Table of contents

Debug & Disassemble

Tiberium Reversal

© **ELF, 2011** \ICQ: *7719116*\ © **CJ, 2011**
ICQ:3708307\ ***Thanks to:*** **Nightshade,**
random, mak, DillerInc ***and everyone else whose help helps in the fight against DRM.*** X-code injection and the virtual machine: theory and practice

---

"If we think of SecuROM v7.33.17 as an Abrams tank without dynamic armor, OllyDbg as an RPG-7, and X-code injection as a cumulative grenade for our grenade launcher, then, just like in reality, such a rearward shot will pierce the armor of this heavy, unwieldy vehicle and achieve its intended target—OEP... Russian engineers are studying the disabled vehicle..."

What is SecuROM ?

SecuROM *(full name:* ***Sony DADC SecuROM) is*** a digital protection for CD/DVD discs unlicensed copying. Year of birth - 1999. ***The publisher*** is Sony Digital Audio Disk Corporation AG (Sony DADC AG). The office is located in Austria *(DO NOT CONFUSE WITH Australia! They also have an office there),* Anif (near Salzburg), Sonystraße 20, 5081, tel. +43 6246 8800. The lead ***author*** is **Reinhard Blaukowitsch .**

The protection type is the application of special physical marks on the disk, which supposedly cannot be copied. Representative office in Russia - Stasovoy St., 4, 119 071 Moscow; Phone: +7 (495) 935 7218 312 Email for sending hacked stuff **-** sales-russia@sonydadc.com

---

INTRO:

While SecuROM 7 is being prepared behind the scenes, I'll tell you what you'll be doing for the next two hours, at least. We have Command & Conquer 3: Tiberium Wars (with patch 1.9) at our disposal. Note right away that the file we need isn't cnc3.exe (but it will be very useful later!). Find \RetailExe\1.9\cnc3game.dat. Load the aforementioned file into the debugger and look for the .securom section.

We understand what we're dealing with. F9... Failed to launch the required security module... The same thing will happen if you don't load anything into Olga and just leave it open! SecuROM actively resists hacker tools, so a workaround is needed. On my 2k3, *IsDebuggerPresent* was patched—XOR EAX, EAX was immediately after the first instruction, so this annoying error message popped up even without a debugger. At first, I thought the protection was detecting the debugger through a hook, and it also became clear that messing with anti-debugging would be tedious. By the way, attaching to the cnc3game.dat process with a debugger is impossible—this operation will terminate it. In the end, it turned out that the protector successfully resists

Procmon and API spies. Let's try to attack the protection side – write any 10 bytes into the .secuROM section and change, for example, TEST EAX, EAX to TEST EBX, EBX... Excellent! A glaring error – the missing file integrity check! A cracker can easily penetrate the protector down to the OEP, despite all the developers' efforts. We won't need stealth devices or API monitors –

Let's simply ignore anti-debugging. In short, I propose we break away from the classics and take a fresh look at protection. So, let's begin. First, I'd like to look at the verification mechanism,
but we're prevented from accessing *GetDriveTypeA (of course,* this WinAPI is there, Ctrl+N in Olka) by a message about a detected debugger. Second, we absolutely must access OEP. Switching back to Olka and analyzing the code with the message active, it becomes clear that the bulk of the protector has already been unpacked. Excellent! Now let's say we first insert the required API jump to our X-code, which can extract the return address, show it to us, put it back, emulate the bytes overlapped by the jump, and return control! ... What? What? :)

PRACTICE

For this chapter, please use the article appendix:
SOURSES.TXT *(ASM code sources)*

In strictly scientific terms, X-code is a meaningful set of bytes injected by an unauthorized person or program into the target process code to perform a specific task. Injecting X-code is often considered a virus-writing topic, so it's a complex and extensive subject. However, in our case, it's much simpler: the X-code we inject performs one simple task: displaying the return address. We'll inject our X-code directly using the **OllyDbg v2** debugger , aka Olya (Olga). For convenience, we'll break our larger task into separate subtasks. 1. **Task:** *Obtain write access to the kernel32 code section.* **Description:** According to the standard, the code section (.text) has the Read/Execute attributes. Therefore, a code like this, where 77E41C00 is an address in the kernel32 code section: MOV BYTE PTR DS: [77E41C00], E9 , would legitimately trigger an access_violation exception, which isn't what we're planning. Since I have two Windows installations, I decided to patch Vista's kernel32 in 2k3 offline (aka bithack). Using WinAPI *VirtualProtect* is still preferable (it's in the X source code!), but I think unconventional solutions are beneficial. In 2k3, a writable kernel32 code section crashes some applications (VisualStudio 2008, OllyDbg 1.10), but they tolerate this behavior in Vista just fine! It's a shame Windows isn't written for hackers – writable code sections and an unpatched *IsDebuggerPresent* aren't good :) **Solution:** Launch 2k3. Launch PeTools. Don't forget to set your library access rights. Add the Write attribute to the code section and recalculate the checksum. Analog

PeTools can use the editbin utility if Visual Studio is installed: for example, to recalculate the checksum, type "editbin /RELEASE E:\Windows\system32\kernel32.dll" in the command line. If Vista starts and in the memory map layout in Olga, in the Access column, the kernel32 code section of any process is marked as RWE, then everything is done correctly! 2. ***Task:*** *Find free space for the X-code in cnc3game.dat; find the address for setting the jump that will transfer control to our X-code; determine which bytes will be covered by the jump; determine to what address control will need to be returned after setting the jump*

Description: There are 11 sections in cnc3game.dat (the PE Header isn't relevant, although it's possible to inject code there if you really want to). The first 6 are definitely unsuitable (text – encrypted code for the game itself; data, rdata, and rsrc – program data area; tls – TLS Callback; rts.ver – has a read-only attribute). Ars – a quick glance reveals obfuscated code; all access attributes are present, so we'll leave it alone. Est – the entry point is located here, along with all access attributes; this is a suitable candidate. The injection target is most often a string of zeros at the end, left to align sections and not used by anyone. Artem – the name seems like a joke from the developers; it's small and meaningless code; the section has all access attributes, but it's best left alone. Celare – definitely all SecuROM data and resources; we'll skip it. According to Olya, the SecuROM section of the same name contains the import table and has all the access attributes, so we'll leave it alone. Now, let's transfer control. Experience shows that it's best to jump from the unpacker. Unlike other structures, its code isn't modified during execution (although in our world, everything is impermanent). We could simply move the entry point to our X code, but that's not necessary for us. **Solution:** The tail of the est section is always free. In my case, our X code will be located starting at address 11DB6D0h. Set a breakpoint when accessing the text section (the unpacker will definitely touch it and notify us). Run the application. We find ourselves in a function with an address, in my case, equal to 11DB1F9h. I chose address 11DB27Eh. The jump will be long, meaning one byte of E9 plus a 32-bit operand, which will take up 4 bytes, for a total of 5 bytes. Address 11DB27Eh contains ADD EDI,ESI and ADD ESI,EBX, which occupy 2 bytes, and MOVZX EBX,BYTE PTR DS:[ECX+7], which takes up a full 4 bytes. Our jump completely overlaps the first two and touches MOVZX with its last byte, meaning all three commands will need to be emulated. Remember this.

them. Now let's calculate where control will need to be returned. In total, all three instructions, when emulated by us, take up 2 + 2 + 4 = 8 bytes. This means 11DB27Eh + 8h = 11DB286h. We will return to this address. We write to address 11DB27E: JMP 11DB6D0 and add three NOPs. It is worth noting that no one prevents us from returning immediately to the address after the jump (more precisely, to the first NOP), however, there is little point in this. 3. **Task:** *The first part of the X-code (preparatory). Emulate the instructions overlapped by the JMP 11DB6D0 jump. Transfer control to the X-code itself. Calculate the address of GetDriveTypeA. Determine which bytes will be overlapped by the jump leading to the second part of our X-code and emulate them. Organize the calculation of the operand of the JMP instruction for the 2nd jump. Correctly write the jump instruction to the second part of the X-code. Return control.* **Description/Solution:** I'll explain three aspects in detail. In our case, the first part of the X-code should ideally be executed once, that's all. However, the unpacker runs in a loop, as indicated by the EAX register, which increments each time. We first check the EAX register for a single fixed value (in my case, 6Ch); if it's equal, we transfer control to the X-code. Obtaining the address of a function from the library involves two APIs: *GetModuleHandle* and *GetProcAdress.*

The EAX register contains the address of the required API.
With the overlapped bytes, *GetDriveTypeA* and most of the Windows API in general have a useful A VC++ compiler feature! The first three instructions in the exported functions of Windows system libraries are MOV EDI, EDI; PUSH EBP; MOV EBP, ESP, which will give us the total of 5 bytes we need, without any additional steps required! Let's examine the operand of the JMP instruction. The problem is that kernel32, like other system libraries, can "dance" around the process's address space even within a single Windows instance.
This means that patching the top of *GetDriveTypeA* with a fixed value is impossible, because, after all, the address you see in the debugger could very well be something other than what you intended, and the jump operand will point to a completely different location! Why? As a reminder, the operand for any jump instruction specifies the number of bytes to jump, not the destination address! Additionally: the x86 operand is written in reverse order, and regardless of a conditional or unconditional jump, the operands are determined identically! To correctly calculate the operand, I first calculated the difference between the entry point in the aforementioned WinAPI and the jump destination address. Then, since the jump will be towards lower addresses, plus taking into account the size of the JMP LONG instruction (5 bytes), I obtained the final value: SUB EAX,cnc3game.011DB6E3 MOV EDI,-5 // EDI = FFFFFFFBh SUB EDI,EAX 4. **Task:** *The second part of the X-code (the main one). Extract the return address. Convert the return address bytes to ACSII for correct output. Display the return address on the screen. Correctly return to the system library and emulate overlapped bytes.*

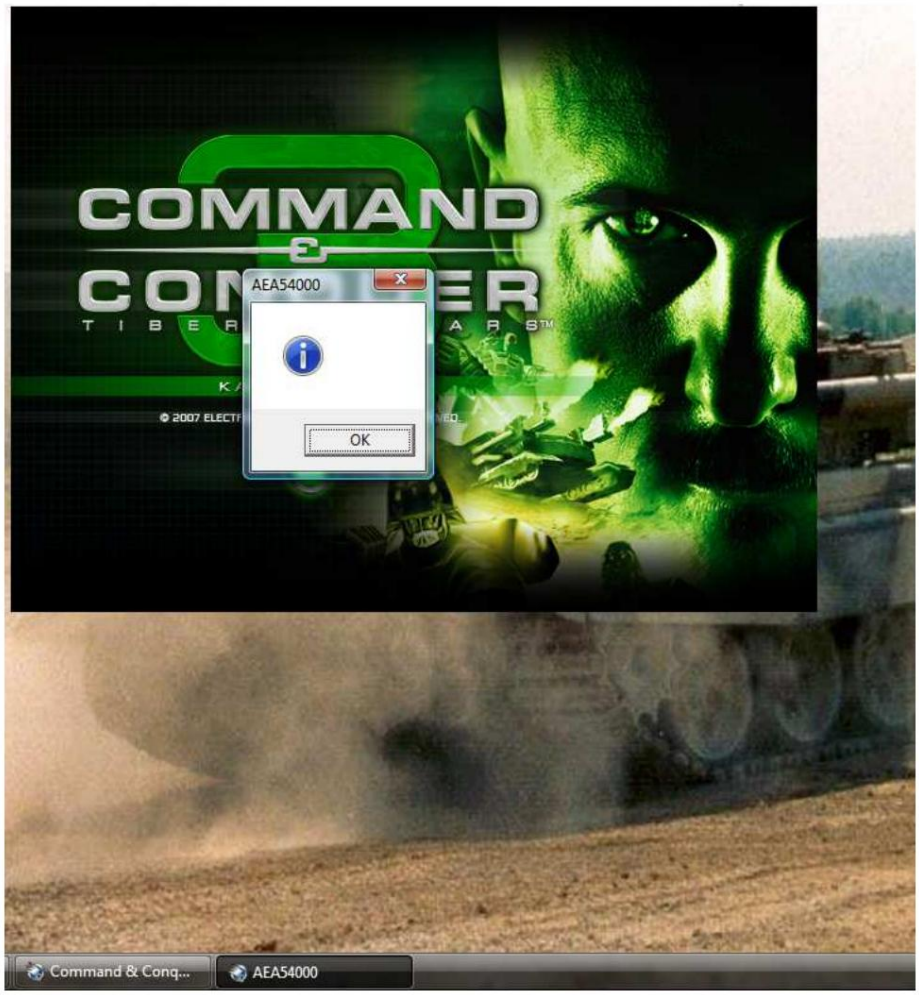
Description/Solution: The process has started, the first part of the X-code has executed successfully, control is transferred to GetDriveTypeA, and then, via the jump we placed, we find ourselves at the beginning of the second part of our presentation. The address is 011DB6E3h. We'll examine the most hardcore aspects of its operation. If you've already figured it out, then we're faced with the problem of correctly displaying the return address. Indeed, if we pass the return address as an argument to *the MessageBoxA API text,* then our satellite's message will contain pure gibberish instead of the return address (by "gibberish" we mean an ASCIIZ string that starts with the return address). After all, for Windows, 00404001 is the address/operand from which to read the string, but not the string itself! Since I don't know of an API that could perform such a conversion (we'll omit _itoa), I decided to write my own code that could perform the required operation. It's actually not that difficult! I reasoned like this: each character in ACSII corresponds to its own code. To determine how each byte of the return address will be encoded, we need to set up a loop. The ACSII codes for numbers 0 through 9 have values 30-39h, while the codes for letters (in our case, numbers) A through F are assigned values 41-46h, respectively. The guiding principle is to compare the return address bytes and the reference value by incrementing the latter and its codes in the ACSII table. If the reference byte is the same as the one being compared, we write the two values of their codes to a specially designated address (a byte has two digits, so we use the high and low parts of the ECX register for the ACSII code of each of the two digits). The last remark concerns the jump from 9 to A - in the ASCII table, their codes are 39h and 41h, respectively, and with an increment of 39h, the next number is 3Ah, which is assigned a different character. To make sure I don't miss the point, I'll say right away that besides *MessageBoxA ,* there are many other options for displaying information: 1) Using *CreateWindowEx.* Create a window with controls (Edit, ListBox) and send it to

I bet secret information using *SendMessage .*

2) To a file using *CreateFile/ WriteFile/ CloseHandle* and their low-level equivalents. Named pipes can also be added here.

3) A very original approach: using DirectX (Draw or even 3D). Fortunately, the game automatically includes the library. However, this requires familiarity with the MelkoSoft interface, and it would be more convenient to write your own separate DLL for this purpose and inject it with the game itself. 5. **Task:** *Test the functionality of the injected X code.*

Correct answer: Besides the green game logo in the background, the first thing we should see is our satellite – a MessageBox displaying 8 hex digits. This is the first call to GetDriveTypeA and its return point in reverse order (if you have the X-code in front of you, you hope you understand why it translates the address in the opposite direction). The first address doesn't refer to cnc3game.dat and is located somewhere in ntdll. The MessageBox will then appear n more times, where n is the number of logical drives on your machine (including virtual drives, of course). Then, a SecuROM window will appear asking you to insert the required disk. After clicking "Retry," the satellite will appear n times, then the window asking you to insert the required disk will appear again. Incorrect answers and their possible causes: The satellite didn't appear, but the game works – The second part of the X-code didn't receive control -> The first part of the X-code mistakenly placed the jump in the wrong place. Program error – Unhandled exception -> Kernel32 code section without a write attribute OR X code erroneously transferred control to another memory location. The satellite appeared, but instead of 8 digits, there's pure nonsense – a return address instead of an ACSII string. The address in the satellite points to a non-existent memory location – the return address was decoded incorrectly into an ACSII string.



"Stirlitz on the line. Center, accept the encryption." Our Stirlitz is already transmitting hex addresses. All that's left is to adjust its scope and record the results on paper. In addition to the already annoying GetDriveTypeA API, we can slightly modify the code and, with the necessary parameters, set up listening for *CreateThread, DriveIoControl, CreateFileA, RegQueryValueExA, and KiUserCallBackDispatcher.* In short, we're working as usual.

A standard API spy. So what's the difference between it and our method? A typical API spy uses standard interprocess *communication procedures (ReadProcessMemory, WriteProcessMemory ,* or lower) and can be easily detected by protectors. Good API spies are few and far between, and they're well known to security developers (this is easy to verify – listen to *CreateFileW* in a protector). With the advent of new versions of security systems, typical API spying methods are slowly becoming a thing of the past. X-code injection has become an innovative and cutting-edge approach, eliminating all the drawbacks of standard API spies and achieving maximum stealth by utilizing the protector's own resources. A real "Stirlitz" behind enemy lines. So, what exactly is SecuROM 7? Obfuscated code, with not much garbage, relying on assembly tricks. It's a treasure trove of all sorts of tricks, you could say. For example: MOV EAX,11D7B1C // Achtung! Starting address of the check zone, present your program breakpoints in expanded form :)
{ MOV ESI,DWORD PTR DS:[EAX] //load the next DWORD ADD DWORD PTR SS:[ESP+10],ESI //add with the previous DWORD

ADD EAX,4 // the next offset DWORD DEC WORD PTR SS:
[ESP+0C] // 114 DWORDs above add JNE SHORT 011D7B76 }
// - while (dword [ESP+0Ch] != 0)

OR BYTE PTR SS:[ESP+10],01 // add one to the low byte SUB DWORD PTR SS:[ESP+10],933 //
subtract the “checksum”, the solitaire is solved if: DWORD [ESP+10] <= 0. (Actually, there should always be zero there; the developers just decided to play it safe if the negative number is zero)

PUSHFD //save flags (EFL = 206h)
... //code designed to distract attention POPFD //we pull out flags
(EFL = 206h)
JBE SHORT 011D7BC1 //so did our solitaire game finally work? (Zero Flag(Z) = 1 or(and) Carry Flag(C) = 1?)

Also, look more often at what you are tracing:

00C49160 PUSHFD
00C49161 MOV EAX,DWORD PTR SS:[ESP]
00C49164 NOP
00C49165 TEST AH,1
00C49168 JE SHORT cnc3game.00C4916F 00C4916A
MOV ECX,7BE 00C4916F XOR
EAX,EAX

It turns out that SONY DADC has also found a very original replacement for the MOV ESI, DWORD PTR instruction DS:[EXIT]:

MOV DWORD PTR SS:[ESP],ESI
XOR ESI,DWORD PTR DS:[ESI]
XOR ESI,DWORD PTR SS:[ESP]

If you set a break on memory access on *IsDebuggerPresent,* then by tracing the procedure that is encountered in the network, you can see:

SUB ESI, EAX
LEA ECX, DWORD PTR DS: [ESI+EBX-5]
MOV BYTE PTR DS: [EAX], 0E9
MOV DWORD PTR DS: [EAX+1], ECX

"Bah! I've seen this somewhere before :)." Like us, SecuROM 7 (AsProtect is also adding to this list) also actively uses the technique of inserting adapters: they calculate the API address, calculate the jump length, add 5 bytes to it (MOV EDI,EDI; PUSH EBP; MOV EBP, ESP), write the adapter, and emulate the opening of a stack frame. Now, if the protector needs to call an API, it does so through the adapter, emulating the opening of a stack frame on its own and getting 5 bytes "earlier" into the library area with the required API. So, we've become familiar with the main technique against API spies. The problem is that the latter are accustomed to attaching hooks to the MOV EDI,EDI instruction.

Which, in our scenario, will never receive control! However, for some unknown reason, SecuROM 7 doesn't use adapters in its work. But this trick should be taken into account in future X-code injection implementations. So, let's return to the "Stirlitz" readings. Unfortunately, time is running out, as is space allocated in the log (the most original phrase in the text). I'll run through the main points. A distinctive feature of this version of the protector is the illegal UD2 instruction (plus hardware breakpoints with them), which are assigned to SEH handlers. I counted only two of them (00C996B0h and 00C99702h). If you analyze them, you will see that they all lead to JMP... and perform absolutely the same task. But first things first. The chain *GetDriveTypeA (set of drives) -> CreateFileA (opening* a drive at the sector level) -

>DeviceIoControl(check for the presence of a disk in a given drive, the number of sectors with tracks)-> *DeviceIoControl(initial* 5 attempts)-> MOV DWORD PTR DS:[EBX+EDI*8+4],EAX (If we emulate the presence of a disk in the drive, we find the head part of the check)-
>QueryPerfomanceCounter (second part of the test) and synchronous streams...
CMP BYTE PTR SS:[EBP-52],BL // check for the presence of a disk in the specified drive

Of course, I could go all out and fully explore the verification mechanism, with all these subchannels and so on. However, I suggest we skip ahead to the dumping process. Once we get to that, you'll immediately understand what the SecuROM version we're examining is running on. A couple more interesting points about GetDriveTypeA and hardware breakpoints. I called this joke X-code flash bag. Knowing the WinAPI location and reading the HKLM\System\MountedDevices branch in the protection mechanism allows you to hide a virtual drive from the protector if necessary, or all drives at once! The second issue is directly related to the call to the API itself and similar ones – in several places, the protector calls them via a series of JMPs and the standard LoadLirary/GetModuleHandle/GetProcAddress trio. The first time I saw this was in SafeDisk v4.5. Regarding hardware breakpoints, as I mentioned above, the aforementioned SEH handlers are also assigned to them. I'm referring to how SecuROM 7 detects the presence of debugging without ZwContinue and similar context operations. The protector doesn't check what caused the exception... Well, that's just an aside! Now, the most interesting part I've prepared! Let's move on to finding the OEP and taking a dump. We'll need **Daemon Tools (or** any other drive emulator) and a mini-dump of the original game disk. However, if we completely disassemble the verification procedure and correct the necessary transitions, we can do without emulation. We need to reach the original entry point and capture a correct dump at any cost, and this task is indeed solvable. The untouched cnc3.exe will help, as it tells us that the developers wrote the game in Microsoft Visual C++ 7.0. The entry point looks like this: 004628DA CALL 004784B8

004628DF JMP 004626FA Actually, the entry point is at the jump operand - 004626FAh. The function at 004784B8h doesn't do anything useful, and I always throw it in the trash. However, we find that it calls several APIs (GetSystemTimeAsFileTime, GetCurrentProcessId...). For us, this means only one thing - if we set our "Stirlitz" to GetSystemTimeAsFileTime and after unpacking, the startup code is indeed in its rightful place... Let's get to implementation. Mount the mini-dump. Launch the game. We are interested in all MessageBoxes after verification: 00DDCE77, 76B414D4, 7C34207B, 0040A5AE... **STOOOP!** The last one, but the call comes from the .text section !!! *We need attach now!* We attach to the process with Olga 1.10 and the dumper and navigate to the last address (we can now attach, *DbgUiRemoteBreakin* is only fixed during the disk check). Incredible! :) We're really looking at OEP! It worked! We jump to address 0040A006 and dump our process. I'll let you in on a secret right away – you can pinpoint the original entry point, but more on that later. Now, the most important thing: SecuROM 7 has the ability to protect itself from dumping. The dump, naturally, is initially inoperable. A quick analysis reveals that some calls (for example, just below, at address 00A400D) don't lead to child procedures, but rather, through a series of JMP DWORD PTR DS:[address] and the "query base," end up in the all-powerful virtual machine's (VM) allocable memory. After it runs, the double word at the jump address magically changes and points to the correct procedure. Moreover, even before accessing the "query database," it's already at its address. Looking at the endlessly long, horrific code of this virtual miracle, one wonders if it's even possible to uncouple SecuROM from the program. Or maybe it's better to try attaching the allocated memory protector to the game as a new section (the vast majority do this, as it's the simplest method, but the most terrifying in terms of optimization).

SecuROM 7 START

For this chapter, please refer to the appendix to the article:
Materials on the operation of the SecuROM add-on protection mechanisms, version 7.33.0017. Before moving on to the VM analysis, it would be a good idea to take a look at the SecuROM 7 add-on armor itself. If you're not interested, you can skip this chapter. Incidentally, I've already briefly touched on some aspects of the protector's operation above, but to fully comprehend everything, it's better to break it down. So:

• Information section

SecuROM Data Block. A special encrypted data block that contains the firmware's security features. The most useful information here is the exact version number. Contained in the PE header between **the MZ files.** stub and **PE_DOS structure.**

CreateThread. In addition to creating two threads for the second part of the check, it's used to create another special thread for displaying messages (like: *Failed to load required security module* , aka your mother! Unload the debugger!).

• Active section

IsDebuggerPresent. An unambiguous function known even to novice programmers. The method is simple and straightforward: during debugging, the BeginDebugged flag is set in the PEB, and WinAPI itself accesses the FS:[18] pointer in the TEB and retrieves the contents of this flag through a reference in the PEB. The first thing that comes to mind is to insert an XOR EAX, EAX instruction... but SONY DADC decided to outsmart reverse engineers by inserting control values greater than one into BeginDebugged and checking whether the function returns them. It's clear that this approach may change in future versions of Windows, so simply check the contents of the aforementioned function using the same template.

```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
RETURN
```

It's not good, but the general specification of the PE format is unlikely (at least, that's what SONY DADC thinks). I'd especially like to note that *IsDebuggerPresent* is the most frequently called function, which typically originates from the VM on SEH UD2/ hardware breakpoint handlers. Countermeasures against detection boil down to my X code initially resetting the flag in the PEB and leaving it alone.

CheckRemoteDebuggerPresent/NtQueryInformationProcess. If IsDebuggerPresent is responsible for If debugging of the current process is detected, the new function under consideration detects the running debugger, regardless of whether the process is being debugged. *CheckRemoteDebuggerPresent* is actually a wrapper for the low-level *NtQueryInformationProcess,* which is called with a *ProcessInfoClass* argument of 7, the number that hides the ProcessDebugPort. Later versions introduced *a ProcessInfoClass* with a value of 0x1F(31)—the same error code 8019. Countermeasure. First, let's look at the *NtQueryInformationProcess* prototype in ntdll:

```
MOV EAX,0A1

MOV EDX,7FFE0300

CALL DWORD PTR DS:[EDX]

RETURN 14
```

As expected, NativeAPI has a __fastcall call declaration. Next, the KiFastSystemCall address is retrieved, which, depending on the processor's modernity, takes us to the kernel.

Via SYSENTER or INT2E. There's only one cool hacker-style way to intercept control for our X code: swap the 7FFE0300 operand with an operand containing our handler's address. This way, control will initially be given to our handler, which will check whether the third argument is equal to 7 or 0x1F. If so, we return zero in the buffer and in EAX.

Parent process filename. Although I had this brilliant idea before researching SecuROM 7, it was no less interesting to see its implementation in the protector. This method is primarily intended for inexperienced researchers who aren't used to renaming the debugger's main executable. The design is simple: use the familiar NtQueryInformationProcess to get the PID of the parent process.
Obviously, if our toy was launched in Olga, then Olga is the parent process. As expected, the parent process inherits all its parent's permissions, but that's not the point... The point is that *ollydbg.exe* was the same five years ago, and it's still the same in 2011. The same goes for *idag.exe , winddbg.exe...*

Hardware Breakpoints. These are a wonderful feature, yet also a scarce one—there are only four per process. The breakpoints are checked twice (all four). The process for setting your own hardware breakpoints is as follows:

1. Control is given to the **UD2 instruction,** which always throws an exception (Illegal instruction)

2. The SEH handler for UD2 looks like this:

```
PUSH EBP
MOV EBP,ESP
PUSHED
MOV EAX,DWORD PTR SS:[EBP+8]
MOV EAX,DWORD PTR DS:[EAX]
MOV EAX,DWORD PTR DS:[12129F0]
MOV DWORD PTR DS:[12022A0],EAX
MOV DWORD PTR DS:[12022A4],26182AEF <- MOV DWORD PTR DS:[12022A4],26182AED (2ÿ)
MOV DWORD PTR DS:[12022A8],EBP
PUSH OFFSET 012022A0
CALL 00D44F40
MOV DWORD PTR DS:[12022A0],0
MOV DWORD PTR DS:[12022A4],0
MOV DWORD PTR DS:[12022A8],0
POPAD
XOR EAX,EAX
MOV ESP,EBP
POP EBP
RETURN
```

   There are two of them: **00C996B0h and 00C99702h – for the first 4 HB and, accordingly, for the second.**

3. Before moving to the Securomov SEH handler, control is always taken over by ntdll.KiUserExceptionDispatcher(pExceptionRecord, pContext). The second argument is a reference to the context structure. CALL 00D44F40 points to the VM, which modifies the Context structure (DR0, DR1, DR2, DR3) on the stack.

4. *Focus! Focus!* After exiting the handler, **ZwContinue** is executed on the next line from UD2. instructions, with the already SECURITY context, i.e. with its hardware breakpoints.
5. The hardware reset point (DR0-DR3) is triggered.
6. The SEH handler (VM) checks the context and address (where the exception occurred), and also stores its check value in DR7. If anything goes wrong, you'll be notified! :)

Software breakpoints and overriding code. While hardware breakpoints have received careful attention, what can we say about software ones? First and foremost, it all starts with the trick mentioned above:

```
MOV EAX,11D7B1C
MOV ESI,DWORD PTR DS:[EAX]
ADD DWORD PTR SS:[ESP+10],ESI …
```

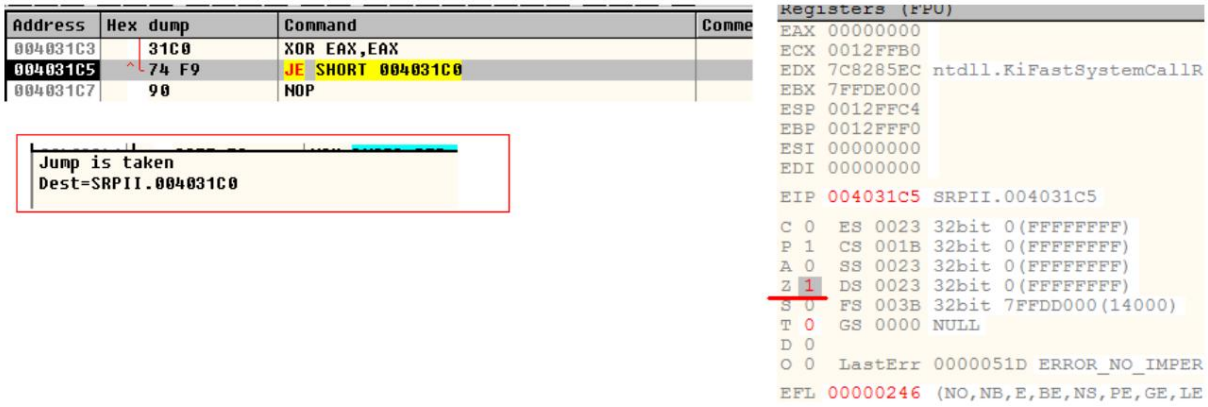
To greatly complicate tracing and detecting important ACM instructions and procedures, the code in question essentially spans the distance between them. This makes sitting and pressing F7, F8, or enabling animated tracing as useless as "shooting a pea against a wall." The developers hoped that reversers would lose their nerve and breed software breakpoints like rabbits —to quickly escape this trap. As a result,

The section's checksum is violated, and instead of the desired destination, the protector sends hapless hackers to the same place they never intended. If you're so keen to trace, there's a way to combat this. First, the trap is characterized by its PUSHFD/POPFD cycle and the following conditional branch. Therefore, it's easy to figure out where control will be transferred while bypassing the cycle. Second, hardware breakpoints are checked a total of only eight times (2 x 4) at the very beginning, so the hardware is on our side.

Incidentally, in "Safe Disk , " the overlay code is made up of "frogs"—scattered conditional and unconditional jumps that follow the same path. Unconditional jumps are when (the most common scenario):

```
XOR EAX, EAX

JZ SOME_LABEL
```

Even though the jump opcode isn't **a JMP,** it will always jump to SOME_LABEL ! This happens because the result of *an exclusive-OR* operation on the same register is always **ZERO! (NULL, ZERO, 0, donut).** Accordingly, the processor's Z (Zero Flag) flag will be set (equal to one), and your processor, obeying the set **Zero Flag,** will execute the jump condition.



Checking for software breakpoints/jumps at the beginning of WinAPI. Setting software breakpoints at the beginning/end of a procedure is de facto standard, needless to say. Fortunately, not all WinAPIs are checked, but *CreateFileA* is included . The most obvious thing is not to set software breakpoints on the first instruction. It's . *FindWindow.* Countermeasure. not even necessary on the second. For example, as a standard, *CreateFileA* generates a UNICODE string and calls

CreateFileW, so you can also use it to call *CreateFileW.* It's a different story with adapters. Whatever you say, it's convenient to use them first, since the first three instructions (prologue) in most hooked WinAPIs are:

MOV EDI, EDI

PUSH EBP

MOV EBP, ESP

To counter X-code, we'll need the system library export table and the image import table, and the fact that the compiler leaves space between them to align functions at multiple addresses. In the first case, the raw offset needs to be shifted up by one byte, which will (or should) hit a NOP. In the second case, we'll fix the function's RVA. This way, with minimal modifications, we'll force the protector to check for NOPs, while the jump below will remain safe. Speaking of the export table, it's worth mentioning that it's sometimes useful to add a handler to *GetProcAddress !*

CreateFileA/ FindWindow/OpenSCManager/RegOpenKey/ FindWindowExA/EnumWindows/GetClassName… I think everything is clear here. The only caveat is with *FindWindowExA.* Unlike *FindWindow,* it has a somewhat expanded list of existing windows to check. Specifically, using *TrayNotifyWnd,* it retrieves the tray handle, from which it enumerates the icon names. And using the *ToolBarWindow32 window handle,* it sets the PID of Windows Explorer (explorer.exe).

FindFirstFileA. A rather paranoid method of protecting against X-code: searching for ntdll*.dll in the folder containing cnc3game.dat. Apparently, the developers intended for the reverser to replace the original ntdll.dll with its own, which is filled with X-code.

GetModuleHandleA. This seemingly harmless function reminded me that it can detect libraries attached to a process using DLL jacking. SecuROM actually attempts to search the process memory for the **asr.dll library (AntiSecuROM?)**

DbgUiRemoteBreakin. Provides a wonderful opportunity to catch the debugger by setting a primary breakpoint. Sony DADC decided to be blunt and, specifically for those who like to attach it, installed an unconditional adapter in the ExitProcess library of the kernel32 library. As a countermeasure to the X-code protector (so to speak), our X-code has the right to strike back and remove the adapter or prevent it from being set altogether.

- **Procedure for checking the disc in the drive ÿ**

Highly recommended reading: http://www.insidepro.com/kk/020/020r.shtml

Split into two parts and is the SAME in versions 7.3x-8.0x

LEVEL 1 ***** SIGNATURE CHECK *******

GetDriveTypeA. A live classic! A hint that the session is about to begin.

CMP EAX, 5 //DRIVE_CDROM

CreateFileA. Opens the specified drive *(File name in the form:* \\.\D), gets the handle.

DeviceIoControl (4 TIMES). The main role here is given to DeviceIoControl, with which you can do or get anything from the device. The most important is the ControlCode, which can
take three values:

1) *(IOCTL_SCSI_PASS_THROUGHT_DIRECT)* **4D014h** or **4D004h.** The first option is information about the physical presence of the disk in the device. Then comes various information about tracks, sectors, etc. The buffer size is always 50h (80 bytes).
CMP BYTE PTR SS:[EBP-52],BL //Is there a disc in the tray?

The first check is just to confirm the disc is in the drive. If everything is OK, then we begin.

the procedure for reading and composing a signature.

At this stage, when executing *WinAPI* **DeviceIoControl,** pay attention to **InBuffer(ESP+8): from the beginning of InBuffer at offset $+14h** there is a pointer to the structure of a special buffer, where, after the call is executed, service data *(for example, the sector number) is placed.*

And now... three divorce tests to pass the first level:

```
Address   Hex dump          Command                                  Comments
010B6FC7      8BC8          MOV ECX,EAX
010B6FC9      E8 3F60F7FF   CALL SOME_CPP_COMPARE_FUNC               CHECK SIGNATURES !
010B6FCE      83E0 1F       AND EAX,0000001F                         AND ?
010B6FD1      3C 1F         CMP AL,1F                                CMP FOR LENGHT!
010B6FD3      9C            PUSHFD
010B6FD4      9C            PUSHFD
010B6FD5      83EC 24       SUB ESP,24
010B6FD8      C74424 20 BBI MOV DWORD PTR SS:[ESP+20],725CB5BB       emulate(!) check software breakpoints. but ELF known...
010B6FE0      C74424 1C 44( MOV DWORD PTR SS:[ESP+1C],44
010B6FE8      895424 18     MOV DWORD PTR SS:[ESP+18],EDX
010B6FEC      BA 546F0B01   MOV EDX,010B6F54
010B6FF1      C14C24 20 00  ROR DWORD PTR SS:[ESP+20],0              Shift out of range
010B6FF6      90            NOP
010B6FF7      897424 14     MOV DWORD PTR SS:[ESP+14],ESI
010B6FFB      0FACD2 00     SHRD EDX,EDX,0                           Shift out of range
010B6FFF      8B32          MOV ESI,DWORD PTR DS:[EDX]
010B7001      87C9          XCHG ECX,ECX
010B7003      017424 20     ADD DWORD PTR SS:[ESP+20],ESI
010B7007      83C2 04       ADD EDX,4
010B700A      66:FF4C24 1C  DEC WORD PTR SS:[ESP+1C]
010B700F      75 EA         JNE SHORT 010B6FFB
010B7011      804C24 20 01  OR BYTE PTR SS:[ESP+20],01
010B7016      8B5424 24     MOV EDX,DWORD PTR SS:[ESP+24]
010B701A      8B7424 20     MOV ESI,DWORD PTR SS:[ESP+20]
010B701E      C1E1 00       SHL ECX,0                                Shift out of range
010B7021      895424 20     MOV DWORD PTR SS:[ESP+20],EDX
010B7025      90            NOP
010B7026      8B5424 18     MOV EDX,DWORD PTR SS:[ESP+18]
010B702A      90            NOP
010B702B      897424 24     MOV DWORD PTR SS:[ESP+24],ESI
010B702F      8B7424 14     MOV ESI,DWORD PTR SS:[ESP+14]
010B7033      83C4 20       ADD ESP,20
010B7036      9D            POPFD
010B7037      90            NOP
010B7038      75 17         JNE SHORT 010B7051                       here ? ;)
```

```
010B869E  .^  EB F3         JMP SHORT 010B8693
010B86A0      9D            POPFD
010B86A1      80BF F2070000 CMP BYTE PTR DS:[EDI+7F2],0
010B86A8      9C            PUSHFD
010B86A9      68 7F2D0000   PUSH 2D7F
010B86AE      75 17         JNE SHORT 010B86C7                       // 4
010B86B0      810424 10590( ADD DWORD PTR SS:[ESP],7508910
010B86B7      C1E0 00       SHL EAX,0                                Shift
010B86BA      810424 C2000( ADD DWORD PTR SS:[ESP],8C0000C2
010B86C1      C1E6 00       SHL ESI,0                                Shift
010B86C4  ^   EB F7         JMP SHORT 010B86BD
010B86C6      25 83EC1CC7   AND EAX,C71CEC83
```

```
010A615A  .   F9            STC
010A615B  .   83D3 00       ADC EBX,0
010A615E  .   8498 C007000( TEST BYTE PTR DS:[EAX+7C0],BL
010A6164  .   895D FC       MOV DWORD PTR SS:[EBP-4],EBX
010A6167  .   B9 37F6FEFF   MOV ECX,FFFEF637
010A616C  .   8D8C29 85090( LEA ECX,[EBP+ECX+10985]
010A6173  .   9C            PUSHFD
010A6174  .   68 053B0000   PUSH 3B05
010A6179  .v  74 15         JE SHORT 010A6190                        // 2
010A617B  .   810424 E7250( ADD DWORD PTR SS:[ESP],420A25E7
010A6182  .   90            NOP
010A6183  .   810424 C2000( ADD DWORD PTR SS:[ESP],8F0000C2
010A618A  .   8BD2          MOV EDX,EDX
010A618C  .^  EB F8         JMP SHORT 010A6186
```

Passing the first check (and subsequent ones) correctly causes **SecuROM** 7-8 to generate a repeating message: *Cannot authenticate the original disc. Your disc may require a different software version.* There is a fourth check, but patching it doesn't affect the final result—Level *1 completed!*

```
010D2AB7    A3 8B83E00F   MOV DWORD PTR DS:[0FE0838B],EAX
010D2ABC    3C 0F         CMP AL,0F
010D2ABE    8B0C24        MOV ECX,DWORD PTR SS:[ESP]
010D2AC1    8D6424 04     LEA ESP,[ESP+4]
```

IMPORTANT! You need to patch the operations that affect the flags, not the transitions themselves! PUSHFD protection preserves the *EFL* state for subsequent transitions (in 7.33.017, there are no more than two), meaning the condition is checked multiple times.

```
010B6FCE AND EAX,0000001F
010B6FD1    3C 1F           CMP AL,1F //AL must be equal to 0x1F(31)
010B6FD3 PUSHFD 9C          or a multiple of this number
.....
010B7036    9D              POPFD
010B7037    90              NOP
010B7038    75 17           JNE SHORT 010B7051 // TRANSITION NOT EXECUTED
                        *** 2 ***
010A615E . 8498 C0070000 TEST BYTE PTR DS:[EAX+7C0],BL //BL = 1. It is clear that the structure [EAX+7C0] = 9 (worked with any != 1)
.....
010B7038    75 17           JNE SHORT 010B7051 //TRANSITION NOT EXECUTED *** 3 ***

010B86A1    80BF F2070000 CMP BYTE PTR DS:[EDI+7F2],0 //[EDI+7F2] = 1 (also worked with other numbers)
010B86A8    9C              PUSHFD
.....
010B86AE    75 17           JNE SHORT 010B86C7 //THE TRANSITION IS
```

EXECUTED. **THE LAST TWO CHECKS (AT LEAST 2) ARE DIRECTLY RELATED TO LEVEL 2 AND MOST LIKELY SET THE BOUNDARIES OF** *THE VIRTUAL BACKUP-REAL PHYSICAL HIT* INTERVAL **(or CYCLE, NUMBER OF ITERATIONS).**

By the way, a characteristic detail in this version for the first signature check is the “fake” calculation of the checksum of the section in the overlapping code.

QueryPerfomanceCounter & **QueryPerfomanceFrequency.** Start counting ticks in normal (preparatory) mode.

LEVEL 2 ******* CHECKING GEOMETRY (CALCULATORS) OR QPC/QPF *******

SetSystemCursor. Temp.ani

It's more complicated here, and at the time of writing, it was largely a matter of luck (the whole problem is a bug with interval calculation). ***The most important thing*** *is that it worked* ***(and more than once!!!):***

Starting with ***a completely wrong disk*** *in the virtual drive (Windows 7 installer). ;)* After everything worked fine on Windows 7 with a fake virtual disk, I decided to try on Windows 8 without *Daemon Tools* , using a regular physical drive and disk. After successfully passing the signature level, the process always terminated with exit code 1. At first, I thought it was a security issue, but then I remembered ***HKEY_CURRENT_USER\Software\SecuROM.*** I deleted the **Key** branch – the process now terminates **after** passing the geometric check (in **the Bin** table, the data structures for timing and geometry checks, the disk label under *XOR* , and something else). In short, the "fiskim" (010DF7A9) is as shown in the image below *(signed int* > 0). If you're really interested in digging into the check, set an access-based breakpoint.

```
010DF7A6    897D 64          MOV DWORD PTR SS:[EBP+64],EDI
010DF7A9    833D BF1D2D01 00 CMP BYTE PTR DS:[12D1DBF],0      -- 2fix
010DF7B0    9C               PUSHFD
010DF7B1    68 AF060000      PUSH 6AF
010DF7B6    76 15            JBE SHORT 010DF7CD               not taken
010DF7B8    810424 F70F3300  ADD DWORD PTR SS:[ESP],330FF7
010DF7BF    C1E1 00          SHL ECX,0                        Shift out
010DF7C2    810424 C3E1DA00  ADD DWORD PTR SS:[ESP],00DAE1C3
010DF7C9    EB FA            JMP SHORT 010DF7C5
```

Everything related to the second level is related to FPU operations.

If one of the checks fails, the process will terminate or an exception will be thrown. Between checks and after the third , the VM is logged into designated islands.

First check of the second geometric level

```
010EAFB2 FLD QWORD PTR DS:[11EDC78]
010EAFB8 MOV ESI,OFFSET 012D1E30
010EAFBD FSUB QWORD PTR DS:[11EDC80]
010EAFC3 FMUL QWORD PTR DS:[14AC050]
010EAFC9 FCOMP QWORD PTR DS:[1483098]
010EAFCF FSTSW AX //AH == 0
010EAFD1 TEST AH,05
010EAFD4 PUSHFD
010EAFD5 PUSH 17F7
010EAFDA JPE SHORT 010EAFF0 ; NOT TAKEN Second
```

geometric level second distribution check

```
010ED310 FLD QWORD PTR DS:[11EDC78]
010ED316 FSUB QWORD PTR DS:[11EDC80]
010ED31C FMUL QWORD PTR DS:[14A4E64]
010ED322 FCOMP QWORD PTR DS:[1483098]
010ED328 FSTSW AX
010ED32A TEST AH,05 //AH == 1
010ED32D PUSHFD
010ED32E PUSH 24F2
010ED333 JPO SHORT 010ED34A                    ; TAKEN !
```

Third duplicated test of the second geometric level

```
010CD164 FLD QWORD PTR DS:[11EDC78]
010CD16A FSUB QWORD PTR DS:[11EDC80]
010CD170 FMUL QWORD PTR DS:[14A4E64]
010CD176 FCOMP QWORD PTR DS:[1483098]
010CD17C FSTSW AX
010CD17E TEST AH,05
010CD181 PUSHFD
010CD182 PUSH 14D
010CD187 MOV ESI,ESI
010CD189 JPE SHORT 010CD1A0
```

Fix for **"Conflict with Emulation Software detected"** ***error :*** **when requesting** ***DeviceIoControl*** **with** ***IOCTL_DISK_PERFORMANCE(0x00070020)*** macro **from minor Threads (parallel geometry checking) must return 0 (zero) in** ***EAX.*** **Sometimes** ***setting (SetAffinityMask)*** **the minimum number of allowed processors (just one) for the target process helps. There's also a special function that queries thread exit codes:**

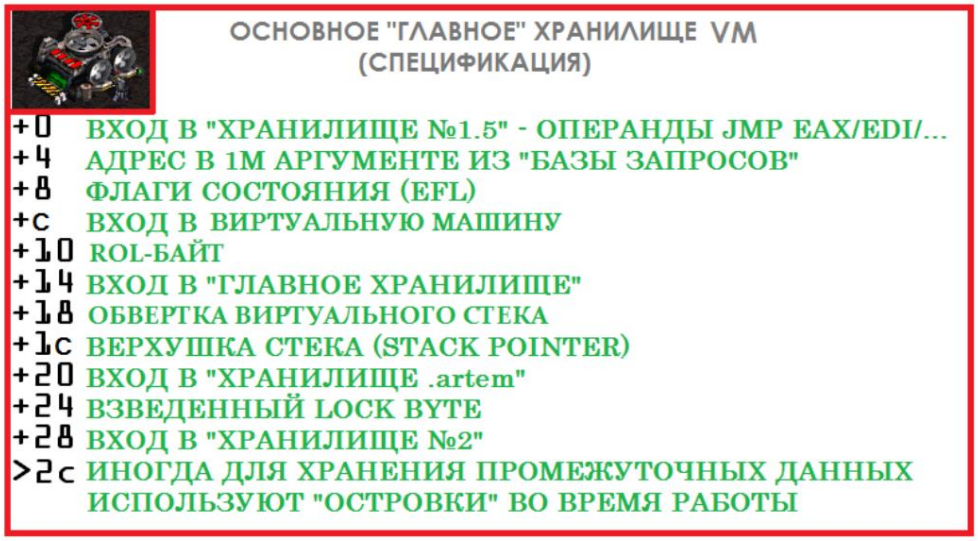
```
010F146C PUSH EBP
010F146D MOV EBP,ESP 010F146F
PUSH EBX 010F1470 PUSH
ESI 010F1471 MOV ESI,DWORD
PTR SS:[ARG.1]
010F1474 XOR EBX,EBX 010F1476
CMP BYTE PTR DS:[ESI],BL 010F1478 PUSH EDI
010F1479 JE SHORT 010F14C7
010F147B CMP BYTE PTR DS:[ESI+1],BL
010F147E JE SHORT 010F14C7 010F1480 PUSH EBX
010F1481 PUSH 1 010F1483 LEA EDI,
[ESI+228]                                   ; /Timeout => 0
                                            ; |WaitAll = TRUE
                                            ; |
010F1489 PUSH EDI 010F148A                  ; |HandleList
PUSH DWORD PTR DS:[ESI+4]                   ; |Count ; |
010F148D MOV DWORD PTR SS:[ARG.1],EBX 010F1490 CALL
DWORD PTR DS:[<&KERNEL32.WaitForMul ; \KERNEL32.WaitForMultipleObjects 010F1496 TEST EAX,EAX 010F1498 JE SHORT 010F149E
010F149A PUSH 5 010F149C JMP
SHORT 010F14C9 010F149E CMP DWORD
PTR DS:[ESI+4],EBX
010F14A1 JLE SHORT 010F14BE 010F14A3
LEA EAX,[ARG.1] ; /pExitCode => OFFSET ARG.1


010F14A6 PUSH EAX
010F14A7 PUSH DWORD PTR DS:[EDI] ; |hThread
010F14A9 CALL DWORD PTR DS:[<&KERNEL32.GetExitCo ; \KERNEL32.GetExitCodeThread 010F14AF CMP DWORD PTR SS:
[ARG.1],1 010F14B3 JNE SHORT 010F14C3 010F14B5 INC
EBX 010F14B6 ADD EDI,4 010F14B9 CMP
EBX,DWORD PTR DS:[ESI+4]


010F14BC JL SHORT 010F14A3 010F14BE
XOR EAX,EAX 010F14C0 INC EAX
010F14C1 JMP SHORT
010F14CA 010F14C3 PUSH 6 010F14C5 JMP
SHORT 010F14C9
010F14C7 PUSH 7 010F14C9 POP EAX
010F14CA POP EDI
010F14CB POP ESI
010F14CC POP EBX
010F14CD POP EBP
010F14CE RETN
```

Regarding files embedded in the protector and created in the temporary folder. I'd like to remind you that access to the temporary folder may be blocked, so functions located in *drm_dyndata_7330017.dll* may be duplicated in the image. And now—the virtual machine! Meet! • **Paul.dll (pasha.dll)**

Paul doesn't need patching. It's enough to fake the results. Previously, security had constants for Paul's response. Like 21 - failed the check, 1 - passed. Or vice versa. There was also something like a list of addresses of security functions in memory (like an IAT). So, in this IAT, security had addresses like "check failed" and "online activation check passed." By simply swapping three dwords, you could always follow the desired path. How it works now? I don't know.

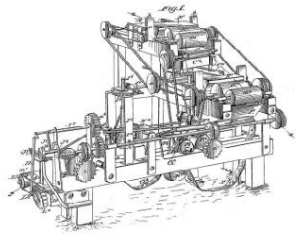
Nightshade

VIRTUAL MATHEMATICS: The
following statement may seem absurd, but in fact, cracking the so-called "SecuROM 7 virtual machine" is quite simple! The more difficult part is explaining how to do it and providing all the details. But, the journey begins with a bang!



The key to understanding the operation of a “virtual machine” (hereinafter simply VM) is a common structure I call “main storage” (the offset from the VM entry is most often +2000h. In this case, it is 20A7000h, and the VM entry is 20A5000h, respectively). This is a spiral around which the main logic of all “islands” dances, or, more objectively, 255 pieces of code, each of which operates according to a single template with the elements of the main storage (+0, +4, +10, 14) and performs only one of its tasks (the total number of tasks is four, plus several specific ones). Obviously, the islands, although different in appearance, are in fact duplicated. With regard to methods #1 and 1A, the constructed chain of islands fits beautifully into one main and simple goal - to put a given value/address at a given address! That's all! Naturally, these two main DWORD The keys are encrypted and stored in a special memory area (the virtual stack), and at the right moment, the appropriate island carrying the decoding algorithm decrypts them, after which another island performs the substitution. Method #2 goes a bit further, but in reality, it can be replaced with a single ACM instruction AND ECX, [NOT_key]. It is tailored to a special forward/backward algorithm. The function that calls it takes two arguments. The first, which specifies the address, copies the keys onto the stack for the algorithm. The second argument serves as an input number, from which the required value will be derived based on the keys. Everything here is tied to **Odd-Even Based** Cryptography (even-odd). Let's move on to a more in-depth analysis.

How to find VM?



JMP SHORT $+

One of the ASCII-Z strings (the most common ones are shown): "<space for rent>"

«You are now a restricted area»

«Nobody move, nobody gets hurt»

For this chapter, please refer to the appendix to the article:
Materials for the operation of the SecuRom virtual machine version 7.33.0017

The following operations on any island are always performed by default:

1) Reading the resource (+4), adding the result to the resource (+C), resulting in the current virtual pointer ESP to the virtual stack.

2) Reading (extraction by analogy with POP EDX, see point 5) by virtual pointer: A) one double (control) word, if the island task is not related to
decryption

B) two double words. The first needs to be decrypted, the second is a control byte. **3) Reading the resource (+10) - the control byte. If the island operated according to 2-B, then the first stage of decryption will be performed with the control byte, the second stage is the XOR mask with the same key. Next, a special byte (or 3 bytes) will be taken from the control word, which will be added (in addition, it is possible: XOR with a specific key, subtraction, addition) to the current ROL byte, the new value will be stored in the resource (+10) for the next island. 4) Obtaining a byte from the control word, multiplying it by 4, adding the resulting offset to storage address #1.5, reading the double word from there, bitwise shifting it to the right ROR (an operation equivalent to division), then adding it to the VM entry address.**

This is how the address of the next island or VM exit is obtained. This algorithm is stable for everyone. 5) As a consequence of point 2, it is naturally necessary to correct the virtual pointer ESP, to the resource (+4) is added: A) 4 bytes, if one double word was used (pulled out) **B)** 8 bytes, if two double words were used (pulled out) **C)** *Exception:* 10(1) bytes. Related to some differences in the operation of VM in SEH-

The handlers of the protection itself and the VM itself in the "Tiberium brawl." To confuse attackers, in the first case, the virtual stack is initially deliberately made "crooked" (not a multiple of 4), and about a dozen "islands" do not carry any payload. After their

Once completed, everything will fall into place.

The instructions ***ADD DWORD PTR DS:[(+4)], 8*** *and* ***ADD DWORD PTR DS:[(+4)],4*** *make it easy to identify the island type!*

6) The actual jump to the next island JMP EAX(JMP EDI/RET) or exit from VM.

Explanation of the above: How are offsets
extracted from the control word (before decoding and subsequent transition to the requested island). In the example, control word = 889AFC25; ROL byte = D7h:

```
SHL EDX,30 ; EDX = FC250000 SHR
EDX,18 ; EDX = 000000FC // received the required control byte (2nd from the right)
ADD DL,CL ; DL = FC ; CL = D7 ; => DL = D3 //ROL byte redistributes the original value!

SHL EDX, 2 ; EDX = 000000D3 * 4 = 0000034C /* increase the result by 4 times!
 (SHL REG_32, 2) The only arithmetic operation performed on the result, without the ROL byte. This gives us the offset for
Storage 1.5 and the cells in the main memory! */

MOV EAX,DWORD PTR DS:[ (+0) ] ; EAX = FFF2C000 //retrieve resource (+0)
ADD EAX,DWORD PTR DS:[ (+C) ] ; EAX = FFF2C000 + 020A5000 = 01FD1000 //forming the virtual address of Storage #1.5

ADD EAX,EDX ; EAX = 01FD1000 + 0000034C = 01FD134C //Resulting offset
we add it up with him
MOV EDX, DWORD PTR DS:[EAX] ; EDX = C003FFF0 //read the encoded offset of the next island The last five instructions have
one equivalent instruction,
which is also used on some islands in the VM (EDX resource(+0) or Storage address #1.5): PUSH DWORD PTR DS:
[EAX*4+EDX]
```

Decoding the received offset from Storage #1.5. A typical operation at the end of an island is a transition to the next one. Let's assume that the encoded offset C003FFF7 (EDI register) was retrieved from Storage #1.5 and CPUID returned 92h in AL (for ROR, the operand is equivalent to dividing by F0h):

```
MOV EAX, 1 //EAX = 1 for the CPUID instruction indicates that the
instruction will return the
so-called CPU signature - information about the processor (model, stepping)

CPUID //coding/decoding of offsets in Storage table #1.5 is performed using this ACM instruction!

AND EAX,FFFFFFDF //get the byte to shift via CPUID
ROR EDI, CL ; EDI = C003FFF7/ F0 = FFFDF000 //DECODE
```

ADD EDI, DWORD PTR DS:[(+C)] ; EDI = FFFDF000 + 020A5000 = 02084000 /*ADD THE RECEIVED OFFSET TO THE VM ENTRY ADDRESS*/

JMP EDI ; EDI = 02084000 //JUMP TO THE NEXT ISLAND In order to put your address in #1.5, you need to perform the reverse actions: 1. Calculate the delta (RVA of the VM entry minus RVA of the beginning of your code) 2. Execute:

```
MOV EAX, 1
CPUID
AND EAX,FFFFFFDF
ROLE EDI, CL
```

Note: Short list of valid EAX register values before calling CPUID.

CPU Signature(EAX = 1)

The bits returned in the EAX register after executing CPUID.

3:0 *– Stepping (processor stepping)*
7:4 *– Model*
11:8 *– Family*
13:12 *– Processor Type*
19:16 *– Extended Model*
27:20 *- Extended Family AMD and Intel have different nuances in the values that the bits return.*

For example, for the AMD Phenom II X4 940 processor, the value returned in EAX will be 00100F42. Accordingly, 42h is used to encode/ decode the Storage 1.5 offset table. Other EAX values are used for CPUID execution.

EAX = 0(CPU Vendor ID)

The EBX, EDX, and ECX registers contain the first 12 bytes of an ASCII string identifying the manufacturer. For example, ***AuthenticAMD*** *indicates that the processor is supplied by* ***Advanced Micro Devices (AMD).***

EAX = 2(Cache and TLB descriptor information)

TLB – Translation Lookaside Buffer. In a nutshell, this buffer is a CPU cache used to increase the speed of virtual address translation.

EAX = 3(CPU serial number)

Depending on the processor model and manufacturer, the serial number is returned in EDX:ECX (or EBX:EAX).

The SecuROM 7 VM *virtual stack* stores **two** types of data: 1) Control word (Control DWORD). Manipulates the VM's operating logic.

Contains 4 bytes, of which: - The next ROL byte

value for the next island. Always present. As standard, 1 byte. Exception: 2 bytes (of which one byte is “glued together”, which will similarly be added to the resource (+10)). - The address of the next island in Storage #1.5 [1 byte] Always present. 1 byte. By

is essentially an encoded offset.

-The address of the cell in the main storage. Depending on the function of the current island. is essentially an encoded offset. **2)** Encrypted data in the size of one DWORD. Such as: The address of the requested function, the address ASCIIZ strings, numbers. This type is used directly by the islands that decrypt them (using the ROL byte).

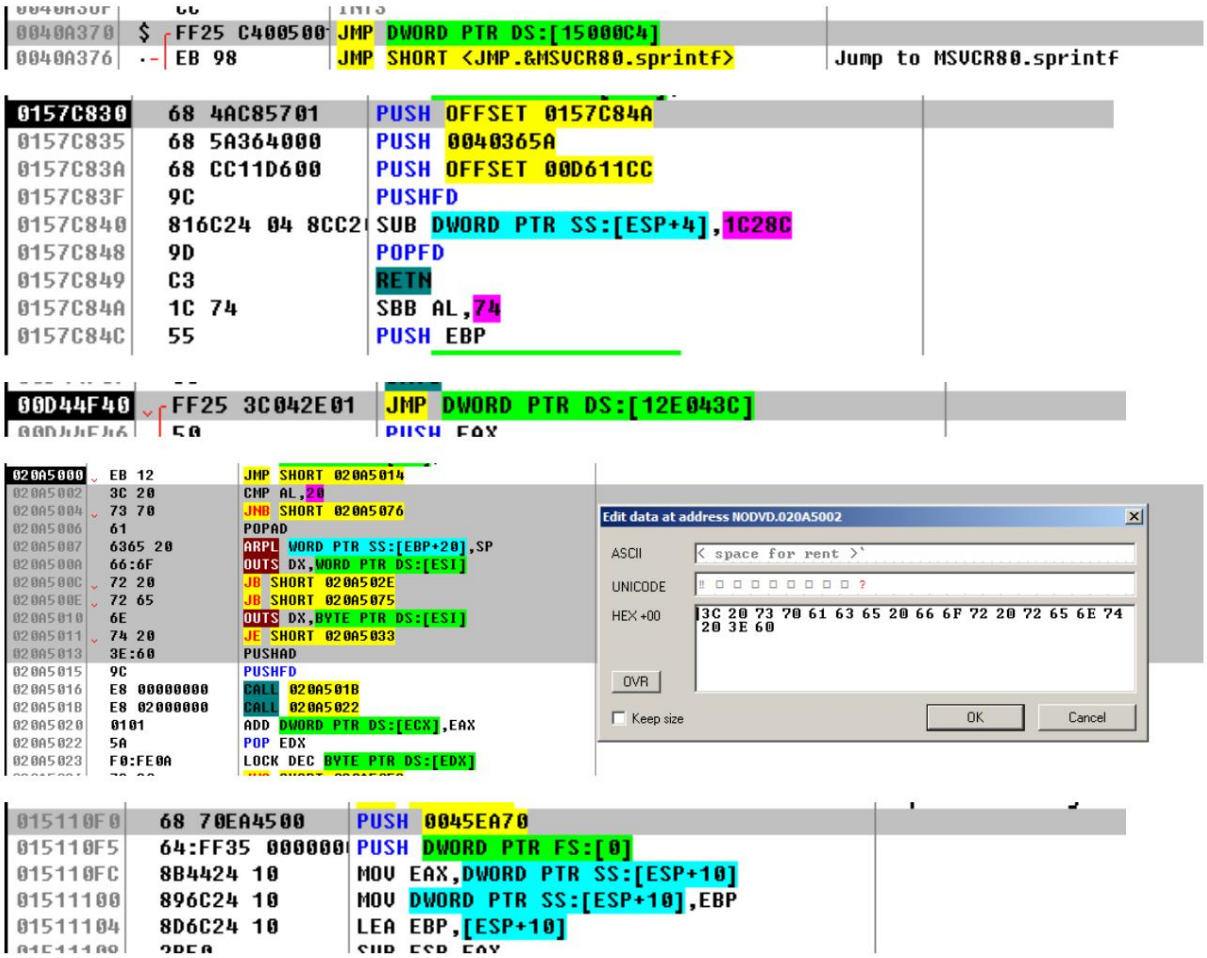
METHOD #1 – CALLING AN INTERNAL *FUNCTION (015110F0)*, THE ADDRESS IS IN THE VIRTUAL STACK	
0040A006 **PUSH 58** 0040A008 **PUSH OFFSET 00B68A40** 0040A00D **CALL 0040A370 //**call an internal function	The main code of the program
0040A370 **JMP DWORD PTR DS:[15000C4]**	"Strelka." First time **-** Going to the base request. During the **VM** operation , there will be replaced by **015110F0,** i.e. after the first we immediately get to the required call function
0157C830 **PUSH OFFSET 0157C84A [0155741C]** 0157C835 **PUSH 0040365A** 0157C83A **PUSH OFFSET 00D611CC** 0157C83F **PUSHFD //**garbage 0157C840 **SUB DWORD PTR SS:[ESP+4],1C28C** 0157C848 **POPFD //** garbage 0157C849 **RETN //00D611CC-1C28C=00D44F40** 0157C84A **1C 74** **// 0155741C** 0157C84C **55** 0157C84D **0100**	2nd argument: **LPDWORD = 0157C84A** The first one starts at the address just below **DWORD** from virtual stack **(0155741C)** 1st argument: **(VOID) 0040365A** It doesn't make sense for method **#1,** because it will replacements for the address of the required procedure closer to exiting **the VM**
00D44F40 **JMP DWORD PTR DS:[12E043C]**	Roadblock. Transfers to **VM.**
JMP SHORT $+14 **<space for rent>** **PUSHED** **PUSHFD** **CALL $+5** **CALL $+7** **Lock byte +1 byte** **POP EDX //lock byte offset** **LOCK DEC BYTE PTR DS:[EDX]** **JNS SHORT START_VM** **CMP BYTE PTR DS:[EDX],0** **PAUSE** **JLE SHORT CMP** **JMP SHORT LOCK** **START_VM:** **MOV EBX,EDI** **MOV ECX,100** **MOV EAX,0** **REP STOS DWORD PTR ES:[EDI] //**CLEARING MAIN STORAGE FROM PREVIOUS CALL… **//** …AND ITS NEW INITIALIZATION **MOV DWORD PTR DS:[EBX],EAX** **POP EAX** **MOV DWORD PTR DS:[EBX+4],EAX** **MOV EAX,DWORD PTR SS:[ESP]** **MOV DWORD PTR DS:[EBX+8],EAX** **MOV DWORD PTR DS:[EBX+0C],EDX** **MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h** **MOV DWORD PTR DS:[EBX+14],EBX**	Spin lock **Space for rent** – ASCII string The first island

MOV DWORD PTR DS:[EBX+1C],ESP **MOV EAX,OFFSET 01212A10** **MOV DWORD PTR DS:[EBX+20],EAX** **MOV EAX,OFFSET 020A6000** **MOV DWORD PTR DS:[EBX+28],EAX** ... **JMP EAX** **MOV EAX,1 //**SECOND ISLAND- nop **SHL EAX,2** **ADD EAX,EBX** ...	
MOV EBX, [V-stack pointer] **ADD EBX,** 4*number of DWORDs to encrypted address **MOV EAX, [EBX] //** **EAX = 243BBBD6** **XOR EAX, 43E2AB9D //EAX = 15000C4** **MOV** [CELL IN DATA STORAGE AREA_1], **EAX** **ADD EBX,** 4*number **of DWORDs** to next encrypted address **MOV EAX, [EBX] //** **EAX = 42B3BB6D** **XOR EAX, 43E2AB9D //EAX = 015110F0** **MOV** [CELL IN DATA STORAGE AREA_2], **EAX** **MOV EAX,** [CELL IN DATA STORAGE AREA_1] **MOV EDX,** [CELL IN DATA STORAGE AREA_2] **MOV [EAX], EDX //** MOVING THE ARROWS!!! **MOV [ARG. 1], EDX //** the one that is VOID **JMP ARG. 1 //**go to the requested internal procedure	What essentially happens in general terms **V-stack pointer – (+1C) in** main storage **43E2AB9D** – decryption key

The VM itself has three operating modes, each with its own number of exits. Meanwhile, we're beginning to explore the VM's various modes: Mode 1 – calling an internal function (which, by the way, can be a single ACM command) within the program being protected; Mode 1A – essentially the same as the first, except that its VM access is typically performed directly through the EAX register (CALL EAX), which generates the request base and immediately passes it to the VM (i.e., CALL 00D44F40); after the final RET, control is transferred to the requested WinAPI; and Mode 2 – radically different from the first two, with its output in the EAX and ECX registers. The request base. The most important first argument is a simple "wrapper" that points to the beginning of the area in the .secuROM section with which the VM will operate. To understand, imagine it as a "virtual stack," and the resource (+4) in the "main storage" as the virtual equivalent of the ESP register, which in this example is initially set to address 0155741C. The second argument is the return address to which the final RET in the VM will transfer control. For the first two methods, its specified value is not used (address 0040365A falls in the middle of the instruction) and will be corrected by the VM in the stack to the address of the final requested function. The opposite situation for method #2: the return address will be just below the request base, and the final RET in the virtual machine will transfer control to it. From the request base, a jump to address 00D44F40 works, a kind of VM checkpoint that always jumps to the beginning of the VM. Even though its equivalent addresses exist, all request bases point to it, and everything passes through it into the VM. Let's examine the beginning of the VM. The first island (I deliberately left it as is; the rest are unobfuscated). First up is obfuscation, which is simple: use mathematical operations to obtain offsets in the main storage (as is standard). For example: MOV EDX,60 SHR EDX,3 ; EDX = 0Ch //resource (+C)

MOV ECX,72
XOR ECX,00000062; ECX = 10h //resource (+10)

However, not all islands are so simple; some employ a long, complex algorithm for obtaining the +4 offset. After obtaining the offsets and performing operations in the main storage, SecuROM returns the pointer to its beginning, subtracting the used offset. It's worth noting that the two registers most often involved in obfuscation are EDX and ECX, their values first saved to the stack. There are few garbage instructions, so knowing the unchangeable operating logic of all islands allows you to drink tea without worrying about quickly removing all code obfuscation.



Let's move on to analyzing the functionality. Standard operations of saving registers and flags and arming the spinlock indicate that only one thread is authorized to work with the VM; the others will wait for the spinlock to be cleared, which occurs upon exiting the VM. Now we need to initialize the main storage, first zeroing out its contents. Note that our virtual ESP or resource (+4) will implicitly store a pointer to the virtual stack. This is indicated by the instruction for the ACh offset from the VM entry, which subtracts the VM entry address from the pointer address (more precisely, its offset is stored). Accordingly, to read the control word from the virtual ESP, the machine needs to add the VM entry address back, which will also be placed in the main storage in the (+C) resource. In addition to (+4), the virtual stack "wrapper" is located at offset +18, which can always be used to determine its beginning.
After resource (+4), resource (+10) and the DWORDs in the virtual stack deserve even more attention. Looking at the former, you can see that its initial value is always 95h. Why 95h? The actual value isn't important; compliance with the rules of the entire system is crucial. Imagine you have two very simple equations: X + Y = 5 (1)

4+Y = 7 (2) If we
ignore the second, then the number 5 can be obtained in different ways, however, only one option is correct, when Y = 3 and X = 2. The latter is the analogue of 95h. Obviously, by name, the resource (+10) is associated with the ACM command ROL, which is familiar to programmers as a bitwise left shift. It carries the main decoding function, and with the help of control bytes, its second operand receives the desired shift each time. However, to complicate life for reversers, islands perform additional operations with the ROL byte: addition, subtraction, XOR with a certain key, so it will not be possible to fit everything neatly into one algorithm and remove everything at once. Thus, everything related to decryption relies on the ROL byte. If you have no idea about this resource, imagine what a virtual machine would look like without it: in the virtual stack, control bytes for operations with the same

The cells in the Storages matched, and looking ahead, the encoded addresses could be calculated directly using a single XOR mask. Thus, by looking at the virtual stack, one could reproduce the VM's operating algorithm with very high accuracy without digging into the islands, not to mention the fact that the islands with the tasks assigned to them would simply duplicate each other. Therefore, "thanks" to the ROL byte, the reverse engineer needs to clarify the details of decoding the actual bytes in the control word on each island. It's worth noting that the ROL byte is used in Storage #1.5 only for decoding its actual control byte, while the algorithm for encrypting/decrypting offsets, which uses the result of CPUID operation in the AL register with a tandem bitwise shift operation, but this time to the right (ROR), is separate. Regarding the remaining resources: since the islands are scattered across the entire allocated memory, the address of which is the result of the *VirutalAlloc API,* there is a "what's where" problem. This is solved by resource (+0) with the aforementioned storage #1.5, which stores the constant offsets from the start of the VM entry for all the islands (essentially, the calculated Delta offsets). In our case, it is 01FD1000, and as expected, its offset is stored in the main storage. Equally important is that the CPUID result can be different for each processor, so it's clear that the offset can only be correctly decrypted using the same CPUID value with which it was encrypted! Separately (slightly below) from the general initializer of the main storage, the other two addresses are stored in resources (+20) and (+28). Essentially, no work is done with them, except that during initialization of storage #2, the EAX register will be placed in the first double word after executing CPUID, the lower part of which will be used as the key, while the island will ignore its own CPUID. It is clear that this permutation will not change the key for ROR. In the second word, which contains the result 0h-EAX(CPUID), the island that retrieves this value performs the inverse operation, which always results in zero, which is added to the ROL byte in the main storage. Here, too, it is clear that the operation is meaningless. In fact, the function of the first island is the initialization of the main storage, which was just discussed. Next, the default operation occurs: reading the control word through the "wrapper" (the "wrapper" is needed only for the first island), then the control word (BCB2C5D6) is retrieved through virtual ESP. Bytes B2 and C5 will be the same for all upper control words retrieved by the first island. It follows that they can be used to identify the beginnings of all existing virtual stacks. Furthermore, they also indicate that the second island will be the same for all calling VMs. As we can see, the VM first fetches byte D6, then adds it to the resource (+10), thereby preparing the next ROL byte for the second island. Byte C5 will be used together with the current control byte (95h) to obtain an offset into storage #1.5. They will initially be added together, then the result will be quadrupled, and the resulting offset will be added to 01FD1000. This process is interrupted by adding 4 bytes to the resource (+4). Clearly, only one control word was used, and our virtual pointer has shifted down (up by higher addresses), thereby preparing address 1557420h for the second island. In short, a typical POP EAX. Let's return to what was interrupted. VM needs to request the encrypted offset of the second island from storage #1.5. Decoding is performed using the combination CPUID(EAX = 1); MOV CL,AL;ROR EDX,CL. The raw offset will be added to the beginning of VM, and the resulting RVA will be popped from the stack into the EAX register. The next jump will take us to the second island. First of all, it becomes obvious that the 6 principles highlighted above and discussed in detail in the first island are implemented unchanged in the second (and in the third, fourth, and so on until the end). It is worth noting that there is a hierarchy in the use of registers. Thus, ECX is assigned to the current ROL byte, EDX to the control word, EAX to the virtual pointer or encrypted double word *(2-*

B), EBX behind the pointer to the main storage, EDI, as an assistant to the previous one, works with resources (an analogy with temporary variables injected by the compiler is appropriate). This explains the presence of a special area in the main storage, used for exchanging data directly between islands. This requirement, for example, is used by islands with the rewrite algorithm in the "arrow." They only need to know what and where to rewrite. Therefore, two islands that will decode the rewrite address (15000C4) and the address of the requested function (015110F0) will place their results in the specified area. And naturally, the control word includes control bytes for this purpose, which point to the resources where the aforementioned addresses are stored. The second island is actually associated with this area, but as you can see, it is an assistant: its function is to copy data from resource to resource. Since the work has just begun, there is naturally nothing to copy, so at this stage it can be considered akin to the nop instruction. By the way, we can formulate a rule: the VM's real work begins with the island that first entered data into the main storage area. Obviously, before that, the others only "twiddled" the ROL byte and "ate" several control words from the virtual stack. The next, third island (it, the fourth, and the fifth are logically interconnected) will be the most intriguing, after all, it's a decryption algorithm. As mentioned above, the VM first needs two double words on the stack: the first is the encrypted address and its

The first needs to be decoded (243BBBD6), and the second is the already familiar control word (4A191A68). I think the guys at SONY DADC were deliberately forced to take this approach (direct addresses in the virtual stack) due to the limitations imposed by the 32-bit processor. The control word only holds 4 bytes, and all are occupied. Therefore, it's impossible to create storage and, consequently, ROL bytes for decrypted addresses; there's simply no room for them. A 64-bit processor with its 16 registers is a different matter! The control word could hold 8 control bytes, significantly expanding the scope for new storage, and doubling the number of registers would provide scope for new tricks. The downside is the increased size of the islands and, as a result, a much greater loss in the performance of the protected application. But let's get back to the 32-bit world: the next thing the island in question does is take care of the next one, preparing a new ROL byte for it. It's mundane, but then comes what they were created for: ROR DL, CL; ROR EDX, 8. Imagine a revolver with a cylinder designed for 4 bullets, the first assembly command acts as a striker, the second rotates the cylinder to the next cartridge: for each byte of the decoded address, a cyclic shift to the right is performed by the value of the control byte! Here it is worth noting one feature - depending on the "range" of the shift, you can obtain multiplication and "mirror reflection". What can I say!!! You can see it all for yourself: When CL = FC; DL = 24 => 42 When CL = 64; DL = 7D => D7 When CL = D7; DL = 21 => 42 In the final step, the result is XORed, with the key being the same for each island with the decryptor (43E2AB9D). Essentially, the algorithm is quite trivial, but as has been mentioned many times, decryption relies on the current ROL byte in the low-order CL register! So, the decryptor has converted the encoded double-word into address 015000C4, which we already know from the "arrow." Next, the first byte is extracted from the control word, multiplied by 4, and, in one word, the result of the aforementioned round dance is written to the "main storage" (020A7110). As you can see, the thesis that the islands are not connected by registers or the stack is correct. The last two operations are pointless to comment on. Unless our virtual stack moves up by the legal two double-words. Let's jump to the fourth. The most astute have already realized that this is also a decryptor. The result of its work is 015110F0, moving on to it, it's easy to guess that this is the destination address of our CALL 0040A370! Excellent! The address will be stored (of course) in cell 020A7290. The fifth island. We have 015110F0 and 015000C4, on the operand of which the “arrow” operates. What remains to be done? Well, translate, of course! And indeed, it translates – the corresponding bytes are hardwired into the control word - the addresses 020A7290 and 020A7110 are extracted. The last word is for POP DWORD PTR DS:[EDX], which performs the magical substitution from the request database for the required function. Thus, subsequent calls to 0040A370(015110F0) will be direct. The sixth island. It seems that the logic of its operation defies any explanation. Where did the address 0022FF6C come from in 020A701C? It was there before (you'll find out below who left it). This is the saved top of the stack at the time of execution of the next ACM instruction of this island. After analyzing the last four islands, its purpose becomes clear. The seventh island, which is another unusual decryptor, extracts the number 24h. Unusual, if you compare the decoded double word 02482334h and the only instruction for decrypting XOR EAX, 02482310h, it is nothing other than

V

DL = 3B => B3

How:

AND EAX,000000FF

XOR AL, 10h

It's easy to guess that only the last byte is needed. Likewise, the familiar 0040365A is stored at address 0022FF90 (0022FF6C + 24h) on the stack. The return address was meaningless from the start, so replacing it with the address of the required procedure is a natural occurrence, especially as the final step. The address is stored in cell 020A70A0, and we're already in the eighth, which decodes and overwrites 015110F0 in cell 020A7290—a kind of safety net. The ninth island summarizes the work of the last three islands, extracting freshly minted addressees from the last-mentioned cells and overwriting the return address. After the ninth island, the standard exit procedure begins: resetting the spin lock, restoring the processor registers and flags, correcting the stack, and the long-awaited transition to the required function at address 015110F0.

METHOD #1A – CALL WinAPI (SetUnhandledExceptionFilter), the offset is taken from the import table	
0044F64C **PUSH 00406E34 //**arg. 2 for VM 0044F657 **PUSH 004011C3 //**arg. 1 for VM 0044F65C **MOV EAX,B3A7335C** 0044F661 **PUSHFD //**garbage 0044F662 **XOR EAX,B3737C1C //**B3A7335C ^ B3737C1C = D44F40 0044F667 **POPFD //**garbage 0044F668 **JMP SHORT 0044F669** 0044F669 **CALL EAX**	The main code of the program Unlike Method No. 1, there is no query base, and the calling method is characteristic **CALL EAX**
00D44F40 **JMP DWORD PTR DS:[12E043C]**	The checkpoint. The characteristic "Strelka" is missing. The arguments have been set – we move straight to the VM.
JMP SHORT $+14 **<space for rent>** **PUSHED** **PUSHFD** **CALL $+5** **CALL $+7** **Lock byte +1 byte** **POP EDX //lock byte offset** **LOCK DEC BYTE PTR DS:[EDX]** **JNS SHORT START_VM** **CMP BYTE PTR DS:[EDX],0** **PAUSE** **JLE SHORT CMP** **JMP SHORT LOCK** **START_VM:** **MOV EBX,EDI** **MOV ECX,100** **MOV EAX,0** **REP STOS DWORD PTR ES:[EDI] //**CLEARING MAIN STORAGE FROM PREVIOUS CALL… **//** …AND ITS NEW INITIALIZATION **MOV DWORD PTR DS:[EBX],EAX** **POP EAX** **MOV DWORD PTR DS:[EBX+4],EAX** **MOV EAX,DWORD PTR SS:[ESP]** **MOV DWORD PTR DS:[EBX+8],EAX** **MOV DWORD PTR DS:[EBX+0C],EDX** **MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h** **MOV DWORD PTR DS:[EBX+14],EBX** **MOV DWORD PTR DS:[EBX+1C],ESP** **MOV EAX,OFFSET 01212A10** **MOV DWORD PTR DS:[EBX+20],EAX** **MOV EAX,OFFSET 020A6000** **MOV DWORD PTR DS:[EBX+28],EAX** **...** **JMP EAX** **MOV EAX,1 //**SECOND ISLAND- nop **SHL EAX,2** **ADD EAX,EBX** **...**	Spin lock **Space for rent** – ASCII string The first island

MOV EBX, [V-stack pointer] **ADD EBX,** 4*number of DWORDs to encrypted address **MOV EAX, [EBX] //** **EAX = 4340FA51** **XOR EAX, 43E2AB9D //EAX = 00A251C** **MOV EAX, [EAX] // SetUnhandledExceptionFilter** **MOV** [CELL IN DATA STORAGE AREA_1] , **MOV EAX,** **EAX** [CELL IN DATA STORAGE AREA_1] **MOV [ARG. 1], EAX //**the one that is VOID **JMP ARG. 1 //** Switch to WinAPI	What essentially happens in general terms **V-stack pointer – (+1C) in** main storage **00A251CC – RVA WinAPI SUEF in import table** **43E2AB9D** – decryption key

Regarding method #1A, some of the calls are subordinate to the PreStaticInitDebug(0044F4D1) procedure. It's cut by PUSH/RET-style adapters. Plus, there are jumps into the middle of ACM instructions. Here's a partial list of the WinAPIs called: SetUnhandledExceptionFilter, GetModuleFileNameA, DeleteFileA (\errors.txt), GlobalFree.

```
0044F669   FFD0          CALL EAX                                  SetUnhandleExceptionFilter
0044F66B   68 00020000   PUSH 200
0044F670   8D85 00FEFFFF LEA EAX,[EBP-200]
0044F676   50            PUSH EAX
0044F677   56            PUSH ESI
0044F678   68 0E284F6F   PUSH 6F4F280E
0044F67D   68 E1114000   PUSH 004011E1
0044F682   9C            PUSHFD
0044F683   817424 08 4E( XOR DWORD PTR SS:[ESP+8],6F9B674E
0044F68B   816C24 08 4E( SUB DWORD PTR SS:[ESP+8],6F9B674E
0044F693   817424 08 6A: XOR DWORD PTR SS:[ESP+8],553B276A
0044F69B   817424 08 6A: XOR DWORD PTR SS:[ESP+8],553B276A
0044F6A3   814424 08 4E( ADD DWORD PTR SS:[ESP+8],6F9B674E
0044F6AB   9D            POPFD
0044F6AC   58            POP EAX
0044F6AD   870424        XCHG DWORD PTR SS:[ESP],EAX
0044F6B0   FFD0          CALL EAX                                  GetModuleFileNameA
0044F6B2   8D85 00FEFFFF LEA EAX,[EBP-200]
0044F6B8   6A 5C         PUSH 5C
0044F6BA   50            PUSH EAX
0044F6BB   68 CFF64400   PUSH 0044F6CF
0044F6C0   FF35 6055A20( PUSH DWORD PTR DS:[<&MSVCR80.strrchr>]
0044F6C6   C3            RETN                                      RET is used as a jump
```

Incidentally, I'd like to point out that, in addition to the "arrows," the VM also restores addresses for some ASCIIZ strings, and in some cases, numeric values (small ones). Similar to the first method, the virtual stack stores an address that points to the WinAPI offset in the game's import table. The procedure entry point is taken from this offset. **METHOD #2 – WORKING WITH FORWARD AND BACKWARD ALGORITHM FUNCTIONS**

FORWARD MOVE ALGORITHM (0152253D) AND ITS ATTACHED VM CALL

FOR REVERSE (1520C0A) IT'S THE SAME EXCEPT THE ABSENCE OF HIDDEN COMMANDS IN VM!

0152253D PUSH EBP (Func_ARG 1. - DWORD, Func_ARG.2 - LPDWORD) ... **01522551 MOV EAX,CFCECDCC //characteristic garbage** **01522556 MOV EAX,00413F5A //typical garbage** **0152255B MOV EAX,1 //typical garbage** **01522560 MOV EAX,0 //typical garbage** **01522565 MOV EAX,CFCECDCC //characteristic garbage** ... **0152256B PUSH 00401283** **01522570 PUSH OFFSET 0152257E**	The main code of the program The characteristic feature of this method is: 1) Full connectivity with two procedures (0152253D) 2) As a consequence of the first, the typical code of two algorithms (P and Ob) 3) Characteristic garbage code in beginning/end of procedures P and Ob (MOV EAX, garbage) 4) Unlike the first two methods for VM, the return address specified in the first argument makes sense (we always return a few bytes lower, since control does not go beyond

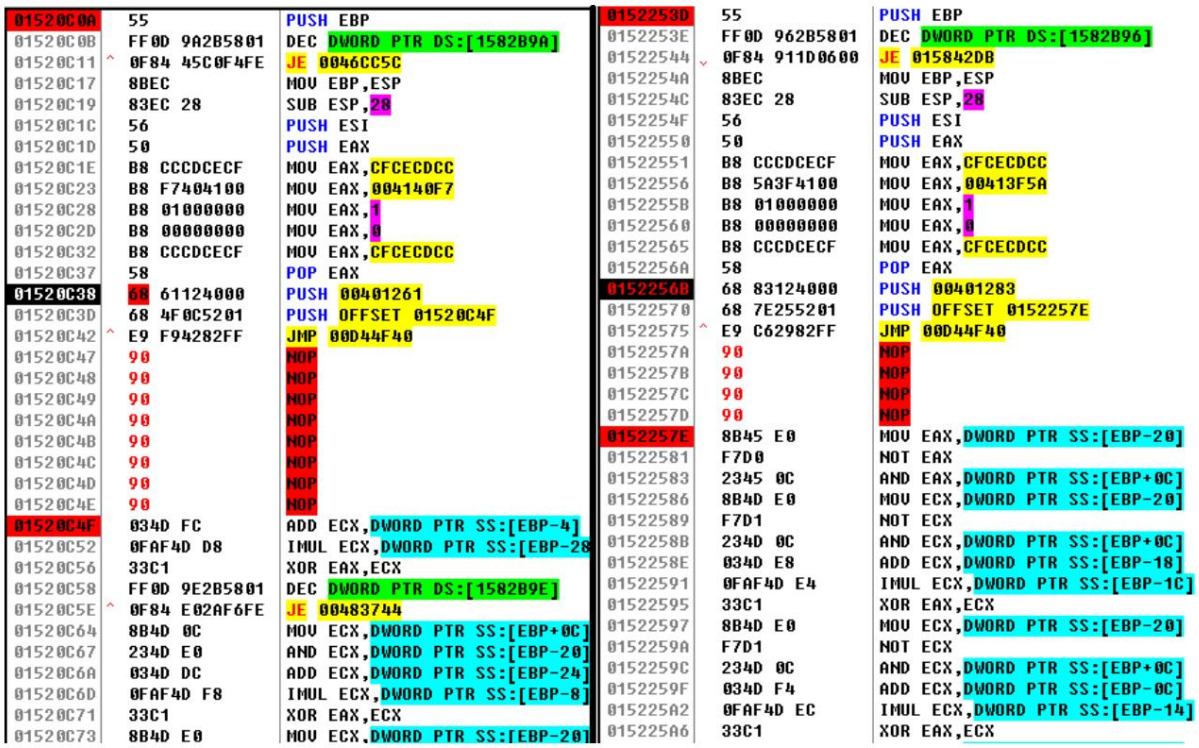
01522575 JMP 00D44F40 //go to VM 0152257A NOP 0152257B NOP 0152257C NOP 0152257D NOP 0152257E MOV EAX,DWORD PTR SS:[EBP-20] //continuation of main procedures … (AND / XOR) 01522662 MOV DWORD PTR SS:[EBP-10],EAX 01522666 MOV EAX,CFCECDCC //characteristic garbage 0152266B MOV EAX,00413F5A //typical garbage 01522670 MOV EAX,0 //typical garbage 01522675 MOV EAX,0 //typical garbage 0152267A MOV EAX,CECDCCCB //characteristic garbage 01522680 MOV EAX, DWORD PTR SS:[EBP-10] //ÿÿÿ ÿÿÿÿÿ EAX! 01522684 LEAVE 01522685 RETURN	basic procedure)
00D44F40 JMP DWORD PTR DS:[12E043C]	The checkpoint. The characteristic "Strelka" is missing. The arguments have been set – we move straight to the VM.

JMP SHORT $+14
<space for rent>
 	
PUSHED
PUSHFD
CALL $+5
CALL $+7
 	
Lock byte +1 byte
 	
POP EDX //lock byte offset
LOCK DEC BYTE PTR DS:[EDX]
JNS SHORT START_VM
CMP BYTE PTR DS:[EDX],0
PAUSE
JLE SHORT CMP
JMP SHORT LOCK
 	
START_VM:
MOV EBX,EDI
MOV ECX,100
MOV EAX,0
REP STOS DWORD PTR ES:[EDI] //CLEARING MAIN STORAGE FROM PREVIOUS CALL…
 	
// …AND ITS NEW INITIALIZATION
 	
MOV DWORD PTR DS:[EBX],EAX
POP EAX
MOV DWORD PTR DS:[EBX+4],EAX
MOV EAX,DWORD PTR SS:[ESP]
MOV DWORD PTR DS:[EBX+8],EAX
MOV DWORD PTR DS:[EBX+0C],EDX
MOV BYTE PTR DS:[EBX+10],95 //CRYPT-BYTE - 95h
	Spin lock
	
	Space for rent – ASCII string
	
	The first island

MOV DWORD PTR DS:[EBX+14],EBX **MOV DWORD PTR DS:[EBX+1C],ESP** **MOV EAX,OFFSET 01212A10** **MOV DWORD PTR DS:[EBX+20],EAX** **MOV EAX,OFFSET 020A6000** **MOV DWORD PTR DS:[EBX+28],EAX** ... **JMP EAX** **MOV EAX,1 //**SECOND ISLAND- nop **SHL EAX,2** **ADD EAX,EBX** ...	
MOV EAX, [Func_ARG.2] **MOV EBX, [EAX+2C]** **MOV [EBP-28], EBX** **MOV EBX, [EAX+30]** **MOV [EBP-24], EBX** **MOV EBX, [EAX+13]** **MOV [EBP-20], EBX** **MOV EBX, [EAX+18]** **MOV [EBP-1C], EBX** **MOV EBX, [EAX+1C]** **MOV [EBP-18], EBX** **MOV EBX, [EAX+20]** **MOV [EBP-14], EBX** **MOV EBX, [EAX+24]** **MOV [EBP-C], EBX** **MOV EBX, [EAX+34]** **MOV [EBP-8], EBX** **MOV EBX, [EAX+28]** **MOV [EBP-4], EBX** **MOV ECX, DWORD PTR SS:[Func_ARG.2]** **MOV ECX, DWORD PTR DS:[ECX]** **MOV EAX, DWORD PTR SS:[Func_ARG.1]** **AND EAX, ECX** **MOV ECX, EAX** **JMP [ARG.1]**	What essentially happens in general terms **Func_ARG.2 –** SECOND PROCEDURE ARGUMENT, pointer to the key table (XOR, AND) Construction: **MOV EBX, [EAX+2C]** **MOV [EBP-28], EBX** – standard data transfer from key table to stack + Hidden ASM commands for the main move (in reality, there are no such instructions in VM, all the work is organized by islands of individual primitives), *BUT* The only island (executed before exiting the VM), which completely duplicates the AND instruction and serves only for the second method: **ANDEAX, [STORAGE CELL_1]**

Highly recommended reading: http://www.iaeng.org/IJAM/issues_v36/issue_1/IJAM_36_1_12.pdf

Method #2. This method has completely different objectives than those discussed above and is part of the same mechanism. Calls to addresses (0152256Bh and 01520C38) and the procedures they reside in, including those that call them. Generally speaking, both methods work with the second argument at offset [EBP+0xC]. Everything here relies on **Odd-Even Based Cryptography.**



```
01520C0A   55              PUSH EBP
01520C0B   FF0D 9A2B5801   DEC DWORD PTR DS:[1582B9A]
01520C11 ^ 0F84 45C0F4FE   JE 0046CC5C
01520C17   8BEC            MOV EBP,ESP
01520C19   83EC 28         SUB ESP,28
01520C1C   56              PUSH ESI
01520C1D   50              PUSH EAX
01520C1E   B8 CCCDCECF     MOV EAX,CFCECDCC
01520C23   B8 F7404100     MOV EAX,004140F7
01520C28   B8 01000000     MOV EAX,1
01520C2D   B8 00000000     MOV EAX,0
01520C32   B8 CCCDCECF     MOV EAX,CFCECDCC
01520C37   58              POP EAX
01520C38   68 61124000     PUSH 00401261
01520C3D   68 4F0C5201     PUSH OFFSET 01520C4F
01520C42 ^ E9 F94282FF     JMP 00D44F40
01520C47   90              NOP
01520C48   90              NOP
01520C49   90              NOP
01520C4A   90              NOP
01520C4B   90              NOP
01520C4C   90              NOP
01520C4D   90              NOP
01520C4E   90              NOP
01520C4F   034D FC         ADD ECX,DWORD PTR SS:[EBP-4]
01520C52   0FAF4D D8       IMUL ECX,DWORD PTR SS:[EBP-28
01520C56   33C1            XOR EAX,ECX
01520C58   FF0D 9E2B5801   DEC DWORD PTR DS:[1582B9E]
01520C5E ^ 0F84 E02AF6FE   JE 00483744
01520C64   8B4D 0C         MOV ECX,DWORD PTR SS:[EBP+0C]
01520C67   234D E0         AND ECX,DWORD PTR SS:[EBP-20]
01520C6A   034D DC         ADD ECX,DWORD PTR SS:[EBP-24]
01520C6D   0FAF4D F8       IMUL ECX,DWORD PTR SS:[EBP-8]
01520C71   33C1            XOR EAX,ECX
01520C73   8B4D E0         MOV ECX,DWORD PTR SS:[EBP-20]
```

```
0152253D   55              PUSH EBP
0152253E   FF0D 962B5801   DEC DWORD PTR DS:[1582B96]
01522544 v 0F84 911D0600   JE 015842DB
0152254A   8BEC            MOV EBP,ESP
0152254C   83EC 28         SUB ESP,28
0152254F   56              PUSH ESI
01522550   50              PUSH EAX
01522551   B8 CCCDCECF     MOV EAX,CFCECDCC
01522556   B8 5A3F4100     MOV EAX,00413F5A
0152255B   B8 01000000     MOV EAX,1
01522560   B8 00000000     MOV EAX,0
01522565   B8 CCCDCECF     MOV EAX,CFCECDCC
0152256A   58              POP EAX
0152256B   68 83124000     PUSH 00401283
01522570   68 7E255201     PUSH OFFSET 0152257E
01522575 ^ E9 C62982FF     JMP 00D44F40
0152257A   90              NOP
0152257B   90              NOP
0152257C   90              NOP
0152257D   90              NOP
0152257E   8B45 E0         MOV EAX,DWORD PTR SS:[EBP-20]
01522581   F7D0            NOT EAX
01522583   2345 0C         AND EAX,DWORD PTR SS:[EBP+0C]
01522586   8B4D E0         MOV ECX,DWORD PTR SS:[EBP-20]
01522589   F7D1            NOT ECX
0152258B   234D 0C         AND ECX,DWORD PTR SS:[EBP+0C]
0152258E   034D E8         ADD ECX,DWORD PTR SS:[EBP-18]
01522591   0FAF4D E4       IMUL ECX,DWORD PTR SS:[EBP-1C]
01522595   33C1            XOR EAX,ECX
01522597   8B4D E0         MOV ECX,DWORD PTR SS:[EBP-20]
0152259A   F7D1            NOT ECX
0152259C   234D 0C         AND ECX,DWORD PTR SS:[EBP+0C]
0152259F   034D F4         ADD ECX,DWORD PTR SS:[EBP-0C]
015225A2   0FAF4D EC       IMUL ECX,DWORD PTR SS:[EBP-14]
015225A6   33C1            XOR EAX,ECX
```

The first describes the forward flow of the algorithm, the second the reverse. It works by performing a logical AND operation on the argument, first with the mask 5CAC5AC5, then flipping the mask (NOT) and performing a logical AND operation with the mask A353A53A (if you XOR the two halves, the result is the original argument). The first half is stored in duplicate. The result is in the EAX register. For example, if the argument is 0790A442, we will have two halves: 04800040 and 310A402. As far as I understand, two numbers, 0 and 1, are extracted from both using magic constants (keys), and each is added to the two copies of the first half (04800040 and 04800041). The resulting XOR operation yields 1. Reversing the address 01520C38 does the exact opposite—from 1 and offsets 0 and 1, we get 0790A442. I'd like to draw your attention to the direct correlation between the odd/even nature of the numbers and the alternation of values in the third column.

Прямой ход - Аргумент/ Обратный ход- Результат	Прямой ход- результат/Обратный ход - аргумент	Прямой ход: AND EAX, 5CAC5AC5	Прямой ход: AND EAX, A353A53A*
0	3DB1F4C7	0	0
A590217B	0	04800041	A110213A
0790A442	1	04800040	0310A402
630A0CFD	2	400808C5	23020438
615BA9FC	3	400808C4	2153A138
2783017F	4	04800045	2303013A
A5D1A57E	5	04800044	A151A53A
401AA8F9	6	400808C1	0012A038
E0092DC0	7	400808C0	A0012500

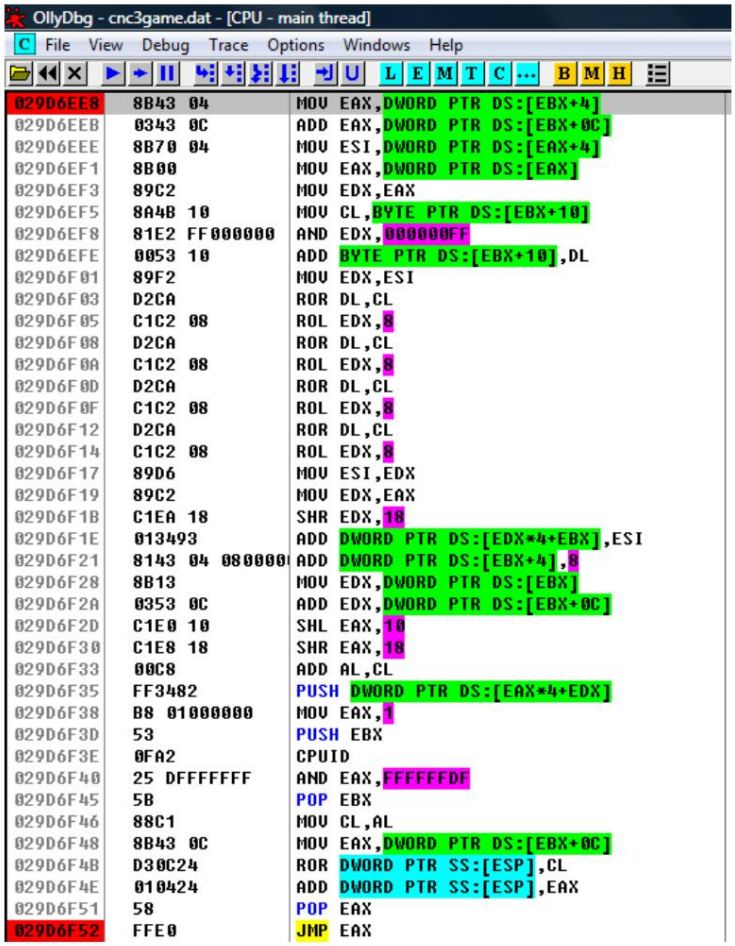
*NOT(5CAC5AC5) = A353A53A
(XOR) 04800041 ^ A110213A = A590217B

The magic constants are located at address 00B93AFC, both calls copy them onto the stack, and the first one additionally contains:

MOV ECX, DWORD PTR SS:[ESP+34] //00B93AFC
MOV ECX, DWORD PTR DS:[ECX] //5CAC5AC5
MOV EAX, DWORD PTR SS:[ESP+38] //the argument itself
AND EAX, ECX //ÿÿÿ AND EAX, 5CAC5AC5

Therefore, method #2 itself effectively hides from view several ACM lines that are part of the algorithm located below (using the return address argument). Unfortunately, I don't know the name of the underlying algorithm, but I can only assume that it can be

Replace it with several equivalent ASM lines. Incidentally, the trace log itself is just over
30,000 lines long, which is explained by the islands running in a loop, copying from address to address. The first two have a 10x lower number! But perhaps the funniest thing is that the developers, determined to somehow increase execution speed, went for broke. Otherwise, how can they explain the fact that the VM has an island with a completely open (without obfuscation) operating algorithm? They gave it away, guts and all :)



VM exit islands deserve a separate discussion. Methods #1 and #1A use the same exit method: ADD EBX,24 MOV DWORD PTR DS:[EBX],0 POPFD POPAD PUSH 0

PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
LEA ESP,[ESP+30]
RETURN 4

If you've already memorized the hierarchy of register usage in VMs, you'll automatically remember that EBX = the start of the main storage.
What's at offset +24 in the main storage? As at the beginning of this chapter, the watchdog number is still there. This implies that the SPIN lock is being reset. The logical conclusion is that any other thread can now access the VM, which clearly indicates that we'll soon be leaving the virtual machine's domain. Our hypothesis is confirmed by the two subsequent instructions restoring flags and processor registers. As they say, that's exactly what was required! Next comes the typical VM trace-covering, i.e., zero is pushed onto the stack 12 times, and 48 bytes (30h) are erased. Ultimately, the top will move back to where it was before the first PUSH 0 and the long-awaited RET 4. In fact, the return occurs based on the second argument, which was replaced in all cases except for the call from the SEH handler (the first is removed from the query database). Oddly enough, upon exit, method #2 again stands out among the first two, one might even say it's ahead, using jumps into the middle of commands: JMP SHORT 2 JMP SHORT 0A

```
JMP SHORT 1
POP EAX
JMP SHORT 9
JMP SHORT 6
PUSH DWORD PTR DS:[EBX+8]
JMP SHORT 0
PUSH 5773
XOR DWORD PTR SS:[ESP],00005757
ADD EBX,DWORD PTR SS:[ESP]
ADD ESP,4
JMP SHORT 2
MOV DWORD PTR DS:[EBX],0
JMP SHORT 8
POPFD
JMP SHORT E
POPAD
JMP SHORT 2
RET 4
```

The virtual machine is too lazy to clean up the stack after itself in the second case. As for XOR, it's obvious, even without calculating it, that the result will be 24h. It's equally obvious that the first PUSH operates on POPFD, i.e., the main storage at offset +8 contains the saved processor flags (EFL). Agree that everything is quite simple and predictable!

Method number	Control byte	Offset from the start (by cells)
1	B6	2D8
1A	83	20C
2	BF	2FC

Actually, if you haven't forgotten, our goal is to completely unbind the Protector from Tiberium. Let's discuss how we'll do that. Well, the second type of VM call seems straightforward – we'll simply rewrite this unknown algorithm, hardcode the keys into the operands, and add the discovered hidden ACM commands. Method 1A shouldn't be a problem either – the required WinAPIs are in the import table, so it's all that simple. Method 1, however, is perhaps the most problematic. Firstly, it's called over 5,000 times, and secondly, how can we quickly grab the values it restores? Of course, the most obvious option is to find those islands from the 255 that are tasked with moving the switches and rewriting the values (POP DWORD PTR DS:[REG_32]), but that's quite time-consuming, and the length of the just-mentioned instruction is 3 bytes shorter than that of JMP, so an SEH handler is essential. A cool hacker solution was finally found – among the 255 different addresses in 1.5, there's one that's unique. Of course, it's the solution for method #1 from VM! In the version of the protector under consideration, it's located at offset 2D8 (control byte B6). After it, we always get to the code requested by the game, which means we already have the destination address! But wait, where do we get the address of the switch itself? If you've been observant, your answer now will be: in the main storage in

The area used for data exchange between islands, and one of the cells stores our desired address, will only be cleared the next time the VM is accessed! Everything would be perfect and would constitute a significant security breach, if, after a test run of the X-code, it turned out that the stolen dump was incomplete. However, SONY DADC suspected that some Russian engineer would be able to get to the bottom of the issue in 2011, so they decided to play it safe – the VM can restore multiple values/addresses in a single run, and it uses the same cells in the main storage for the decoded values/addresses. Thus, the X-code only steals the last restored value at the exit! It follows that the most obvious option is the correct one. The final step will be to rebuild the program code section to its normal form. Including:

replacement

PUSH 01234567

RIGHT

Directly to JMP 01234567; removal of foreign

implementations such as:

0044F4D2 DEC DWORD PTR DS:[158297A]
0044F4D8 IS NODVD.00482DE5

00482DE5 MOV DWORD PTR DS:[158297A],18D 00482DEF JMP
NODVD.0044F4DE replacing arrow calls with direct

ones, moving procedures and variable addresses from the .secuROM section to .text and .data. By the way, use Olka's Call Tree (Ctrl+K); it's indispensable at the last step. Oh! Oh yeah! I almost forgot! The transition to OEP is indeed performed from the VM – starting from the first SEH handler on UD2, which naturally transfers control to the VM. On the 52nd call, of course, using method

#1, control is transferred to the original entry point. In the main storage, its **address (0040A2C7)** will be at offset A0. Now, let me remind you of the secret to getting into OEP with precision. The SONY DADC team's mistake is obvious—there's an island of VM exit in storage #1.5. There are two virtually identical options here:

1. We replace the address of the exit island with our X-code, which expects the first address
   Return to the code section. It's interesting to note that besides the return address, there are other clues, for example, the ROL byte equals EAh. A clear reference to **Electronic Arts.** In general, I'll just say one thing about this: the SecuROM developers clearly have a good sense of humor, and that's a good thing!

   The only nuance is to correctly calculate the encoded value of the initial address of our interceptor for writing to the table of Storage No. 1.5 and rewrite it.
2. Modify the exit island itself by implementing a jump or directly writing the interceptor code. There's one pitfall when writing the interceptor code here: the virtual machine doesn't initially allocate
   a separate memory area for the exit island, so it's always located within other code or often at the very end of the allocated memory. This means there may not be enough space for the entire interceptor code. It's worth noting that the exit islands for methods #1 and #1A are often located close to each other. The first option, in my opinion, is more elegant. However, many who have read this paragraph will likely take a simpler approach – they'll do everything in the debugger using conditional breakpoints.

The .ars section itself—the code located there is responsible for creating the SecuROM 7 virtual machine (WinAPI *VirtualAlloc*, REPNE MOVSB from the .securom section, and XOR BYTE PTR DS:[EAX],CL). It's also worth noting another anomalous thing: at address 00C93AE3

called by VM.

```
00C93ABA   4A              DEC EDX
00C93ABB   4A              DEC EDX
00C93ABC   75 2A           JNE SHORT 00C93AE8
00C93ABE   C745 C4 985CAE4 MOV DWORD PTR SS:[EBP-3C],41AE5C98
00C93AC5   EB 10           JMP SHORT 00C93AD7
00C93AC7   C745 C4 FFA18E9 MOV DWORD PTR SS:[EBP-3C],938EA1FF
00C93ACE   EB 07           JMP SHORT 00C93AD7
00C93AD0   C745 C4 39BD1A2 MOV DWORD PTR SS:[EBP-3C],221ABD39
00C93AD7   A1 F0292101     MOV EAX,DWORD PTR DS:[12129F0]
00C93ADC   8945 C0         MOV DWORD PTR SS:[EBP-40],EAX
00C93ADF   8D45 C0         LEA EAX,[EBP-40]
00C93AE2   50              PUSH EAX
00C93AE3   E8 58140B00     CALL 00D44F40
00C93AE8   EB 07           JMP SHORT 00C93AF1
00C93AEA   2222            AND AH,BYTE PTR DS:[EDX]
00C93AEC   2200            AND AL,BYTE PTR DS:[EAX]
00C93AEE   06              PUSH ES
00C93AEF   0001            ADD BYTE PTR DS:[ECX],AL
00C93AF1   7E 02           JLE SHORT 00C93AF5
00C93AF3   EB 69           JMP SHORT 00C93B5E
00C93AF5   83EC 14         SUB ESP,14
```

However, it's called in an unusual way (using Method #1) – control is transferred to the next short jump instruction, and it's transferred as usual – via the return address specified by the call, i.e., it doesn't appear in the main storage cells at the end! Incidentally, this is where my address-robbing X code failed. If you NOP the call, the game will function without errors. The assumption that fake VM calls exist is confirmed by the fact that this call is parsed: first, it places the number 10h in the cell and decrements it. It's the only number in the cell, which means that, based on other VM operations, it can't do anything else with it! The decrement operation is repeated several times with different numbers until the cell is zero. Then come other meaningless values and operations, such as reading a cell in Storage #1.5 at offset +400 (control byte FFh), which naturally contains zero! The main storage cells also contain the address of the procedure where the call is located – 00C9A350. Moreover, all this code is located in the .ars section, not in .secuROM, which is not typical for Method #1. Incidentally, .ars contains SEH for UD2, and in general, the procedure looks more like a verification code found in add-on armor (for example, TF verification). Summarizing what I just said, I'll say one thing – don't expect developers to always do absolutely everything according to the same template. Another rule is quite obvious: query databases must always match the game code to which they are assigned! But it's also clear that the query databases in Methods #1 and 1-A can be swapped! Although the VM will indeed perform operations based on a substituted database (or more precisely, a virtual stack), the final result is that the requested procedure/ASM command will be from the substituted database, which bodes ill for the code operating the requested procedure. And if we remember that some databases restore addresses and initial variables, expect a ton of unhandled exceptions. Well then! The article is steadily drawing to a close. Let's draw conclusions and summarize. A serious cumulative weapon for destroying SecuROM v7's main add-on armor was presented, based on the miscalculations of the developers of the protective system. Of course, this isn't know-how in the true sense, but rather an attempt to rationalize existing theory and demonstrate the operation of our cumulative projectile using a real-world example. As you can see, it works! No matter how terrifying SecuROM v7 and its VM may initially seem, one fact will ultimately remain: it's really simple! Especially if I make another mind-blowing statement: Good news for those who want to disassemble the protector in OllyDbg v2.0, forgetting about all the phantoms and other similar stealth plugins. Incredibly, this simple and quite obvious trick works before 7.40 (remember, only under the following condition: instead of *ollydbg.exe* there will be *elf.exe*... in short, anything but the original name of the debugger executable file):

```
MOV ESI,DWORD PTR FS:[18] ; < NEW ENTRY POINT
MOV ESI,DWORD PTR DS:[ESI+30]

MOV DWORD PTR DS:[ESI],0 PUSH                    ; WE ACTIVATE THE "CAPE" COMPLEX
011DBA77 MOV                                        ; ASCII "NTDLL"
EAX,DWORD PTR DS:[<&KERNEL32.GetModuleHandleA>]
CALL EAX
PUSH 011DBA7E                                       ; ASCII "NtQueryInformationProcess"
PUSH EAX
MOV EAX,DWORD PTR DS:[<&KERNEL32.GetProcAdress>]
CALL EAX
MOV ESI,DWORD PTR DS:[<&KERNEL32.VirtualProtect>]
PUSH 011DBA9A
PUSH 40
MOV EDI,EAX
PUSH 100
PUSH EAX
CALL ESI
ADD EDI,6
MOV EAX,011DB8B7
XCHG DWORD PTR DS:[EDI],EAX ;KiFastSystemCall call substitution
SUB EDI,6
MOV DWORD PTR DS:[11DB8BB],EAX
MOV AX,40
MOV WORD PTR DS:[11DBA9A],AX
PUSH 011DBA9A
PUSH 20
PUSH 100
PUSH EDI
```

```
CALL ESI
CHORUS, IT WAS
MOV EXIT,EXIT
JMP 011D7B30                                    ; RETURNING TO THE PREVIOUS ENTRY POINT
011DB8B7        C0B8 1D010003F
011DB8BE        7F 00
MOV ECX,DWORD PTR SS:[ESP+0C] ;OUR INTERCEPTOR CODE MOV CH,7 ;
if(ProcessInfoClass == ProcessDebugPort)…
SUB CH,CL
JNE SHORT 011DB8D8
XOR EAX,EAX
MOV ECX,DWORD PTR SS:[ESP+10]
MOV DWORD PTR DS:[ECX],EAX ADD ESP,4
RETN 14 What's
```

the trick?

Hah! Sleight of hand, no cheating. And anyway, the solution was already given in the previous chapter, so all you have to do is try this trick against the protector yourself. And don't be surprised when the first message from the protector is a request to insert a disk, rather than a complaint about the failure to launch the security module. The only thing is, our protective code doesn't protect against your set hardware breakpoints and software breakpoints initially controlled by WinAPI, but this problem is also solvable if desired. But first of all, I want to draw attention to WinAPI *VirtualProtect.* The injection of X-code was based on the need to gain write access to a section of code, and I did it in the simplest, but highly not recommended, way. Using *VirtualProtect is* the most correct way to change page attributes; in this case, the protection attribute changes from

PAGE_EXECUTE_READ(20h) to PAGE_EXECUTE_READWRITE(40h) and vice versa.

Before I **conclude ,** I'd like to say a few words. To fully understand the above, you must first understand that this is a technical article, not fiction. This means that when reading it step by step, you should also follow it in a debugger! This is the only way to get to know SecuROM 7 better and understand how slowly it runs with VM! I may not have revealed all the secrets, and the X-code provided in the sources is in its original, unoptimized form. I'm also happy to note that the following versions of the 7th series in Cnc3: Tiberium Wars – Kane Wrath and GTA IV (7.35) are structurally identical (oh, if only they'd changed the ROL byte in VM... 95h), but with some improvements and refinements (especially with error code 8019...). Incidentally, in the case of GTA IV, the product's online activation library, ***paul.dll, comes into play.*** If anything is unclear about the version we're reviewing, carefully study the call breakdown with the obfuscation removed; everything is so obvious there that this article was unnecessary. In short, I've left you with the opportunity to spend your time productively :)

CONCLUSION:

Well, what can I say? When the president himself said we need innovation and technological progress? X-code injection is relatively new, but it's certainly a promising direction worth pursuing! Protector developers aren't sitting idle either; at the time of writing, the 8th series of SecuROM is alive and well. If you're looking at your computer right now, and the monitor displays a welcoming **OllyDbg v2** image with the cnc3game.dat module and SecuRom_7 Profiler loaded, then I can only advise you to "never give up, even if it's difficult, even if you lack the strength and knowledge to achieve your goal!" I myself didn't reach my goal right away; the path is always thorny. Yes! It seemed so difficult, my hands dropped, thoughts of "No! I can't do it!" And yet, persistence and hard work were rewarded! After two weeks, I knew as much about the virtual machine as its creators. I've learned a lot, adopted a lot, adopted a lot, rethought some of my approaches to disassembling and debugging, and my knowledge base has expanded significantly. I'd simply like to thank SONY Digital Audio Disk Corporation for all their efforts in creating an interesting and truly successful product... for hackers in particular :) Now all that's left for me to do is wish you luck in your future reverse engineering endeavors, and until we meet again on the pages of the magazine .

GLOSSARY

SecuROM 7 VM *– (my* definition) a collection of code consisting of "islands" and using "storages" to restore data required when accessing it, such as addresses of called internal functions of the protected program, data of the called function itself, WinAPI requests, and certain operations hidden in special islands (method #2). Their primary purpose is to prevent dumping of the protected application. ***"Storages"*** are a structural unit, memory areas specially allocated by SecuROM that store all information required for VM operation. Accessing a storage resource is performed as an offset from the beginning of a given storage. The "Main Storage" is the fundamental unit upon which the virtual machine operates. In the figure below, Storage #1.5 is always the RVA provider for all islands and VM exits.

Ячейка	Зашифр offset	Декод offset	Адрес	Контрольный б...
13D803F8	F8540003	FE150000	151B0000	000000FE
13D803F4	FFE80003	FFFA0000	17000000	000000FD
13D803F0	F4940003	FD250000	142B0000	000000FC
13D803EC	F4D00003	FD340000	143A0000	000000FB
13D803E8	AC904D43	EB241350	022A1350	000000FA
13D803E4	F5E00003	FD780000	147E0000	000000F9
13D803E0	FF780003	FFDE0000	16E40000	000000F8
13D803DC	AC907603	EB241D80	022A1D80	000000F7
13D803D8	F9280003	FE4A0000	15500000	000000F6
13D803D4	AC9034C3	EB240D30	022A0D30	000000F5
13D803D0	FBFC0003	FEFF0000	16050000	000000F4
13D803CC	F7AC0003	FDEB0000	14F10000	000000F3
13D803C8	F8040003	FE010000	15070000	000000F2
13D803C4	AC905563	EB241558	022A1558	000000F1

A "resource in storage" is a special address with a strictly defined offset from the beginning of the storage. I essentially refer to this definition as the main storage, with offsets ranging from 0 to 2Ch.

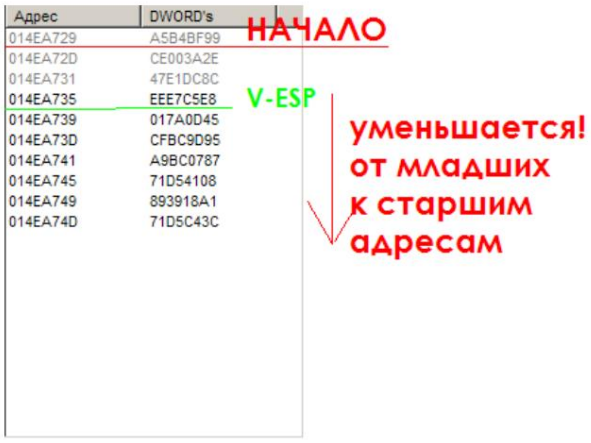
"Islands" are a structural unit, a separate section of code in the VM, the end of which is identified by JMP EAX(EDI), RET, or an immediate pointer. Sometimes jumps are followed by calls to reversers or garbage code, although reading the former is optional. Their common standard is to work with resources (+4) and (+10) in the "main store" and "store #1.5" to obtain the address of the next "island." The functions performed by one island can be as follows: *1) Copying addresses (numbers) from a cell to a cell in the main store or copying to cells from requested addresses outside the stores. 2) Decoding encrypted addresses stored in the virtual stack 3) Overwriting addresses on the stack (for methods #1 and #1A - the return address, for method #2 - the EAX and ECX registers stored on the stack) 4) Editing the arrow for method #1* ***"Query base"*** - forms a custom request to the VM from two arguments. 1) A pointer to the Virtual stack (wrapper) 2) The address to which control should be transferred after exiting the VM. Essentially, this makes sense for method #2. In method #1, it is replaced with the address of the requested internal procedure of the toy/ASM instruction, and in method #1A - the address of the requested WinAPI. For UD2/hardware breakpoints in SEH, it is equal to zero, in which case control is transferred to the next instruction after *CALL 00D44F40* ***"Virtual stack"*** - the concept is similar to a regular machine stack. The allocated memory area and the size of one double word (DWORD). But an important difference is the inverted structure of the virtual: its top is always the lowest address, and the virtual pointer is shifted from the lowest to the highest addresses. That is, the Securom virtual stack does not "grow from top to bottom," but "decreases from bottom to top." In its virtual stack, the VM has only two types of data: *1) Control word.* The byte content can vary (depending on the island "load") from 1 to 3 for each action (1 byte is the standard). First of all, this is the next value for the ROL byte on the next island. Next in importance is the offset at which the address of the next island or the VM exit is retrieved from Storage #1.5. Next in importance are the offsets for cells in the main storage - addresses are entered/ retrieved there.

Or any intermediate values in VM operation. Regarding control byte offsets: After retrieving the control byte, its true value will be decoded, naturally using
the current value of the ROL byte (i.e., they are stored encrypted on the virtual stack, so to speak). The true value will then be multiplied by 4 (x4) - in the vast majority of cases, this operation is performed by the ACM instruction **SHL *REG_32,* 2** (a bitwise left shift by 2 positions is equal to integer multiplication by 4!), and the resulting raw offset will be added to the starting address of the desired storage. *2) Encrypted address/ number.* The simultaneous use of all two data types by an island indicates that it is performing an address decryption procedure (note that in this case, the control word contains a byte that points to the cell in the main storage where it will be stored. Again, for other islands that require this address, their control words will also contain a pointer to the same cell). Using a revolving algorithm (in EDX, *the Encrypted Address/ Number taken from the virtual stack;*
CL, the ROL byte): ROL DL, CL ROL EDX, 8. This will be executed a total of four times, with the final XOR with a specific key performing the decryption.

The following rules apply: - *One checkword per*
island. One encrypted address in the virtual stack per island with a decryption algorithm. - For an island with a decryption algorithm, the checkword always contains a byte pointer to a free cell in the main storage where the
result is to be placed. Naturally, the islands that will use this result are notified via their checkwords of the offset from which the address should be read. (Naturally, due to the ROL byte wrapping, the values of the checkbyte pointers in this case do not match; furthermore, the location of such a checkbyte pointer in the checkword can always be different.)

- If the island uses only the control word, the virtual stack pointer (aka resource (+4) in the main storage) moves down to higher addresses by 4 bytes (one double word). - If the island uses both the control word and the encrypted address, the virtual stack pointer moves down to higher addresses by 8 bytes (two double
words). The instructions ***ADD DWORD PTR DS:[(+4)], 8*** *and* ***ADD DWORD PTR DS:[(+4)], 4*** *in the obfuscated island code make it very easy to discern its purpose!* ***The "virtual stack wrapper"*** is an address that is always included in the first argument of the query base. This is how the first island determines the start of the virtual stack. For method #1, the start address of the virtual stack itself is usually located immediately after the query base.

The "Virtual Pointer" or ***V-ESP*** is a resource (+4) in the main storage. Similar to the direct purpose of the ESP register, it points to the top of the virtual stack.



It is by this that the island first learns what double word is intended for it.

Storage Rules: *The maximum allowable offset for* ***Storage #1.5*** *and* ***the Main Storage*** *cannot exceed 3FCh, since FFh*4 = 3FCh*

(FFh(255) is the maximum permissible value in one byte - the control byte)

The minimum allowable offset for the Main Storage cannot be less than 2Ch (minimum control byte for cells = Bh), since the area below is occupied by a special resource structure of the Main Storage itself.

"Jumping into the middle of an instruction" *is a well-known trick based on the rule of binary code disassembly. Similar to the example Chris gave in his well-known book, "The Art of Disassembly" (Publisher: BVH-Petersburg, 2008): we have a certain set of letters -* ***PODOROGEEKHALAMASHINA.*** *The processor begins to try out variants in order to construct a valid sentence from it (correctly disassemble all four instructions):* ***P PODOROGEEKHALAMASHINA, P PODOROGEEKHALAMASHINA,… P PODOROGEEKHALAMASHINA…*** *until it reaches the only correct variant -* ***P PODOROGEEKHALAMASHINA.*** *Naturally, this is the layout that the hacker will see in their window. Note that the word* ***CAR*** *contains another familiar word -* ***TIRE.*** *Using the transition to the "middle" of the word (ASM instructions) CAR, after the word* ***DRIVEN,*** *you can construct a new sentence:* ***A TIRE WAS DRIVEN ALONG THE ROAD.***

"Delta", "delta calculation" - *The virtual machine is always located at new addresses (results of* ***VirutalAlloc);*** *naturally, it is impossible to bind to specific values, so it is necessary to know the current actual location of the code in memory.* ***Delta is*** *the difference between the current actual location of the code and its declared location during compilation (or from some fixed address). A typical set of instructions for calculating delta: CALL delta delta: POP EAX SUB EAX, offset delta // EAX will contain the delta value. In the virtual machine, delta offsets can be found in Storage #1.5. The calculation is performed by subtracting the RVA of the islands from the RVA of the VM entry island.*

LINKS

https://exelab.ru/f/index.php?action=vthread&forum=13&topic=19719

http://tuts4you.com/download.php?view.2090

At the very end, thanks to *Nightshade,* I discovered, quite by chance, that there was an English-language article on the SecuROM 7.30 virtual machine. I was pleased to learn that our conclusions generally coincided. However, the article presented the VM structure somewhat differently, and I missed some aspects of its operation. http://www.playground.ru/cheats/4932/

NoDVD for CnC3: Tiberium Wars v1.9. A distinctive feature is the virtual machine embedded in the .memory section. This article uses it as the main example. Beginners are recommended to start with it. ftp://ftp.ea.com/pub/eapacific/cnc3/CNC3_patch109_russian.exe

The full version of the protector is available at http://www.exelab.ru/rar/dl/CRACKLAB.rU_11.rar

SecuROM 4 ÿ Empire Earth 2. http://www.exelab.ru/art/?action=view&id=316

SecuROM 7 ÿ FEAR http://web.textfiles.com/software/secuROM.txt

English-language article on SecuROM

http://En.wikipedia.org/wiki/cpuid

All available information about the CPUID instruction http://SecuROM.com

Captain Obvious. However, they're unlikely to share any valuable information there. http://pid.gamecopyworld.com/

The **ProtectionID** tool allows you to accurately determine the protector version : http://rutracker.org/forum/viewtopic.php?t=3637042

Another gift from us to Electronic Arts http://en.wikipedia.org/wiki/SecuROM

At the moment, Wikipedia only has an English-language article : http://www.ollydbg.de/version2.html

The latest version of OllyDbg v2 (currently with plugin support) is always available for download: http://www.pcweek.ru/themes/detail.php?ID=52199

Article: SecuROM Copy Protection http://www.lki.ru/text.php?id=4868

Article: Hacking vs. Security. (The paragraph *"He let us down" is especially amusing.)** http://citforum.ru/security/articles/analis/

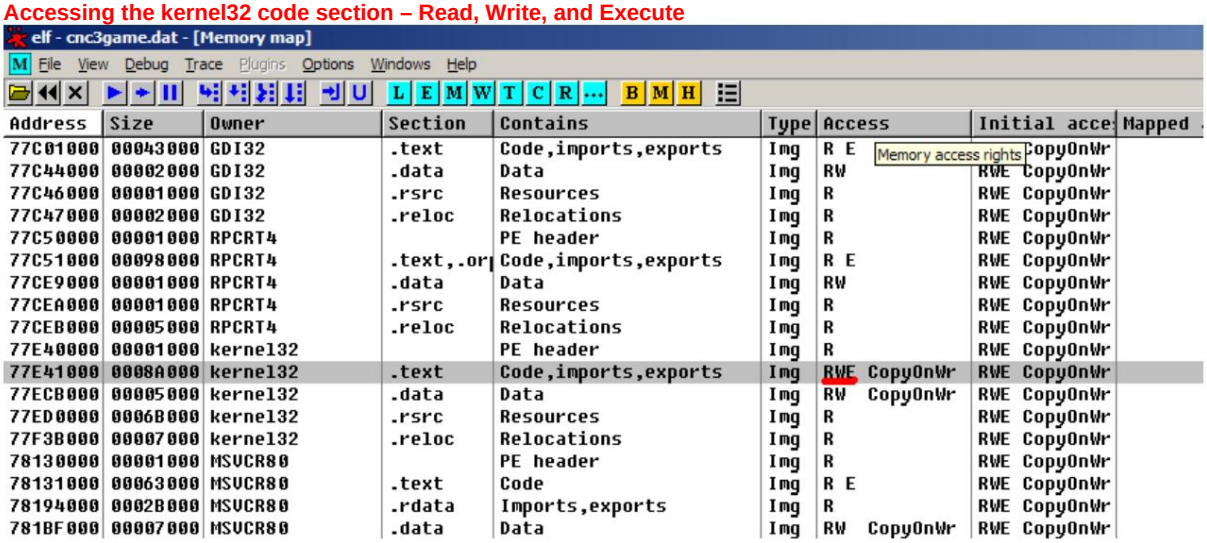
Article: Analysis of the market for software copy protection and hacking protection*

***The authors were bought outright by Protection Technology.**

SecuRom_7 Profiler v1.0 – ready to go :)

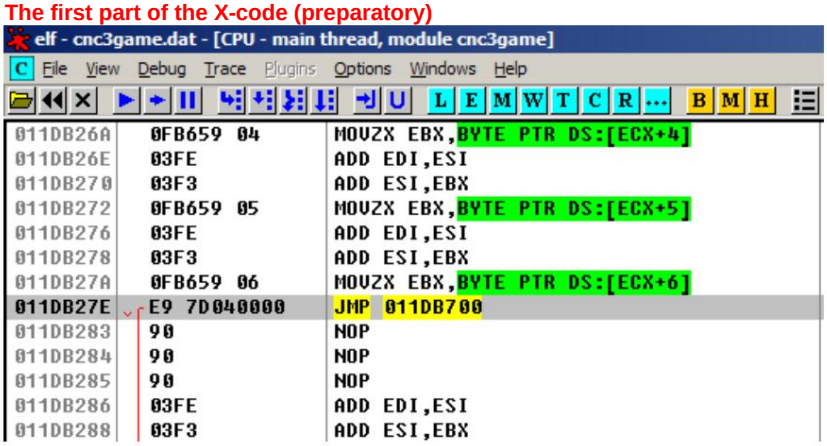
Extended image part

Accessing the kernel32 code section – Read, Write, and Execute



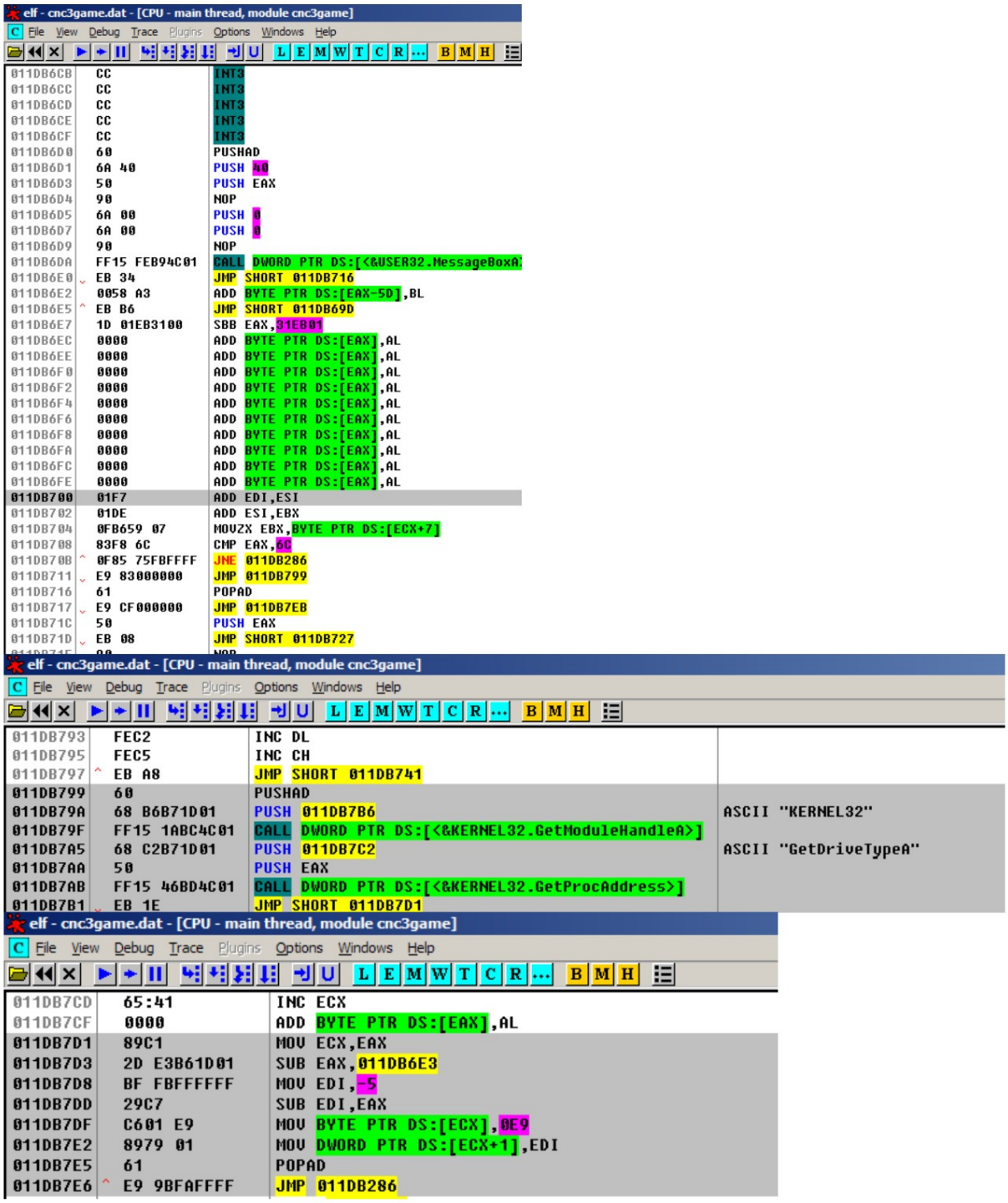
Address	Size	Owner	Section	Contains	Type	Access	Initial acce	Mapped
77C01000	00043000	GDI32	.text	Code,imports,exports	Img	R E	CopyOnWr	
77C44000	00002000	GDI32	.data	Data	Img	RW	RWE CopyOnWr	
77C46000	00001000	GDI32	.rsrc	Resources	Img	R	RWE CopyOnWr	
77C47000	00002000	GDI32	.reloc	Relocations	Img	R	RWE CopyOnWr	
77C50000	00001000	RPCRT4		PE header	Img	R	RWE CopyOnWr	
77C51000	00098000	RPCRT4	.text,.or	Code,imports,exports	Img	R E	RWE CopyOnWr	
77CE9000	00001000	RPCRT4	.data	Data	Img	RW	RWE CopyOnWr	
77CEA000	00001000	RPCRT4	.rsrc	Resources	Img	R	RWE CopyOnWr	
77CEB000	00005000	RPCRT4	.reloc	Relocations	Img	R	RWE CopyOnWr	
77E40000	00001000	kernel32		PE header	Img	R	RWE CopyOnWr	
77E41000	0008A000	kernel32	.text	Code,imports,exports	Img	RWE CopyOnWr	RWE CopyOnWr	
77ECB000	00005000	kernel32	.data	Data	Img	RW CopyOnWr	RWE CopyOnWr	
77ED0000	0006B000	kernel32	.rsrc	Resources	Img	R	RWE CopyOnWr	
77F3B000	00007000	kernel32	.reloc	Relocations	Img	R	RWE CopyOnWr	
78130000	00001000	MSVCR80		PE header	Img	R	RWE CopyOnWr	
78131000	00063000	MSVCR80	.text	Code	Img	R E	RWE CopyOnWr	
78194000	0002B000	MSVCR80	.rdata	Imports,exports	Img	R	RWE CopyOnWr	
781BF000	00007000	MSVCR80	.data	Data	Img	RW CopyOnWr	RWE CopyOnWr	

The first part of the X-code (preparatory)

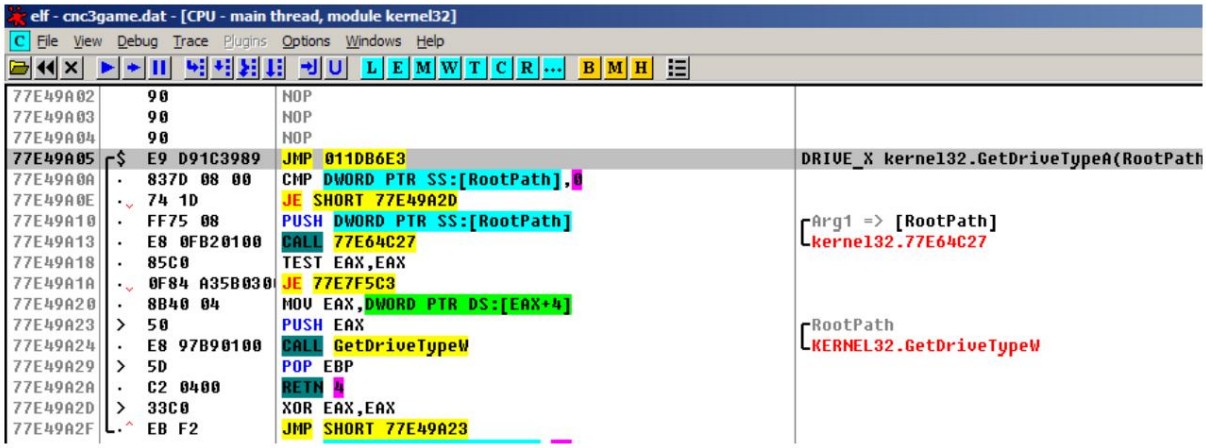



```
011DB26A   0FB659 04       MOVZX EBX,BYTE PTR DS:[ECX+4]
011DB26E   03FE            ADD EDI,ESI
011DB270   03F3            ADD ESI,EBX
011DB272   0FB659 05       MOVZX EBX,BYTE PTR DS:[ECX+5]
011DB276   03FE            ADD EDI,ESI
011DB278   03F3            ADD ESI,EBX
011DB27A   0FB659 06       MOVZX EBX,BYTE PTR DS:[ECX+6]
011DB27E   E9 7D040000     JMP 011DB700
011DB283   90              NOP
011DB284   90              NOP
011DB285   90              NOP
011DB286   03FE            ADD EDI,ESI
011DB288   03F3            ADD ESI,EBX
```

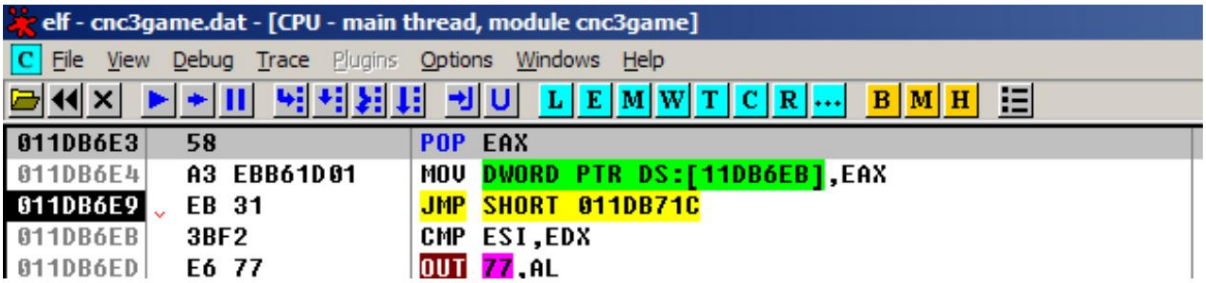
Machine Translated by Google

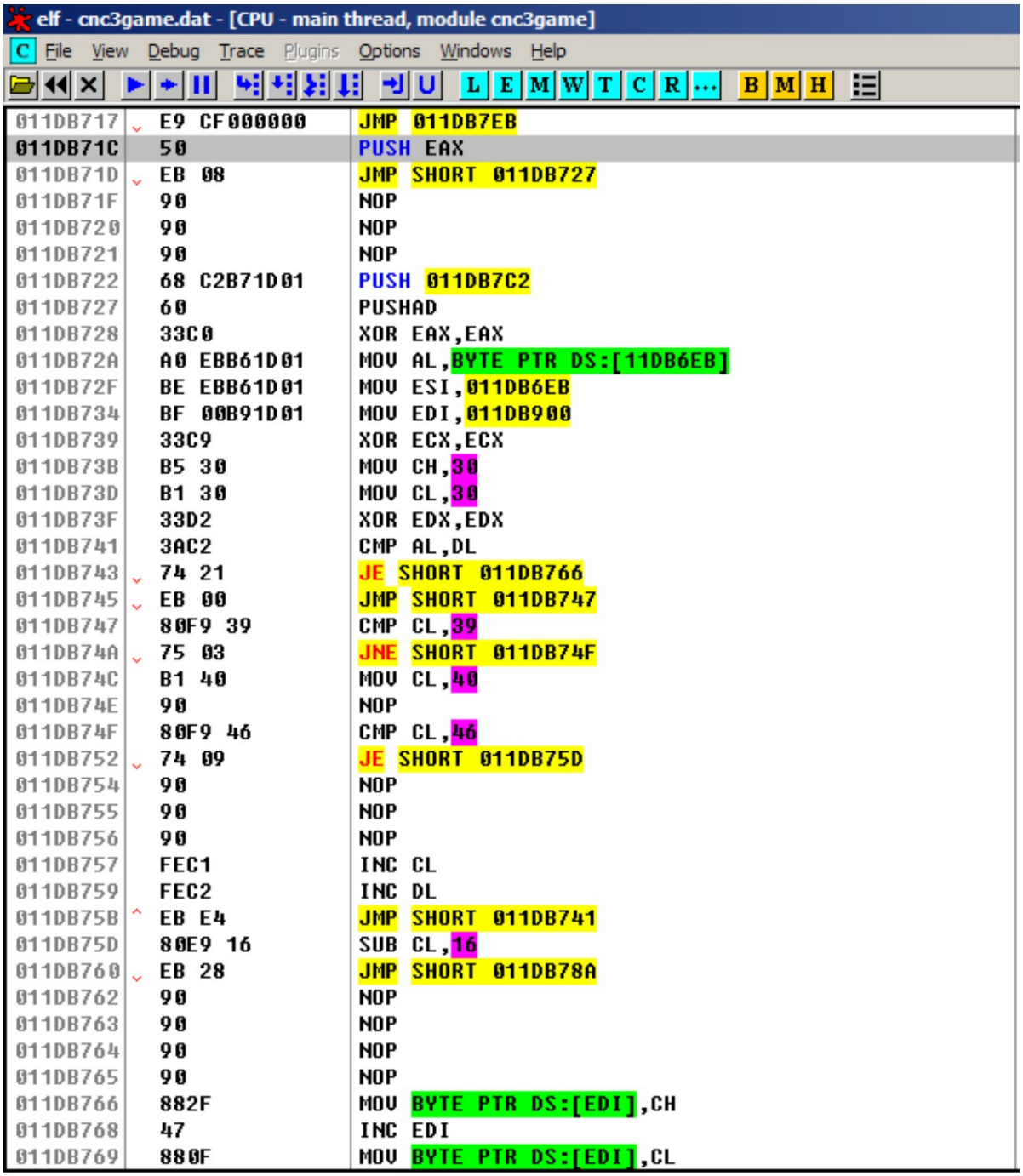


The result of running the first part of the X code

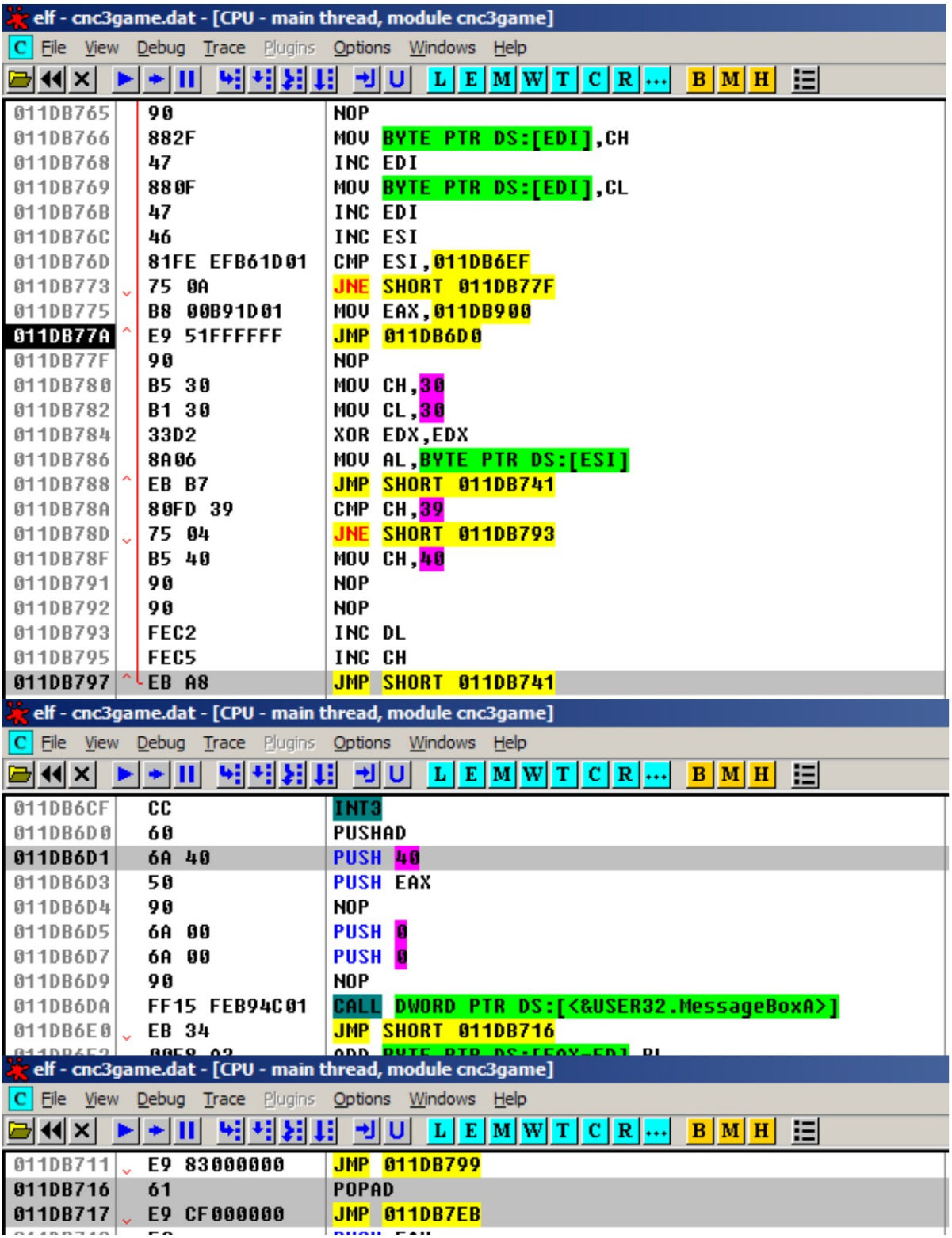


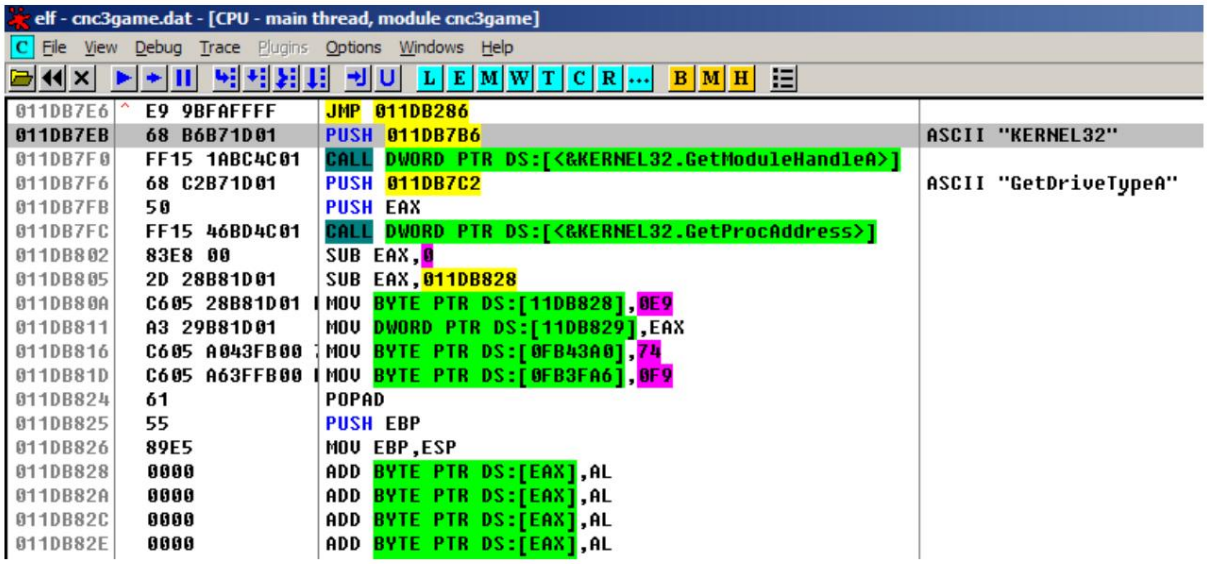
The second part of the X-code (the interceptor code itself)



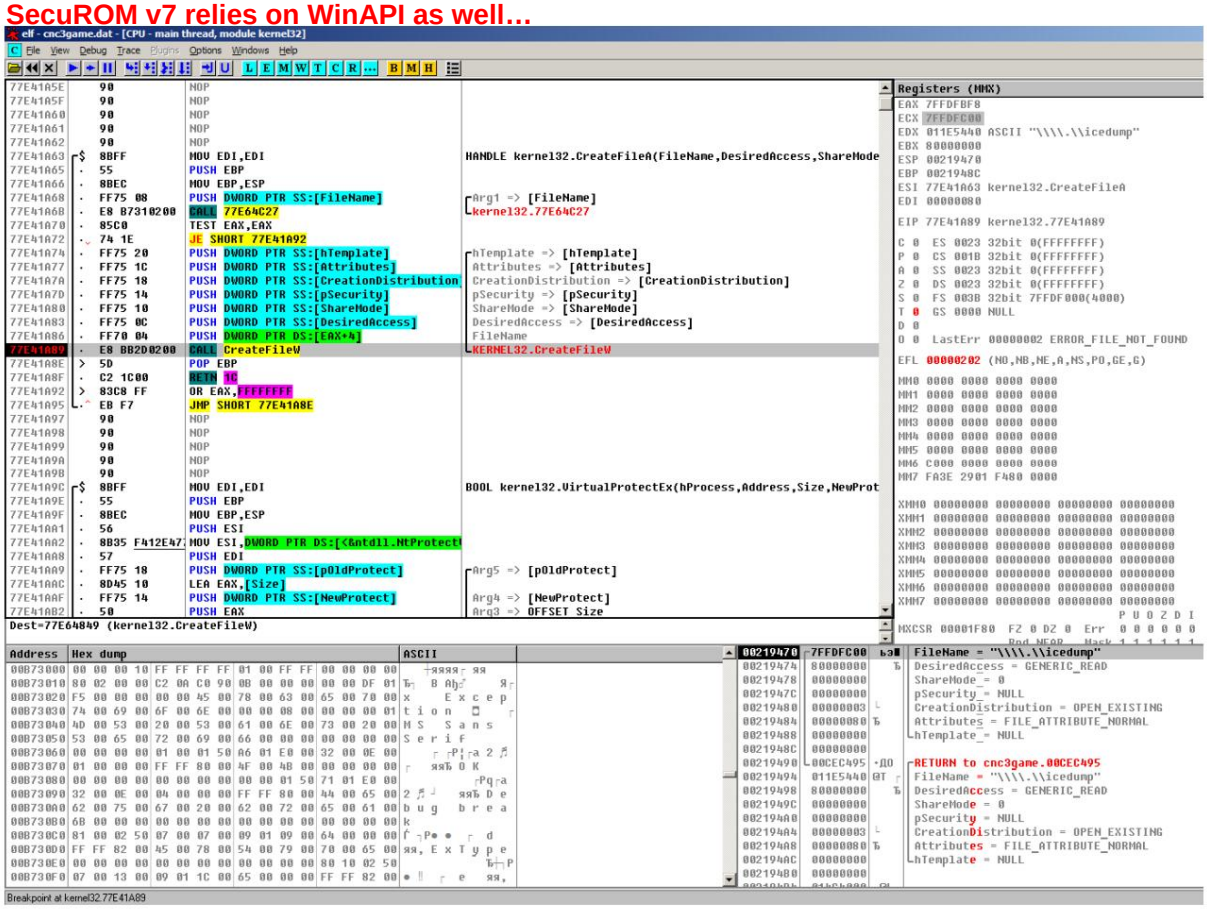


Machine Translated by Google

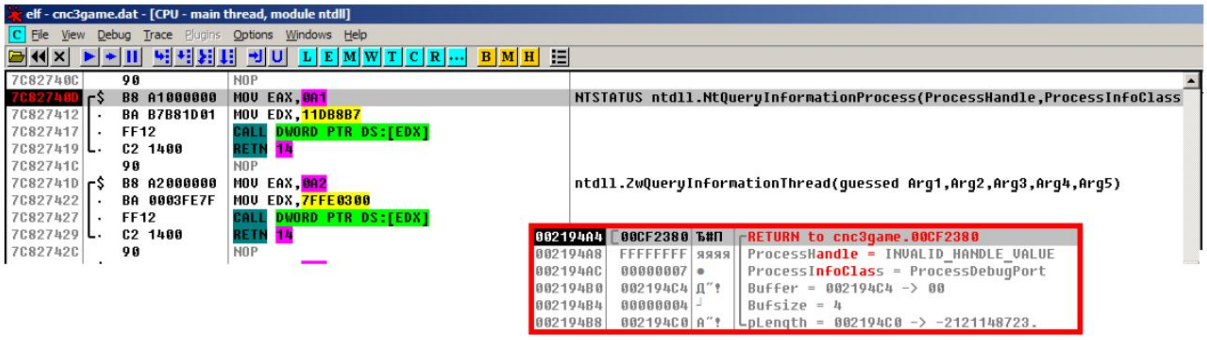




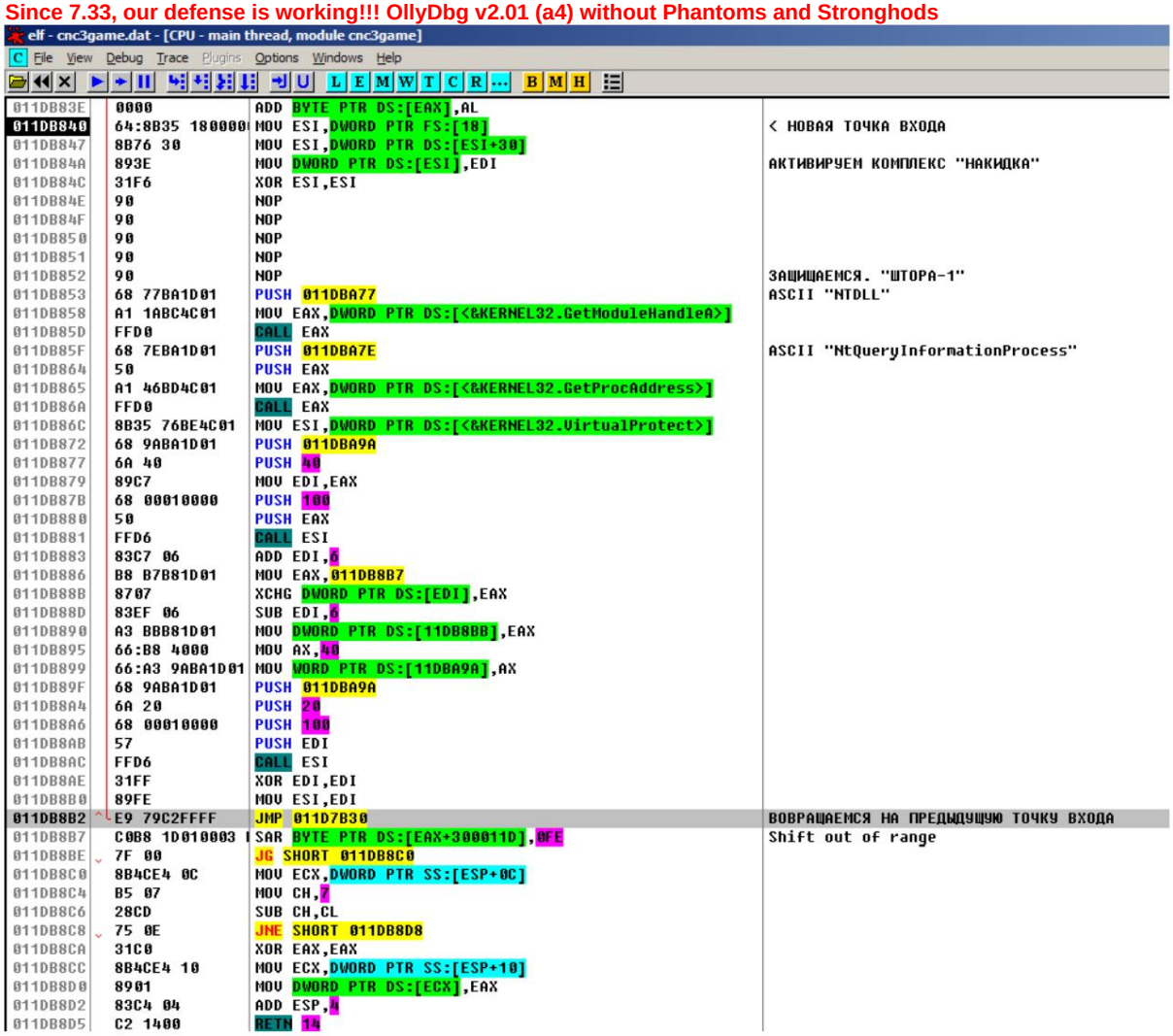
SecuROM v7 relies on WinAPI as well…



…The same applies to low-level functions from ntdll. Pay attention to the return address and argument. *ProcessInfoClass*. However, in any case, the debugger won't be detected—we've "blinded" ourselves.



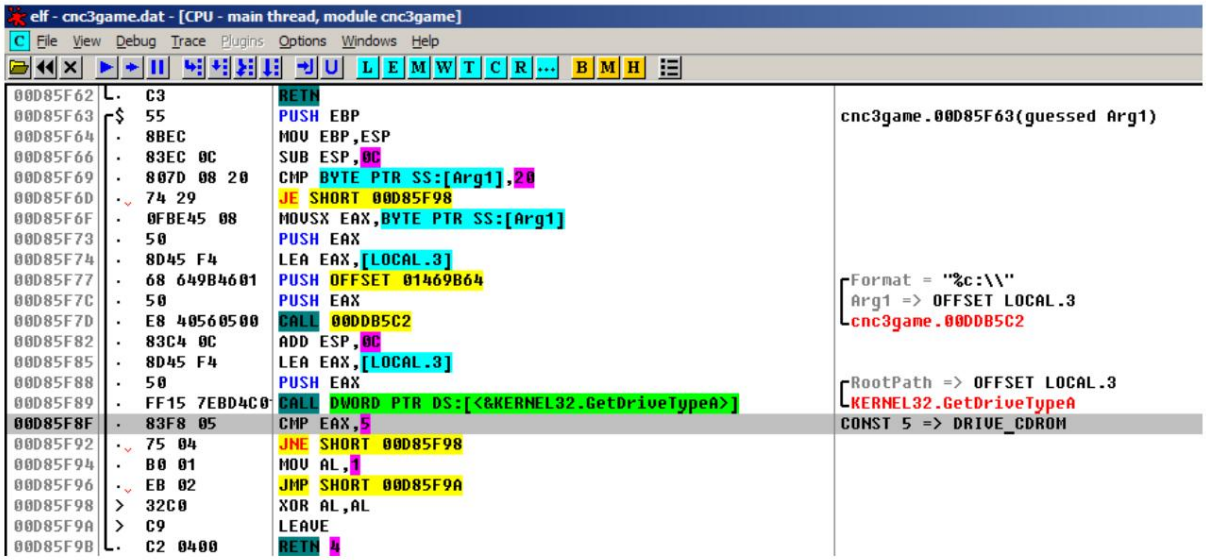
Since 7.33, our defense is working!!! OllyDbg v2.01 (a4) without Phantoms and Stronghods



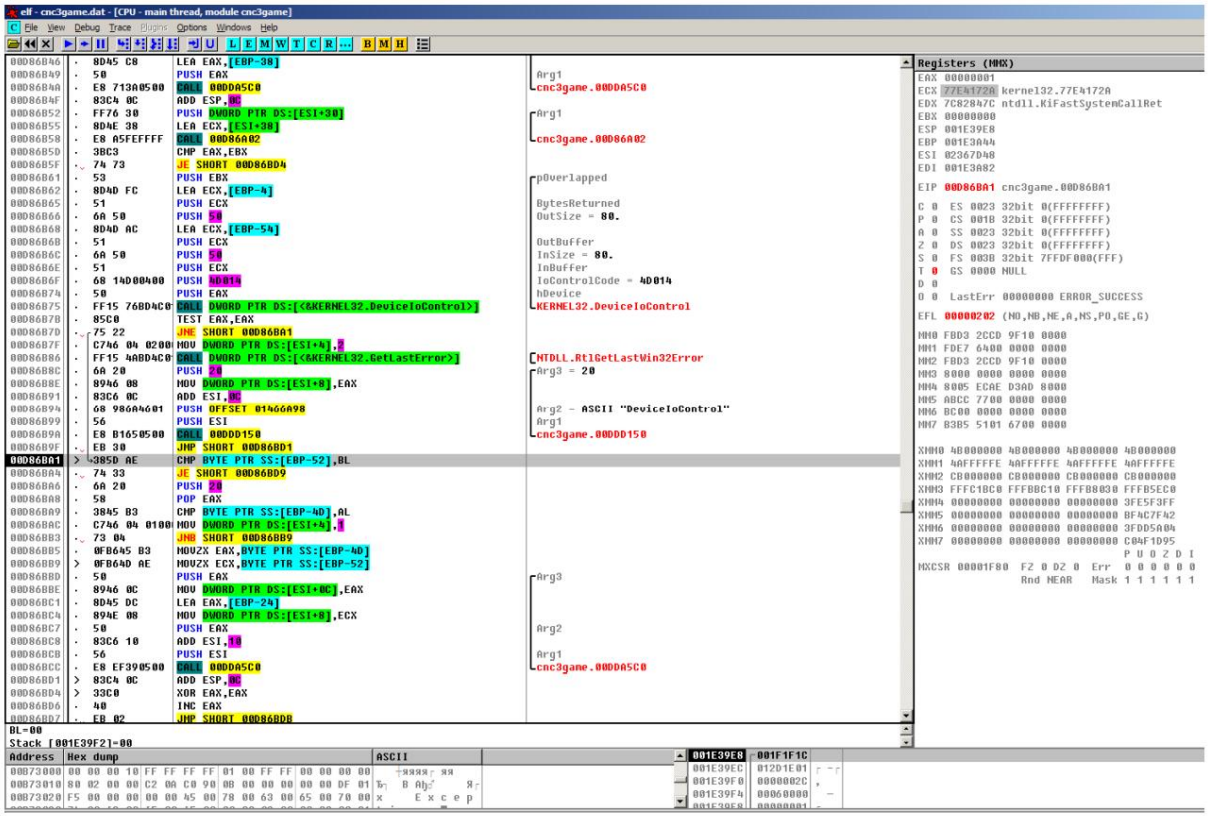
Disc Check Circuit in Drive PLEASE

NOTE – EVERYTHING IS FORMATED AS TYPICAL FUNCTIONS WITH A PROLOGUE AND EPILOGUE

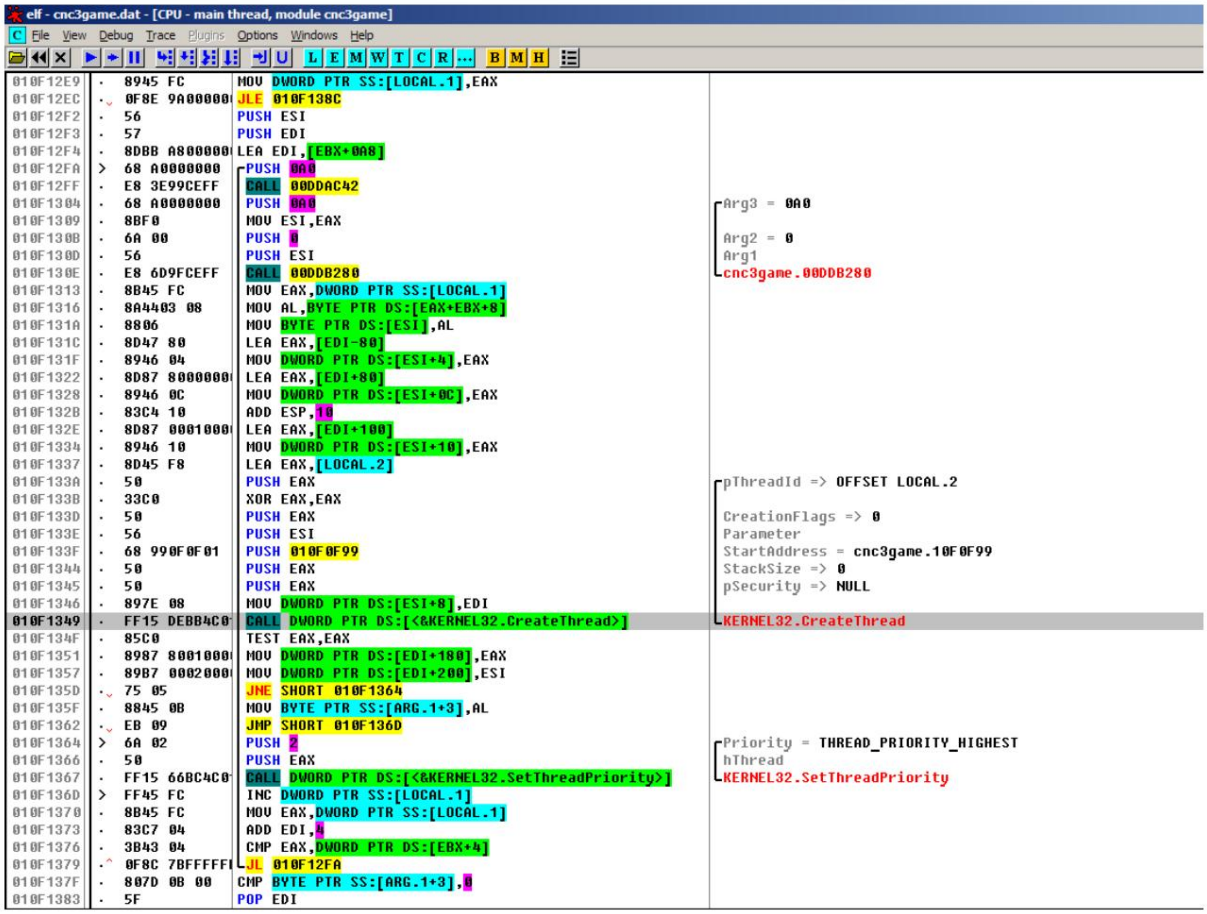
GetDriveTypeA to search for drives



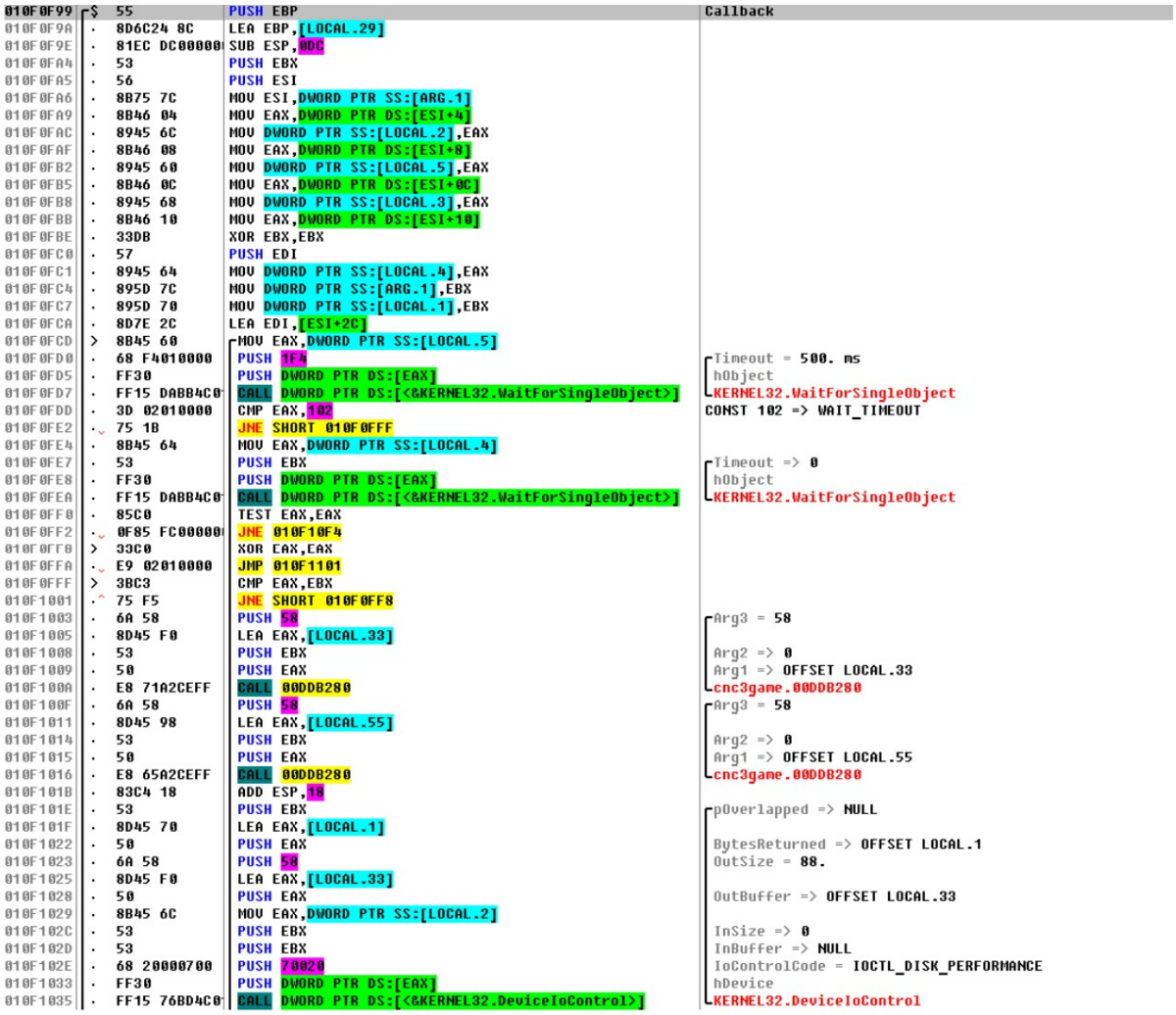
CreateFileA and DeviceIoControl are the two key functions in the first part of the test.



The second part of the test is synchronous threads, DeviceIoControl and QueryPerfomanceCounter

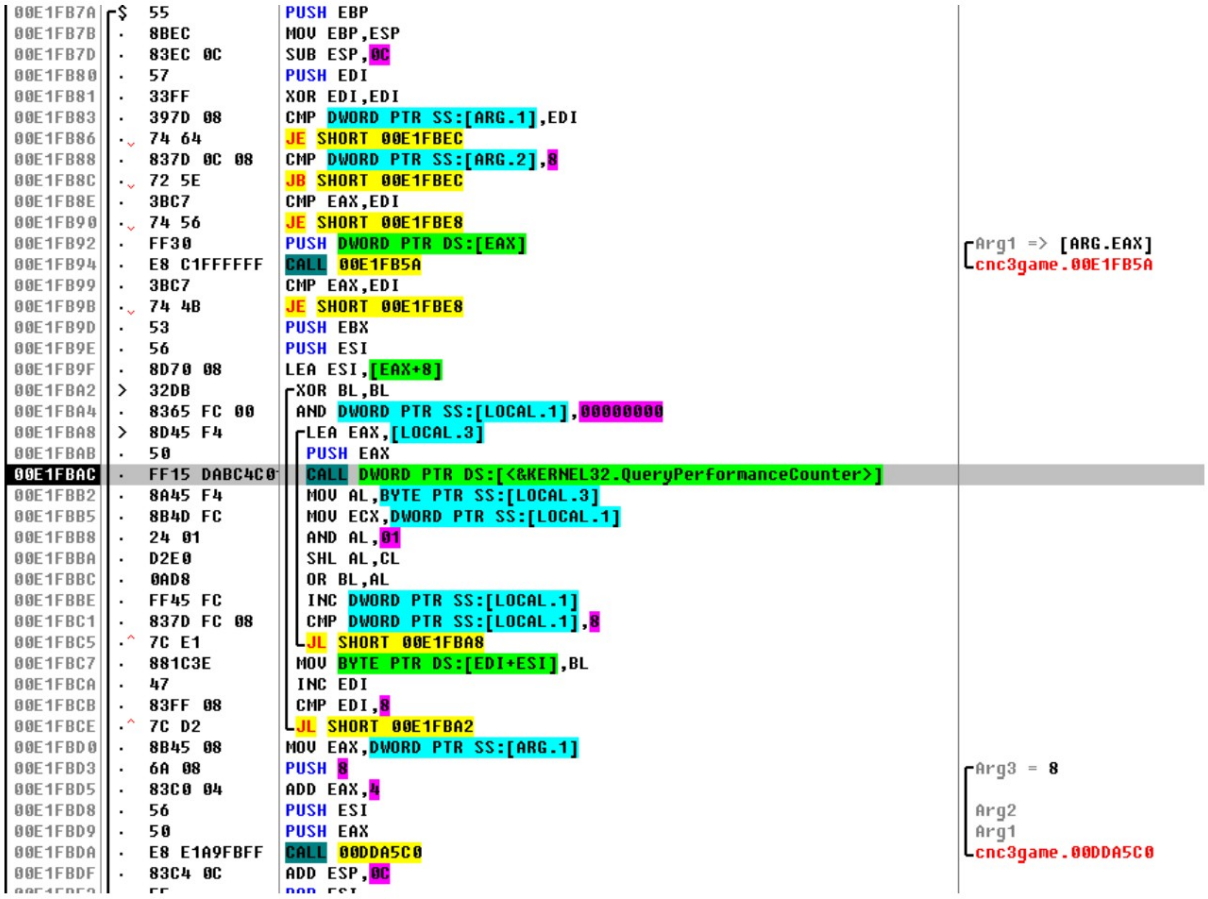


Machine Translated by Google

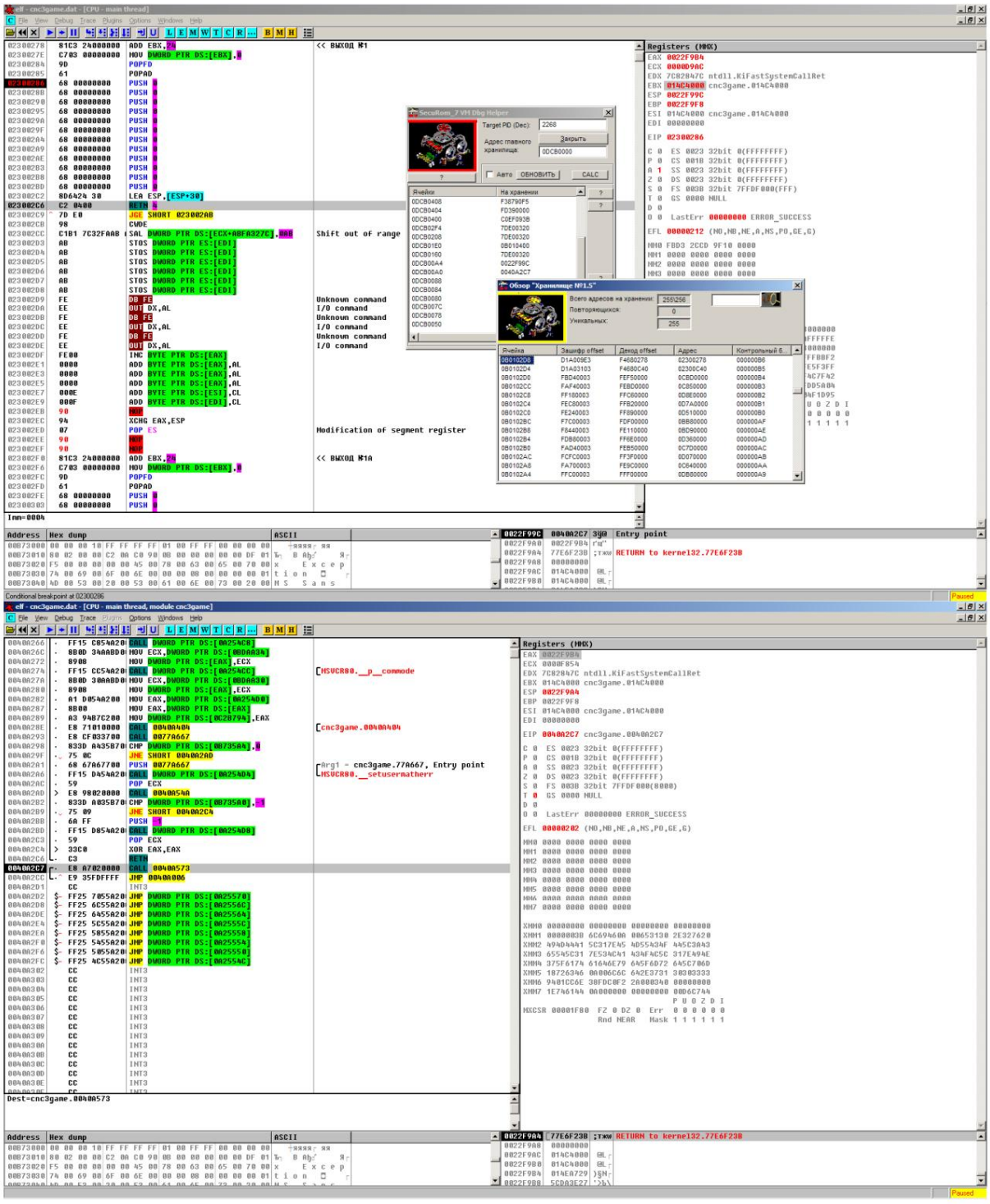


```
010F0F99 ┌$  55              PUSH EBP                                 Callback
010F0F9A │·  8D6C24 8C       LEA EBP,[LOCAL.29]
010F0F9E │·  81EC DC00000(   SUB ESP,0DC
010F0FA4 │·  53              PUSH EBX
010F0FA5 │·  56              PUSH ESI
010F0FA6 │·  8B75 7C         MOV ESI,DWORD PTR SS:[ARG.1]
010F0FA9 │·  8B46 04         MOV EAX,DWORD PTR DS:[ESI+4]
010F0FAC │·  8945 6C         MOV DWORD PTR SS:[LOCAL.2],EAX
010F0FAF │·  8B46 08         MOV EAX,DWORD PTR DS:[ESI+8]
010F0FB2 │·  8945 60         MOV DWORD PTR SS:[LOCAL.5],EAX
010F0FB5 │·  8B46 0C         MOV EAX,DWORD PTR DS:[ESI+0C]
010F0FB8 │·  8945 68         MOV DWORD PTR SS:[LOCAL.3],EAX
010F0FBB │·  8B46 10         MOV EAX,DWORD PTR DS:[ESI+10]
010F0FBE │·  33DB            XOR EBX,EBX
010F0FC0 │·  57              PUSH EDI
010F0FC1 │·  8945 64         MOV DWORD PTR SS:[LOCAL.4],EAX
010F0FC4 │·  895D 7C         MOV DWORD PTR SS:[ARG.1],EBX
010F0FC7 │·  895D 70         MOV DWORD PTR SS:[LOCAL.1],EBX
010F0FCA │·  8D7E 2C         LEA EDI,[ESI+2C]
010F0FCD │>  8B45 60         ┌MOV EAX,DWORD PTR SS:[LOCAL.5]
010F0FD0 │·  68 F4010000     │PUSH 1F4                                ┌Timeout = 500. ms
010F0FD5 │·  FF30            │PUSH DWORD PTR DS:[EAX]                 │hObject
010F0FD7 │·  FF15 DABB4C0(   │CALL DWORD PTR DS:[<&KERNEL32.WaitForSingleObject>]  └KERNEL32.WaitForSingleObject
010F0FDD │·  3D 02010000     │CMP EAX,102                             CONST 102 => WAIT_TIMEOUT
010F0FE2 │·⌄ 75 1B           │JNE SHORT 010F0FFF
010F0FE4 │·  8B45 64         │MOV EAX,DWORD PTR SS:[LOCAL.4]
010F0FE7 │·  53              │PUSH EBX                                ┌Timeout => 0
010F0FE8 │·  FF30            │PUSH DWORD PTR DS:[EAX]                 │hObject
010F0FEA │·  FF15 DABB4C0(   │CALL DWORD PTR DS:[<&KERNEL32.WaitForSingleObject>]  └KERNEL32.WaitForSingleObject
010F0FF0 │·  85C0            │TEST EAX,EAX
010F0FF2 │·⌄ 0F85 FC00000(   │JNE 010F10F4
010F0FF8 │>  33C0            │XOR EAX,EAX
010F0FFA │·⌄ E9 02010000     │JMP 010F1101
010F0FFF │>  3BC3            │CMP EAX,EBX
010F1001 │·^ 75 F5           │JNE SHORT 010F0FF8
010F1003 │·  6A 58           │PUSH 58                                 ┌Arg3 = 58
010F1005 │·  8D45 F0         │LEA EAX,[LOCAL.33]                      │
010F1008 │·  53              │PUSH EBX                                │Arg2 => 0
010F1009 │·  50              │PUSH EAX                                │Arg1 => OFFSET LOCAL.33
010F100A │·  E8 71A2CEFF     │CALL 00DDB280                           └cnc3game.00DDB280
010F100F │·  6A 58           │PUSH 58                                 ┌Arg3 = 58
010F1011 │·  8D45 98         │LEA EAX,[LOCAL.55]                      │
010F1014 │·  53              │PUSH EBX                                │Arg2 => 0
010F1015 │·  50              │PUSH EAX                                │Arg1 => OFFSET LOCAL.55
010F1016 │·  E8 65A2CEFF     │CALL 00DDB280                           └cnc3game.00DDB280
010F101B │·  83C4 18         │ADD ESP,18
010F101E │·  53              │PUSH EBX                                ┌pOverlapped => NULL
010F101F │·  8D45 70         │LEA EAX,[LOCAL.1]                       │
010F1022 │·  50              │PUSH EAX                                │BytesReturned => OFFSET LOCAL.1
010F1023 │·  6A 58           │PUSH 58                                 │OutSize = 88.
010F1025 │·  8D45 F0         │LEA EAX,[LOCAL.33]                      │
010F1028 │·  50              │PUSH EAX                                │OutBuffer => OFFSET LOCAL.33
010F1029 │·  8B45 6C         │MOV EAX,DWORD PTR SS:[LOCAL.2]          │
010F102C │·  53              │PUSH EBX                                │InSize => 0
010F102D │·  53              │PUSH EBX                                │InBuffer => NULL
010F102E │·  68 20000700     │PUSH 70020                              │IoControlCode = IOCTL_DISK_PERFORMANCE
010F1033 │·  FF30            │PUSH DWORD PTR DS:[EAX]                 │hDevice
010F1035 │·  FF15 76BD4C0(   │CALL DWORD PTR DS:[<&KERNEL32.DeviceIoControl>]  └KERNEL32.DeviceIoControl
```

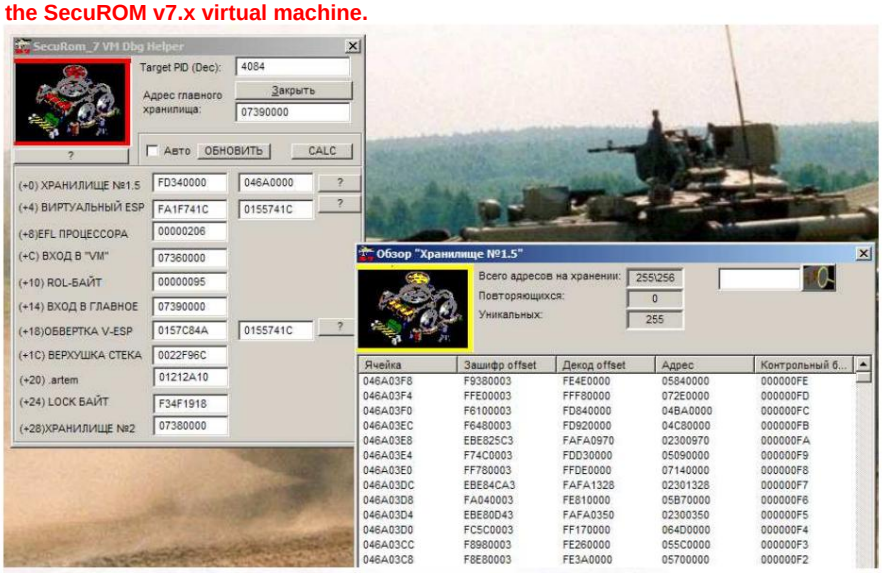
Machine Translated by Google

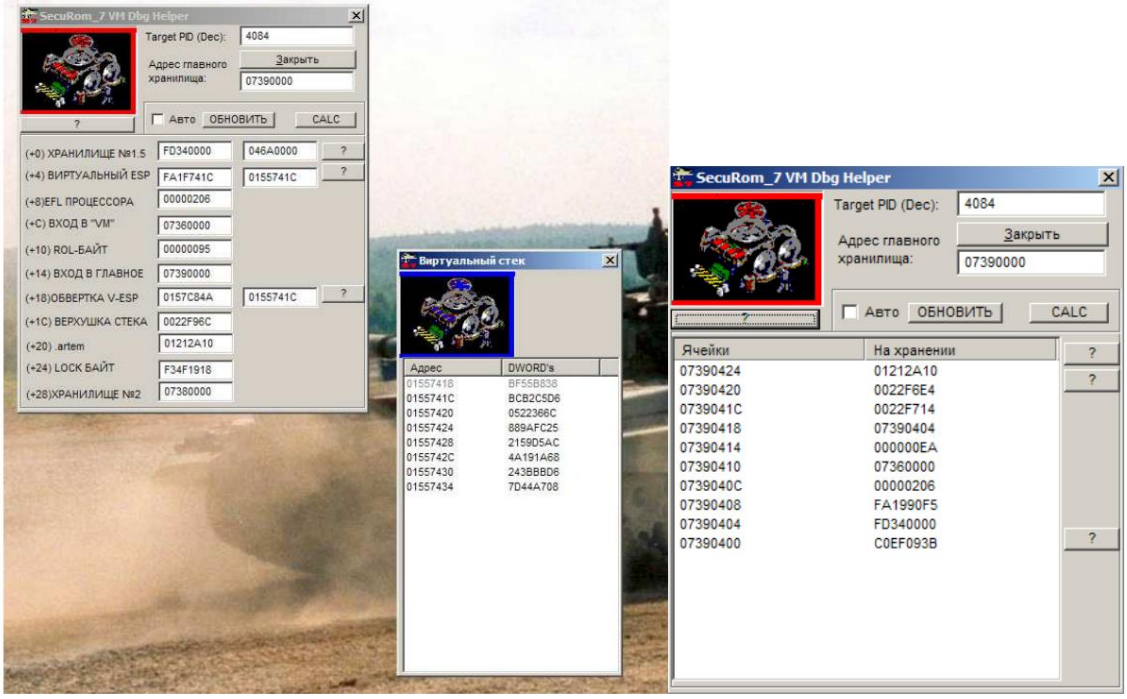


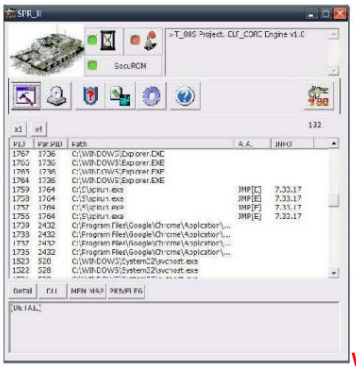
Comrade Commander, mission accomplished! OEP successfully reached! Address: 0040A2C7. VM Exit 52. The first image shows: 1. The exit islands for methods 1 and 1A are located next to each other. This is not an uncommon situation. 2. SecuRom_7 Profiler: Island address in #1.5 and OEP address in cell +0A of the Main Storage



SecuRom_Profiler v1.0 is our first successful step towards visualizing the operating parameters of the SecuROM v7.x virtual machine.







Whether there will be a second one depends only on you! ;)