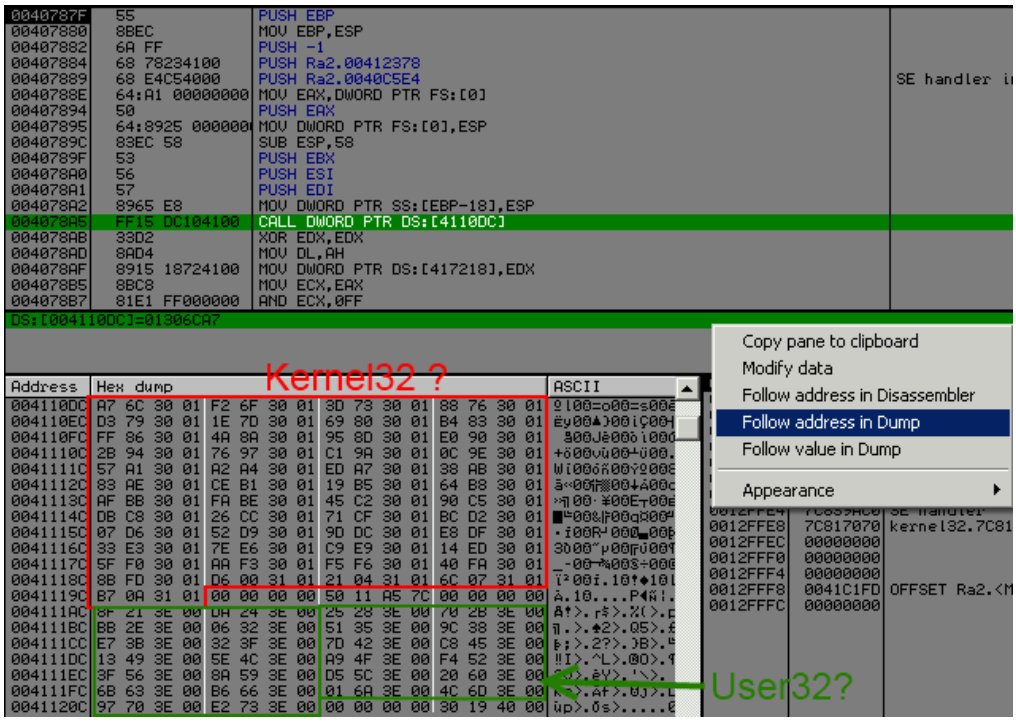
```
0041C1FD >  55              PUSH EBP
0041C1FE    8BEC            MOV EBP,ESP
0041C200    60              PUSHAD
0041C201    B8 7BC24100     MOV EAX,Ra2.0041C27B
0041C206    2D FDC14100     SUB EAX,OFFSET Ra2.<ModuleEntryPoint>
0041C20B    0305 7CC24100   ADD EAX,DWORD PTR DS:[41C27C]
0041C211    C705 FDC14100 E>MOV DWORD PTR DS:[<ModuleEntryPoint>],0E9
0041C21B    A3 FEC14100     MOV DWORD PTR DS:[41C1FE],EAX
0041C220    68 C9C04100     PUSH Ra2.0041C0C9                           ; ASCII "USER32.
0041C225    68 BBC04100     PUSH Ra2.0041C0BB                           ; ASCII "KERNEL3
0041C22A    68 09C04100     PUSH Ra2.0041C009
0041C22F    68 9BC04100     PUSH <&KERNEL32.GetModuleHandleA>
0041C234    A0 21C04100     MOV AL,BYTE PTR DS:[41C021]
0041C239    3C 01           CMP AL,1
0041C23B    74 07           JE SHORT Ra2.0041C244
0041C23D    B8 00000000     MOV EAX,0
0041C242    EB 03           JMP SHORT Ra2.0041C247
0041C244    8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0041C247    50              PUSH EAX
0041C248    E8 33000000     CALL Ra2.0041C280
0041C24D    83C4 14         ADD ESP,14
0041C250    83F8 00         CMP EAX,0
0041C253    74 1C           JE SHORT Ra2.0041C271
0041C255    C705 FDC14100 C>MOV DWORD PTR DS:[<ModuleEntryPoint>],0C2
0041C25F    C705 FEC14100 0>MOV DWORD PTR DS:[41C1FE],0C
0041C269    50              PUSH EAX
0041C26A    A1 ABC04100     MOV EAX,DWORD PTR DS:[<&KERNEL32.ExitProcess>]
0041C26F    FFD0            CALL EAX
0041C271    61              POPAD
0041C272    5D              POP EBP
0041C273    EB 06           JMP SHORT Ra2.0041C27B
0041C275    72 16           JB SHORT Ra2.0041C28D
0041C277    61              POPAD
0041C278    1360 0D         ADC ESP,DWORD PTR DS:[EAX+D]
0041C27B  - E9 FFB5FEFF     JMP Ra2.0040787F
```

Look at 0x0041C271, popad (we restore all our registers), pop ebp, jmp to 0x0041C27B, and again a jump to ... OEP.
After setting an hbp at this address we are here :

```
0040787F    55              PUSH EBP
00407880    8BEC            MOV EBP,ESP
00407882    6A FF           PUSH -1
00407884    68 78234100     PUSH Ra2.00412378
00407889    68 E4C54000     PUSH Ra2.0040C5E4
```

Fix redirect call

In this version of safedisc 2, it's the same difficulty in my opinion, but it's take more time to write call fixer, because there are some funny anti dump tricks.
Let's see the call jsut after oep :



It looks like version 1, but they set up new "protection" for fixing it.
I will not get into detail but explain you what they did and how to defeat it.

```
01306CA7    68 B413EABF     PUSH BFEA13B4
01306CAC    9C              PUSHFD
01306CAD    60              PUSHAD
01306CAE    54              PUSH ESP
01306CAF    68 E76C3001     PUSH 1306CE7
01306CB4    E8 3729D10E     CALL ~df394b.100195F0
01306CB9    83C4 08         ADD ESP,8
01306CBC    6A 00           PUSH 0
01306CBE    58              POP EAX
01306CBF    61              POPAD
01306CC0    9D              POPFD
01306CC1    C3              RET
```

Routine are exactly the same than my previous post about version 1 subersion 41, they compute the addr and then ret to it.
BUT ! now they check on the stack from where you have called this routine, and if it's an unknow address, it will compute a random api address.
So when we will want to fix import, we will have to scan code section find 0xFF15 (call dword [rdata]) and push the addr + 6.
I will spare you from crash, because it's not the only protection ... after making a call fixer, i encoutered a second problem, you can have several call to the same offset to rdata section, and in function of where you called it it will compute different api address :

```
0040521E   CALL DWORD PTR DS:[4110F4]              ; Return to 7C91FE01 (ntdll.RtlGetLastW
004078F3   CALL DWORD PTR DS:[4110F4]              ; Return to 7C812FAD (kernel32.GetComma
```

As you can these 2 calls call the same routine, but ret on different api.
So for our call fixer we will have to create a temporary kernel32 and user32 table, and fix each call dword ptr to call the good index :

- Make a temporary dword table size of nb api, fill with null at start
- Scan code section, compute address api with safedisc routine
- Hook return safedisc routine, and put address into a register
- If computed address is not known find place (null bytes) into temporary table, and fix destination of the call dword ptr with new index into the table.
- If computed adress is already known and index into the table didn't change, do nothing, else fix destination of the call dword ptr.
- When done juste memcpy new table

How to hook safedisc routine ?

```
100183DF    FF15 44800310   CALL DWORD PTR DS:[<&KERNEL32.SetEvent>]     ; kernel32.SetEve
100183E5   /EB 07           JMP SHORT ~df394b.100183EE
100183E7    8BDB            MOV EBX,EBX
100183E9   /70 06           JO SHORT ~df394b.100183F1
100183EB   |90              NOP
100183EC   /71 03           JNO SHORT ~df394b.100183F1
100183EE  ^\EB F7           JMP SHORT ~df394b.100183E7
100183F1    8B65 0C         MOV ESP,DWORD PTR SS:[EBP+C]
100183F4    61              POPAD
100183F5    9D              POPFD
100183F6    C3              RET
```

We will remplace the JMP SHORT ~df394b.100183EE, to jump to our code (in my case i used a dll, why because dll injection FTW !, no it's a simply way to fuck this anti dump, and not assemble code in ollydbg each time, yes safedisc 2 call fixer is longer than safedisc 1).

So if you are following my tuts and code some shit, your dump will crash... we need to fix 3 more tricks.
The problem is mov REG, [rdata] ... call REG :

```
References in Ra2:.text to 00411084..00411087, item 3
Address=0040E719
Disassembly=MOV ESI,DWORD PTR DS:[411084]
Comment=DS:[00411084]=01302435

References in Ra2:.text to 004110A4..004110A7, item 0
Address=0040350E
Disassembly=MOV EBP,DWORD PTR DS:[4110A4]
Comment=DS:[004110A4]=01303E8D

References in Ra2:.text to 00411124..00411127, item 0
Address=0040310F
Disassembly=MOV EBX,DWORD PTR DS:[411124]
Comment=DS:[00411124]=0130A7ED

References in Ra2:.text to 00411130..00411133, item 0
Address=00402CF5
Disassembly=MOV EDI,DWORD PTR DS:[411130]
Comment=DS:[00411130]=0130B1CE
```

So we will need to fix mov edi, [rdata], ebp, ebx, edi, see my call fixer for explanation but same thing like before.
An another tricks is jmp to Stxt774 a section found inside the binary.

```
0040C488  - E9 98EB0000     JMP Ra2.0041B025

....


0041B025    53              PUSH EBX
0041B026    E8 00000000     CALL Ra2.0041B02B
0041B02B    870424          XCHG DWORD PTR SS:[ESP],EAX
0041B02E    9C              PUSHFD
0041B02F    05 D5FFFFFF     ADD EAX,-2B
0041B034    8B18            MOV EBX,DWORD PTR DS:[EAX]
0041B036    6BDB 01         IMUL EBX,EBX,1
0041B039    0358 04         ADD EBX,DWORD PTR DS:[EAX+4]
0041B03C    9D              POPFD
0041B03D    58              POP EAX
0041B03E    871C24          XCHG DWORD PTR SS:[ESP],EBX
0041B041    C3              RET
```

This routine will just compute an addr to rdata section, and ret to it, so for fixing this we will have to replace jmp by call [rdata], but wait jmp addr it's only 5 bytes and call [rdata] is equals 6 bytes, but don't worry when we will ret from this routine we will ret at 0x0040C488 + 6 (in the example above), so we have enough place to fix it.
And the last tricks is similar than this one seen below, it is jmp [rdata], and we have enough place too for fixing it too.

Now you have all the pieces to understand my fix import dll :

```
.386
.model flat,stdcall
option casemap:none

include         \masm32\include\kernel32.inc
includelib      \masm32\lib\kernel32.lib


.const
; user32 addr rdata start
user32_rdata    =       004111ACh
; kernel32 addr rdata start
kernel32_rdata  =       0041105Ch

code_start              =       00401000h
code_end                =       00411000h
code_size               =       00010000h

; Addr to patch for our hook
addr_to_patch   =       100183E5h

start_rdata             =       00411000h
size_rdata              =       00003000h

stxt_section    =       0041B000h
stxt_end                =       stxt_section + 00001000h

.data?
OldProtect              dd      ?
addrcall                dd      9 dup (?)
nbwrite                 dd      ?
addrapi                 dd      ?
not_real                dd      ?
kernel32_table  dw      051h dup (?,?,?,?)
user32_table    dw      01Ah dup (?,?,?,?)

.code

LibMain proc parameter1:DWORD, parameter2:DWORD, parameter3:DWORD
```

```
        pushad
        pushfd

        ; Set full access for beeing able to fix jmp to call and rdata section
        invoke VirtualProtect, addr_to_patch, 5, 40h, addr OldProtect
        invoke VirtualProtect, code_start, code_size, 40h, addr OldProtect
        invoke VirtualProtect, start_rdata, size_rdata, 40h, addr OldProtect

        ; Set hook
        mov     eax, addr_to_patch
        mov byte ptr [eax], 0E9h
        mov [eax + 1], Hook - addr_to_patch - 5

; Scan section text
SearchCall:
        mov eax, code_start

Scantext:
        inc eax
        cmp eax, code_end
        jae end_scan
        cmp word ptr[eax], 015FFh       ; call [rdata]
        je call_type1
        cmp word ptr[eax], 025FFh       ; Jmp [rdata]
        je call_type1
        cmp word ptr[eax], 0358Bh       ; MOV ESI, ...
        je call_type1
        cmp word ptr[eax], 02D8Bh       ; MOV EBP, ...
        je call_type1
        cmp word ptr[eax], 01D8Bh       ; MOV EBX, ...
        je call_type1
        cmp word ptr[eax], 03D8Bh       ; MOV EDI, ...
        je call_type1
        cmp byte ptr[eax], 0E9h         ; Jmp Stxt774
        je jmp_type
        jmp Scantext

end_scan:
        ; copy temporary table to original position
        mov ecx, 051h
        lea esi, kernel32_table
        mov edi, kernel32_rdata
        rep movsd
        mov ecx, 01Ah
        lea esi, user32_table
        mov edi, user32_rdata
        rep movsd
    mov eax, 1
        popfd
        popad
    retn

; Our hook function edx = addr of the resolved api
Hook:
        mov [addrapi], edx
        mov esp, dword ptr ss:[ebp + 0Ch]
        popad
        popfd
        pop edi
        pop edi
        retn

; fix jump
jmp_type:
        mov edx, [eax + 1]
        lea     edx, [edx + eax + 5]
        .if edx >= stxt_section && edx  <= stxt_end
                push next_jmp
                jmp edx
        .endif
        jmp Scantext
next_jmp:
        xor ecx, ecx
        mov     ebx, [addrapi]
        ; is a jump to user32 or kernel32 rdata
        .if ebx >= 07E000000h
                lea edi, user32_table
                mov esi, user32_rdata
        .else
                lea edi, kernel32_table
                mov esi, kernel32_rdata
        .endif
        .while (dword ptr [edi + ecx * 4] != 0)
                .if dword ptr [edi + ecx * 4] == ebx
                        mov word ptr[eax], 015FFh
                        mov edx, esi
                        lea edx, [edx + ecx * 4]
                        mov dword ptr [eax + 2], edx
                        jmp @F
                .endif
                inc ecx
        .endw
        mov dword ptr [edi + ecx * 4], ebx
```

```
        mov word ptr[eax], 015FFh
        mov edx, esi
        lea edx, [edx + ecx * 4]
        mov dword ptr [eax + 2], edx
@@:
        jmp Scantext

; We will scan if the ptr to rdata section
; is in kernel32 table or go check if it is
; in user32 table
call_type1:
        mov edx, [eax + 2]
        mov     ebx, kernel32_rdata

next_kernel32:
        cmp dword ptr [ebx], 0
        je user32
        cmp ebx, edx
        je is_kernel32
        add ebx, 4
        jmp next_kernel32

; We found it, let's fix this
is_kernel32:
        push next_scan
        mov ecx, eax
        add ecx, 6
        push ecx
        jmp dword ptr [ebx]
next_scan:
        xor ecx, ecx
        mov     ebx, [addrapi]
        .while (dword ptr [kernel32_table + ecx * 4] != 0)
                .if dword ptr [kernel32_table + ecx * 4] == ebx
                        mov edx, kernel32_rdata
                        lea edx, [edx + ecx * 4]
                        ; fix index
                        mov dword ptr [eax + 2], edx
                        jmp @F
                .endif
                inc ecx
        .endw
        mov dword ptr [kernel32_table + ecx * 4], ebx
        mov edx, kernel32_rdata
        lea edx, [edx + ecx * 4]
        mov dword ptr [eax + 2], edx
@@:
        jmp Scantext

; Same thing like below but for user32
user32:
        mov edx, [eax + 2]
        mov     ebx, user32_rdata

next_user32:
        cmp dword ptr [ebx], 0
        je Scantext
        cmp ebx, edx
        je is_user32
        add ebx, 4
        jmp next_user32

is_user32:
        push next_scan_user32
        mov ecx, eax
        add ecx, 6
        push ecx
        jmp dword ptr [ebx]
next_scan_user32:
        xor ecx, ecx
        mov     ebx, [addrapi]
        .while (dword ptr [user32_table + ecx * 4] != 0)
                .if dword ptr [user32_table + ecx * 4] == ebx
                        mov edx, user32_rdata
                        lea edx, [edx + ecx * 4]
                        mov dword ptr [eax + 2], edx
                        jmp @F
                .endif
                inc ecx
        .endw
        mov dword ptr [user32_table + ecx * 4], ebx
        mov edx, user32_rdata
        lea edx, [edx + ecx * 4]
        mov dword ptr [eax + 2], edx
@@:
        jmp Scantext

LibMain endp

end LibMain
```

And the make.bat :

```
@echo off

if exist "inject.obj" del "inject.obj"
if exist "inject.dll" del "inject.dll"

\masm32\bin\ml /c /coff "inject.asm"
if errorlevel 1 goto end

\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL "inject.obj"
if errorlevel 1 goto end

:end
pause
```

For injecting the dll, i used a olly plugin writtent by baboon, big thanks to him :]
Then after injecting dll, dump process, use importrec, and enjoy !

Bonus

And now ! the lulz part :]
In the folder of red alert 2, we can see ra2.exe (protected by safedisc) and game.exe (not protected), ra2.exe will simply launch game.exe. So removing safedisc was a long solution for breaking it.
But they used lame protection to watch if the process was launched by ra2.exe or not.

```
004916E4  |.  50            PUSH EAX                                                ;
004916E5  |.  6A 00         PUSH 0                                                  ;
004916E7  |.  6A 00         PUSH 0                                                  ;
004916E9  |.  FF15 30527800 CALL DWORD PTR DS:[<&KERNEL32.CreateMutexA>]            ;
004916EF  |.  8BF0          MOV ESI,EAX
004916F1  |.  FF15 F0517800 CALL DWORD PTR DS:[<&KERNEL32.GetLastError>]             ;
004916F7  |.  3D B7000000   CMP EAX,0B7
004916FC  |.  0F94C3        SETE BL
004916FF  |.  85F6          TEST ESI,ESI
00491701  |.  74 07         JE SHORT game2.0049170A
```

What !? They are just checking if a mutex has been created or not, ok let's nop this.
A second check is :

```
0049173A  |.  68 20037C00   PUSH game2.007C0320
0049173F  |.  6A 01         PUSH 1
00491741  |.  6A 02         PUSH 2
00491743  |.  FF15 28527800 CALL DWORD PTR DS:[<&KERNEL32.OpenEventA>]
00491749  |.  8BF0          MOV ESI,EAX
0049174B  |.  85F6          TEST ESI,ESI
0049174D  |.  74 22         JE SHORT game2.00491771
```

Ok a check if OpenEventA worked or not let's nop it too.

Kind of triggers ?

But there is another problem if we fix game.exe, i started a skirmish party for lulz, and after 15 seconds of playing the computer (IA) leave the match and i am victorius. WTF ?!

Ok it's clear the launcher (ra2.exe) send message throw PostThreadMessage() and game.exe set a value if a received well this message :

```
00491791  |.  8BF1          MOV ESI,ECX
00491793  |.  817E 04 EFBE0>CMP DWORD PTR DS:[ESI+4],0BEEF
0049179A  |.  75 4A         JNZ SHORT game2.004917E6
0049179C  |.  68 A4037C00   PUSH game2.007C03A4
004917A1  |.  E8 4A51F7FF   CALL game2.004068F0
004917A6  |.  8B46 0C       MOV EAX,DWORD PTR DS:[ESI+C]
004917A9  |.  83C4 04       ADD ESP,4
004917AC  |.  6A 00         PUSH 0                                  /BaseAddr = NULL
004917AE  |.  6A 00         PUSH 0                                  |MapSize = 0
004917B0  |.  6A 00         PUSH 0                                  |OffsetLow = 0
004917B2  |.  6A 00         PUSH 0                                  |OffsetHigh = 0
004917B4  |.  68 1F000F00   PUSH 0F001F                             |AccessMode = F001F
004917B9  |.  50            PUSH EAX                                |hMapObject
004917BA  |.  FF15 24527800 CALL DWORD PTR DS:[<&KERNEL32.MapViewOfFileEx>]     \MapViewOf
004917C0  |.  85C0          TEST EAX,EAX
004917C2  |.  A3 4C9E8300   MOV DWORD PTR DS:[839E4C],EAX
```

This routine is called for checking the type of message in our case 0xBEEF, and then if the MapViewOfFileEx() is well done, it will set a value in 0x839E4C. By watching reference to this immediate constant, we can see this :

```
004917FA  |.  A1 4C9E8300   MOV EAX,DWORD PTR DS:[839E4C]
004917FF  |.  83C4 04       ADD ESP,4
00491802  |.  85C0          TEST EAX,EAX
00491804  |.  75 03         JNZ SHORT game2.00491809
00491806      32C0          XOR AL,AL
00491808  |.  C3            RET
```

Just replace xor al, al by mov al, 1 and (trigger?) is fix.

10/09/2011

Subversion 41

reverse safedisc

Introduction

No this is post will not be about SVN (a software versioning).
We will just see a litlle difference in the subversion 41 of safedisc 1.
I invite you to read this post about version 1, before reading this.

Anti debug

It is the same stuff than before, you have to use EBFE tricks too.
This part is exactly the same than the previous post.

Call redirection

Each call to Kernel32 or User32 api, are done through dplayerx.dll as usual :

```
013E5BD5    68 6712EABF     PUSH BFEA1267
013E5BDA    9C              PUSHFD
013E5BDB    60              PUSHAD
013E5BDC    54              PUSH ESP
013E5BDD    68 1B000000     PUSH 1B
013E5BE2    68 00000000     PUSH 0
013E5BE7    FF15 F75B3E01   CALL DWORD PTR DS:[13E5BF7]                ; dplayerx.00E7531
013E5BED    83C4 0C         ADD ESP,0C
013E5BF0    6A 00           PUSH 0
013E5BF2    58              POP EAX
013E5BF3    61              POPAD
013E5BF4    9D              POPFD
013E5BF5    C3              RET
```

But in this revision for each api you will have to push a predefined value (random?) like 0xBFEA1267 in this example.
We can see the number of the api to call, and 0 or 1 for kernel32 or user32.
But ! after the call, we haven't got a jmp dword for jumping to the resolved address api
Because now the routine dplayerx.00E75310, will GetProcAddress() and then ret to this address, code at 0x013E5BED will never be executed.
So we must fix the previous code for fixing the iat :

```
013E36C5    33DB            XOR EBX,EBX
013E36C7    BA 50F04C00     MOV EDX,OFFSET SC3U.__imp__GetStartupInfoA@4 ; MOV EDX,4CF050 
013E36CC    8B02            MOV EAX,DWORD PTR DS:[EDX]
013E36CE    8B40 01         MOV EAX,DWORD PTR DS:[EAX+1]        ; Retrive the (random) valu
013E36D1    50              PUSH EAX
013E36D2    9C              PUSHFD
013E36D3    60              PUSHAD
013E36D4    54              PUSH ESP
013E36D5    6A 10           PUSH EBX                    ; Numero api
013E36D7    6A 00           PUSH 0                              ; 0 (Kernel32)
013E36D9    FF15 AB363E01   CALL DWORD PTR DS:[13E36AB]     ; dplayerx.00E35310
013E36DF    8B4424 14       MOV EAX,DWORD PTR SS:[ESP+14]       ; Addr of api
013E36E3    A3 F0BF4F00     MOV DWORD PTR DS:[4FBFF0],EAX       ; Save it
013E36E8    61              POPAD
013E36E9    9D              POPFD
013E36EA    A1 F0BF4F00     MOV EAX,DWORD PTR DS:[4FBFF0]
013E36EF    8902            MOV DWORD PTR DS:[EDX],EAX  ; Fix
013E36F1    43              INC EBX                             ; Next api
013E36F2    83FB 50         CMP EBX,50                  ; No more api ?
013E36F5    74 06           JE SHORT 013E36FD
013E36F7    83C2 04         ADD EDX,4
013E36FA  ^ EB D0           JMP SHORT 013E36CC
013E36FA    CC              INT3
```

Don't forget to set full access(write) to your rdata section.
And now let's patch in dplayerx.dll :

```
00E33B13    8B65 0C         MOV ESP,DWORD PTR SS:[EBP+C]
00E33B16    61              POPAD
00E33B17    9D              POPFD
00E33B18    C3              RET
```

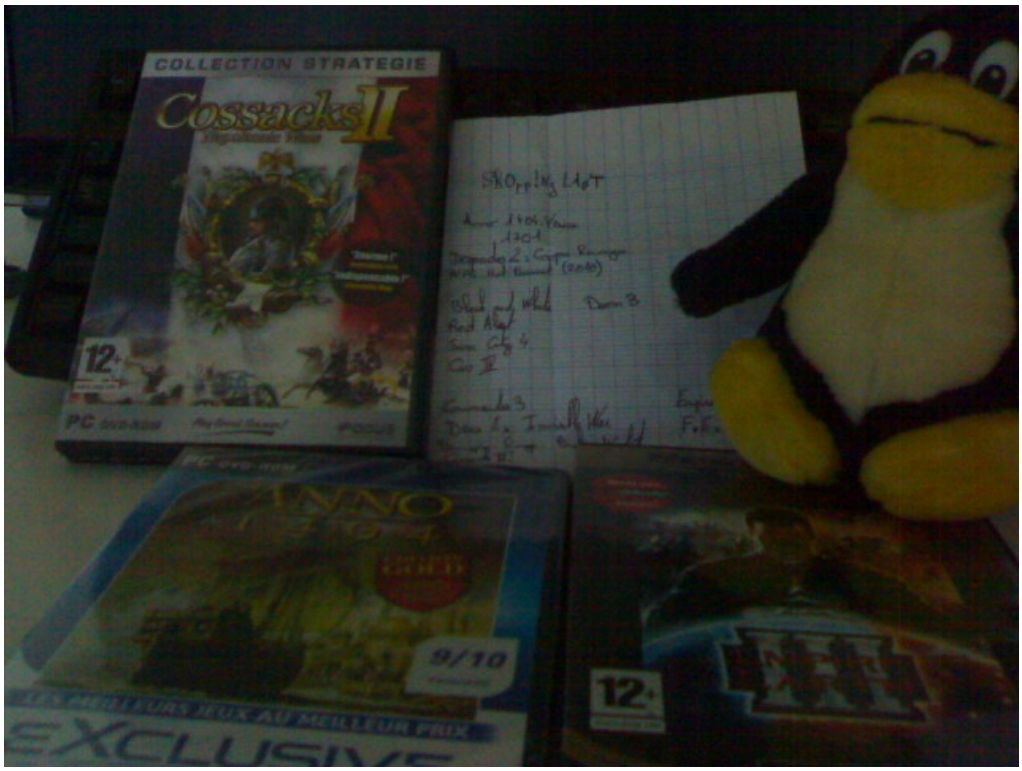
We will nop the popad and popfd instruction and replace ret by a jmp to our code (just after call instruction) :

```
00E33B13    8B65 0C         MOV ESP,DWORD PTR SS:[EBP+C]
00E33B16    90              NOP
00E33B17    90              NOP
00E33B18  - E9 C2FB5A00     JMP 013E36DF
```

Now set new origin, run it,then do the same thing with user32 api, change edx start value,push 0 by push 1, and 50 by 29. Now you can dump fix the iat with ImportRec and enjoy your game :]

Conlusion

This post is not very important but just to see a little difference, and how to fix it.
Btw today is Saturday, girlz go shopping for new shoes, guyz buy their alcohol for saturday night party, and me I wrote a list of fun (protection) games and buy some of them :



I just need time for beeing able to publish my research about this new stuff.

03/09/2011

Unsafedisc

reverse crypto safedisc toolz

Introduction

I'm actually doing an internship, so it's difficult to work on my personal project.
But today i found some times to finish the first release of my unsafedisc, actually it will work only with version 1.11.0 because i haven't got enough game with safedisc protection.
But when i will touch my pay, i will have the possibility to buy some old school games :]

Tiny Encryption Algorithm

In my last post about safedisc, i said dplayerx.dll was here for decrypting some sections of the icd file, like .data and .text. I reversed all the stuff from this dll, there is a lot obfuscation inside by using stc, jb, jmp instruction, the only solution i found was to trace the code and reconstruct with my own hand. If you want to check the decrypt routine look at the begining of segment text2 inside dplayerx.dll.
The interesting thing to know is the key that is present in this form :
ABCD - ABCD - ABCD - ABCD
It uses a 128-bit key with the same 32-bit pattern.
So we are able to bruteforce it, but how to know it's the good key ?
At the begining i was thinking about decrypting the 64-bit blocks where oep is, and check if the value equal to push ebp, mov ebp, esp (0x55 0x89 0xe5), but it's not enough, it was a really bad idea.
After switching on my brain, i looked at .data section :

game.icd.section-2.dmp

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00054DD0	14	44	E5	57	EA	FC	99	AE	A7	E6	ED	0A	01	E0	65	92
00054DE0	A7	E6	ED	0A	01	E0	65	92	8E	94	D8	D9	E5	C2	73	E3
00054DF0	67	E6	12	99	72	F2	C1	40	55	47	77	49	F1	99	63	F5
00054E00	0B	6D	8F	8A	85	9C	26	AB	17	F7	DF	9D	A4	36	37	5E
00054E10	7A	62	8C	D9	FA	CB	A1	BE	69	C4	26	D7	D9	E8	14	21
00054E20	71	09	E1	E2	C5	96	72	28	C3	D8	38	D8	E6	0F	D6	A3
00054E30	75	73	E6	30	24	6F	4A	1A	C5	7F	5C	39	8C	76	65	4E
00054E40	F5	CF	C4	3E	61	63	2E	F4	6E	CB	20	0E	8A	03	4D	C5
00054E50	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054E60	A7	E6	ED	0A	01	E0	65	92	9E	BC	ED	6A	6D	3C	2C	E2
00054E70	28	A2	6E	58	40	C7	0A	01	E9	BB	E8	48	8D	DE	29	B6
00054E80	D3	2C	93	D9	D2	E5	8C	56	A7	E6	ED	0A	01	E0	65	92
00054E90	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054EA0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054EB0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054EC0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054ED0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054EE0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054EF0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F00	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F10	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F20	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F30	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F40	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F50	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F60	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F70	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F80	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054F90	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FA0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FB0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FC0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FD0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FE0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92
00054FF0	A7	E6	ED	0A	01	E0	65	92	A7	E6	ED	0A	01	E0	65	92

We can easily see the same pattern : 0xA7E6ED0A 0x01E06592, TEA use electronic codebook mode (ECB), if we got in our example two 64-bit block of "0" they will have the same cipher.
With the end of section data, we are able to bruteforce easily the key :].

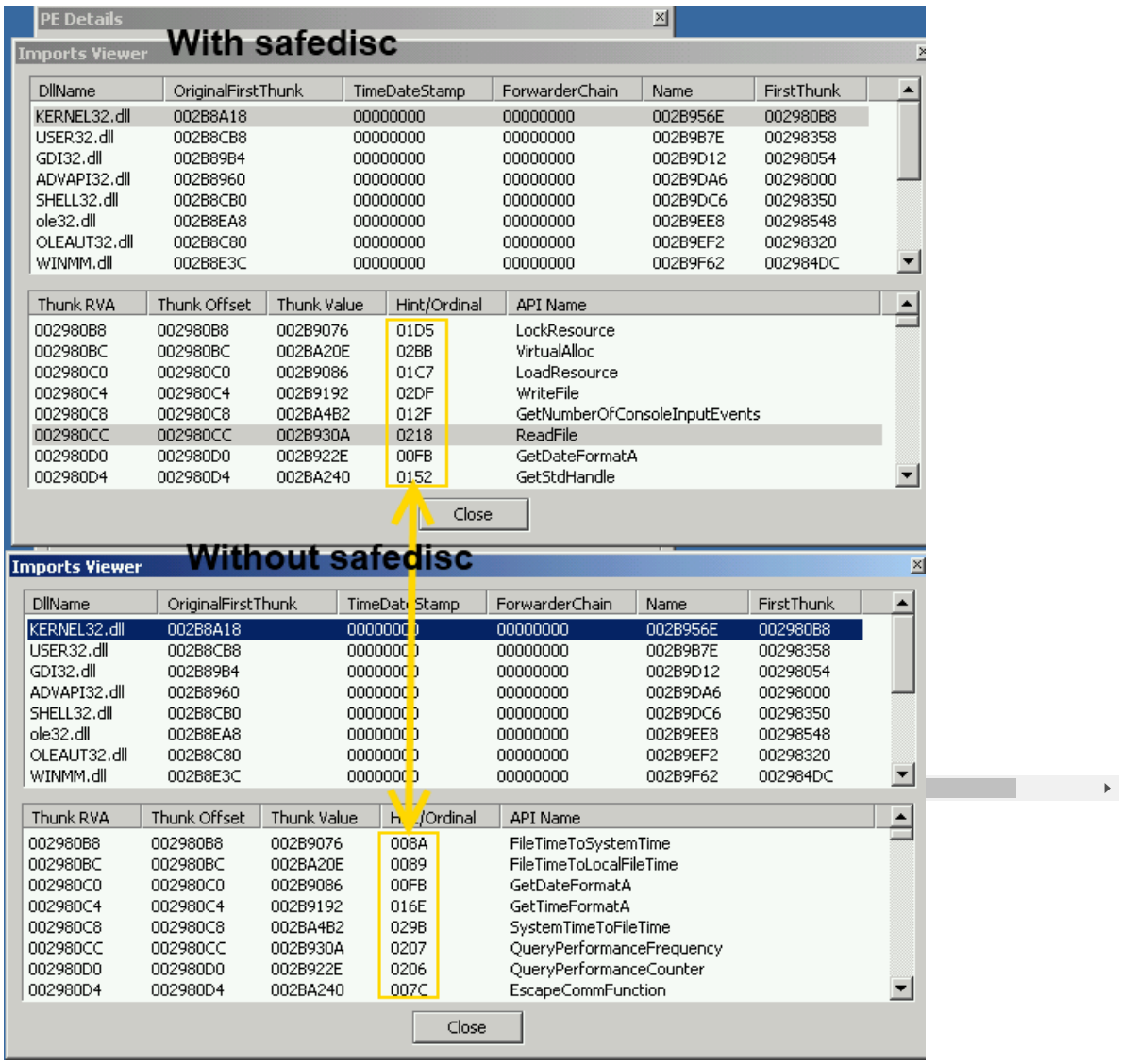
Monte Carlo algorithm

The last part for breaking safedisc, is reconstruct all the iat. By reading information from woodmann, and my reversed stuff from dplayerx.dll especially this routine :

```
00D42420    51              PUSH ECX
00D42421    8B4424 10       MOV EAX,DWORD PTR SS:[ESP+10]               ; tea key
00D42425    53              PUSH EBX
00D42426    8B5C24 0C       MOV EBX,DWORD PTR SS:[ESP+C]                ; Allocated memory
00D4242A    55              PUSH EBP
00D4242B    56              PUSH ESI
00D4242C    8B7424 18       MOV ESI,DWORD PTR SS:[ESP+18]               ; Nb max api 0x99 (
00D42430    33ED            XOR EBP,EBP
00D42432    33C9            XOR ECX,ECX
00D42434    57              PUSH EDI
00D42435    8B38            MOV EDI,DWORD PTR DS:[EAX]
00D42437    3BF5            CMP ESI,EBP
00D42439    896C24 10       MOV DWORD PTR SS:[ESP+10],EBP
00D4243D    76 11           JBE SHORT dplayerx.00D42450
00D4243F    33C0            XOR EAX,EAX
00D42441    41              INC ECX
00D42442    890483          MOV DWORD PTR DS:[EBX+EAX*4],EAX
00D42445    8BC1            MOV EAX,ECX
00D42447    25 FFFF0000     AND EAX,0FFFF
00D4244C    3BC6            CMP EAX,ESI
00D4244E  ^ 72 F1           JB SHORT dplayerx.00D42441
00D42450    3BF5            CMP ESI,EBP
00D42452    76 5E           JBE SHORT dplayerx.00D424B2
00D42454    69FF 6D5AE835   IMUL EDI,EDI,35E85A6D
00D4245A    33D2            XOR EDX,EDX
00D4245C    81C7 E9621936   ADD EDI,361962E9
00D42462    85F6            TEST ESI,ESI
00D42464    8BC6            MOV EAX,ESI
00D42466    74 05           JE SHORT dplayerx.00D4246D
00D42468    42              INC EDX
00D42469    D1E8            SHR EAX,1
00D4246B  ^ 75 FB           JNZ SHORT dplayerx.00D42468
00D4246D    81E2 FFFF0000   AND EDX,0FFFF
00D42473    8BC7            MOV EAX,EDI
00D42475    8BCA            MOV ECX,EDX
00D42477    D3E8            SHR EAX,CL
00D42479    B9 20000000     MOV ECX,20
00D4247E    2BCA            SUB ECX,EDX
```

```
00D42480    0FAFC6          IMUL EAX,ESI
00D42483    D3E8            SHR EAX,CL
00D42485    8BCD            MOV ECX,EBP
00D42487    81E1 FFFF0000   AND ECX,0FFFF
00D4248D    3BC1            CMP EAX,ECX
00D4248F    74 14           JE SHORT dplayerx.00D424A5
00D42491    8B148B          MOV EDX,DWORD PTR DS:[EBX+ECX*4]
00D42494    895424 20       MOV DWORD PTR SS:[ESP+20],EDX
00D42498    8B1483          MOV EDX,DWORD PTR DS:[EBX+EAX*4]
00D4249B    89148B          MOV DWORD PTR DS:[EBX+ECX*4],EDX
00D4249E    8B4C24 20       MOV ECX,DWORD PTR SS:[ESP+20]
00D424A2    890C83          MOV DWORD PTR DS:[EBX+EAX*4],ECX
00D424A5    45              INC EBP
00D424A6    8BD5            MOV EDX,EBP
00D424A8    81E2 FFFF0000   AND EDX,0FFFF
00D424AE    3BD6            CMP EDX,ESI
00D424B0  ^ 72 A2           JB SHORT dplayerx.00D42454
00D424B2    5F              POP EDI
00D424B3    5E              POP ESI
00D424B4    5D              POP EBP
00D424B5    5B              POP EBX
00D424B6    59              POP ECX
00D424B7    C3              RET
```

This routine will just build a table of ascending dwords, with a size of nb api, and sort the table into the correct order, using two consts, and morphing the decrypt key with them.



As you can see all the ordinal value of each IMAGE_THUNK_DATA differ, we have to reverse their algorithm for reconstruct well all IMAGE_THUNK_DATA.

Conclusion

That's all for understanding all the suff i coded in masm : my_unsafedisc.rar.

Screenshot

25/08/2011

NOD or GDI

reverse red alert safedisc c-dilla

Introduction

It's my first english article so please be cool.
If you haven't noticed yet, I (only) like old video games (see SNES), today i'm going to play with Red Alert : Tiberian Sun.
This game is protected by an old commercial protection : Safedisc aka C-dilla.

Detection

Safedisc version 1 can be recognized by several files on the CD :

00000001.TMP
CLOKSPL.EXE
DPLAYERX.DLL
SECDRV.SYS

And also the existence of two executables, Game.EXE and Game.ICD.
We can also recognize this protection by her signature :

GAME.EXE

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00024A10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00024A20	42	6F	47	5F	20	2A	39	30	2E	30	26	21	21	20	20	59	BoG_ *90.0&!! Y
00024A30	79	3E	00	00	01	00	00	00	0B	00	00	00	00	00	00	00	y>
00024A40	01	00	00	00	0B	00	00	00	00	00	00	00	00	00	00	00	

The signature is "BoG_ *90.0&!! Yy>" followed by 3 unsigned integers : the version, subversion an revision number.

Or simply by using Protection ID :

```
Scanning -> C:\Westwood\SUN\GAME.EXE
File Type : 32-Bit Exe (Subsystem : Win GUI / 2), Size : 239351 (03A6F7h) Byte(s)
-> File has 1271 (04F7h) bytes of appended data starting at offset 03A200h
[File Heuristics] -> Flag : 00000000000001001100000000000101 (0x0004C005)
[!] Safedisc v1.11.000 detected !
[CompilerDetect] -> Visual C++ 5.0
- Scan Took : 0.759 Second(s)
```

So in this article we will talk about how to defeat Safedisc version 1.11.0000 using a ring3 debugger.
Yes because all over the internet, i just found article / tutorial using Softice.

Anti Debug

So let's open Game.exe into OllyDbg.
We can found two anti debug tricks related to a ring 3 debugger.
The first anti debug trick is several call to IsDebuggerPresent(), and they also check this manually :

```
004212C6   .  64:A1 1800000>MOV EAX,DWORD PTR FS:[18]
004212CC   .  8B48 30       MOV ECX,DWORD PTR DS:[EAX+30]
004212CF   .  0FB641 02     MOVZX EAX,BYTE PTR DS:[ECX+2]
004212D3   .  85C0          TEST EAX,EAX
```

So by putting this code where you want, and change the origin ("New origin here" => Ctrl + Gray *), we can defeat this trick.

```
004197C5      64:A1 1800000>MOV EAX,DWORD PTR FS:[18]
004197CB      8B40 30       MOV EAX,DWORD PTR DS:[EAX+30]
004197CE      83C0 02       ADD EAX,2
004197D1      C600 00       MOV BYTE PTR DS:[EAX],0
00419E35    ^\E9 5675FFFF   JMP GAME.<ModuleEntryPoint>
```

The second trick is several call to ZwQueryInformationProcess(), with ProcessInformationClass argument set to ProcessDebugPort for checking if the process is being run under the control of a ring 3 debugger.
I know, it can be simply bypass by using the Phant0m plugin, but i wanted to write a script to defeat it :

```
var is_ProcessDebugPort
var buffer
var ZwQueryInformationProcess

gpa "ZwQueryInformationProcess", "ntdll.dll"
bphws $RESULT, "x"
mov ZwQueryInformationProcess, $RESULT
add $RESULT, C
bphws $RESULT, "x"
mov is_ProcessDebugPort, 0

cnt:
eob break
run

break:
cmp eip, ZwQueryInformationProcess
je begin_zwqueryinformationprocess

end_zwqueryinformationprocess:
cmp is_ProcessDebugPort, 1
jne cnt
mov is_ProcessDebugPort, 0
mov [buffer], 0
jmp cnt

begin_zwqueryinformationprocess:
cmp [esp + 8], 7
jne cnt
mov buffer, [esp + C]
log buffer
mov is_ProcessDebugPort, 1
jmp cnt
```

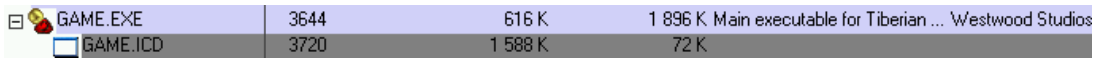
Explanation :

> Put an hardware breakpoint at the entry point of the function, and at the end.
> Check if the function was called with ProcessInformationClass argument = ProcessDebugPort (0x7).
> If it was called with the fact mentionned before, we set the buffer to 0 (The process is not being run under ring 3 debugger).

No we can work with our executables without problems.
There are another anti debug tricks, like calls to CreateFileA() with argument : "\.SICE" and "\.NTICE", for checking is softice is running.
Also in this version, they load a driver (secdrv.sys) but i have noticed nothing about anti debug techniques.
I don't know what the purpose of this driver in this version of safedisc (1.11.0000), i tried to do some shit with DeviceIoControl() api but without result.
Apparently, the driver is safe in this version while others are : CVE-2007-5587.

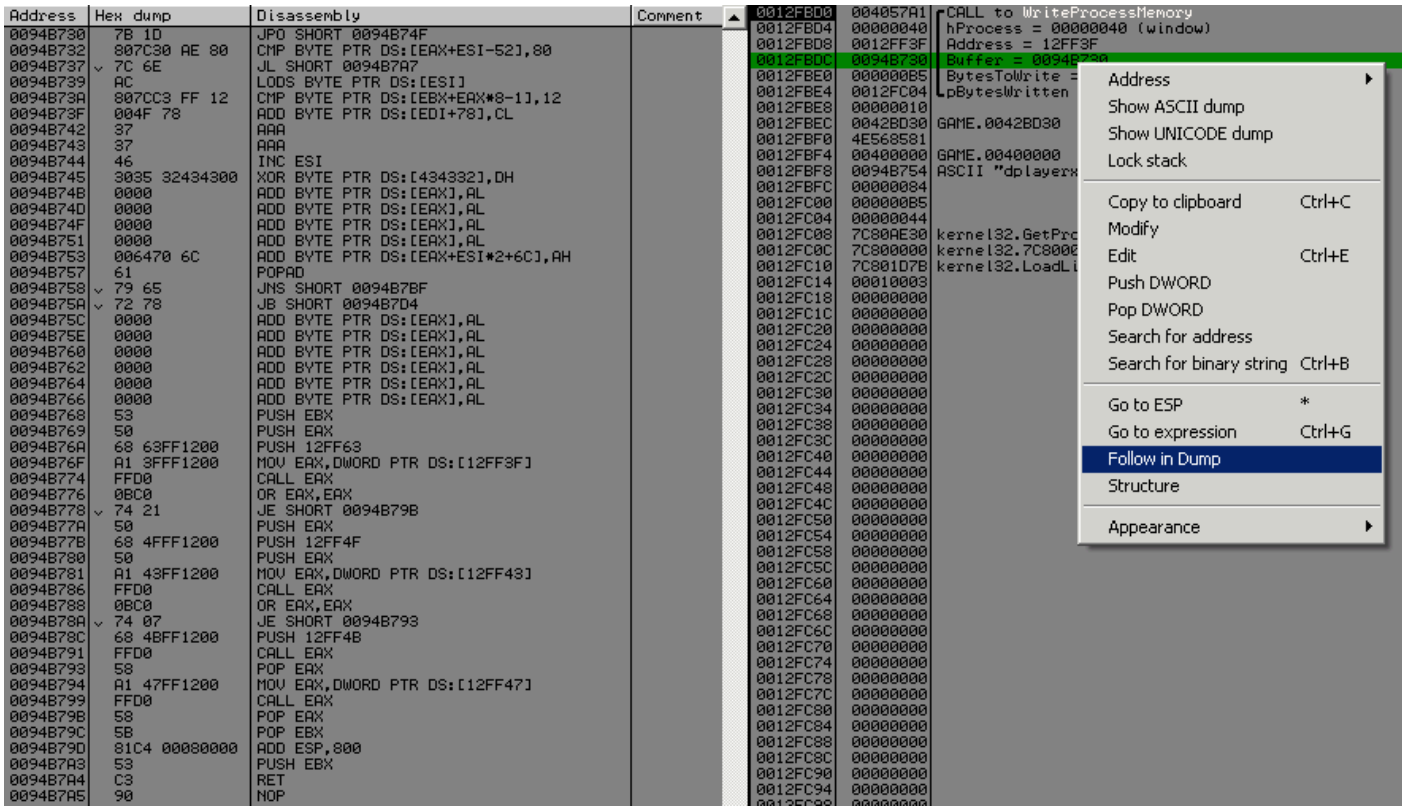
*.ICD

At the beginning of the post, I mentionned the existence of two files Game.EXE and Game.ICD.
Game.EXE executable is only a loader which decrypts and loads the protected game executable in the encrypted ICD file :

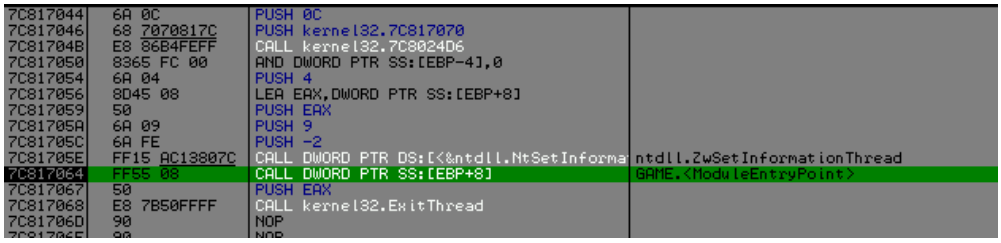



```
0012FCD4  004058F3  CALL to CreateProcessA
0012FCD8  0042CD80  ModuleFileName = "H:\INSTALL\GAME.ICD"
0012FCDC  00152350  CommandLine = """H:\INSTALL\GAME.EXE"""
0012FCE0  00000000  pProcessSecurity = NULL
0012FCE4  00000000  pThreadSecurity = NULL
0012FCE8  00000001  InheritHandles = TRUE
0012FCEC  00000004  CreationFlags = CREATE_SUSPENDED|CREATE_NEW_CONSOLE|NORMAL_PRIORITY_CLASS
0012FCF0  00000000  pEnvironment = NULL
0012FCF4  00000000  CurrentDir = NULL
0012FCF8  0042C178  pStartupInfo = GAME.0042C178
0012FCFC  0042BD30  pProcessInfo = GAME.0042BD30
```

Safedisc uses a debug locker technique with buffer overflow to the start the real executable (ICD file).
So let's see the buffer used by WriteProcessMemory() :



You can notice the different call eax, LoadLibrary("dplayerx.dll"), GetProcAdress("Ox77F052CC"), then it call this function, for decrypting all sections of the icd file (it uses TEA).
So if we want to attach a debugger to the icd process, remember my last post about EBFE or CC tricks, and notice the PUSH EBP RET (0x53C3), we can replace these opcode by EBFE, then attach olly to the process.
We can see that if DEP is on, the game will crash (failed?).
So let's replace (0x53C3 by 0xEBFE), continue the program (ResumeThread() will be called) and open a new ollydbg and attach to this process.
Then pause the program (F12), and replace JMP SHORT 0012FFB2 by PUSH EBX RET.
Trace the code you will arrive into kernel32, continue tracing, then at the entry point of the icd file.



So we are here and the fun part can begin :] :

```
006854E3 >  55              PUSH EBP
006854E4    8BEC            MOV EBP,ESP
```

Call redirection

In Safedisc, all the calls to Kernel32 or User32 api are done through dplayerx.dll.
Let's take an example :

```
00685509    FF15 54826900   CALL DWORD PTR DS:[<&KERNEL32.CreateDirectoryA>] ; DS:[0069825

...

01213423    60              PUSHAD
01213424    68 67000000     PUSH 67
01213429    68 00000000     PUSH 0
0121342E    FF15 44342101   CALL DWORD PTR DS:[1213444]                      ; dplayerx.00
01213434    83C4 08         ADD ESP,8
01213437    61              POPAD
01213438    FF25 3E342101   JMP DWORD PTR DS:[121343E]                       ; Jump to api
```

You can notice a call to dplayerx.00D4E9D0, with a stack where we pushed 0x67 and 0x0.
What does it mean ?
It simple, 0x0 is for telling the routine that it's a kernel32 api, and 0x67 is the number of the api to call.
This routine will GetProcAddress() for the specified api, then jump to it.
And what about User32 api ?

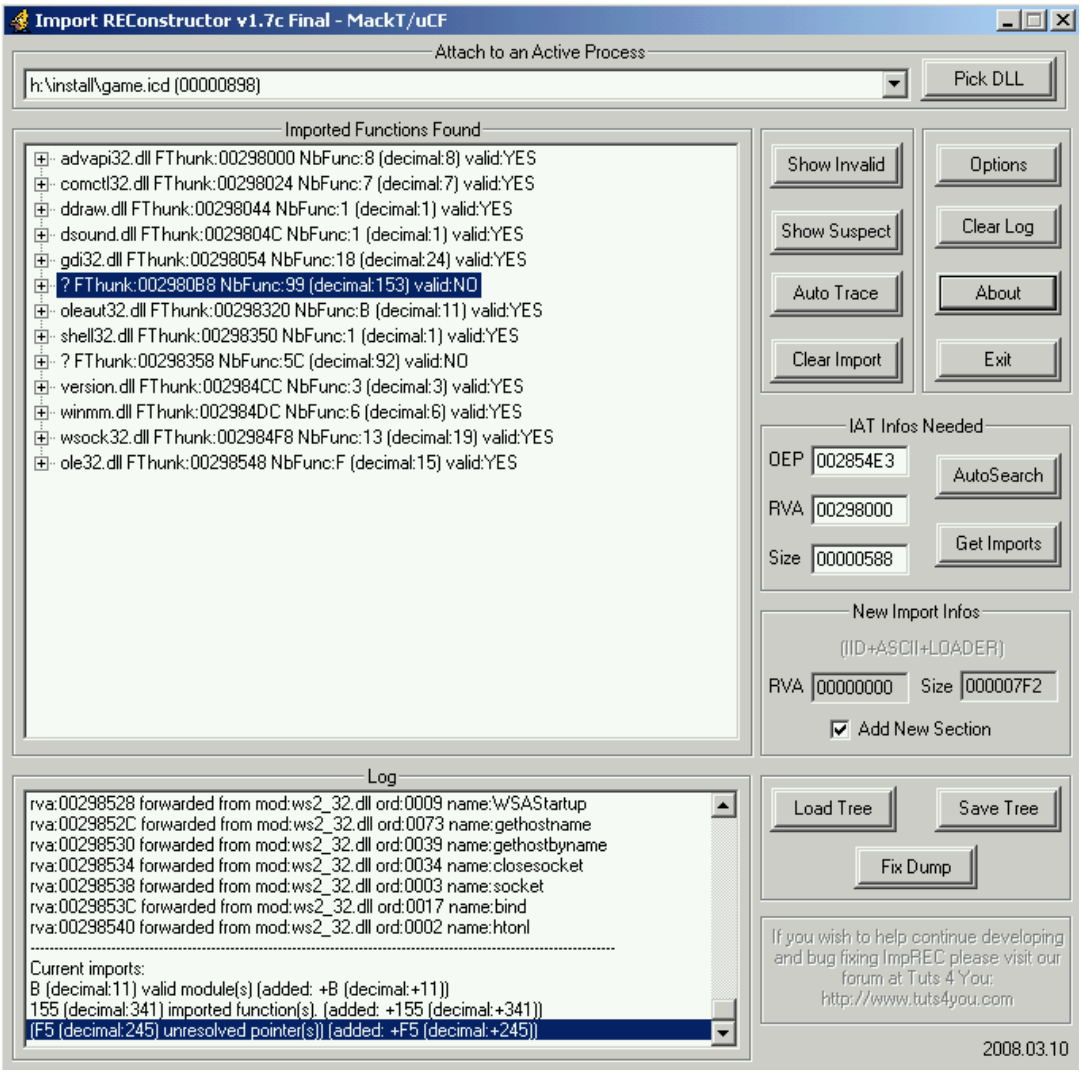
```
00D750B7    60              PUSHAD
00D750B8    68 43000000     PUSH 43
00D750BD    68 01000000     PUSH 1
```

```
00D750C2    FF15 D850D700   CALL DWORD PTR DS:[D750D8]
00D750C8    83C4 08         ADD ESP,8
00D750CB    61              POPAD
00D750CC  - FF25 D250D700   JMP DWORD PTR DS:[D750D2]
```

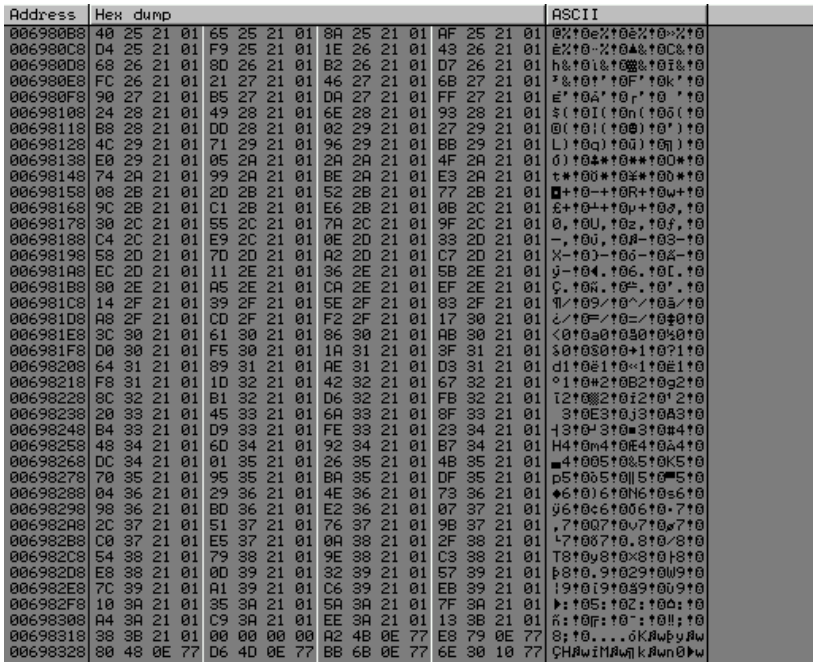
Same thing but we pushed 0x1 instead of 0x0, and jump to the resolved address.

Fix this

The problem is if we dump the executable and try to reconstruct the iat, it will fail, because all calls to kernel32 or user32 api must be solved by dplayerx.dll :



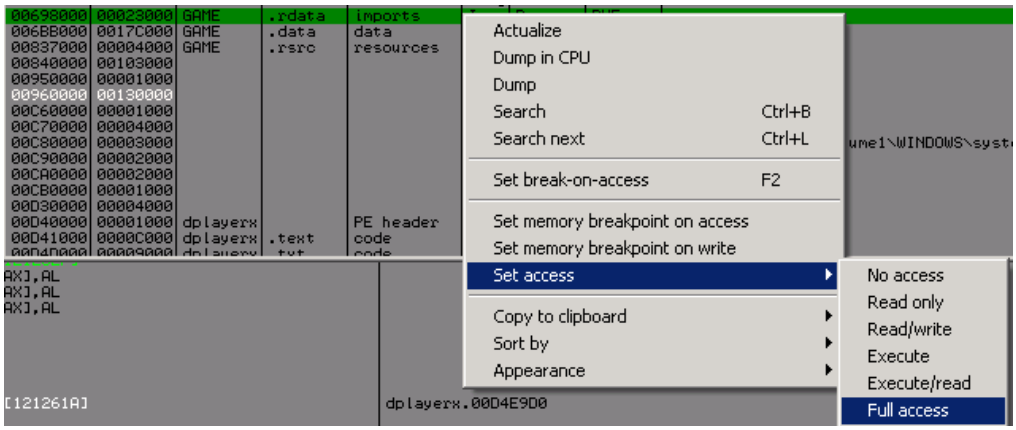
So why not using their own routine (dplayerx.00D4E9D0) to fix our iat :] ?
Let's take a look at rdata section for all kernel32 redirection :



It starts at 0x6980b8 and ends at 0x69831b

```
>>> hex(((0x69831b - 0x6980b8) / 4) + 1)
'0x99'
```

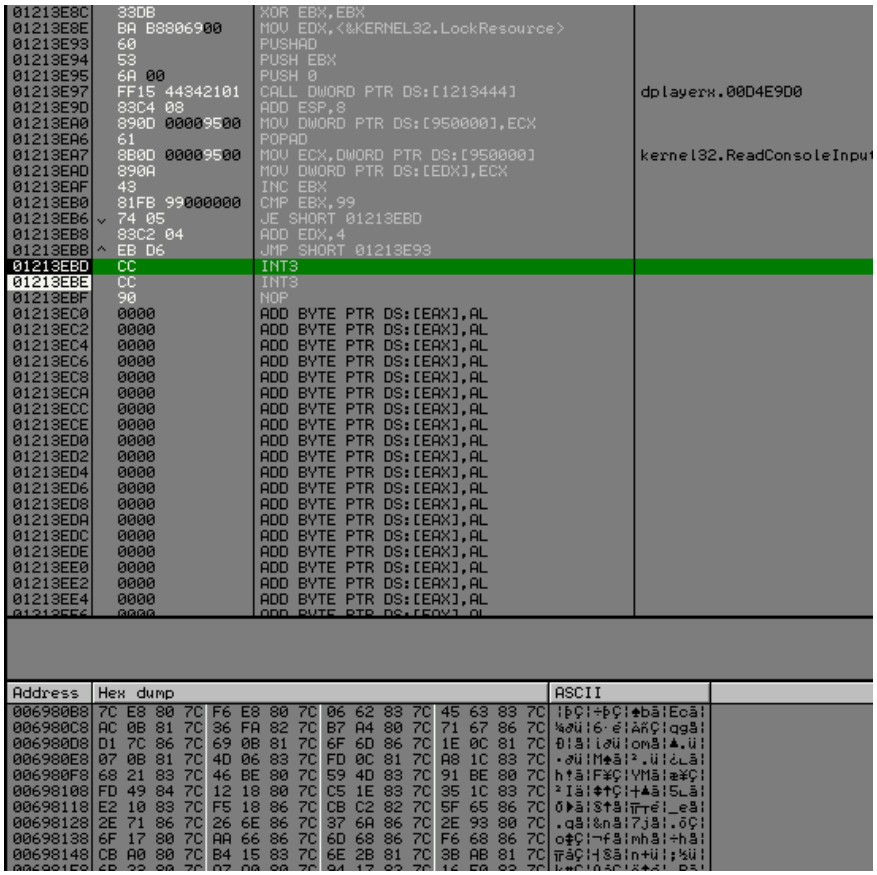
So we got 0x99 call to fix, like ImportRec said us : NbFunc:99(decimal:153) Valid:NO.
Be careful because the rdata section is not Writable, go fix it :



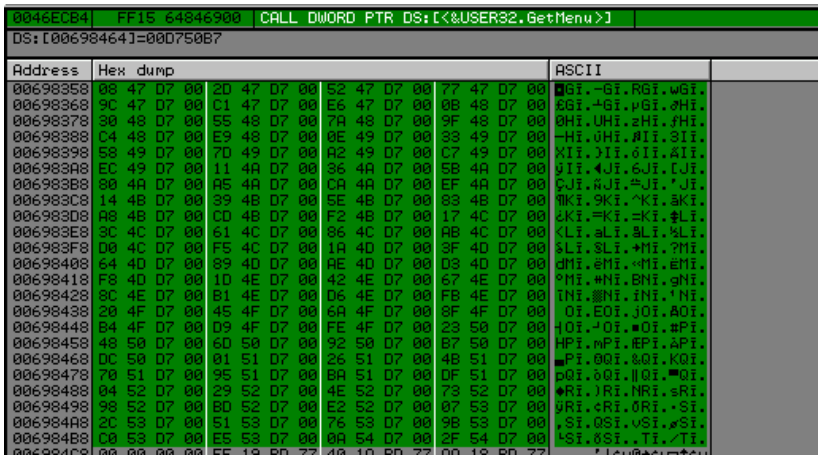
No choose any memory region wich is executable and code this :

```
01213DEB    33DB              XOR EBX,EBX                          ; we start with ap
01213DED    BA B8806900       MOV EDX,006980b8                   ; Start of the rdata kernel
01213DF2    60                PUSHAD
01213DF3    53                PUSH EBX
01213DF4    6A 00             PUSH 0                             ; Kernel32 api
01213DF6    FF15 61252101     CALL DWORD PTR DS:[1212561]            ; dplayerx.00D4E9D0
01213DFC    83C4 08           ADD ESP,8
01213DFF    890D 00009500     MOV DWORD PTR DS:[950000],ECX      ; save the api address
01213E05    61                POPAD
01213E06    8B0D 00009500     MOV ECX,DWORD PTR DS:[950000]
01213E0C    890A              MOV DWORD PTR DS:[EDX],ECX                ; fix the api addr
01213E0E    43                INC EBX                                   ; next api
01213E0F    81FB 99000000     CMP EBX,99                                ; no more api to f
01213E15    74 05             JE SHORT 01213E1C
01213E17    83C2 04           ADD EDX,4                                 ; next api address
01213E1A  ^ EB D6             JMP SHORT 01213DF2
01213E1C    CC                INT3                                      ; please stop
```

When we got out of the func dplayerx.00D4E9D0, ecx equals the address of api wich will be called, so we save this in a memory region writable.
And then we fix this in the rdata section for each api for kernel32.
Set a new origin at the begining of your code and run it.
Olly break, and our rdata look pretty nice :

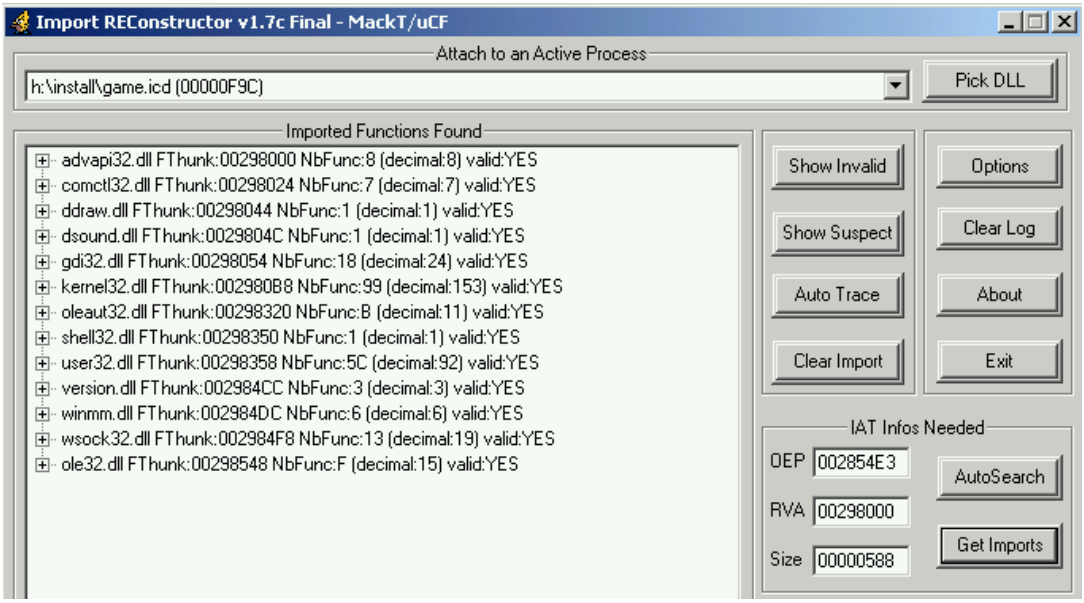


We need to do the same thing for user32.




```
>>> hex(((0x6984C4 - 0x698358) / 4) + 1)
'0x5b'
```

We will use the same code as above but change the start value of edx to 0x698358, PUSH 0 to PUSH 1 and cmp EBX, 99 to cmp EBX, 5C.
Run it and now come back into import rec :



Everything is ok, we can dump and fix the iat without problems. Safedisc has been removed correctly :].
BUT ! there is an another problem when we launch the game it ask us for the a cd ....

This is not the purpose of the article but you should copy all *.mix files form the cd into the tiberian sun directory and look around 0x004DCBAE ;)

Conclusion

I know over the internet there are several unwrappers for safedisc but with closed source, so i'm actually studying differents versions of safedisc, for writting a similar tools but with source included.

Useful links

http://www.woodmann.com/fravia/artha_safedisc.htm
http://www.winehq.org/pipermail/wine-users/2002-April/007910.html

---

Pages : **1**