Armadillo 4.20

Removing the armour: a naked animal

{***W**ritten **b**y* **AndreaGeddon**}
{andreageddon@gmail.com}

[RET]
[www.reteam.org]

INTRO

Armadillo is a nice animal, and a nice protection system! It is divided into two parts, the parent (debugger) and the child (debuggee). The code section is protected by a Copymem system and some code is emulated using nanomites. Also API emulation and redirection is used, a lot of nice features! The target I am using is Armadillo itself version 4.20. Let's start reversing.

STEP ONE: The ASM loader – Loading the loader

I start by having a general look at the PE, we can see that the import table seems to be correct (lots of API imports from kernel32, user32 and gdi32) but I don't think it will be easy, so probably the real IT is loaded at runtime and is present elsewhere. Also, the section table is interesting

```
 entry point = 004c9000

  From     - To        Section  Size
 -------------------------------------
  00400000 - 00401000  PE       1000
  00401000 - 00448000  .text    47000
  00448000 - 0044B000  .rdata   3000
  0044B000 - 00479000  .data    2E000
  00479000 - 004C9000  .text1   50000
  004C9000 - 004D9000  .adata   10000
  004D9000 - 004F9000  .data1   20000
  004F9000 - 00639000  .pdata   140000
  00639000 - 007D0000  .rsrc    18E000
```

It seems that there are two executables joined in a unique PE. The entry point is located in the .adata section. This is the data section of the second executable. So we can assume that the initial code will be a pre loader that prepares the execution of the second executable, which resides in the .text1 section. Also this could mean that this second executable is the code of the protection core, and the first executable (.text) is the real application. Now that we have a general idea of the structure of the crypter, we can start tracing.

The beginning is as usual:

```
004C9000   PUSHAD
004C9001   CALL      004C9006
004C9006   POP       EBP
```

The address of the start of the loader is stored into ebp, so the loader can access static data in the loader referring to the displacement from ebp. This is done because a loader has no fixed addresses, its location is known only at runtime.
We soon find some polymorphic code:

```
004C9007   50              PUSH EAX
004C9008   51              PUSH ECX
004C9009   0FCA            BSWAP EDX
004C900B   F7D2            NOT EDX
004C900D   9C              PUSHFD
004C900E   F7D2            NOT EDX
004C9010   0FCA            BSWAP EDX
004C9012   EB 0F           JMP SHORT Armadill.004C9023
004C9014   B9 EB0FB8EB     MOV ECX,EBB80FEB
004C9019   07              POP ES
004C901A   B9 EB0F90EB     MOV ECX,EB900FEB
004C901F   08FD            OR CH,BH
004C9021   EB 0B           JMP SHORT Armadill.004C902E
004C9023   F2              PREFIX REPNE:
004C9024   EB F5           JMP SHORT Armadill.004C901B
004C9026   EB F6           JMP SHORT Armadill.004C901E
004C9028   F2              PREFIX REPNE:
004C9029   EB 08           JMP SHORT Armadill.004C9033
004C902B   FD              STD
004C902C   EB E9           JMP SHORT Armadill.004C9017
004C902E   F3              PREFIX REP:
004C902F   EB E4           JMP SHORT Armadill.004C9015
004C9031   FC              CLD
004C9032   E9 9D0FC98B     JMP 8C159FD4
004C9037   CA F7D1          RETF 0D1F7
004C903A   59              POP ECX
004C903B   58              POP EAX
```

This is usually the common form of the garbage code used by armadillo, there are some variants but the general structure is always similiar to this, so don’t waste time tracing these pieces of code, just skip them and focus on the real instructions. After this, we get to the first decryption routines

```
004C9157   XOR DWORD PTR DS:[EAX],EBX
004C9159   CMP DWORD PTR DS:[EAX],5478
004C915F   JE SHORT Armadill.004C9165
004C9161   XOR DWORD PTR DS:[EAX],EBX
004C9163   JMP SHORT Armadill.004C9151
004C9165   MOV ECX,9B4F
```

```
004C916A   AND EBX,0FF
004C9170   ADD EAX,3
004C9173   INC EAX
004C9174   XOR BYTE PTR DS:[EAX],BL
004C9176   DEC ECX
004C9177   JNZ SHORT Armadill.004C9173
004C9179   JMP SHORT Armadill.004C917F
```

This simply decrypts some of the following code (starting at 004c917f). We will encounter a lot of decryption routines. Now we see some anti-debug code:

```
004C917F   PUSH EAX
004C9180   CALL Armadill.004C91A9
004C9185   MOV ECX,DWORD PTR SS:[ESP+C]
004C9189   ADD DWORD PTR DS:[ECX+B8],2
004C9190   XOR EAX,EAX
004C9192   MOV DWORD PTR DS:[ECX+4],EAX
004C9195   MOV DWORD PTR DS:[ECX+8],EAX
004C9198   MOV DWORD PTR DS:[ECX+C],EAX
004C919B   MOV DWORD PTR DS:[ECX+10],EAX
004C919E   MOV DWORD PTR DS:[ECX+14],EAX
004C91A1   MOV DWORD PTR DS:[ECX+18],155
004C91A8   RETN
004C91A9   XOR EAX,EAX
004C91AB   PUSH DWORD PTR FS:[EAX]
004C91AE   MOV DWORD PTR FS:[EAX],ESP
004C91B1   XOR EAX,EAX
004C91B3   POP DWORD PTR DS:[EAX]
004C91B5   POP DWORD PTR FS:[0]
```

This code sets an exception handler (located at 004c9185) and at the line 004c91b3 it accesses a zero address to trigger the seh. In the seh we can see that the code zeroes registers (in the context provided by exception handler parameter) dr0, dr1, dr2, dr3, dr6 and sets in dr7 the bits corresponding to L0 – L3 and LE. In practice this interfers with the debugger, avoiding hardware memory breakpoints which rely on dr0-dr3. After this the exception handler quits returning the exception continuable value (0), so the execution is restored to the line 004c91b5 and the program continues running. Note that in the line 004c91b5 the unlinking of the exception record made with opcode 64:67:8F06 0000 sometimes causes troubles to ollydebug, in other packers I got troubles due to a massive use of these superfluous opcodes.
We continue on and we find some other junk code:

```
004C91C1   ...
...
004C91E7   POPAD
004C91E8   MOV EBX,DWORD PTR SS:[EBP+EDI*4+9CBA]
004C91EF   ADD EBX,EBP
004C91F1   MOV EDX,205
```

```
004C91F6   ADD EDX,EBP
004C91F8   PUSH EDX
004C91F9   CALL Armadill.004C9235
004C91FE   CALL Armadill.004C968C
004C9203   JMP EBX
004C9205   CMP EDI,4
004C9208   JNZ SHORT Armadill.004C91C1
```

This code dispatches the execution using a “call table” located at ebp+09CBA. EDI is the counter, we execute four functions. All this code is just used to decrypt the code that follows. At 004C91E8 in ebx is stored the pointer to the function taken from the function array, pointer which is relative to the loader starting offset, so ebp must be added to obtain the real address of the function. The if we trace the call to 004C9235 we find just a “JMP EBX” there, so the code jumps to the function.
The various functions are quite easy:

1°
```
  004D2C7F   MOV EAX,238
  004D2C84   RETN
```

2°
```
  004D2BFD   ADD EAX,EBP
  004D2BFF   RETN
```

3°
```
  004D2C44   MOV ECX,98AF
  004D2C49   RETN
```

4°
```
  004D2B0D   INC BYTE PTR DS:[EAX]
  004D2B44   ADD BYTE PTR DS:[EAX],CL
  004D2B6C   ROR BYTE PTR DS:[EAX],6
  004D2BA4   XOR BYTE PTR DS:[EAX],CL
  004D2BCC   INC EAX
  004D2BCD   DEC ECX
  004D2BCE   TEST ECX,ECX
  004D2BD0   JNZ Armadill.004D2AE7
  004D2BD6   RETN
```

This is the code without all the garbage polymorphic stuff. Basically it gets the relative pointer of the encrypted block in eax, relocates it by adding the loader base address, gets the size of the encrypted data in ecx, and then goes to the decrypt cycle, which performs a inc-add-ror-xor loop on each byte. The code being decrypted is located at 004C9238 and its length is 98AF bytes. You will find a lot of these loops, all with this similar structure (there are some variants), so just avoid tracing them and go directly to the following code that will be decrypted (usually you just need to go beyond the cmp edi, xx).
After another call table of eight functions, we arrive to:

```
004C9894   MOV ESI,EAX
004C9896   AND ESI,FFFF0000
004C989C   MOV EBX,0BF1
004C98A1   ADD EBX,EBP
004C98A3   CALL EBX
004C98A5   PUSHAD
```

EBX is the address of a function that returns the base address of kernel32, its parameter is passed in esi and is the return address of the code that called the entry point of the exe from kernel32. This return address was incremented by 0x13800, I don't know why, probably for some win9x compatibility problem. Of course the kernel32 base address is calculated by the code and not using GetModuleHandle or LoadLibrary. As we expect, after this the code:

```
004C999B   PUSH EAX
004C999C   CALL Armadill.004C9B04
```

finds the address of GetProcAddress,

```
004C9A40   PUSH EAX
004C9A41   CALL Armadill.004C9B04
```

finds the address of LoadLibraryA

```
004C9ADD   CALL EAX
```

makes a call to LoadLibraryA to have user32 handle

```
004C9AF0   PUSH EAX
004C9AF1   MOV EAX,DWORD PTR SS:[EBP+1608]
004C9AF7   CALL EAX
```

get the address of FindWindowA. Now

```
004C9EDD   INC BYTE PTR DS:[ESI]
004C9EDF   ROR BYTE PTR DS:[ESI],4
004C9EE2   XOR BYTE PTR DS:[ESI],CL
004C9EE4   INC ESI
004C9EE5   LOOPD SHORT Armadill.004C9EDD
004C9EE7   POPAD
```

this code decrypts a pseudo import table:

```
004CA471  49 73 44 65 62 75 67 67 65 72 50 72 65 73 65 6E
IsDebuggerPresen
004CA481  74 00 57 72 69 74 65 50 72 6F 63 65 73 73 4D 65
t.WriteProcessMe
004CA491  6D 6F 72 79 00 52 65 61 64 50 72 6F 63 65 73 73
mory.ReadProcess
```

```
004CA4A1  4D 65 6D 6F 72 79 00 57 61 69 74 46 6F 72 44 65
Memory.WaitForDe
004CA4B1  62 75 67 45 76 65 6E 74 00 47 65 74 56 65 72 73
bugEvent.GetVers
004CA4C1  69 6F 6E 00 47 65 74 4D 6F 64 75 6C 65 48 61 6E
ion.GetModuleHan
004CA4D1  64 6C 65 41 00 47 65 74 43 6F 6D 6D 61 6E 64 4C
dleA.GetCommandL
004CA4E1  69 6E 65 41 00 47 65 74 50 72 6F 63 41 64 64 72
ineA.GetProcAddr
004CA4F1  65 73 73 00 4C 6F 61 64 4C 69 62 72 61 72 79 41
ess.LoadLibraryA
004CA501  00 47 65 74 53 74 61 72 74 75 70 49 6E 66 6F 41
.GetStartupInfoA
004CA511  00 57 72 69 74 65 46 69 6C 65 00 53 65 74 45 6E
.WriteFile.SetEn
004CA521  76 69 72 6F 6E 6D 65 6E 74 56 61 72 69 61 62 6C
vironmentVariabl
004CA531  65 41 00 47 65 74 45 6E 76 69 72 6F 6E 6D 65 6E
eA.GetEnvironmen
004CA541  74 56 61 72 69 61 62 6C 65 41 00 47 65 74 43 75
tVariableA.GetCu
004CA551  72 72 65 6E 74 50 72 6F 63 65 73 73 49 64 00 53
rrentProcessId.S
004CA561  6C 65 65 70 00 43 72 65 61 74 65 46 69 6C 65 41
leep.CreateFileA
004CA571  00 43 72 65 61 74 65 54 6F 6F 6C 68 65 6C 70 33
.CreateToolhelp3
004CA581  32 53 6E 61 70 73 68 6F 74 00 50 72 6F 63 65 73
2Snapshot.Proces
004CA591  73 33 32 46 69 72 73 74 00 50 72 6F 63 65 73 73
s32First.Process
004CA5A1  33 32 4E 65 78 74 00 43 6C 6F 73 65 48 61 6E 64
32Next.CloseHand
004CA5B1  6C 65 00 00 00 00 00 00 00 00 00 00 00 00 00 00
le..............
```

Now every API address is stored in the import array, which starts at [EBP + 15B4]. I am talking of import array because this is not a real import table. It’s just a list of functions that will be used by the loader. Note that in every API about first five bytes are checked

```
004C9F51   MOV EDI,EAX
004C9F53   MOV ECX,4
004C9F58   MOV EAX,660
004C9F5D   SHR EAX,3
004C9F60   REPNE SCAS BYTE PTR ES:[EDI]
004C9F62   TEST ECX,ECX
004C9F64   JE SHORT Armadill.004C9F7D
```

0x660 shr 3 = 0xCC, so if you want to set a breakpoint on these api's you will need to put it a few lines after the API entry point, otherwise armadillo will set an internal error status value

```
004C9F6E   MOV DWORD PTR SS:[EBP+9620],6675636B
```

6675636B = ascii for "fuck", so the execution will terminate. Whenever you see the code keeping that error value, it means that you are in the wrong place! After the import addresses are built, the import names are re-crypted. Here we come:

```
004CA81C   CALL DWORD PTR SS:[EBP+15F0]
```

an interesting call to CreateFileA to open the file \\.\BCMDMCCP. This should be the driver of SpywareBlaster, so maybe armadillo has tampering problems with it. Now there is a really annoying part!

There are about ONE HUNDRED decryption layers; all are made with the call method we have seen above. Also seh are used among decryption routines. Note that the first of these layers will trigger a seh that will be resumed at line 004d25c5; where there is an invalid op code. Ollydbg will panice with this. You will need to trace it with shift-f9 to go on without problems. If you want you can trace all the hundred layers, but if you want to save your precious time then after all these layers there is a call to GetVersion, so just break on it and it's done. Get out from GetVersion and we come to:

```
004CEB6A   CPUID
004CEB6C   RDTSC
```

The important thing is RDTSC. Soon after this we have a seh trigger.
Inside we find:

```
004CF297   CPUID
004CF299   RDTSC
004CF29B   SUB EAX,DWORD PTR SS:[ESP]
004CF29E   ADD ESP,4
004CF2A1   CMP EAX,5FFFFFF
004CF2A6   JB SHORT Armadill.004CF2AF
004CF2A8   ADD DWORD PTR DS:[ECX+B8],67
004CF2AF   XOR EAX,EAX
004CF2B1   RETN
```

another RDTSC. This instruction reads the timestamp counter of the CPU (that is, the number of cycles). So, in the seh it reads again the timestamp counter and subtracts from it the previously read timestamp value. If the difference is bigger than 0x5FFFFFF then too much time is passed from one rdtsc to the other, and this means that someone is probably tracing the code (since tracing is

much more slower than real execution), so it tries to do something on context.eip, but the ecx pointer is zeroed, giving you another exception inside the exception handler itself.
Ok the end is near! We arrive to:

```
004CF499   ROR BYTE PTR DS:[EBX],CL
004CF49B   XOR BYTE PTR DS:[EBX],CL
004CF49D   INC EBX
004CF49E   DEC ECX
004CF49F   TEST ECX,ECX
004CF4A1   JNZ SHORT Armadill.004CF499
```

which is the code that decrypts the .text1 section. This is the code section of the protection. This is first layer.
We find the second at:

```
004CF5F5   XOR BYTE PTR DS:[EBX],AL
004CF5F7   INC EBX
004CF5F8   DEC ECX
004CF5F9   TEST ECX,ECX
004CF5FB   JNZ SHORT Armadill.004CF5F5
```

You can now look at 00479000 and you will see all the code is decrypted. So we must trace just a few instructions and we find:

```
004CF904   POPAD
004CF905   JMP EBX
```

where ebx is 004B4BE3. This is a jump to the code in the .text1 section, so the boring ASM loader is now finished! You can see in this entry point the standard msvc code for WinMain, so you now can dump the executable and disassemble it and you will see the disassembly of the protection code. Let’s go for it.

STEP 2: Fatherland – a parent takes care of his child

Armadillo is now running the code that will become the parent process. The same code will also split into the child process which we will see it later. First of all we let the armadillo run under Ollydbg. We will get a crash of olly itself: it seems nowadays everyone is using this trick, which consists of sending a malformed debug string (lots of %s) which will cause a buffer overflow. So to avoid any problems, put a breakpoint on OutputDebugStringA and modify the string parameter that is passed to it into a simple string (for example “hi\0”). There are two calls to OutputDebugStringA. Ok, at the beginning the loader is checking some string (INFO, REGISTER and so on) they are the parameters passed to the command line, going on we will find this:

```
004A07CD   MOV EAX,DWORD PTR DS:[4D93B4]
004A07D2   XOR EAX,DWORD PTR DS:[4D93C8]
```

```
004A07D8   PUSH EAX
004A07D9   CALL DWORD PTR DS:[<&KERNEL32.GetCurrent>;
kernel32.GetCurrentProcessId
004A07DF   PUSH EAX
004A07E0   PUSH Armadill.004D9674                   ; ASCII
"%X::DA%08X"
004A07E5   LEA ECX,DWORD PTR SS:[EBP-128]
004A07EB   PUSH ECX
004A07EC   CALL Armadill.004B47AA
004A07F1   ADD ESP,10
004A07F4   LEA EDX,DWORD PTR SS:[EBP-128]
004A07FA   PUSH EDX
004A07FB   PUSH 0
004A07FD   PUSH 1F0001
004A0802   CALL DWORD PTR DS:[<&KERNEL32.OpenMutexA>;
kernel32.OpenMutexA
004A0808   TEST EAX,EAX
004A080A   JE SHORT Armadill.004A0810
004A080C   MOV BYTE PTR SS:[EBP-24],0
004A0810   MOV EAX,DWORD PTR SS:[EBP-24]
```

It uses the string %X::DA%08X and transforms it in **PID**::DA**NUMBER** where PID is the process id of the actual current (parent) process and number is usually a fixed number given from the line 004A07CD (it is xored). The final string is so composed, for example for me it is 938::DAC858ADB2. Of course pid will be different at every execution. This string will be the mutex name used for the OpenMutexA function. Why does it try to open a mutex that has never been created??? Simple, this mutex will be checked also in the child process, so this is the point where the process knows if it must be launched in parent or child mode. If the mutex does not exist, the process will be parent, otherwise it will be child. Now we are going to trace the parent process, later we will see how to trace the child process. We soon find other similar code:

```
004A0C04   CALL DWORD PTR DS:[<&KERNEL32.OpenMutexA>;
kernel32.OpenMutexA
004A0C0A   TEST EAX,EAX
004A0C0C   JNZ Armadill.004A0E8C
```

Same as above, there are these two checks on the mutex before the code splits to child / parent flow. After this there is a call to IsDebuggerPresent, so it would be better if you set the PEB.BeingDebugged flag to zero every time you want to trace armadillo.
NOTE:
On winxpsp2 the PEB and TEB are no more at fixed addresses: usually PEB was always at 7FFDF000 and first TEB was 0x1000 bytes before PEB. Now they are in random addresses, so to find the PEB when the process starts you can look at TEB.Peb (TEB + 0x30), since the TEB address is indicated near the FS segment address base in Ollydbg.

Soon after this we see:

```
004A0D55   MOV EAX,DWORD PTR FS:[30]
004A0D5B   MOVZX EAX,BYTE PTR DS:[EAX+2]
004A0D5F   OR AL,AL
004A0D61   JNZ SHORT Armadill.004A0D81
```

A direct check to PEB.BeingDebugged without passing from IsDebuggerPresent, so don't use breaks on such an API. Now let's enter in this call.

```
004A0E3C   push    ecx
004A0E3D   call    004A35D8
```

In the following lines you see some calculus, just skip it, we will analyze it later; for now keep in mind what you have seen in those lines, that is a CreateFileMapping and some memory working on it and pointers adjustment. We come to a fundamental code:

```
004A4B05   CALL DWORD PTR DS:[<&KERNEL32.GetModuleF>;
kernel32.GetModuleFileNameW
004A4B0B   TEST EAX,EAX
004A4B0D   JNZ SHORT Armadill.004A4B16
004A4B0F   XOR AL,AL
004A4B11   JMP Armadill.004A74D2
004A4B16   MOV EAX,DWORD PTR DS:[4E0310]
004A4B1B   PUSH EAX
004A4B1C   LEA ECX,DWORD PTR SS:[EBP-7B4]
004A4B22   PUSH ECX
004A4B23   PUSH 0
004A4B25   PUSH 0
004A4B27   PUSH 4
004A4B29   PUSH 1
004A4B2B   PUSH 0
004A4B2D   PUSH 0
004A4B2F   CALL DWORD PTR DS:[<&KERNEL32.GetCommand>;
kernel32.GetCommandLineW
004A4B35   PUSH EAX
004A4B36   LEA EDX,DWORD PTR SS:[EBP-770]
004A4B3C   PUSH EDX
004A4B3D   CALL DWORD PTR DS:[<&KERNEL32.CreateProc>;
kernel32.CreateProcessW
```

Armadillo re-executes itself. The process being created is suspended but it's not created in debug mode. In the call

```
004A4C01   CALL Armadill.004A906D
```

Armadillo with ReadProcessMemory reads two bytes from the child process (the two bytes at entry point) and changes them into EB FE with a WriteProcessMemory, which is a self jumping instruction. This is made so that the child process once running will loop its

execution on the entry point. Once we get out from that call we find

```
004A4C13   CALL Armadill.004A929B
```

This is just a ResumeThread, Sleep, SuspendThread, used to run the program until it loops on the entry point.

```
004A4C25   CALL DWORD PTR DS:[<&KERNEL32.ResumeThre>;
kernel32.ResumeThread
004A4C2B   MOV EAX,DWORD PTR DS:[4E0310]
004A4C30   MOV ECX,DWORD PTR DS:[EAX+8]
004A4C33   PUSH ECX
004A4C34   CALL DWORD PTR DS:[<&KERNEL32.DebugActiv>;
kernel32.DebugActiveProcess
004A4C3A   MOV DWORD PTR SS:[EBP-20],EAX
004A4C3D   MOV EDX,DWORD PTR DS:[4E0310]
004A4C43   MOV EAX,DWORD PTR DS:[EDX+4]
004A4C46   PUSH EAX
004A4C47   CALL DWORD PTR DS:[<&KERNEL32.SuspendThr>;
kernel32.SuspendThread
004A4C4D   MOV ECX,DWORD PTR SS:[EBP-3C]
004A4C50   PUSH ECX
004A4C51   PUSH 0
004A4C53   MOV EDX,DWORD PTR DS:[4E0310]
004A4C59   PUSH EDX
004A4C5A   CALL Armadill.004A906D
```

The parent process attaches as a debugger to the child process. The call at 004A4C5A uses that function again to restore the bytes at the entry point removing the self-jumping op code. Again there is some calculus that for now we will skip until we arrive to

```
004A4DD9   CALL DWORD PTR DS:[<&KERNEL32.WaitForDeb>;
kernel32.WaitForDebugEvent
004A4DDF   TEST EAX,EAX
004A4DE1   JE Armadill.004A7493
```

OK, the parent process is waiting for the child events, so of course we are going to look what kind of events armadillo is interested in, and why. Just in case you miss the event constant identifiers, here it is a little list:

```
-------------------------------------------------------
EXCEPTION_DEBUG_EVENT        1
CREATE_THREAD_DEBUG_EVENT    2
CREATE_PROCESS_DEBUG_EVENT   3
EXIT_THREAD_DEBUG_EVENT      4
EXIT_PROCESS_DEBUG_EVENT     5
LOAD_DLL_DEBUG_EVENT         6
UNLOAD_DLL_DEBUG_EVENT       7
OUTPUT_DEBUG_STRING_EVENT    8
```

```
RIP_EVENT                    9

EXCEPTION_GUARD_PAGE                  80000001
EXCEPTION_DATATYPE_MISALIGNMENT       80000002
EXCEPTION_BREAKPOINT                  80000003
EXCEPTION_SINGLE_STEP                 80000004

EXCEPTION_ACCESS_VIOLATION            C0000005
EXCEPTION_IN_PAGE_ERROR               C0000006
EXCEPTION_ILLEGAL_INSTRUCTION         C000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION    C0000025
EXCEPTION_INVALID_DISPOSITION         C0000026
-------------------------------------------------------
```

in [ebp-0A24] you have the event type, so following the code and eliminating all the garbage you can make a map of the event handlers:

```
004a4f1f - EXCEPTION_DEBUG_EVENT
    004a4f67 - EXCEPTION_GUARD_PAGE
    004a5635 - EXCEPTION_ACCESS_VIOLATION
    004a5adf - EXCEPTION_BREAKPOINT
    004a6668 - STATUS_INVALID_HANDLE
    004a6668 - EXCEPTION_STACK_OVERFLOW
    004a6668 - STATUS_HANDLE_NOT_CLOSEABLE

004a6682 - CREATE_THREAD_DEBUG_EVENT

004a670b - EXIT_THREAD_DEBUG_EVENT

004a676c - EXIT_PROCESS_DEBUG_EVENT

004a78d6 - OUTPUT_DEBUG_STRING_EVENT

004a68ed - RIP_EVENT

004a6901 - CREATE_PROCESS_DEBUG_EVENT

004a6e0b - LOAD_DLL_DEBUG_EVENT

004a7446 - ContinueDebugEvent
```

the ones we are interested in are the EXCEPTION_GUARD_PAGE and EXCEPTION_BREAKPOINT. The rest are not important for our purposes, they regard the normal debugging cycle. For example, the createthread event keeps track of newly created threads and stores their handles, and so on. So for now let’s examine the event sequence, until we see the use of the two handlers we are interested in:

```
CREATE_PROCESS
LOAD_DLL
```

```
LOAD_DLL
LOAD_DLL
LOAD_DLL
CREATE_THREAD
EXCEPTION_DEBUG_EVENT  EXCEPTION_BREAKPOINT (first time its
ignored)
-----------------------------------------------------------------
--
EXIT_THREAD_DEBUG_EVENT  calls exit thread handler
EXCEPTION_ACCESS_VIOLATION  (violations in the loader, pop fs:eax
and other useless stuff)
   does nothing, goes to continuedebugevent
EXCEPTION_ACCESS_VIOLATION (4cca84, pop fs:eax in loader)
   does nothing, goes to continuedebugevent
CREATE_THREAD
LOAD_DLL (005b1800 uxtheme.dll)
LOAD_DLL (77be0000 msvcrt)
LOAD_DLL (77f40000 advapi32)
LOAD_DLL (77da0000 rpcrt4)
LOAD_DLL (5d4d0000 comctl32.dll)
LOAD_DLL (76360000 comdlg32.dll)
LOAD_DLL (77e90000 shlwapi.dll)
LOAD_DLL (7c9d0000 shell32.dll)
LOAD_DLL (773a0000 comctl32.dll relocated)
LOAD_DLL (770f0000 oleaut32.dll)
LOAD_DLL (774b0000 ole32.dll)
  call to OutputDebugString of the father
  call to OutputDebugString of the father
LOAD_DLL (71a30000 ws2_32.dll)
LOAD_DLL (71a20000 ws2help.dll)
LOAD_DLL (66BB0000 inetmib1.dll)
LOAD_DLL (72d60000 iphlpapi.dll)
LOAD_DLL (71ef0000 snmpapi.dll)
LOAD_DLL (71a50000 wsock32.dll)
LOAD_DLL (76d00000 mprapi.dll)
LOAD_DLL (77c90000 activeds.dll)
LOAD_DLL (76dd0000 adsldpc.dll)
LOAD_DLL (5bc70000 netapi32.dll)
LOAD_DLL (76f20000 wldap32.dll)
LOAD_DLL (76ae0000 atl.dll)
LOAD_DLL (76e40000 rtutils.dll)
LOAD_DLL (71b80000 samlib.dll)
LOAD_DLL (778f0000 setupapi.dll)
EXCEPTION_ACCESS_VIOLATION 00e4e065 in security.dll (ghost dll)
  .text:1002E063                 xor     eax, eax
  .text:1002E065                 mov     [eax], eax
  father does nothing
EXCEPTION_ACCESS_VIOLATION  00e4e2ca  same as above
EXCEPTION_ACCESS_VIOLATION  same as above, and so all the
following
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXIT_THREAD
```

```
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
LOAD_DLL (73390000 msvbvm60.dll)
EXCEPTION_ACCESS_VIOLATION  00e4e4fb
EXCEPTION_ACCESS_VIOLATION  00e4edbf
EXCEPTION_ACCESS_VIOLATION  00e4eff0
EXCEPTION_ACCESS_VIOLATION  00e4f221
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  ooe4f452
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4f683
EXCEPTION_ACCESS_VIOLATION  00e4e2ca
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION   "
LOAD_DLL (77bd0000 version.dll)
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e4fb
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e065
EXCEPTION_ACCESS_VIOLATION  00e4e72c
EXCEPTION_ACCESS_VIOLATION  00e4e95d
-------------------------------------------------------
EXCEPTION_GUARD_PAGE  00441cc0! first exception, we found OEP!
```

Okay you should have a similar event sequence. All the DLL's you see are probably loaded when the Armadillo is building the REAL import table. The first exception breakpoint event is ignored from the parent as it's just the normal DebugBreak call inside the newly created process. It's far from the program code itself. The interesting thing is the first GUARD_PAGE exception that we have, at address 00441CC0.

STEP 3: COPYMEM2 – swimming in a page pool

If we look at the memory of the child process at 00441CC0 we see:

```
seg000:00441CC0                 push    ecx
seg000:00441CC1                 mov     ebp, [edi]
seg000:00441CC3                 mov     dl, 0FBh
seg000:00441CC5                 push    404D13h
seg000:00441CCA                 stosd
```

```
seg000:00441CCB                 test    [esi], bl
seg000:00441CCD                 inc     esp
seg000:00441CCE                 retn
```

nonsense code! It is obvious that the page is encrypted. If we look at the process memory regions, for example with Lord-PE, we see that all code sections have the attribute GUARD_PAGE, except the one we are looking now. GUARD_PAGE in fact is a “one-shot access alarm” (I love this definition in the msdn!). However, all pages are in GUARD_PAGE, this means ALL PAGES ARE ENCRYPTED, and they will shoot a GUARD_PAGE notification when executed the first time. So this 00441CC0 is the FIRST line executed from the original code section. This means that this is the original entry point of the packed program. The problem now is: how do we dump the program? If we let it run we are not sure that ALL pages will be decrypted, and in fact you can try and see that even executing the program some pages remain encrypted. The simplest idea is: we force armadillo to decrypt all pages. We must examine the PAGE_GUARD handler.
I will paste the most important lines of a generic page decryption:

```
004A4FB9   MOV ECX,DWORD PTR DS:[EAX+24]    ; get exception
address

004A4FC2   CMP DWORD PTR DS:[4E032C],0      ; check if the number
of the decrypted pages is 0

004A5187   MOV ECX,DWORD PTR SS:[EBP-A3C]   ; ecx = fault address
004A518D   SUB ECX,DWORD PTR DS:[4E0314]    ; ecx -= 00401000
004A5193   SHR ECX,0C                       ; ecx = page number
004A5196   MOV DWORD PTR SS:[EBP-A34],ECX

004A5363   CMP DWORD PTR SS:[EBP-A34],0     ; check that
004A536A   JL Armadill.004A5618             ; 0 <= page number <=
47
004A5370   MOV ECX,DWORD PTR SS:[EBP-A34]
004A5376   CMP ECX,DWORD PTR DS:[4E0328]
004A537C   JGE Armadill.004A5618
```

We see a variable (004e032c) here that keeps the number of decrypted pages (break on this handler more times and you see that number being increased), then the page number is calculated from the virtual address, so the program can use it as an index for the page array in the code section. There is some calculus then stepping into two calls there is the code that we need

```
004A7E49   PUSH EAX                         ; old protection
004A7E4A   PUSH 4                           ; PAGE_READWRITE
004A7E4C   PUSH 1000                        ; size of the region
004A7E51   MOV ECX,DWORD PTR SS:[EBP-14]
```

```
004A7E54   PUSH ECX                          ; virtual address of
the faulting guarded page
004A7E55   MOV EDX,DWORD PTR DS:[4E0310]
004A7E5B   MOV EAX,DWORD PTR DS:[EDX]
004A7E5D   PUSH EAX                          ; child process handle
004A7E5E   CALL DWORD PTR DS:[<&KERNEL32.VirtualPro>;
kernel32.VirtualProtectEx
```

The page of the faulting address is turned back in the PAGE_READWRITE attribute.

```
004A7EE1   LEA ECX,DWORD PTR SS:[EBP-8]
004A7EE4   PUSH ECX                          ; ptr to number of
bytes read
004A7EE5   PUSH 1000                         ; number o bytes to
read
004A7EEA   MOV EDX,DWORD PTR DS:[4E033C]
004A7EF0   PUSH EDX                          ; pBuffer
004A7EF1   MOV EAX,DWORD PTR SS:[EBP-14]
004A7EF4   PUSH EAX                          ; virtual address of
the faulting page
004A7EF5   MOV ECX,DWORD PTR DS:[4E0310]
004A7EFB   MOV EDX,DWORD PTR DS:[ECX]
004A7EFD   PUSH EDX                          ; child process handle
004A7EFE   CALL DWORD PTR DS:[<&KERNEL32.ReadProces>;
kernel32.ReadProcessMemory
```

The parent process reads the memory of the page that was accessed and was in GUARD_PAGE protection.

```
004A88D2   CMP EDX,DWORD PTR SS:[EBP-28]
004A88D5   JNB SHORT Armadill.004A88EF
004A88D7   MOV EAX,DWORD PTR SS:[EBP-24]
004A88DA   MOV CL,BYTE PTR DS:[EAX]
004A88DC   XOR CL,BYTE PTR SS:[EBP-20]
004A88DF   MOV EDX,DWORD PTR SS:[EBP-24]
004A88E2   MOV BYTE PTR DS:[EDX],CL
004A88E4   MOV EAX,DWORD PTR SS:[EBP-24]
004A88E7   ADD EAX,1
004A88EA   MOV DWORD PTR SS:[EBP-24],EAX
004A88ED   JMP SHORT Armadill.004A88CF
```

This is the cycle that decrypts the memory read from the faulting page. It’s not the only one, another one is at

```
004A89F3   MOV EAX,DWORD PTR SS:[EBP-4]
004A89F6   CMP EAX,DWORD PTR SS:[EBP-C]
004A89F9   JNB SHORT Armadill.004A8A13
004A89FB   MOV ECX,DWORD PTR SS:[EBP-4]
004A89FE   MOV EDX,DWORD PTR DS:[ECX]
004A8A00   XOR EDX,DWORD PTR SS:[EBP-2C]
004A8A03   MOV EAX,DWORD PTR SS:[EBP-4]
```

```
004A8A06   MOV DWORD PTR DS:[EAX],EDX
004A8A08   MOV ECX,DWORD PTR SS:[EBP-4]
004A8A0B   ADD ECX,4
004A8A0E   MOV DWORD PTR SS:[EBP-4],ECX
004A8A11   JMP SHORT Armadill.004A89F3
```

Once the page is decrypted, it is written back to the process

```
004A8CD6   LEA EDX,DWORD PTR SS:[EBP-8]
004A8CD9   PUSH EDX                          ; number of bytes
written
004A8CDA   PUSH 1000                         ; number of bytes to
write
004A8CDF   MOV EAX,DWORD PTR DS:[4E033C]
004A8CE4   PUSH EAX                          ; pBuffer
004A8CE5   MOV ECX,DWORD PTR SS:[EBP-14]
004A8CE8   PUSH ECX                          ; address of the
faulting page
004A8CE9   MOV EDX,DWORD PTR DS:[4E0310]
004A8CEF   MOV EAX,DWORD PTR DS:[EDX]
004A8CF1   PUSH EAX                          ; handle of the
child process
004A8CF2   CALL DWORD PTR DS:[<&KERNEL32.WriteProce>;
kernel32.WriteProcessMemory
```

Finally, the attribute of the page is changed again:

```
004A8D93   LEA ECX,DWORD PTR SS:[EBP-18]
004A8D96   PUSH ECX
004A8D97   MOV EDX,DWORD PTR SS:[EBP-10]
004A8D9A   PUSH EDX
004A8D9B   PUSH 1000
004A8DA0   MOV EAX,DWORD PTR SS:[EBP-14]
004A8DA3   PUSH EAX
004A8DA4   MOV ECX,DWORD PTR DS:[4E0310]
004A8DAA   MOV EDX,DWORD PTR DS:[ECX]
004A8DAC   PUSH EDX
004A8DAD   CALL DWORD PTR DS:[<&KERNEL32.VirtualPro>;
kernel32.VirtualProtectEx
```

The page is changed to PAGE_EXECUTE_READ.
Now see this interesting compare:

```
004A7ABA   MOV EDX,DWORD PTR DS:[4E032C]     ; number of
decrypted pages
004A7AC0   CMP EDX,DWORD PTR DS:[4D97E4]     ; pool threshold
004A7AC6   JLE Armadill.004A7CA5
```

If the number of decrypted pages is below the threshold, then the execution goes to the ContinueDebugEvent so that the exception is repaired, and the child process continues execution with the decrypted page. If the number of the decrypted pages goes above

the threshold, then armadillo checks which pages are decrypted and how often they are used, then it re-encrypts them back to the process so that the number of decrypted pages is always under the threshold value. With this system the program is NEVER completely decrypted in memory, but at max only the page pool is decrypted at the same time. The page pool is the number of pages allowed to be simultaneously decrypted and present in memory. Usually the pool size is the half of the total number of code section pages. This means we can't dump the executable. Besides, if we try to dump it, usually the dumpers will fail because of the GUARD_PAGE attribute interfering with the ReadProcessMemory used to dump the executable image.

The first thing I did was to write a decrypter for the pages, but the decryption uses random keys that change at every execution, so the simplest solution is to force the routine to decrypt all pages, then dump the decrypted child process. To do this, we simply raise the threshold value so it is never reached, then we modify the code to loop for every page, so, start debugging Armadillo, and arrive at the first PAGE_GUARD exception. I executed all of the handler until it's end, at address 004a7446. Now the entry point page at 00441000 si decrypted, and we write our shell code.
I used the following steps:

[004d97e4] is set to 0x100. This is the threshold value, since the program has 0x48 code pages the re-encryption will never occur.
[004a7482] is set to 00400000. This is the page address that each time is passed as the faulting address.
I searched every occurrence of the faulting EIP given for this exception, and found it in these memory locations

0012bc40
0012bc4c
0012bc50
0012cd80
0012cd8c
0012cd90
0012cdbc

With every iteration the shell code must overwrite these with our page address (004a7482).
Here is the shell code:

```
004A7446   MOV EAX,DWORD PTR DS:[4A7484]   ; get page address
004A744B   ADD EAX,1000                    ; goto next page
004A7450   MOV DWORD PTR DS:[4A7484],EAX   ; set next page address
to all faultin address locations
004A7455   MOV DWORD PTR DS:[12BC40],EAX
004A745A   MOV DWORD PTR DS:[12BC4C],EAX
004A745F   MOV DWORD PTR DS:[12BC50],EAX
004A7464   MOV DWORD PTR DS:[12CD80],EAX
```

```
004A7469   MOV DWORD PTR DS:[12CD8C],EAX
004A746E   MOV DWORD PTR DS:[12CD90],EAX
004A7473   MOV DWORD PTR DS:[12CDBC],EAX
004A7478   CMP EAX,Armadill.00441000        ; loop from 00401000
to 00441000
004A747D   JLE Armadill.004A4F67            ; jump back to the
beginning of the page_guard handler
004A747F   JMP 004A7446                     ; we arrive here when
all pages before entry point are decrypted
```

This loop will cycle for all the pages until the entry point page, which is already decrypted, so we must not execute the handler on it. You just need to change a bit of the shell code to decrypt pages from the page after the entry point to the end of the section:

```
004A7446   MOV EAX,DWORD PTR DS:[4A7484]    ; now this starts
being 00441000 (entry point page)
004A744B   ADD EAX,1000                     ; and we go to next
page
004A7450   MOV DWORD PTR DS:[4A7484],EAX
004A7455   MOV DWORD PTR DS:[12BC40],EAX
004A745A   MOV DWORD PTR DS:[12BC4C],EAX
004A745F   MOV DWORD PTR DS:[12BC50],EAX
004A7464   MOV DWORD PTR DS:[12CD80],EAX
004A7469   MOV DWORD PTR DS:[12CD8C],EAX
004A746E   MOV DWORD PTR DS:[12CD90],EAX
004A7473   MOV DWORD PTR DS:[12CDBC],EAX
004A7478   CMP EAX,Armadill.00448000        ; loop until last code
section page
004A747D   JLE Armadill.004A4F67
```

After this shell code we finally will have decrypted all the code section. We can now dump the process and we have the code and data sections decrypted. Of course the exe will not run we must still fix some things.

STEP 4: IMPORT TABLE – Rebuilding after war

We fix the entry point of the dumped PE (oep is 00441CC0), and disassemble the dumped exe.
First thing I see is just the ep:

```
.text:00441CC0 55                      push    ebp             ;
sub_441CC0
.text:00441CC1 8B EC                   mov     ebp, esp
.text:00441CC3 6A FF                   push    0FFFFFFFFh
.text:00441CC5 68 D0 95 44 00          push    offset unk_4495D0
.text:00441CCA 68 5C 1A 44 00          push    offset
unknown_libname_1
.text:00441CCF 64 A1 00 00 00 00       mov     eax, large fs:0
```

```
.text:00441CD5 50                       push    eax
.text:00441CD6 64 89 25 00 00 00 00     mov     large fs:0, esp
.text:00441CDD 83 EC 58                 sub     esp, 58h
.text:00441CE0 53                       push    ebx
.text:00441CE1 56                       push    esi
.text:00441CE2 57                       push    edi
.text:00441CE3 89 65 E8                 mov     [ebp+var_18], esp
.text:00441CE6 FF 15 54 31 FC 00        call    dword ptr
ds:0FC3154h
```

This is again the standard entry point of a visual studio compiled exe. The call we see here should be a call to GetVersion, but instead of the API address, we find that horrible 0FC3154. This means that API addresses are redirected using a wrapper or something like this.

Also, we see from the op code that the import table information is completely destroyed, we don't have the FirstThunk or OriginalFirstThunk information. Everything relies on the memory bridge at 0FC3154. You can look at the code and see that a lot of imports are redirected towards that memory bridge. So what are we waiting for? Let's dump that bridge from the child process. You can use LordPE to see the child process memory regions and their size. The region we are dumping goes from 00FC2000 to 00FCB000. It's just a data region, so with IDA you can transform all the bytes in dwords. I used this little script

```
static DwordZone(Start, End)
{
   auto i;
   for(i=Start; i<End; I = i+4)
   {
      MakeDword(i);
   }
}
```

Lad it as an idc file and call it from idc command window passing starting and ending addresses. All the bridge bytes will be dworded. We see a lot of zeroed memory, a lot of strange nonsense dwords, the interesting thing is this little piece:

```
 garbage
seg000:00FC3050                 dd 0E865B5h    ; redirection /
emulation
seg000:00FC3054                 dd 0E8665Ch    ; redirection /
emulation
seg000:00FC3058                 dd 77D1C2B2h   ; direct thunk
seg000:00FC305C                 dd 0E865F9h    ; redirection /
emulation
seg000:00FC3060                 dd 0E865ACh    ; redirection /
emulation
seg000:00FC3064                 dd 77D1FD80h   ; direct thunk
```

```
 ...
seg000:00FC35A8                 dd 77D18F9Dh   ; direct thunk
seg000:00FC35AC                 dd 0E865BBh    ; redirection /
emulation
seg000:00FC35B0                 dd 0E8A46Ah    ; redirection /
emulation
seg000:00FC35B4                 dd 0E8673Eh    ; redirection /
emulation
garbage
```

This is our IAT. We see API thunks (those 77xxxxxx addresses), and some other strange 00E8xxxx thunk. This makes me think that some api's are called directly, some others are wrapped by another memory layer (the 00E8* one). Let's look again with LordPE at the child process memory. This time our attention should be attracted by the memory around this bridge:

```
address     size        protect      state
-----------------------------------------
00E70000    00001000    RW           COMMIT
00E71000    00033000    XRW          COMMIT
00EA4000    00003000    R            COMMIT
00EA7000    00013000    RW           COMMIT
00EBA000    00007000    R            COMMIT
```

The first block is of 0x1000 bytes, then the second is an executable block... it seems this is a dll! We can look at the module list of the child process, but there is no module at imagebase 00E70000, probably this is a dll that armadillo loaded by itself. Let's dump it!

We look into LordPE and it is a normal dll, it has imports and exports. In the export window and we also see its name string: "security.dll". There is only one export: SetFunctionAddresses. Ok, let's disassemble this dll and look at 0E8A452:

```
.text:00E8A452 sub_E8A452      proc near               ; DATA
XREF: .data:00EA8F64
.text:00E8A452                 mov     eax, dword_EB7D60
.text:00E8A457                 retn
.text:00E8A457 sub_E8A452      endp
```

where

```
.data:00EB7D60 dword_EB7D60    dd 0A280105h
```

This function just returns the value A280105, which is the value returned by GetVersion. This is API emulation. Luckily there are only a few emulated APIs. Other kinds of APIs are wrapped with some types of wrappers; for example:

```
.text:00E899FE                 push    ebp
```

```
.text:00E899FF                mov     ebp, esp
.text:00E89A01                push    ecx
.text:00E89A02                push    ebx
.text:00E89A03                push    esi
.text:00E89A04                push    edi
.text:00E89A05                pusha
.text:00E89A06                mov     edx, dword_EB7D58
.text:00E89A0C                add     edx, 64h
.text:00E89A0F                call    edx                 ; call
GetTickCount
.text:00E89A11                mov     edx, dword_EB7D14
.text:00E89A17                add     edx, 64h
.text:00E89A1A                mov     ecx, 5
.text:00E89A1F
.text:00E89A1F loc_E89A1F:                                ; CODE
XREF: sub_E899FE+26
.text:00E89A1F                cmp     byte ptr [edx], 0CCh
.text:00E89A22                jz      short loc_E89A34
.text:00E89A24                loop    loc_E89A1F
.text:00E89A26                push    [ebp+arg_C]
.text:00E89A29                push    [ebp+arg_8]
.text:00E89A2C                push    [ebp+arg_4]
.text:00E89A2F                push    [ebp+arg_0]
.text:00E89A32                call    edx             ; call
MessageBoxA
.text:00E89A34
.text:00E89A34 loc_E89A34:                                ; CODE
XREF: sub_E899FE+24
.text:00E89A34                mov     [ebp+var_4], eax
.text:00E89A37                popa
.text:00E89A38                mov     eax, [ebp+var_4]
.text:00E89A3B                pop     edi
.text:00E89A3C                pop     esi
.text:00E89A3D                pop     ebx
.text:00E89A3E                leave
.text:00E89A3F                retn    10
```

This wrapper calls MessageBoxA, before it calls GetTickCount to fool automatic tracers. First five bytes of the API are checked to see if there is a breakpoint on it (0xCC byte). API addresses are stored in the security dll but are subtracted by 0x64 bytes. That’s why you see those add edx, 64h. In another kind of wrapper the API call is encrypted instead:

```
.text:00E8667A                push    ebp
.text:00E8667B                mov     ebp, esp
.text:00E8667D                push    ebx
.text:00E8667E                push    esi
.text:00E8667F                mov     esi,
ds:EnterCriticalSection
.text:00E86685                push    edi
.text:00E86686                push    offset unk_EB7058
```

```
.text:00E8668B                call    esi ; EnterCriticalSection
.text:00E8668D                push    offset unk_EB7070
.text:00E86692                call    esi ; EnterCriticalSection
 ...
.text:00E867A5                xchg    eax, ecx
; encrypted block
.text:00E867A6                push    edi
.text:00E867A7                pop     edi
.text:00E867A8                nop
.text:00E867A9                xchg    eax, ecx
.text:00E867AA                cmp     [esi+2CE2EBCBh], esi
.text:00E867B0                sbb     cl, dl
.text:00E867B2                popf
.text:00E867B3                and     eax, 0F52E793Ch
.text:00E867B8                rep aas
.text:00E867BA                lodsd
.text:00E867BB                xchg    eax, ebp
.text:00E867BC                mov     ecx, 0F87CD732h
.text:00E867C1                in      al, dx
 ...
.text:00E86842                mov     esi,
ds:LeaveCriticalSection
.text:00E86848                push    offset unk_EB7070
.text:00E8684D                call    esi ; LeaveCriticalSection
.text:00E8684F                push    offset unk_EB7058
.text:00E86854                call    esi ; LeaveCriticalSection
.text:00E86856                mov     eax, edi
.text:00E86858                pop     edi
.text:00E86859                pop     esi
.text:00E8685A                pop     ebx
.text:00E8685B                pop     ebp
.text:00E8685C                retn    4
```

there are calls to Enter/LeaveCriticalSections (again it fools automatic tracers), then the call to the real API (in this case this wrapper calls LoadLibraryA) is in an encrypted code block. It is decrypted at runtime, the call is executed and then the block is re-crypted. There are some other simpler variants of these wrappers, where there are no tricks, or just some calls to time function. If you can’t figure out what API is called you can assemble a call to the wrapper in the child process and see where it goes. This introduces a new problem!

STEP 4.1: SECURITY.DLL – tracing the child process

Having this self loaded DLL in the child process means we need to debug it. How can we debug the child if the father is part of its execution? Well we can at least debug the loader until the oep of the application. Then PAGE_GUARD and BREAKPOINT faults will not be handled and the program won’t run, but that’s not a problem for now. First of all start the parent with olly. We break on

WaitForDebugEvent and let it run. Ok, we break on the first debug event, that is CREATE_PROCESS. Coming out from the API we are at

```
004A4DD9   CALL DWORD PTR DS:[<&KERNEL32.WaitForDeb>;
kernel32.WaitForDebugEvent
004A4DDF   TEST EAX,EAX
004A4DE1   JE Armadill.004A7493
004A4DE7   MOV EAX,DWORD PTR SS:[EBP-204]
```

If we trace the CREATE_PROCESS event handler we find

```
004A0802   CALL DWORD PTR DS:[<&KERNEL32.OpenMutexA>;
kernel32.OpenMutexA
004A0808   TEST EAX,EAX
004A080A   JE SHORT Armadill.004A0810
```

going on we see

```
004A6948   PUSH EAX   ; mutex name
004A6949   PUSH 0
004A694B   PUSH 0
004A694D   CALL DWORD PTR DS:[<&KERNEL32.CreateMute>;
kernel32.CreateMutexA
```

where the mutex name is PID::DANUMBER, and the mutex we have seen above. So reassuming we have:

```
Start Armadillo Execution
Check for PID::DANUMBER mutex. Does it exist:
  NO:  go on,
       start armadillo again in debug mode,
       trap CREATE_PROCESS and in this handler create
PID::DANUMBER
       where PID is the child process PID.
  YES: this execution must switch to child mode,
       load security.dll,
       prepare other stuff for copymem and nanomites
       and then run the original program.
```

also two other mutexes are created, named DILLOOEP and DILLOCREATE. We execute this handler so that per-process initialization is done correctly. Note that we also encounter this line

```
004A6989   MOV EDX,DWORD PTR DS:[ECX+4]
004A698C   PUSH EDX
004A698D   CALL DWORD PTR DS:[<&KERNEL32.ResumeThre>;
kernel32.ResumeThread
```

It resumes the main thread of the application. You must not execute it, just set edx to zero so the ResumeThread will fail. Why? Simple. Our idea is to detach the parent debugger, and

attach ourselves as debugger. If we resume the thread, when we detach, the debugger will be shut down, and the thread suspend count will go to zero, running the thread before we can attach to it. Once this resume is failed, we let the Armadillo run, and wait for the second break on WaitForDebugEvent. We exit from the function, and when at the line 004A4DDF we assemble

```
004A4DDF   PUSH 0A64              ; pid of your child process
004A4DE4   CALL kernel32.DebugActiveProcessStop
```

Now the parent is detached. Don’t close the parent debugged process. We open another instance of olly and attach to the child process (use its pid to recognize it). We will break on ntdll.DbgBreakPoint. We can set a bpx on program entry point (004C9000), and we must go to view->threads and resume the thread, because when we detached we missed the ContinueDebugEvent, so the thread suspend count was not updated.

Okay, once you resume the thread, you can run the child and it will break on the entry point. Now we can step through the child. We directly set a break on OpenMutexA, so we go to the parent / child forking code. We have two breaks, on the code we have just seen in step 2:

```
004A07FA   PUSH EDX
004A07FB   PUSH 0
004A07FD   PUSH 1F0001
004A0802   CALL DWORD PTR DS:[<&KERNEL32.OpenMutexA>;
kernel32.OpenMutexA
...
004A0BFC   PUSH EAX
004A0BFD   PUSH 0
004A0BFF   PUSH 1F0001
004A0C04   CALL DWORD PTR DS:[<&KERNEL32.OpenMutexA>;
kernel32.OpenMutexA
```

If you correctly executed the CREATE_PROCESS event handler in the parent, these function will return valid handles. So now that the mutexes are valid, the execution will go to the child flow. First thing we find is this call is:

```
004A0EAC   CALL Armadill.0049F70B
```

Step over it (remember that you must have a break on OutputDebugStringA to avoid olly crash) and you will see that after this call the dll is mapped and ready to be used (look in view->memory and see the region in 00E70000). We continue and see some calls to the dll, until we get to

```
004A0ED3   CALL DWORD PTR DS:[4DFE30]
```

This is a call to the dll, then if you let the program run you will arrive at

```
00441CC0   INC EDI
00441CC1   JMP FAR D984:09ED26B8
00441CC8   PUSH ESI
00441CC9   POPAD
00441CCA   CMP AL,10
```

This is the entry point, breaking on access. Now you can assemble the calls to the redirected apis, and trace them if needed. Of course you can trace the dll loading, just enter in the call to 0049F70B. The code allocates memory for the dll then maps it, then there is a call to:

```
0049F7E4   PUSH 0
0049F7E6   PUSH 1
0049F7E8   MOV EDX,DWORD PTR DS:[4E003C]
0049F7EE   PUSH EDX
0049F7EF   CALL DWORD PTR DS:[4E0040]    ;  dll entry point
```

the address 00EA31DD, which is the entry point of the dll. If you dump the dll, the image base will be 0x10000000, change it to 00E70000 so when disassembling it you will have the same memory addresses you see in the child process. NOTE that 0x00E70000 is a dynamic allocation, it will probably be a different address on other computers. After this there is a call to the only exported API from that dll

```
0049F844   PUSH ECX
0049F845   PUSH Armadill.004A1948
0049F84A   PUSH Armadill.0049F6F9
0049F84F   PUSH Armadill.004A1CD7
0049F854   PUSH Armadill.0047CF90
0049F859   PUSH Armadill.0047C32E
0049F85E   PUSH Armadill.0047BF2B
0049F863   PUSH Armadill.0047BED2
0049F868   PUSH Armadill.0047BEB8
0049F86D   PUSH Armadill.0047B7D0
0049F872   MOV EDX,DWORD PTR DS:[4E0158]        ;
Armadill.00400000
0049F878   PUSH EDX
0049F879   CALL DWORD PTR SS:[EBP-58]
;SetFunctionAddresses
```

it sets some function pointer.

```
0049F95B   LEA ECX,DWORD PTR SS:[EBP-3C]
0049F95E   PUSH ECX
0049F95F   CALL DWORD PTR SS:[EBP-10]
```

Here there are the calls to OutputDebugStringA. First thing I want to know is emulated apis. I know that there is the emulation code for GetVersion, we have seen above that the GetVersion value is placed at 00EB7D60, so we go to the call to the security.dll entry point, and set a memory breakpoint on 00EB7D60. We will break into

```
00E95AC3   CALL DWORD PTR DS:[EA4224]              ;
kernel32.GetVersion
00E95AC9   MOV DWORD PTR DS:[EB7D60],EAX
```

Okay, that's what we wanted to know. If we continue stepping we find some other emulated api

```
00E95B1A   CALL DWORD PTR DS:[EA4220]              ;
kernel32.GetCommandLineA
00E95B20   MOV DWORD PTR DS:[EB7D64],EAX
...
```

Our emulation information is built, now we know emulated apis:

```
.data:00EB7D5C dword_EB7D5C    dd 140000h              ; DATA
XREF: sub_E8A458 r
.data:00EB7D5C                                         ;
GetProcessHeap
.data:00EB7D60 dword_EB7D60    dd 0A280105h            ; DATA
XREF: sub_E8A452 r
.data:00EB7D60                                         ;
GetVersion
.data:00EB7D64 dword_EB7D64    dd 142378h              ; DATA
XREF: sub_E8A45E r
.data:00EB7D64                                         ;
GetCommandLineA
.data:00EB7D68 dword_EB7D68    dd 4E4h                 ; DATA
XREF: sub_E8A464 r
.data:00EB7D68                                         ; GetACP
```

only four. We are lucky. In the same way we find other iat information:

```
00E954BC   MOV EAX,DWORD PTR DS:[ EA422C ]
00E954C1   SUB EAX,64
00E954C4   MOV DWORD PTR DS:[EB7CFC],EAX           ;
kernel32.7C80FEC9

00E954D4   MOV EAX,DWORD PTR DS:[EA41E0]
00E954D9   SUB EAX,64
00E954DC   MOV DWORD PTR DS:[EB7D00],EAX           ;
kernel32.7C8100B5
...
```

and so on.

STEP 4.2: IMPORTS – Analysis and rebuilding

Now we know the emulated apis, and we know how to determine redirected apis. What we need to do is change the address of the redirection bridge with the address of the real API thunk in the program’s original code section.
We have:

```
 ORIGINAL CODE               BRIDGE            SECURITY.DLL
 ------------------------------------------------------------
 dword ptr ds:0FC3154h  -->  Redirection  -->  API wrapper
                             API Thunk         API emulation
```

The first thing we do is scan the original code and see what imports in the code are redirected to the security.dll. I have lost the code of such a scanner, but you can easly adapt the following code of the rebuilder to make it. Well, once the scan is complete you have a list of 0x00E8xxxx addresses that are used in the code section. With the previous step we have seen how to find the API to which they point to, so we will have the following list:

```
0x00E86EAE  GetEnvironmentVariableA
0x00E880CB  WriteFile
0x00e899fe  MessageBoxA
0x00e8667a  LoadLibraryA
0x00e85aa1  GetProcAddress
0x00e89e64  RegCreateKeyExA
0x00e898b2  EndDialog
0x00e8a74c  DialogBoxParamA
0x00e89a42  GetWindowTextA
0x00e88414  CloseHandle
0x00e8a23e  RegQueryValueA
0x00e8944d  FindFirstFileA
0x00e87a04  CreateFileA
0x00e87e8e  ReadFile
0x00e882a0  SetFilePointer
0x00e874f6  CreateThread
0x00e89975  GetModuleHandleA
0x00e8749c  ExitThread
0x00e88357  GetFileSize
0x00e88659  CreateFileMappingA
0x00e88980  MapViewOfFile
0x00e88aef  UnmapViewOfFile
0x00e89915  GlobalLock
0x00e898e5  GlobalUnlock
0x00e87398  ExitProcess
0x00e8a6e8  CreateDialogParamA
0x00e8a46a  FindResourceA
0x00e89945  SetHandleCount
0x00e87451  TerminateProcess
```

```
     0x00e8a452  GetVersion
     0x00e8a45e  GetCommandLineA
     0x00e871a2  GetEnvironmentStringsW
     0x00e86fac  GetEnvironmentStringsA
     0x00e88493  GetFileType
     0x00e8a464  GetACP
```

Now it’s done. We have everything we need to rebuild the import table.
First of all we add a new section to build the IAT

```
 SECTION   VAddress   Vsize      ROffset    Rsize      FLAGS
 ------------------------------------------------------------
 .myiat    003C7000   00005000   003C7000   00005000   E00000E0
```

Then in the first 0x200 bytes we put the import table itself, with all descriptors. I made descriptors for all the dlls of the process, though they will not be in the import table, so remember to delete the unwanted ones after the rebuilding.

```
->Import Table
   1. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CB200
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFF0  ("ntdll.dll")
    FirstThunk:          0x003CB200

    Ordinal/Hint API name
    ------------ ---------------------------------------

   2. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7400
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFE0  ("kernel32.dll")
    FirstThunk:          0x003C7400

    Ordinal/Hint API name
    ------------ ---------------------------------------

   3. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7600
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFD0  ("user32.dll")
    FirstThunk:          0x003C7600

    Ordinal/Hint API name
```

```
    ------------ ----------------------------------------

   4. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7800
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFC0  ("gdi32.dll")
    FirstThunk:          0x003C7800

    Ordinal/Hint API name
    ------------ ----------------------------------------

   5. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7A00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFB0  ("uxtheme.dll")
    FirstThunk:          0x003C7A00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   6. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7C00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBFA0  ("msvcrt.dll")
    FirstThunk:          0x003C7C00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   7. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C7E00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF90  ("advapi32.dll")
    FirstThunk:          0x003C7E00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   8. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8000
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF80  ("rpcrt4.dll")
```

```
    FirstThunk:          0x003C8000

    Ordinal/Hint API name
    ------------ ----------------------------------------

   9. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8200
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF70  ("comctl32.dll")
    FirstThunk:          0x003C8200

    Ordinal/Hint API name
    ------------ ----------------------------------------

   10. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8400
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF60  ("comdlg32.dll")
    FirstThunk:          0x003C8400

    Ordinal/Hint API name
    ------------ ----------------------------------------

   11. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8600
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF50  ("shlwapi.dll")
    FirstThunk:          0x003C8600

    Ordinal/Hint API name
    ------------ ----------------------------------------

   12. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8800
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF40  ("shell32.dll")
    FirstThunk:          0x003C8800

    Ordinal/Hint API name
    ------------ ----------------------------------------

   13. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8A00
```

```
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF30  ("comctl32.dll")
    FirstThunk:          0x003C8A00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   14. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8C00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF20  ("oleaut32.dll")
    FirstThunk:          0x003C8C00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   15. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C8E00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF10  ("ole32.dll")
    FirstThunk:          0x003C8E00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   16. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9000
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBF00  ("ws2_32.dll")
    FirstThunk:          0x003C9000

    Ordinal/Hint API name
    ------------ ----------------------------------------

   17. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9200
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBEF0  ("ws2help.dll")
    FirstThunk:          0x003C9200

    Ordinal/Hint API name
    ------------ ----------------------------------------
```

```
   18. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9400
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBEE0  ("inetmib1.dll")
    FirstThunk:          0x003C9400

    Ordinal/Hint API name
    ------------ ----------------------------------------

   19. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9600
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBED0  ("iphlpapi.dll")
    FirstThunk:          0x003C9600

    Ordinal/Hint API name
    ------------ ----------------------------------------

   20. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9800
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBEC0  ("snmpapi.dll")
    FirstThunk:          0x003C9800

    Ordinal/Hint API name
    ------------ ----------------------------------------

   21. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9A00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBEB0  ("wsock32.dll")
    FirstThunk:          0x003C9A00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   22. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9C00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBEA0  ("mp3api.dll")
    FirstThunk:          0x003C9C00
```

```
    Ordinal/Hint API name
    ------------ ----------------------------------------

   23. ImageImportDescriptor:
    OriginalFirstThunk:  0x003C9E00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE90  ("activeds.dll")
    FirstThunk:          0x003C9E00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   24. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CA000
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE80  ("adsldpc.dll")
    FirstThunk:          0x003CA000

    Ordinal/Hint API name
    ------------ ----------------------------------------

   25. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CA200
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE70  ("netapi32.dll")
    FirstThunk:          0x003CA200

    Ordinal/Hint API name
    ------------ ----------------------------------------

   26. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CA400
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE60  ("wldap32.dll")
    FirstThunk:          0x003CA400

    Ordinal/Hint API name
    ------------ ----------------------------------------

   27. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CA600
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
```

```
    ForwarderChain:      0x00000000
    Name:                0x003CBE50  ("atl.dll")
    FirstThunk:          0x003CA600

    Ordinal/Hint API name
    ------------ ----------------------------------------

   28. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CA800
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE40  ("rtutils.dll")
    FirstThunk:          0x003CA800

    Ordinal/Hint API name
    ------------ ----------------------------------------

   29. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CAA00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE30  ("samlib.dll")
    FirstThunk:          0x003CAA00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   30. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CAC00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE20  ("setupapi.dll")
    FirstThunk:          0x003CAC00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   31. ImageImportDescriptor:
    OriginalFirstThunk:  0x003CAE00
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE10  ("msvbvm60.dll")
    FirstThunk:          0x003CAE00

    Ordinal/Hint API name
    ------------ ----------------------------------------

   32. ImageImportDescriptor:
```

```
    OriginalFirstThunk:  0x003CB000
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x003CBE00  ("version.dll")
    FirstThunk:          0x003CB000

    Ordinal/Hint API name
    ------------ ----------------------------------------

   33. ImageImportDescriptor:
    OriginalFirstThunk:  0x000D902C
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x000E2378  ("KERNEL32.dll")
    FirstThunk:          0x000D902C

    Ordinal/Hint API name
    ------------ ----------------------------------------
    Bad RVA in thunk !
    ...

   34. ImageImportDescriptor:
    OriginalFirstThunk:  0x000D91D0
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x000E265E  ("USER32.dll")
    FirstThunk:          0x000D91D0

    Ordinal/Hint API name
    ------------ ----------------------------------------
    Bad RVA in thunk !
    ...

   35. ImageImportDescriptor:
    OriginalFirstThunk:  0x000D9000
    TimeDateStamp:       0x00000000  (GMT: Thu Jan 01 00:00:00
1970)
    ForwarderChain:      0x00000000
    Name:                0x000E2706  ("GDI32.dll")
    FirstThunk:          0x000D9000

    Ordinal/Hint API name
    ------------ ----------------------------------------
    Bad RVA in thunk !
    ...
```

The last three import descriptors are the ones that originally were in the IT.
Here is the code for the rebuilder:

>>>>>>>> SEE Importalizer\data.h <<<<<<<<<<
>>>>>>>> SEE Importalizer\main.cpp <<<<<<<<<<

The program scans the code section to find all addresses relative to the memory bridge (0x00FC3*). When found, it looks in the bridge for the bytes at the given address. If these bytes are not an API thunk then they are the address of the security.dll, so from our associative array it extracts the API thunk corresponding to the security.dll wrapper. Once the API thunk is found, the program writes it in our newly created import table. As a result we will obtain an executable with all imports pointing to the correct API addresses. We created the import table so that the pointer of the OriginalFirstThunk is the same of the FirstThunk. Now that we have the correct import table and the correct import address table, we can use any import rebuilder, and our OriginalFirstThunk will be created. Now we have a perfect import table.

STEP 5: NANOMITES – Here comes the pain!

The program is dumped, decrypted and rebuilt, but it still is not running. Let’s have a look at the code, everything is ok, the imports are ok, let’s look at the WinMain:

```
00401D03 55                   push    ebp
00401D04 8B EC                mov     ebp, esp
00401D06 81 EC 38 0C 00 00    sub     esp, 0C38h
00401D0C 53                   push    ebx
00401D0D 56                   push    esi
00401D0E 57                   push    edi
00401D0F CC                   int     3
00401D10 D4 C6                aam     0C6h
00401D12 00 0B                add     [ebx], cl
00401D14 8B 45 08             mov     eax, [ebp+8]
00401D17 33 DB                xor     ebx, ebx
00401D19 88 5D FF             mov     [ebp-1], bl
00401D1C 88 5D FD             mov     [ebp-3], bl
```

int 3? What is it doing there? If we look around we find a lot of those int 3’s. Of course this means we need to debug the parent process in the EXCEPTION_BREAKPOINT handler. If we have a fast look, we break on that handler (004a5adf), we see a call to GetThreadContext which gives us the context.eip = 00401D10, then at the end of the handler we have a call to SetThreadContext, where context.eip is being set to 00401D14. So basically int 3 is just a change of eip, a jump. Now we must analyse the handler; here is the code of it, with no garbage and divided into blocks:

```
------------------
BLOCK 1
```

```
------------------

.text1:004A5ADF                 mov     dword ptr [ebp-1178h], 10h
.text1:004A5AE9                 mov     eax, ds:dword_4D9378
.text1:004A5AEE                 xor     eax, ds:dword_4D93B4
.text1:004A5AF4                 xor     eax, ds:dword_4D937C
.text1:004A5AFA                 mov     [ebp-1174h], eax
.text1:004A5B00                 mov     ecx, ds:dword_4D938C
.text1:004A5B06                 xor     ecx, ds:dword_4D9370
.text1:004A5B0C                 xor     ecx, ds:dword_4D93B8
.text1:004A5B12                 mov     [ebp-1170h], ecx
.text1:004A5B18                 mov     edx, ds:dword_4D9384
.text1:004A5B1E                 xor     edx, ds:dword_4D93C8
.text1:004A5B24                 xor     edx, ds:dword_4D93F0
.text1:004A5B2A                 mov     [ebp-116Ch], edx
.text1:004A5B30                 mov     eax, ds:dword_4D93E8
.text1:004A5B35                 xor     eax, ds:dword_4D93C0
.text1:004A5B3B                 xor     eax, ds:dword_4D93AC
.text1:004A5B41                 mov     [ebp-1168h], eax
.text1:004A5B47                 mov     ecx, ds:dword_4D93EC
.text1:004A5B4D                 xor     ecx, ds:dword_4D9398
.text1:004A5B53                 xor     ecx, ds:dword_4D9374
.text1:004A5B59                 mov     [ebp-1164h], ecx
.text1:004A5B5F                 mov     edx, ds:dword_4D93E4
.text1:004A5B65                 xor     edx, ds:dword_4D93A8
.text1:004A5B6B                 xor     edx, ds:dword_4D93F4
.text1:004A5B71                 mov     [ebp-1160h], edx
.text1:004A5B77                 mov     eax, ds:dword_4D9378
.text1:004A5B7C                 xor     eax, ds:dword_4D938C
.text1:004A5B82                 xor     eax, ds:dword_4D9384
.text1:004A5B88                 xor     eax, ds:dword_4D93D0
.text1:004A5B8E                 mov     [ebp-115Ch], eax
.text1:004A5B94                 mov     ecx, ds:dword_4D93B4
.text1:004A5B9A                 xor     ecx, ds:dword_4D9370
.text1:004A5BA0                 xor     ecx, ds:dword_4D93C8
.text1:004A5BA6                 xor     ecx, ds:dword_4D93E4
.text1:004A5BAC                 mov     [ebp-1158h], ecx
.text1:004A5BB2                 mov     edx, ds:dword_4D93E8
.text1:004A5BB8                 xor     edx, ds:dword_4D93EC
.text1:004A5BBE                 xor     edx, ds:dword_4D93E4
.text1:004A5BC4                 xor     edx, ds:dword_4D93A8
.text1:004A5BCA                 mov     [ebp-1154h], edx
.text1:004A5BD0                 mov     eax, ds:dword_4D93C0
.text1:004A5BD5                 xor     eax, ds:dword_4D9398
.text1:004A5BDB                 xor     eax, ds:dword_4D93A8
.text1:004A5BE1                 xor     eax, ds:dword_4D9378
.text1:004A5BE7                 mov     [ebp-1150h], eax
.text1:004A5BED                 mov     ecx, ds:dword_4D9378
.text1:004A5BF3                 xor     ecx, ds:dword_4D93EC
.text1:004A5BF9                 xor     ecx, ds:dword_4D938C
.text1:004A5BFF                 mov     [ebp-114Ch], ecx
.text1:004A5C05                 mov     edx, ds:dword_4D93B4
```

```
.text1:004A5C0B                 xor     edx, ds:dword_4D93C0
.text1:004A5C11                 xor     edx, ds:dword_4D9384
.text1:004A5C17                 mov     [ebp-1148h], edx
.text1:004A5C1D                 mov     eax, ds:dword_4D938C
.text1:004A5C22                 xor     eax, ds:dword_4D93E8
.text1:004A5C28                 xor     eax, ds:dword_4D93E4
.text1:004A5C2E                 mov     [ebp-1144h], eax
.text1:004A5C34                 mov     ecx, ds:dword_4D9370
.text1:004A5C3A                 xor     ecx, ds:dword_4D93C8
.text1:004A5C40                 xor     ecx, ds:dword_4D93E8
.text1:004A5C46                 mov     [ebp-1140h], ecx
.text1:004A5C4C                 mov     edx, ds:dword_4D9378
.text1:004A5C52                 xor     edx, ds:dword_4D9370
.text1:004A5C58                 xor     edx, ds:dword_4D93E8
.text1:004A5C5E                 xor     edx, ds:dword_4D93EC
.text1:004A5C64                 mov     [ebp-113Ch], edx
.text1:004A5C6A                 mov     eax, ds:dword_4D93B4
.text1:004A5C6F                 xor     eax, ds:dword_4D9384
.text1:004A5C75                 xor     eax, ds:dword_4D93C0
.text1:004A5C7B                 xor     eax, ds:dword_4D9398
.text1:004A5C81                 mov     [ebp-1138h], eax
.text1:004A5C87                 xor     ecx, ecx
.text1:004A5C89                 mov     cl, ds:FirstBreak
.text1:004A5C8F                 test    ecx, ecx
.text1:004A5C91                 jz      ContinueEvent

.text1:004A61E4                 push    2CCh
.text1:004A61E9                 push    0
.text1:004A61EB                 lea     edx, [ebp-1468h]
.text1:004A61F1                 push    edx
.text1:004A61F2                 call    AllocBuffer
.text1:004A61F7                 add     esp, 0Ch
.text1:004A61FA                 mov     dword ptr [ebp-1468h],
10001h
.text1:004A6204                 lea     eax, [ebp-1468h]
.text1:004A620A                 push    eax
.text1:004A620B                 mov     ecx, [ebp-1194h]
.text1:004A6211                 push    ecx
.text1:004A6212                 call    ds:GetThreadContext

----------------------
BLOCK 2
--------------------

.text1:004A625C                 mov     dword ptr [ebp-146Ch], 0
.text1:004A6266                 push    0FFFFFFFFh
.text1:004A6268                 push    4
.text1:004A626A                 lea     edx, [ebp-13B0h]
.text1:004A6270                 push    edx
.text1:004A6271                 call    GetFirstParameter
.text1:004A6276                 add     esp, 0Ch
.text1:004A6279                 mov     [ebp-FirstParameterV], eax
```

```
------------------------
block 3
------------------------

.text1:004A627F                 mov     eax, [ebp-FirstParameterV]
.text1:004A6285                 xor     edx, edx
.text1:004A6287                 mov     ecx, 10h
.text1:004A628C                 div     ecx
.text1:004A628E                 mov     [ebp-119Ch], edx
.text1:004A6294                 mov     edx, [ebp-13B0h]
.text1:004A629A                 push    edx
.text1:004A629B                 mov     eax, [ebp-119Ch]
.text1:004A62A1                 call    ds:off_4DDD3C[eax*4]
.text1:004A62A8                 add     esp, 4
.text1:004A62AB                 mov     [ebp-146Ch], eax

------------------------
block 4
------------------------

.text1:004A62B1                 mov     dword ptr [ebp-1470h], 0
.text1:004A62BB                 mov     ecx, [ebp-119Ch]
.text1:004A62C1                 mov     edx,
ds:dword_4E0230[ecx*4]
.text1:004A62C8                 mov     [ebp-1190h], edx
.text1:004A62CE
.text1:004A62CE loc_4A62CE:
.text1:004A62CE                 mov     eax, [ebp-1470h]
.text1:004A62D4                 cmp     eax, [ebp-1190h]
.text1:004A62DA                 jge     short near ptr byte_4A6338
.text1:004A62DC
.text1:004A62DC loc_4A62DC:
.text1:004A62DC                 mov     eax, [ebp-1190h]
.text1:004A62E2                 sub     eax, [ebp-1470h]
.text1:004A62E8                 cdq
.text1:004A62E9                 sub     eax, edx
.text1:004A62EB                 sar     eax, 1
.text1:004A62ED                 mov     ecx, [ebp-1470h]
.text1:004A62F3                 add     ecx, eax
.text1:004A62F5                 mov     [ebp-1474h], ecx
.text1:004A62FB                 mov     edx, [ebp-119Ch]
.text1:004A6301                 mov     eax,
ds:dword_4E01D0[edx*4]
.text1:004A6308                 mov     ecx, [ebp-1474h]
.text1:004A630E                 mov     edx, [ebp-146Ch]
.text1:004A6314                 cmp     edx, [eax+ecx*4]
.text1:004A6317                 jbe     short loc_4A632A
.text1:004A6319                 mov     eax, [ebp-1474h]
.text1:004A631F                 add     eax, 1
.text1:004A6322                 mov     [ebp-1470h], eax
.text1:004A6328                 jmp     short loc_4A6336
```

```
.text1:004A632A
.text1:004A632A loc_4A632A:
.text1:004A632A                 mov     ecx, [ebp-1474h]
.text1:004A6330                 mov     [ebp-1190h], ecx
.text1:004A6336
.text1:004A6336 loc_4A6336:
.text1:004A6336                 jmp     short loc_4A62CE

------------------------
 block 5
------------------------

.text1:004A635E                 mov     edx, [ebp-119Ch]
.text1:004A6364                 mov     eax,
ds:dword_4E01D0[edx*4]
.text1:004A636B                 mov     ecx, [ebp-1470h]
.text1:004A6371                 mov     edx, [eax+ecx*4]
.text1:004A6374                 cmp     edx, [ebp-146Ch]
.text1:004A637A                 jnz     ContinueEvent

------------------------
 block 6
------------------------

.text1:004A63D7                 mov     eax, [ebp-119Ch]
.text1:004A63DD                 mov     ecx,
ds:dword_4E0270[eax*4]
.text1:004A63E4                 mov     edx, [ebp-1470h]
.text1:004A63EA                 mov     eax, [ecx+edx*4]
.text1:004A63ED                 mov     [ebp-1488h], eax
.text1:004A63F3                 mov     ecx, [ebp-13A8h] ; eflags
.text1:004A63F9                 and     ecx, 0FD7h
.text1:004A63FF                 mov     [ebp-1478h], ecx
.text1:004A6405                 mov     edx, [ebp-1488h]
.text1:004A640B
.text1:004A640B loc_4A640B:
.text1:004A640B                 and     edx, 0FF000000h
.text1:004A6411                 shr     edx, 18h
.text1:004A6414                 mov     [ebp-1484h], edx
.text1:004A641A                 mov     eax, [ebp-1488h]
.text1:004A6420                 and     eax, 0FFFFFFh
.text1:004A6425                 mov     [ebp-1480h], eax
.text1:004A642B                 mov     ecx, [ebp-13BCh]
.text1:004A6431                 push    ecx
.text1:004A6432                 mov     edx, [ebp-1478h]
.text1:004A6438                 push    edx             ; eflags
.text1:004A6439                 mov     eax, [ebp-1480h]
.text1:004A643F                 push    eax
.text1:004A6440                 mov     ecx, [ebp-1484h]
.text1:004A6446                 call    ds:off_4D97E8[ecx*4]
.text1:004A644D                 add     esp, 0Ch
.text1:004A6450                 mov     [ebp-147Ch], eax
```

```
.text1:004A6456                 mov     edx, [ebp-147Ch]
.text1:004A645C                 and     edx, 1
.text1:004A645F                 test    edx, edx
.text1:004A6461                 jz      near ptr
SkipSecondCalculus

----------------------
 block 7
---------------------

.text1:004A648D SecondCalculus:
.text1:004A648D                 mov     eax, [ebp-119Ch]
.text1:004A6493                 mov     ecx,
ds:dword_4E0190[eax*4]
.text1:004A649A                 mov     eax, [ebp-1470h]
.text1:004A64A0                 xor     edx, edx
.text1:004A64A2                 mov     esi, 10h
.text1:004A64A7                 div     esi
.text1:004A64A9                 mov     eax, [ebp-1470h]
.text1:004A64AF                 mov     ecx, [ecx+eax*4]
.text1:004A64B2                 xor     ecx, [ebp+edx*4-1174h]
.text1:004A64B9                 mov     edx, [ebp-13B0h]
.text1:004A64BF                 add     edx, ecx
.text1:004A64C1                 mov     [ebp-13B0h], edx
                          goto setthreadcontext
.text1:004A64C1 ; ---------------------------------------------
---------------------------
.text1:004A64C7                 db 51h, 0Fh, 0C9h, 0F7h, 0D1h,
50h, 0F7h, 0D0h, 0B8h, 6Dh ; goto 004a65d6

----------------------
 block 8
---------------------

.text1:004A6515 SkipSecondCalculus
.text1:004A6515          db 70h, 7, 7Ch, 3, 0EBh, 5, 0E8h, 74h,
0FBh, 0EBh, 0F9h
.text1:004A6515
.text1:004A6520 ; ---------------------------------------------
---------------------------
.text1:004A6520                 mov     eax, [ebp-119Ch]
.text1:004A6526                 mov     ecx,
ds:dword_4E02B8[eax*4]
.text1:004A652D                 mov     edx, [ebp-1470h]
.text1:004A6533                 xor     eax, eax
.text1:004A6535                 mov     al, [ecx+edx]
.text1:004A6538                 mov     ecx, [ebp-13B0h]
.text1:004A653E                 add     ecx, eax
.text1:004A6540                 mov     [ebp-13B0h], ecx
                           goto setthreadcontext

setthreadcontext:
```

  updates information in the context, goto ContinueDebugEvent
ContinueEvent:
  continue the debugging cycle

All these code blocks use static or dynamic arrays of data and functions, some functions are executed dynamically. Maybe using some image will clarify the situation:

BLOCK 1
Makes a static calculus, which is always the same, and gets the context of the faulting instruction

BLOCK 2		
Calls a function that given the context.eip address calculates a 32bit number using an array of data	--->	Array of 0x400 bytes

BLOCK 3				
Uses an array of 16 functions to compute a 32bit number from context.eip	--->	Array of 16 functions, each function uses a max of 4 different subfunctions	--->	4 sub functions

BLOCK 4		
Uses an array of 16 8-bit values to extract a byte from an array containing 16 dynamic memory data arrays	--->	Array of 16 8-bit values (stored as dwords) Array of 16 dynamic memory blocks containing the data

BLOCK 5		Array of 16 dynamic

---------- Just makes a check using an array of 16 dynamic memory blocks	--->	memory arrays

BLOCK 6 ---------- Uses an array of 16 dynamic memory blocks to extract the index of the function that must be executed from an array of 256 functions	--->	Array of 256 static functions, each one calls some stati subfunctions, and other dynamic functions determined dynamically	--->	array of 256 dynamic functions, which call a sublayer of functions

BLOCK 7 ---------- extracts a dword from an array of 16 memory arrays, then xores this dword with another dword taken from an array of other 16 dwords, the obtained number is the number of bytes that will be jumped	--->	Array of 16 dynamic memory data arrays Array of 16 dword used for xoring

BLOCK 8 ---------- uses an array of 16 dynamic memory arrays to extract the number of bytes that will be jumped	--->	Array of 16 dynamic data arrays containing number of displacement bytes

Okay this is the structure of the nanomites processing. We see that the EIP and EFLAGS from the context of the child process are used in the calculus, so what is now clear is that nanomites are used to emulate jumps, both conditional or unconditional. Every 0xCC in the code represents a jump. What we can do now is try to fix every 0xCC to its original opcode, or make an emulator of

these jumps. Fixing the code is not easy, we should code a scanner which supports instruction tracing. Armadillo infact can’t recognize if a 0xCC is really a nonomite or not, so for example if we scan the code section and we meet the bytes of the instruction

```
mov eax, 0xCC
```

we would trash the original instruction trying to fix this non-nanomite. Even having a disassembling engine, it would be easy to make mistakes and patch wrong 0xCCs. So what I did is coded the emulator. I took all the functions of the breakpoint handler, and inlined them in C, then dumped all the data used by the emulation blocks and the functions. It needed to be modified a little, but with some time the work is done. Once I had all the emulation functions, I just had to code a parent process which makes emulation:

```
>>>>>>>> SEE Nanomulator\main.cpp <<<<<<<<<<
>>>>>>>> SEE Nanomulator\core.h <<<<<<<<<<
>>>>>>>> SEE Nanomulator\core.cpp <<<<<<<<<<
>>>>>>>> SEE Nanomulator\coredata.cpp <<<<<<<<<<
```

THE END

Thanx to Arilou and Neoxquick who helped me with Armadillo. Thanx to my personal beta-reader, Devine9.
Greetings to all my RET friends and all UIC guys.
A super greet to the super The_Enigma aka Giulia :)
GoodBye!

[AndreaGeddon] andreageddon@gmail.com my mail
[RET] www.reteam.org RET's great site
[UIC] www.quequero.org Italian University of Cracking