AthCon 2011

Capture the Flag Reversing Challenge

June 2011

George Nicolaou
ishtus{ατ}astalavista{δοτ}com

Glafkos Charalambous
glafkos{ατ}astalavista{δοτ}com

Table of Contents

Table of Figures

Chapter 1. Introduction

Hello and welcome to the submitted report document on Athcon 2011 Capture the Flag Reversing Part. This year's challenge includes three (3) files:

1. %s%s%sAthcon_2011%s%s%s
2. %s%s%sGet_It_All%s%s%s.exe
3. Passes.rar

The solution requirements included inside the challenge archive state the following:
Solution: Find the password to unlock the Passes.rar file. In case you solve it, in order to claim the ticket you have to write a proper documentation about how you did it and the key points of the protection. In case you just send the correct password, you will *not* have the right to claim the ticket which will be given at the next person that will send a proper documentation along with a valid password.

Our laboratory includes:

- Windows 7 x32 running on VirtualBox
- OllyDBG Debugger v1.10 (slightly modified)
- ODBGScript Plugin
- OllyAdvanced Plugin

In order to identify and explore the possible anti-reversing methodologies used in this challenge, all "additional" features incorporated with our tools were disabled prior to analysis.

This report is divided into two parts, first the analysis of %s%s%sGet_It_All%s%s%s executable module in *Chapter 2*, and then the analysis of %s%s%sAthcon_2011%s%s%s module in Chapter 3. Both chapters contain visual and textual information explaining the functionality of each module and its contribution to the challenge. Please note that each section must be read in the appearing order for a less confusing and clearer understanding. The final chapter of this report, Chapter 4 Appendix, contains the scripts and data structures used in this challenge. If something is unfamiliar to you or you believe that the scripts are poorly commented you will most likely find what you are looking within the two main chapters.

Chapter 2. Executable "%s%s%sGet_It_All%s%s%s.exe"

This chapter includes the analysis steps taken to identify the purpose of the *%s%s%sGet_It_All%s%s%s.exe* application which will be referred to as GIA for the remainder of this report. Before actually engaging in any actual analysis, we identified that the two files (excluding the .rar file) were actually PE executables. This led us to the conclusion that the GIA application was either a process loader or a dynamic linker.

2.1 Anti-Reversing Techniques

Our first step was to load the GIA in Olly. Unfortunately we received the error message in *Figure 1* with Olly unexpectedly breaking inside the Windows loader procedure.

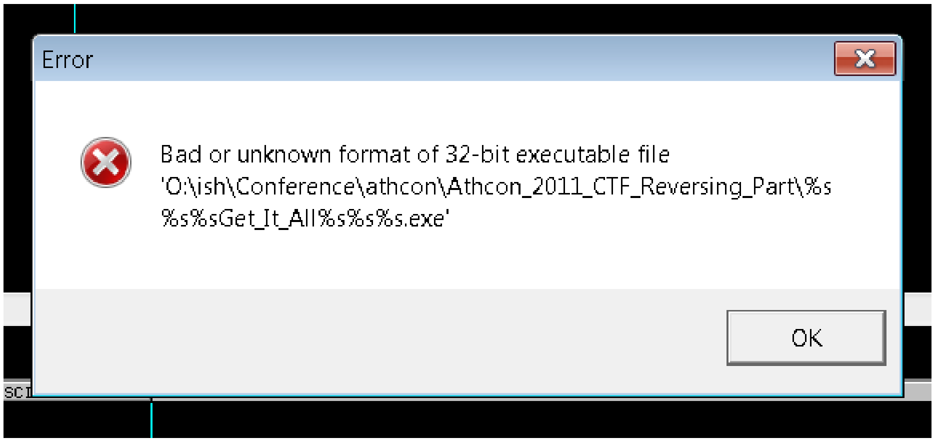


Figure 1 Initial Error Message

Such an error indicates possible tampering with the executable's PE header. This can also be verified by opening the Memory View window in Olly, as shown in *Figure* 2. There are a number of tools and libraries (eg the pefile project) that identify inconsistencies with the PE header structure. However, we decided to stick with Olly and verify the PE header manually.

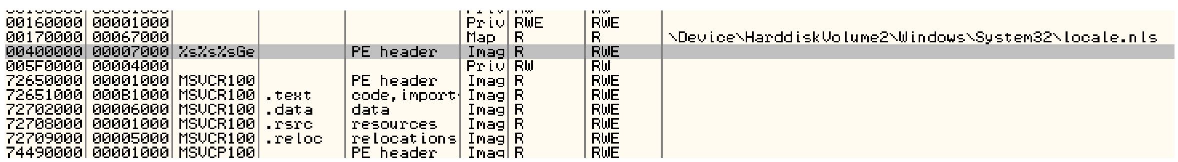


Figure 2 Memory View

During the header analysis we identified that the NumberOfRvaAndSizes element had an invalid value, as shown in *Figure 3*.

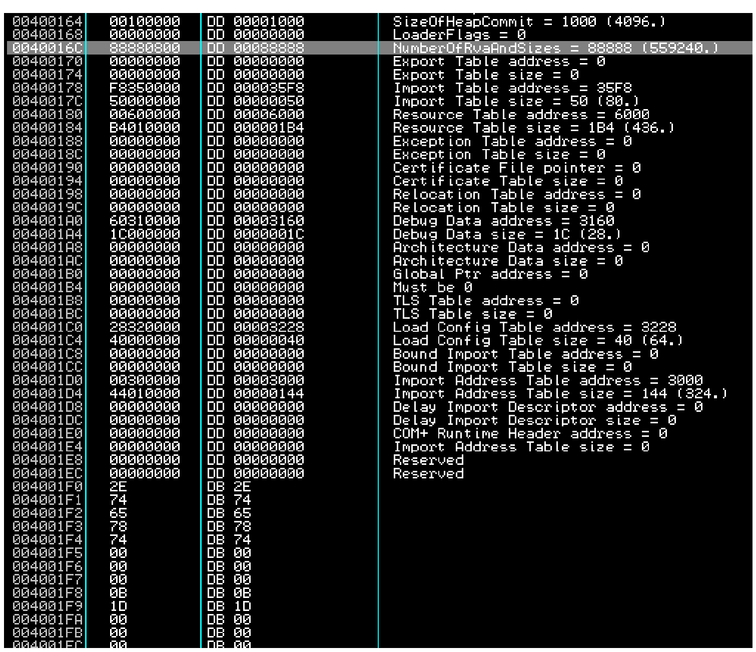


Figure 3 GIA PE Header

By default, this element displays the number of directory entries in the header and its value is equal to the defined `IMAGE_NUMBEROF_DIRECTORY_ENTRIES` of the programmer's linker. For this specific executable, PE Header version and almost for all executables the number of entries is 16 decimal and 0x10 hexadecimal. To bypass this protection all we have to do is load our favorite hex editor navigate to the offset 0x16C from the beginning of the file, since NumberOfRvaAndSizes Address = 0040016c and BaseAddress = 00400000 therefore,

$$0040016C - 00400000 = 16C$$

and patch the number 0x00000010 in the appropriate little-endian format as shown in *Figure 4*.

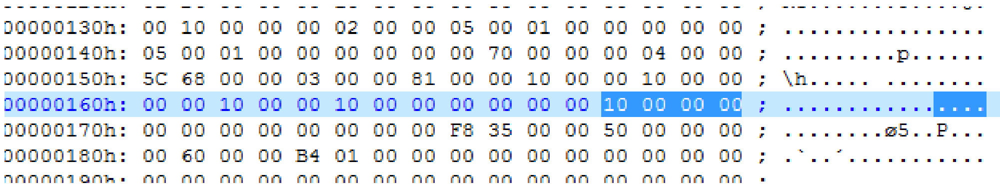


Figure 4 Patching NumberOfRvaAndSizes

Reloading the executable we came across another issue that has to do with a bug in Olly debugger. That is the confusion of the interpreter when it comes across executables that contain the "%s%s%s" pattern in their name. This is shown in *Figure 5*.

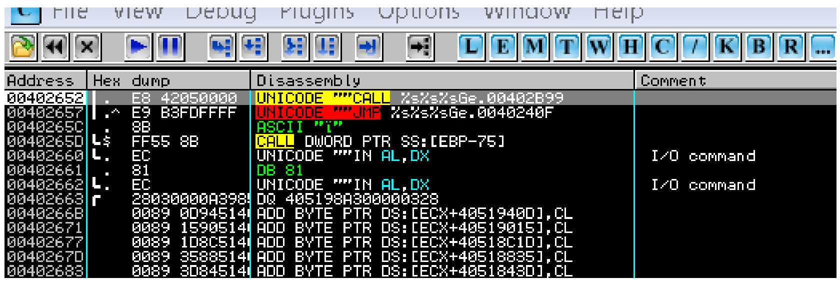


Figure 5 %s%s%s Bug

To overcome this issue you might consider renaming the executable. However, that could be proven to be a bad solution since the author of this challenge would most likely have a reason for naming the files in such a way. Other proper solutions include using a plugin (such as OllyAdvanced) to repair the bug, or even running an additional analysis of this module using Analysis > Analyse Code. We decided to use the OllyAdvanced plugin that patches both of the NumberOfRvaAndSizes and format string issue.

The next step was to identify the actual WinMain function of the executable, since the entry point at 00402652 is the entry point of the compiler and linker generated executable file. In order to locate it, you mostly have to rely on experience. Once you load the code and hit the Entry Point, experience will tell you that this program was compiled using Visual Studio, next (for VS) you follow the jump instruction until you find the appropriate call matching the number of arguments and argument types for this compiler. The CALL to the WinMain function is illustrated in *Figure 6.*

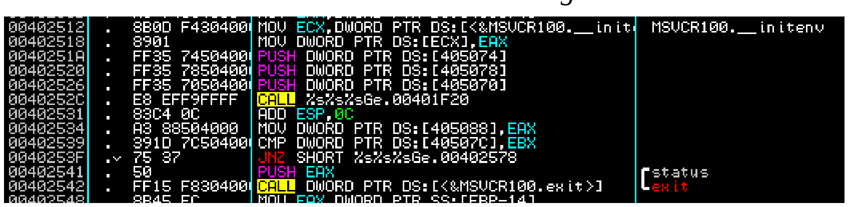


Figure 6 WinMain Called at 0040252C

Viewing the WinMain function we immediately understand that the program is constructed in such a way thus confusing the recursive traversal algorithm used by OllyDBG. This algorithm identifies control transfer instructions such as branches (JMP, JNE, JE, etc), procedure CALL and RET instructions during the sequential analysis of the code and once they are reached, the analysis continues at the address pointed by them. The resulting code inside the WinMain procedure is illustrated in *Figure 7*.

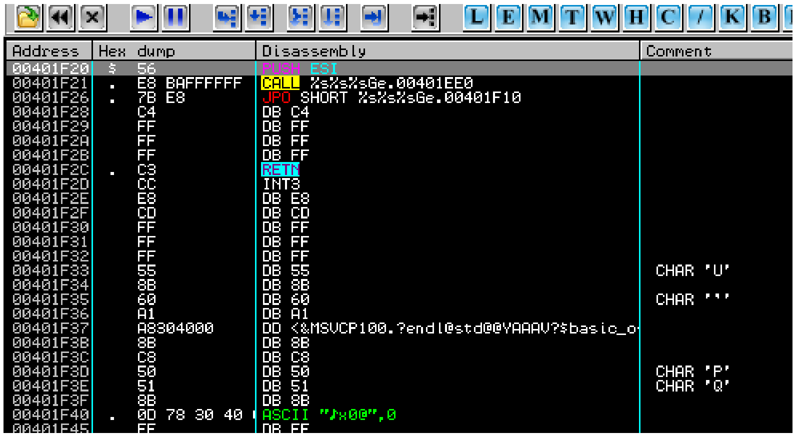


Figure 7 WinMain Procedure of GIA

Following the first call (see *Figure* 8) we come across a function that alters the dword element pointed by the stack register by adding the number 1 to it. That element is actually the return address to the caller procedure right after the CALL instruction. Therefore, by adding the number one to the return address the program is instructed to return one byte after the address specified by the CALL instruction from the caller procedure thus skip one byte.

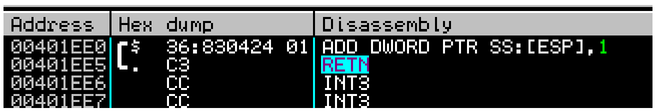


Figure 8 Obfuscation Function 1

This technique, other than the fact that it confuses OllyDBG's code analyzer, it also confuses it's tracing mechanisms since stepping over the call involves placing a breakpoint at the next instruction, which is skipped when returning from that function. Therefore, the breakpoint would never be hit and the program will continue execution. The usage of such functions can be proven quite annoying during the dynamic analysis process which requires single stepping through the code and identifying its purpose. In order to avoid accidentally stepping over such a call you would have to replace them with the appropriate NOP instructions. In this case, we'll have to remove the CALL instruction which is 5 bytes long, plus the number of bytes it skips which is 1 byte as shown in *Figure 8*.

Replacing the obfuscation function call and an additional byte with the 6 NOP instructions, another function call reveals itself (see *Figure 9*). Following the CALL (see *Figure 10*) we come across a similar function to the one seen in *Figure 8*.

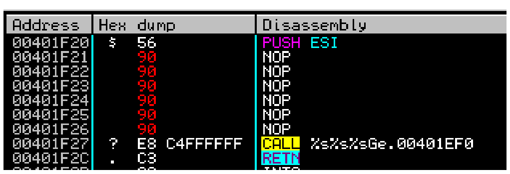


Figure 9 NOP-ing Obfuscation Function

This function CALLS the first function we've seen in *Figure 8*, skips the single byte after the CALL which is translated as a PUSHAD instruction, then adds the number two to the return address pointed by the stack register. Therefore, this function whenever called, skips the next two bytes from the return address specified by the CALL instruction. In order to remove this obfuscation function we'd have to replace 5 bytes for the CALL instruction and the 2 bytes skipped. Doing this reveals another obfuscation instruction that skips the next 3 bytes; replacing that reveals the actual code. However, such obfuscation calls exist in most parts of the program and can be proven to be extremely annoying during analysis.

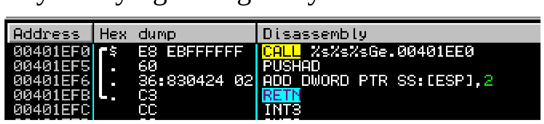


Figure 10 Inside Obfuscation Function 2

Coding your own obfuscation function is not that hard; all you have to do is instruct the compiler to skip the standard function header and footer instructions that deal with frame instantiation by declaring the function as “naked” and finally adding the necessary assembly instructions.

```
#define add2junks obfuscate_2(); __asm _emit 0x43 __asm _emit 0x60

void  __declspec(naked) obfuscate_2()
{
        __asm {
                add dword ptr [esp], 2
        }
}

void foo(void)
{
        //code omitted
        add2junks;
        //code omitted
}
```

Figure 11 Creating your own Junk Code-Obfuscation Function

Removing each of the obfuscation functions manually can be proven to be a time consuming process. Therefore, using the ODBGScript plugin we wrote a script (see Get_It_All De-obfuscation Script for ODBGScript in Appendix) that automatically de-obfuscates the GIA executable. We’ve identified the following obfuscation functions shown in *Table 1* and by retrieving the list of references to those functions we made the appropriate patches.

Function Address	Bytes Skipped	Actual Bytes Patched
00401EE0	1	5 + 1 = 6
00401EF0	2	5 + 2 = 7
00401F00	3	5 + 3 = 8
004010F0	4	5 + 4 = 9
00401F10	5	5 + 5 = 10

Table 1 Obfuscation Function Addresses

By running the script multiple times (about 5 times) until no references to the above functions are found we’ve managed to successfully remove the obfuscation protection of GIA.

2.2 GIA Analysis

Our initial analysis began by inspecting all references to external library functions and strings. However, the biggest proportion of anti-reversing techniques attempt to eliminate such information by dereferencing or dynamically loading them. In the case of GIA we can easily identify (after removing any obfuscation traces) a number of key API functions calls such as:

1. CreateProcessA
   http://msdn.microsoft.com/en-us/library/ms682425%28v=vs.85%29.aspx
2. OpenProcess
   http://msdn.microsoft.com/en-us/library/ms684320%28v=vs.85%29.aspx
3. WaitForDebugEvent
   http://msdn.microsoft.com/en-us/library/ms681423%28v=vs.85%29.aspx
4. GetThreadContext
   http://msdn.microsoft.com/en-us/library/ms679362%28v=vs.85%29.aspx
5. SetThreadContext
   http://msdn.microsoft.com/en-us/library/ms680632%28v=vs.85%29.aspx
6. WriteProcessMemory
   http://msdn.microsoft.com/en-us/library/ms681674%28v=vs.85%29.aspx
7. ContinueDebugEvent
   http://msdn.microsoft.com/en-us/library/ms679285%28v=vs.85%29.aspx

Reviewing the purpose of these functions one can easily deduct the functionality of GIA as a loader program that executes a process and acts as a debugger for it.

We began our dynamic analysis by reviewing the code step by step beginning from the WinMain function. The first instruction block we came across (illustrated in *Figure 12*) deals with printing out the message "Hello Reverser...so you think you can get it all?!?! (Athcon 2011 - CTF)".

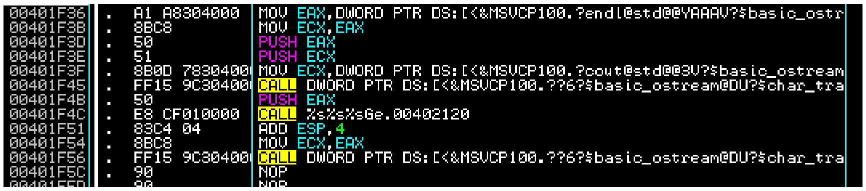


Figure 12 Print Message Block

The second instruction block (see *Figure 13*) instructs the system to pause the thread for 0x3E8 (=1000) milliseconds using the Sleep function.

Figure 13 Sleep(1000)

The next block (see *Figure* 14) is not that interesting since it appears to initialize a class object. We will later see that this object is essentially used for storing information such as the child's process handle, the thread's context retrieved using the GetThreadContext function and so on. For the moment all we need to know is that the size of this class/object is equal to 0x3A4 (=932) bytes. We note down the address of the allocated object in order to observe and understand its structure.

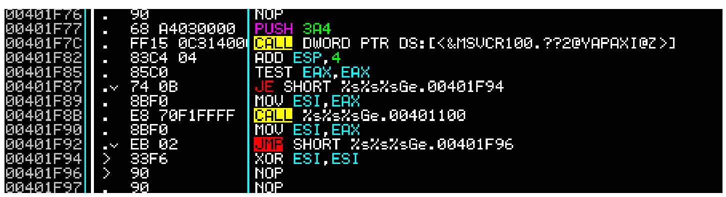


Figure 14 Object Allocation

The final code analysis block (see *Figure* 15) is our main point of interest for this program. It invokes two functions responsible for creating the process and debugging it, thus communicating the required information for the execution of %s%s%sAthcon_2011%s%s%s. The final pages of this chapter will explore that process in detail revealing some of the secrets behind the anti-reversing protections of this challenge.

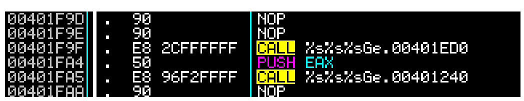


Figure 15 GIA Execution and Debugging Functions

Having noted the address allocated for the identified object, we follow the call located at 00401F9F which leads into a stub function (possibly a constructor). Following the second call we reach the entry point of the header-less function located at 004011E0 (shown in *Figure 16*). The first three instructions handle spilling the EBX register to the stack and moving the value in EAX into the EBX and ECX registers. EAX contains the pointer to the allocated object given by the code analysis block in *Figure 14* and assigned to during the execution of the stub function in 00401ED0.

There are two CALL instructions inside this function, the first is still unknown to us at the moment and appears to be calling a function within the current executable module. The second one appears to be calling an address, within the Import Address Table of the executable located at 00403000, which points to the OpenProcess function inside the kernel32.dll module.

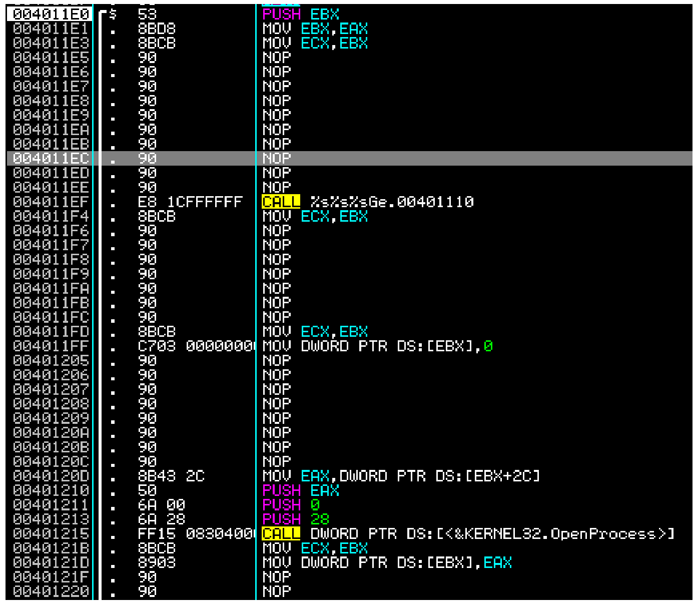


Figure 16 GIA CreateProcess & OpenProcess Function

Following the first CALL instruction we land in a function, protected by the /GS buffer security check cookie. *Figure 17* and *18* display the initialization and checking process of the cookie. The GS protection in this function reveals the existence of local buffer arguments or buffer manipulation intends, with the most likely scenario being a string initialization used when creating the %s%s%sAthcon_2011%s%s%s process.

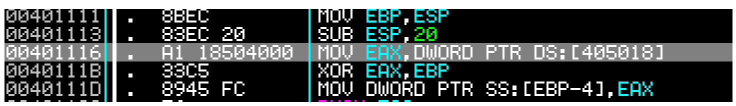


Figure 17 GS Cookie Initialization

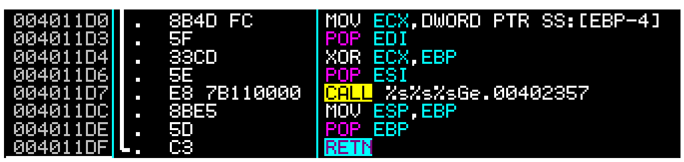


Figure 18 Stack Cookie Check

The first code analysis block we define (see *Figure 19*) loads to ESI the address of EBX+34 (EBX containing the base address of the allocated object and was assigned to in 004011E1, see *Figure 16*) and assigns the value 0x44 (=68) to it. Additionally, the same element pointed to by ESI has 0x44 of its bytes set to 0x00 using the memset function. This reveals that the element at unknown_object+34 is most likely a buffer of size 48 and possibly a DWORD array of 17 elements.

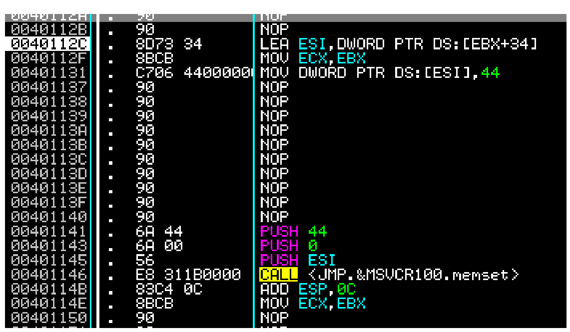


Figure 19 Function 00401110 First Code Analysis Block

The next code analysis block handles the rest of the initializations for the unknown object. EDI is set to the address of EBX+24 (base address of allocated object) and the four adjacent DWORD values are set to the value of EAX that was XORed at the beginning of this block. Additionally the DWORD value pointed to by EBX+60 as well as the WORD value pointed to be EBX+64 are set to 1.

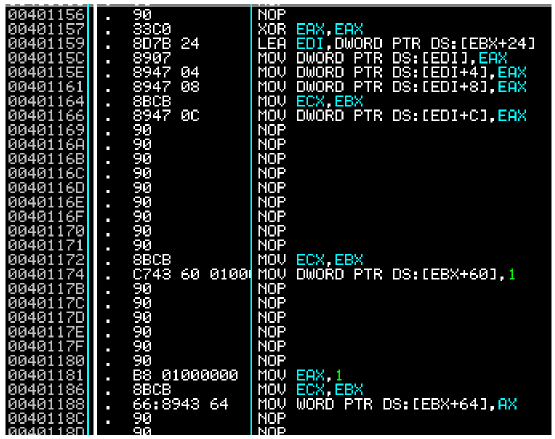


Figure 20 Function 00401110 Second Code Analysis Block

The next code analysis block (see *Figure 21*) contains a "string" class type initialization function (CALL 004010C0) which allocates a string structure and copies to it the "%s%s%sAthcon_2011%s%s%s" ASCII string. Next the CreateProcessA API function is called with the following arguments:

CreateProcessA(

lpApplicationName = "%s%s%sAthcon_2011%s%s%s",

lpCommandLine = NULL,

lpProcessAttributes = NULL,

lpThreadAttributes = NULL,,

bInheritHandles = TRUE,

dwCreationFlags = DEBUG_PROCESS | CREATE_SUSPENDED,

lpEnvironment = NULL,

lpCurrentDirectory = NULL,

lpStartupInfo = unknown_struct + 24

lpProcessInformation = uknown_struct + 34

);

Therefore we deduct that the element at *unknown_struct+34* is a STARTUPINFO structure and the element at *unknown_struct+34* is a PROCESS_INFORMATION structure. Finally, the next CALL to function 00401A30 doesn't appear to be doing anything worth mentioning. Exiting this function the GS stack cookie is checked for any possible stack based overflows, the stack is rebalanced and the function returns to its caller.

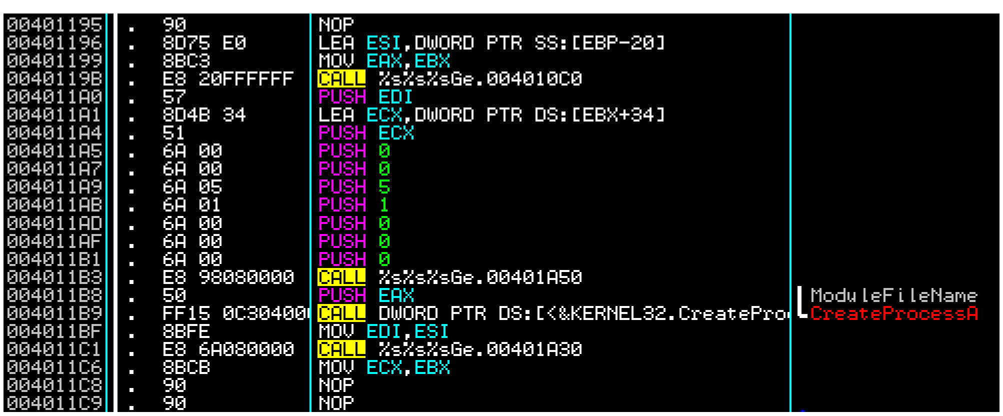


Figure 21 Function 00401110 Third Code Analysis Block

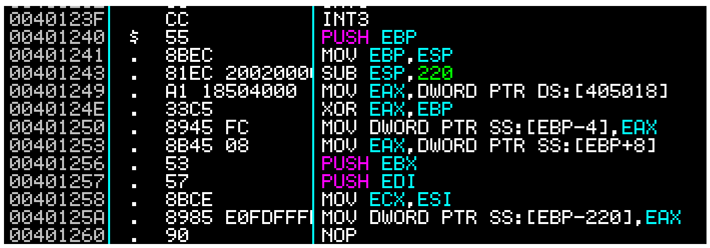
Revisiting *Figure 16* we can clearly identify the call to OpenProcess with the following arguments:

```
OpenProcess(
        dwDesiredAccess = VM_OPERATION | VM_WRITE,
        bInheritHandle = FALSE,
        dwProcessId = EBX+2C
);
```

Where EBX+2C contains the dwProcessId given by the PROCESS_INFORMATION structure set by the CreateProcessA function. Additionally, the result of OpenProcess (returned in EAX) is stored inside the first element of the *unknown_structure*. Finally, the process returns to the stub function which returns to WinMain preserving the OpenProcess result in EAX.

2.3 Function 00401240

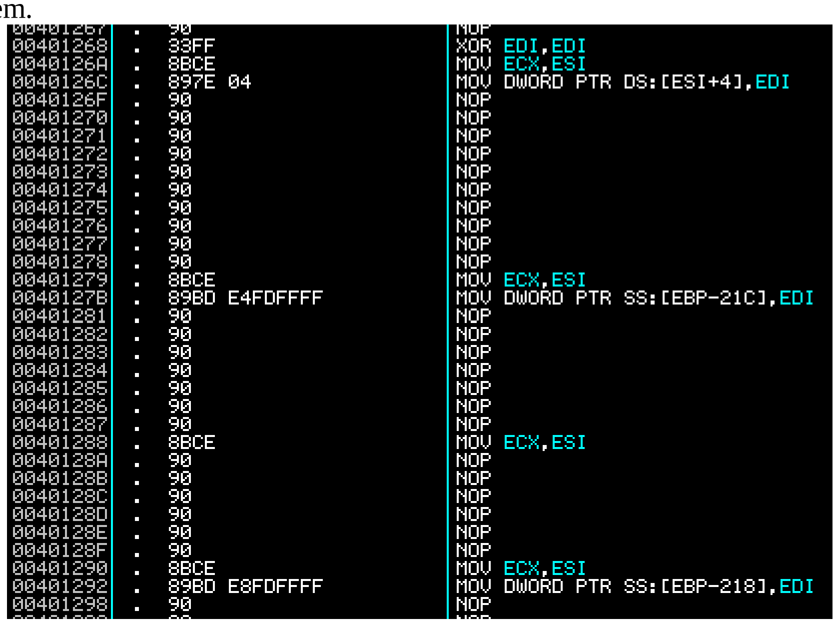
Next the value of EAX is PUSHed as an argument to the function located at 00401240 (see *Figure 15*) which is executed next. This function's header (see *Figure 22*) reveals three things; the first being the large number of allocated bytes inside the stack (SUB ESP, 0x220), second the initialization of a GS stack cookie and third the storage of the first argument (EBP+8) inside the first declared local variable (EBP-220).



```
0040123F      CC              INT3
00401240  $   55              PUSH EBP
00401241  .   8BEC            MOV EBP,ESP
00401243  .   81EC 2002000    SUB ESP,220
00401249  .   A1 18504000     MOV EAX,DWORD PTR DS:[405018]
0040124E  .   33C5            XOR EAX,EBP
00401250  .   8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00401253  .   8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
00401256  .   53              PUSH EBX
00401257  .   57              PUSH EDI
00401258  .   8BCE            MOV ECX,ESI
0040125A  .   8985 E0FDFFF    MOV DWORD PTR SS:[EBP-220],EAX
00401260  .   90              NOP
```

Figure 22 00401240 Header

The second code analysis block for this function initializes the second *unknown_struct+4* element (ESI+4) and two local variables at EBP-21C and EBP-218 to NULL by XORing EDI with itself and assigning it to them.



```
00401268  .   33FF            XOR EDI,EDI
0040126A  .   8BCE            MOV ECX,ESI
0040126C  .   897E 04         MOV DWORD PTR DS:[ESI+4],EDI
0040126F  .   90              NOP
00401270  .   90              NOP
00401271  .   90              NOP
00401272  .   90              NOP
00401273  .   90              NOP
00401274  .   90              NOP
00401275  .   90              NOP
00401276  .   90              NOP
00401277  .   90              NOP
00401278  .   90              NOP
00401279  .   8BCE            MOV ECX,ESI
0040127B  .   89BD E4FDFFFF   MOV DWORD PTR SS:[EBP-21C],EDI
00401281  .   90              NOP
00401282  .   90              NOP
00401283  .   90              NOP
00401284  .   90              NOP
00401285  .   90              NOP
00401286  .   90              NOP
00401287  .   90              NOP
00401288  .   8BCE            MOV ECX,ESI
0040128A  .   90              NOP
0040128B  .   90              NOP
0040128C  .   90              NOP
0040128D  .   90              NOP
0040128E  .   90              NOP
0040128F  .   90              NOP
00401290  .   8BCE            MOV ECX,ESI
00401292  .   89BD E8FDFFFF   MOV DWORD PTR SS:[EBP-218],EDI
00401298  .   90              NOP
```

Figure 23 00401240 Second Code Analysis Block

Next, the function initializes a set of addresses (see *Figure* 24) and numbers (see *Figure 25*) by storing them inside the pre-allocated stack space for local variables. Tracing your debuggee up to the point located right after the initialization allows you to view a nicely structured memory dump of the initialized variables.

The second analysis block contains a number of debugging API calls. First, the thread id given in *unknown_struct+28* (ESI+28) which contains the handle hThread retrieved by the CreateProcessA PROCESS_INFORMATION structure is send to the ResumeThread API function thus resuming the created process. Next, the WaitForDebugEvent API is called with the *unknown_struct+78* (ESI+78) as a pointer to a DEBUG_EVENT structure and -1 representing the INFINITE waiting time thus blocking the current GIA thread until a debug event occurs.

```
004012B3  .  C785 F4FEFFFF 8B2E4000  MOV DWORD PTR SS:[EBP-10C],%s%s%sGe.00402E8B
004012BD  .  C785 F8FEFFFF B82E4000  MOV DWORD PTR SS:[EBP-108],%s%s%sGe.00402EB8
004012C7  .  C785 FCFEFFFF F92E4000  MOV DWORD PTR SS:[EBP-104],%s%s%sGe.00402EF9
004012D1  .  C785 00FFFFFF 3B2F4000  MOV DWORD PTR SS:[EBP-100],%s%s%sGe.00402F3B
004012DB  .  C785 04FFFFFF 912F4000  MOV DWORD PTR SS:[EBP-FC],%s%s%sGe.00402F91
004012E5  .  C785 08FFFFFF 07334000  MOV DWORD PTR SS:[EBP-F8],%s%s%sGe.00403307
004012EF  .  C785 0CFFFFFF 76334000  MOV DWORD PTR SS:[EBP-F4],%s%s%sGe.00403376
004012F9  .  C785 10FFFFFF AA324000  MOV DWORD PTR SS:[EBP-F0],%s%s%sGe.004032AA
00401303  .  C785 14FFFFFF 3B334000  MOV DWORD PTR SS:[EBP-EC],%s%s%sGe.0040333B
0040130D  .  C785 18FFFFFF C22E4000  MOV DWORD PTR SS:[EBP-E8],%s%s%sGe.00402EC2
00401317  .  C785 1CFFFFFF 082F4000  MOV DWORD PTR SS:[EBP-E4],%s%s%sGe.00402F08
00401321  .  C785 20FFFFFF 4A2F4000  MOV DWORD PTR SS:[EBP-E0],%s%s%sGe.00402F4A
0040132B  .  C785 24FFFFFF A02F4000  MOV DWORD PTR SS:[EBP-DC],%s%s%sGe.00402FA0
00401335  .  C785 28FFFFFF B52F4000  MOV DWORD PTR SS:[EBP-D8],%s%s%sGe.00402FB5
0040133F  .  C785 2CFFFFFF B0304000  MOV DWORD PTR SS:[EBP-D4],%s%s%sGe.004030B0
00401349  .  C785 30FFFFFF EA304000  MOV DWORD PTR SS:[EBP-D0],%s%s%sGe.004030EA
00401353  .  C785 34FFFFFF 19314000  MOV DWORD PTR SS:[EBP-CC],%s%s%sGe.00403119
0040135D  .  C785 38FFFFFF 5C314000  MOV DWORD PTR SS:[EBP-C8],%s%s%sGe.0040315C
00401367  .  C785 3CFFFFFF D6324000  MOV DWORD PTR SS:[EBP-C4],%s%s%sGe.004032D6
00401371  .  C785 40FFFFFF 13334000  MOV DWORD PTR SS:[EBP-C0],%s%s%sGe.00403313
0040137B  .  C785 44FFFFFF 53334000  MOV DWORD PTR SS:[EBP-BC],%s%s%sGe.00403353
```

Figure 24 00401240 Init Addresses

```
004014F5  .  C785 ECFDFFFF 95050000  MOV DWORD PTR SS:[EBP-214],595
004014FF  .  C785 F0FDFFFF B8050000  MOV DWORD PTR SS:[EBP-210],5B8
00401509  .  C785 F4FDFFFF 77050000  MOV DWORD PTR SS:[EBP-20C],577
00401513  .  C785 F8FDFFFF 35050000  MOV DWORD PTR SS:[EBP-208],535
0040151D  .  C785 FCFDFFFF DF040000  MOV DWORD PTR SS:[EBP-204],4DF
00401527  .  C785 00FEFFFF 19020000  MOV DWORD PTR SS:[EBP-200],219
00401531  .  C785 04FEFFFF 3A020000  MOV DWORD PTR SS:[EBP-1FC],23A
0040153B  .  C785 08FEFFFF 76030000  MOV DWORD PTR SS:[EBP-1F8],376
00401545  .  C785 0CFEFFFF 85030000  MOV DWORD PTR SS:[EBP-1F4],385
0040154F  .  C785 10FEFFFF BE080000  MOV DWORD PTR SS:[EBP-1F0],8BE
00401559  .  C785 14FEFFFF 78080000  MOV DWORD PTR SS:[EBP-1EC],878
00401563  .  C785 18FEFFFF 36080000  MOV DWORD PTR SS:[EBP-1E8],836
0040156D  .  C785 1CFEFFFF E0070000  MOV DWORD PTR SS:[EBP-1E4],7E0
00401577  .  C785 20FEFFFF CB070000  MOV DWORD PTR SS:[EBP-1E0],7CB
00401581  .  C785 24FEFFFF D0060000  MOV DWORD PTR SS:[EBP-1DC],6D0
0040158B  .  C785 28FEFFFF 96060000  MOV DWORD PTR SS:[EBP-1D8],696
00401595  .  C785 2CFEFFFF 67060000  MOV DWORD PTR SS:[EBP-1D4],667
0040159F  .  C785 30FEFFFF 24060000  MOV DWORD PTR SS:[EBP-1D0],624
004015A9  .  C785 34FEFFFF AA040000  MOV DWORD PTR SS:[EBP-1CC],4AA
004015B3  .  C785 38FEFFFF 6D040000  MOV DWORD PTR SS:[EBP-1C8],46D
004015BD  .  C785 3CFEFFFF 2D040000  MOV DWORD PTR SS:[EBP-1C4],42D
004015C7  .  C785 40FEFFFF 00040000  MOV DWORD PTR SS:[EBP-1C0],400
004015D1  .  C785 44FEFFFF 13090000  MOV DWORD PTR SS:[EBP-1BC],913
004015DB  .  C785 48FEFFFF CD080000  MOV DWORD PTR SS:[EBP-1B8],8CD
004015E5  .  C785 4CFEFFFF 8B080000  MOV DWORD PTR SS:[EBP-1B4],88B
```

Figure 25 00401240 Init Numbers

```
00401793  .  8B4E 28          MOV ECX,DWORD PTR DS:[ESI+28]
00401796  .  51               PUSH ECX
00401797  .  FF15 20304000    CALL DWORD PTR DS:[<&KERNEL32.ResumeThread>]
0040179D  .  8BCE             MOV ECX,ESI
0040179F  .  90               NOP
004017A0  .  90               NOP
004017A1  .  90               NOP
004017A2  .  90               NOP
004017A3  .  90               NOP
004017A4  .  90               NOP
004017A5  >  8BCE             MOV ECX,ESI
004017A7  .  90               NOP
004017A8  .  90               NOP
004017A9  .  90               NOP
004017AA  .  90               NOP
004017AB  .  90               NOP
004017AC  .  90               NOP
004017AD  .  90               NOP
004017AE  .  90               NOP
004017AF  .  90               NOP
004017B0  .  90               NOP
004017B1  .  6A FF            PUSH -1
004017B3  .  8D7E 78          LEA EDI,DWORD PTR DS:[ESI+78]
004017B6  .  57               PUSH EDI
004017B7  .  FF15 18304000    CALL DWORD PTR DS:[<&KERNEL32.WaitForDebugEvent>]
004017BD  .  8BCE             MOV ECX,ESI
004017BF  .  90               NOP
004017C0  .  90               NOP
004017C1  .  90               NOP
004017C2  .  90               NOP
004017C3  .  90               NOP
004017C4  .  90               NOP
004017C5  .  90               NOP
004017C6  .  833F 05          CMP DWORD PTR DS:[EDI],5
004017C9  .  8BCE             MOV ECX,ESI
004017CB  .v 0F84 C5010000    JE %s%s%sGe.00401996
004017D1  .  90               NOP
004017D2  .  90               NOP
004017D3  .  90               NOP
004017D4  .  90               NOP
004017D5  .  90               NOP
004017D6  .  90               NOP
004017D7  .  90               NOP
004017D8  .  90               NOP
004017D9  .  90               NOP
004017DA  .  8B96 80000000    MOV EDX,DWORD PTR DS:[ESI+80]
004017E0  .  52               PUSH EDX
004017E1  .  6A 00            PUSH 0
004017E3  .  6A 5A            PUSH 5A
004017E5  .  FF15 10304000    CALL DWORD PTR DS:[<&KERNEL32.OpenThread>]
004017EB  .  8BCE             MOV ECX,ESI
004017ED  .  8946 04          MOV DWORD PTR DS:[ESI+4],EAX
```

Figure 26 00401240 Second Code Analysis Block

Next, the DWORD value pointed to by EDI which is the first element dwDebugEventCode of the DEBUG_EVENT structure is checked against number 5 (EXIT_PROCESS_DEBUG_EVENT) and if equal the code jumps to the location 00401996 which exits the function.

The next call to OpenThread receives the dwThreadId (ESI+80) located inside *unknown_struct+80* (after the call to WaitForDebugEvent), with a FALSE (0) bInheritHandle and THREAD_SUSPEND_RESUME, THREAD_GET_CONTEXT, THREAD_SET_CONTEXT and THREAD_QUERY_INFORMATION flags set in dwDesiredAccess. The result is stored inside *unknown_struct+4*.

Next (see *Figure 27*), the element at location *unknown_struct+84* (ESI+84), which resides within the DEBUG_EVENT structure and contains the first element of the union "u" which is an EXCEPTION_RECORD structure inside a EXCEPTION_DEBUG_INFO structure with element name ExceptionCode, is checked with the value 80000003 that stands for the EXCEPTION_BREAKPOINT definition. If the two checking values are not equal then the code branches at location 0040195F where the DBG_EXCEPTION_NOT_HANDLED message is send back to the system.

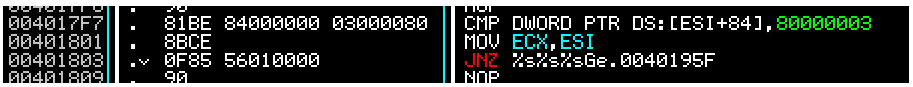


Figure 27 00401240 EXCEPTION_BREAKPOINT check

If the exception code is equal to a breakpoint then the program enters a looping state (see *Figure 28*) in which the value inside *unknown_struct+90* (ESI+90) that contains the ExceptionAddress pointer is checked against the local DWORD address elements beginning from EBP-10C and ending at EBP-8. You can clearly see that EDI acts as a counter and an array index pointer calculating the location of each element by multiplying the number of DWORDs (since each element is of DWORD length) to the address of the first element. (see 00401827 CMP EAX, DWORD PTR SS:[EBP+**EDI*4-**10C]). If no such item is found then the program branches to location 00401933 where the DBG_CONTINUE message is send back to the system. This information reveals that the addresses in *Figure 24* are actually expected exception addresses in the "%s%s%sAthcon_2011%s%s%s" module.

Figure 28 00401240 Loop

When a matching address is found the JE 00401842 instruction branches the execution flow onto the next code analysis block (see *Figure 29*). There, the address of *unknown_struct+D8* is moved into EBX and the element pointed to by that address is set to 0x1003F. Next, the GetThreadContext API function is called with EBX and ECX as its arguments. ECX is set to *unknown_struct+4* (ESI+4), which is already known to us from *Figure 26*. In addition, we've just identified EBX as an element in *unknown_struct+D8* and given the definition of GetThreadContext that memory location contains a CONTEXT structure. Therefore, the value 0x1003F is a set of flags (defined as `CONTEXT_ALL`) assigned to the first element of CONTEXT named ContextFlags. Note that this program was compiled under the i386 architecture therefore, all references to CONTEXT and flags are intended for that architecture.

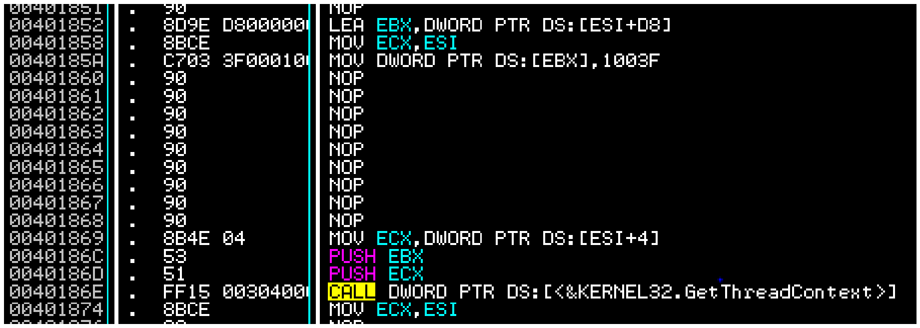


Figure 29 00401240 GetThreadContext Code Analysis Block

Figure 30 illustrates the next steps taken by the program right after the CONTEXT structure is populated by the GetThreadContext function. The value in *unknown_struct+19C* (ESI+19C) which is the element at location 0x19C - 0xD8 = 0xC4 from the beginning of the CONTEXT structure is decremented by 0x04. Reviewing the definition of the structure, we identify that this element is the stack pointer (ESP) of the debuggee. Next, the local table address element pointed to by the EDI index (which hasn't changed since *Figure 28*) and contains the exception address, is assigned to the element in *unknown_struct+190* (ESI+190) which points to the instruction pointer (EIP) in the CONTEXT struct. Next, the same address (ESI+190) is moved into EAX, incremented by 5 and assigned to local variable located at EBP-218. Finally, the same index number in EDI is used to retrieve a value from the second table (illustrated in *Figure 25*) which is then added to *unknown_struct+190* (ESI+190) containing the EIP register of the debuggee.

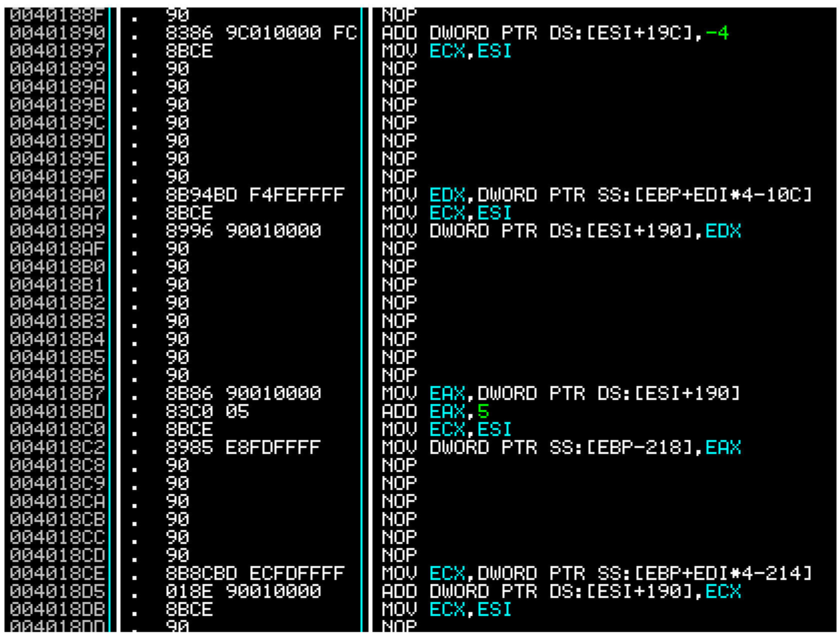


Figure 30 00401240 Managing CONTEXT flags

The next code analysis block contains calls to two API functions. First, the SetThreadContext function is called with EDX, containing the thread's handle (assigned to from ESI+4) and EBX, containing the CONTEXT structure address, therefore effectively changing the altered registers. Next, the WriteProcessMemory function is called, with the 5th argument being the address of a local variable in EBP-21C, which by definition of WriteProcessMemory is a SIZE_T lpNumberOfBytesWritten value; the 4th argument being the number of bytes to write, which is set to 4; the 3rd argument being the local variable/buffer at EBP-218, which is assigned to in *Figure 30* and is the exception address plus 5 bytes; the second argument being the address in *unknown_struct+19C*, which is the altered stack pointer (ESP) of the debuggee; finally, the first argument in EAX is loaded from the local variable in EBP-220 which is assigned to in *Figure 22* from the first and only argument of this function and contains the process handle.

```
004018E3 | .   90              | NOP
004018E4 | .   8B56 04         | MOV EDX,DWORD PTR DS:[ESI+4]
004018E7 | .   53              | PUSH EBX
004018E8 | .   52              | PUSH EDX
004018E9 | .   FF15 04304000   | CALL DWORD PTR DS:[<&KERNEL32.SetThreadContext>]
004018EF | .   8BCE            | MOV ECX,ESI
004018F1 | .   90              | NOP
004018F2 | .   90              | NOP
004018F3 | .   90              | NOP
004018F4 | .   90              | NOP
004018F5 | .   90              | NOP
004018F6 | .   90              | NOP
004018F7 | .   90              | NOP
004018F8 | .   90              | NOP
004018F9 | .   90              | NOP
004018FA | .   90              | NOP
004018FB | .   8B96 9C010000   | MOV EDX,DWORD PTR DS:[ESI+19C]
00401901 | .   8D85 E4FDFFFF   | LEA EAX,DWORD PTR SS:[EBP-21C]
00401907 | .   50              | PUSH EAX
00401908 | .   8B85 E0FDFFFF   | MOV EAX,DWORD PTR SS:[EBP-220]
0040190E | .   6A 04           | PUSH 4
00401910 | .   8D8D E8FDFFFF   | LEA ECX,DWORD PTR SS:[EBP-218]
00401916 | .   51              | PUSH ECX
00401917 | .   52              | PUSH EDX
00401918 | .   50              | PUSH EAX
00401919 | .   FF15 1C304000   | CALL DWORD PTR DS:[<&KERNEL32.WriteProcessMemory>]
0040191F | .   85C0            | TEST EAX,EAX
00401921 | .v  74 08           | JE SHORT %s%s%sGe.0040192B
```

Figure 31 00401240 SetThreadContext and WriteProcessMemory

Finally, the next code analysis block (see *Figure 32*) continues the debugged process with a DBG_CONTINUE (0x10002) status and carries on the debug loop.

```
0040193E | .   8B8E 80000000   | MOV ECX,DWORD PTR DS:[ESI+80]
00401944 | .   8B56 7C         | MOV EDX,DWORD PTR DS:[ESI+7C]
00401947 | .   68 02000100     | PUSH 10002
0040194C | .   51              | PUSH ECX
0040194D | .   52              | PUSH EDX
0040194E | .   FF15 14304000   | CALL DWORD PTR DS:[<&KERNEL32.ContinueDebugEvent>]
00401954 | .   8BCE            | MOV ECX,ESI
```

Figure 32 00401240 ContinueDebugEvent

2.4 Conclusions

Reviewing GIA module's behavior one can clearly validate some of the assumptions made in previous sections. The Get/SetThreadContext process is essentially "emulating" the state of a program right after the execution of a CALL instruction. Just like the return address is calculated as being the immediate address after the call instruction, then PUSHed inside the stack as a piece of procedural linking information within the newly constructed stack frame. So does GIA subtracts from the stack pointer a value equal to the stack width (namely 4 bytes in x86) and assigns to the element pointed by it a value equal to the exception address plus 5 bytes (the return address). Therefore, the 5 bytes could stand for a CALL instruction that was replaced or removed intentionally to render the debuggee useless without the use of GIA module.

Chapter 3. Executable %s%s%sAthcon_2011%s%s%s

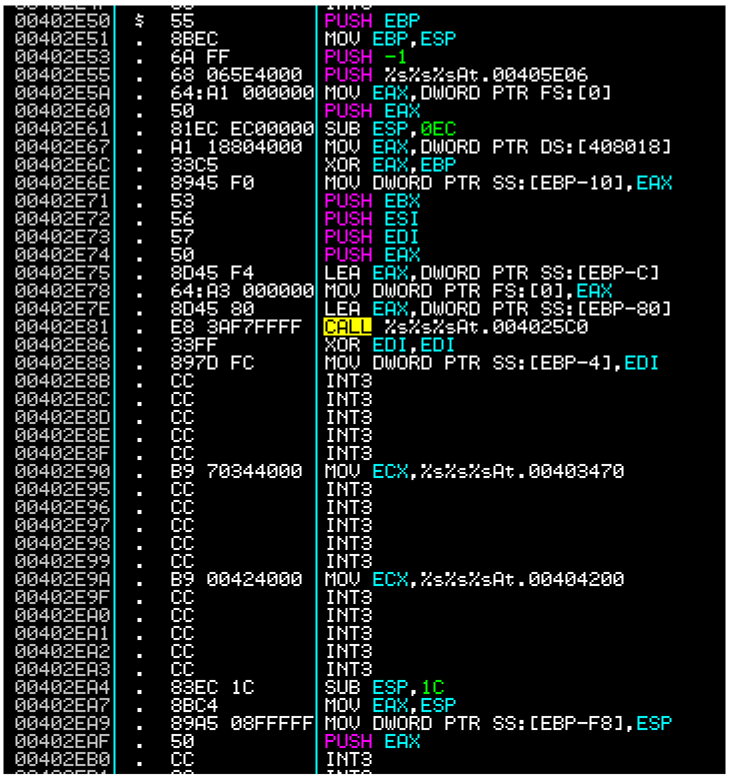
This chapter covers the dynamic and static analysis of "%s%s%sAthcon_2011%s%s%s" module. For the remainder of this report we will refer to it as "Athcon module".

Loading up the module in OllyDBG we come across the same PE anti-reversing techniques (NumberOfRvaAndSizes and format string) as GIA. Refer to previous section 2.1 which describes how to bypass them. In addition, the WinMain function is partially encrypted from address 00402F96 until it's exit.

3.1 INT3 Exceptions

Navigating to the WinMain function (located at 00402E50) of the application we observe a number of INT3 instructions inlined along with the code. A simple analysis can reveal the following:

1. All INT3 inline blocks have size equal to 5 bytes.
2. The first INT3 instruction of each block is located in an address contained inside the array initialized by GIA (see *Figure 24*).



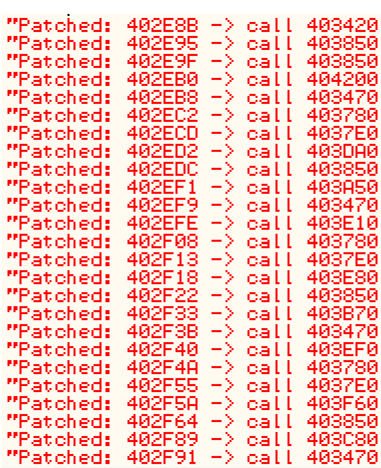
```
00402E50  $  55             PUSH EBP
00402E51  .  8BEC           MOV EBP,ESP
00402E53  .  6A FF          PUSH -1
00402E55  .  68 065E4000    PUSH %s%s%sAt.00405E06
00402E5A  .  64:A1 000000   MOV EAX,DWORD PTR FS:[0]
00402E60  .  50             PUSH EAX
00402E61  .  81EC EC00000   SUB ESP,0EC
00402E67  .  A1 18804000    MOV EAX,DWORD PTR DS:[408018]
00402E6C  .  33C5           XOR EAX,EBP
00402E6E  .  8945 F0        MOV DWORD PTR SS:[EBP-10],EAX
00402E71  .  53             PUSH EBX
00402E72  .  56             PUSH ESI
00402E73  .  57             PUSH EDI
00402E74  .  50             PUSH EAX
00402E75  .  8D45 F4        LEA EAX,DWORD PTR SS:[EBP-C]
00402E78  .  64:A3 000000   MOV DWORD PTR FS:[0],EAX
00402E7E  .  8D45 80        LEA EAX,DWORD PTR SS:[EBP-80]
00402E81  .  E8 3AF7FFFF    CALL %s%s%sAt.004025C0
00402E86  .  33FF           XOR EDI,EDI
00402E88  .  897D FC        MOV DWORD PTR SS:[EBP-4],EDI
00402E8B  .  CC             INT3
00402E8C  .  CC             INT3
00402E8D  .  CC             INT3
00402E8E  .  CC             INT3
00402E8F  .  CC             INT3
00402E90  .  B9 70344000    MOV ECX,%s%s%sAt.00403470
00402E95  .  CC             INT3
00402E96  .  CC             INT3
00402E97  .  CC             INT3
00402E98  .  CC             INT3
00402E99  .  CC             INT3
00402E9A  .  B9 00424000    MOV ECX,%s%s%sAt.00404200
00402E9F  .  CC             INT3
00402EA0  .  CC             INT3
00402EA1  .  CC             INT3
00402EA2  .  CC             INT3
00402EA3  .  CC             INT3
00402EA4  .  83EC 1C        SUB ESP,1C
00402EA7  .  8BC4           MOV EAX,ESP
00402EA9  .  89A5 08FFFFF   MOV DWORD PTR SS:[EBP-F8],ESP
00402EAF  .  50             PUSH EAX
00402EB0  .  CC             INT3
```

Figure 33 Athcon WinMain

Since there were no apparent indications that the INT3 blocks served any purpose other than signaling the GIA module we decided to calculate the emulated CALL addresses and manually replace each block with them. Initially, we reversed engineered from the debug loop function, the necessary information to calculate programmatically each address (see Appendix 4.3).

However, we eventually decided to use ODBGScript to reduce the amount of time required to calculate and patch. The resulting script initializes the two tables from *Figure* 24 and *Figure* 25, takes the address currently pointed to by EIP, locates INT3 blocks of size 5 bytes, looks up the table and finally patches the required CALL instruction. For the purpose of simplicity and genericality the script pops up a message asking for the user's permission before continuing, whenever a block doesn't match an address inside the address table. The script's source code is located in Appendix 4.4. In order to execute the script, navigate to WinMain and run it until the Yes/No message box appears. This should sufficient enough to patch the first batch of INT3 blocks.

Once the script finishes executing the following messages should appear in OllyDBG's Log window. Each one denotes the address of the patch and the call instruction that replaced the INT3 block.



```
"Patched: 402E8B -> call 403420
"Patched: 402E95 -> call 403850
"Patched: 402E9F -> call 403850
"Patched: 402EB0 -> call 404200
"Patched: 402EB8 -> call 403470
"Patched: 402EC2 -> call 403780
"Patched: 402ECD -> call 4037E0
"Patched: 402ED2 -> call 403DA0
"Patched: 402EDC -> call 403850
"Patched: 402EF1 -> call 403A50
"Patched: 402EF9 -> call 403470
"Patched: 402EFE -> call 403E10
"Patched: 402F08 -> call 403780
"Patched: 402F13 -> call 4037E0
"Patched: 402F18 -> call 403E80
"Patched: 402F22 -> call 403850
"Patched: 402F33 -> call 403B70
"Patched: 402F3B -> call 403470
"Patched: 402F40 -> call 403EF0
"Patched: 402F4A -> call 403780
"Patched: 402F55 -> call 4037E0
"Patched: 402F5A -> call 403F60
"Patched: 402F64 -> call 403850
"Patched: 402F89 -> call 403C80
"Patched: 402F91 -> call 403470
```

Figure 34 ODBGScript Log

3.2 Analysis of WinMain Unencrypted Body

The following analysis is rather slightly abstract in comparison to the one reported for GIA in previous sections. Most of the internal and somewhat irrelevant structures and functions are not covered in detail since they are not specifically required for the completion of this challenge. A number of function calls have been labeled according to their functionality and a short description is given whenever they come up.

The first code analysis block in *Figure 35* calls a function (00402E81) labeled by us as "VirtualProtect" since inside it the VirtualProtect API is invoked with the following arguments:

VirtualProtect(

lpAddress = 00401000 (beginning of the code section),

dwSize = 0x4000,

flNewProtect = 0x40 (= PAGE_EXECUTE_READWRITE),

EBP-24 (A local variable discarded on return)

);

The reason behind this API call is to allow the program to self-alter or polymorph it's code. In short, polymorphism is the intended incorporation of encryption and/or code manipulation within a program, allowing it to decrypt and/or alter its own code dynamically while executing in memory.

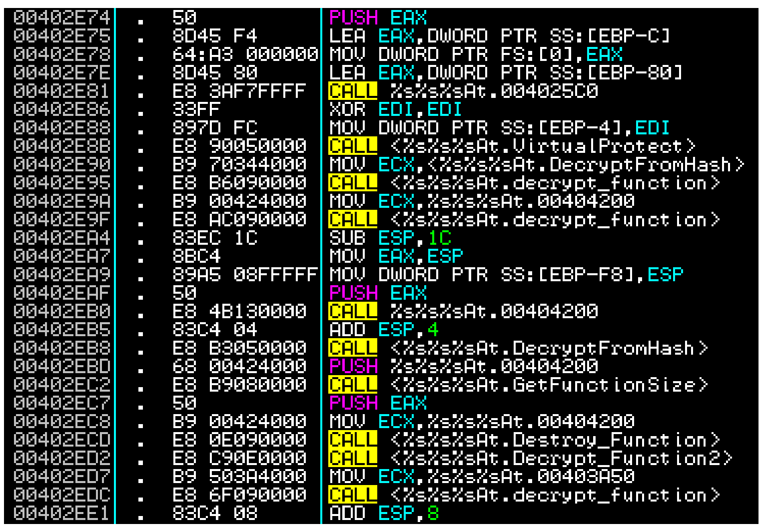


Figure 35 WinMain First Code Analysis Block

The CALL instructions labeled as "decrypt_function" take as an argument the address of an encrypted function in ECX and applies a byte by byte XOR decryption loop (see *Figure* 36) with a key equal to the high 8bit byte of AH (assigned to at 00403876 by EDI which at that point holds the function's address). For example, if the function's address is 0040**12**34 then the decryption key is 0x12.

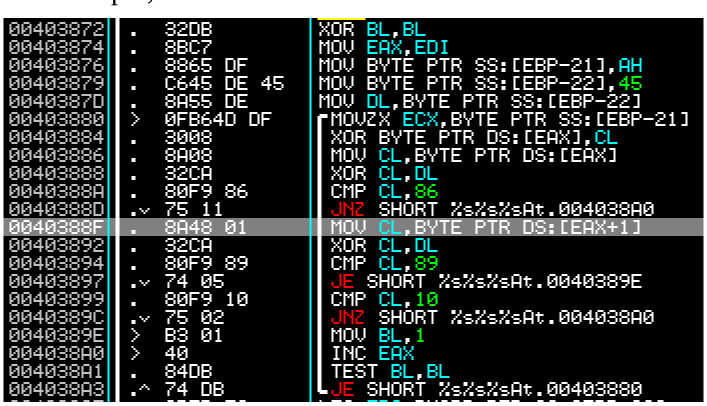


Figure 36 00403850 Decrypt_Function Function

The loop ends when BL is not equal to 0, that is when the instruction at 0040389E is executed. This is possible only if the execution flow follows blocks 0040388F and/or 00403899. To do so the byte pointed to by EAX (currently decrypted byte) must be equal to 0x86 when XORed with the number 0x45. The byte satisfying this equation is 0xC3 (RETN instruction mnemonic). Next, if the byte pointed to by EAX+1 is equal to 0x89 when XORed with 0x45 (therefore equal to 0xCC) then jump to location 0040389E setting the byte in BL to 1.

Next, the decrypted function (00404200) is called with a stack address as an argument. Its purpose is to retrieve the name of the application, hash it using the SHA-512 algorithm, allocate a buffer, convert the hash into an ASCII string inside that buffer and return it. The buffer is then used in function 00403470 (called at 00402EB8) to bitwise XOR the rest of the WinMain function (which is currently encrypted). However, this is just the first of many chained decryption functions that make use of hash values to decrypt the remaining instructions of WinMain, making this algorithm the main protection used in Athcon_2011. The first valid hash signature is:

"59d9dfa6e92f95f281e4bbb7ec6b15bd495d7e12bc26fd3e9ee281856781b4100ce596eb4f294ef1e00735e46c3e1bf43b7a6110332652d08eda8da6523e0041"

Another interesting function is the one called at address 00402EC2 (in WinMain) and is labeled "GetFunctionSize". This function works in a similar way with decryption_function, it takes a single argument in the stack which is the pointer to a function and returns its size in bytes.

Next, the function located at 00404200 is "destroyed" by the function located at 004037E0 and labeled "Destroy_Function". The destroy function takes two arguments, first the function's address in ECX and second the function's size in bytes as a PUSHed argument. Inside, the rand function from msvcr100.dll is called to determine the bytes to replace the function's instructions with.

The function labeled Decrypt_Function2 and illustrated in *Figure 37*, decrypts a single function pointed to by EAX (see address 00403DE3). The pointer to that function is set in EAX from EBP-28 which in turn is set at 00403DCD. At that point EAX holds the resulting value from XOR EAX, ESI. EAX is equal to 0x23BC (set at 00403DC0) and ESI holds the address 004067FC (set at 00403DB3). The resulting function's address is 00404440; we labeled it "Parenting" for obvious reasons that will be revealed later. The XOR key currently residing in [EBP-21] is set to the low EAX value (AL) at 00403DD1; that value in turn is being set (before the PUSHAD/POPAD instructions) to the address of EBP-20 (not its contents). Since the low address bytes of the stack remain the same even after address randomization, the XOR key remains the same and equal to 0xF0. Finally, by observing the loop factors, we can identify the length (in bytes) of the decrypted function. Since ECX acts as a counter to the loop and is set to 0x315, we immediately assume that the function is equal to or more than 0x314 bytes.

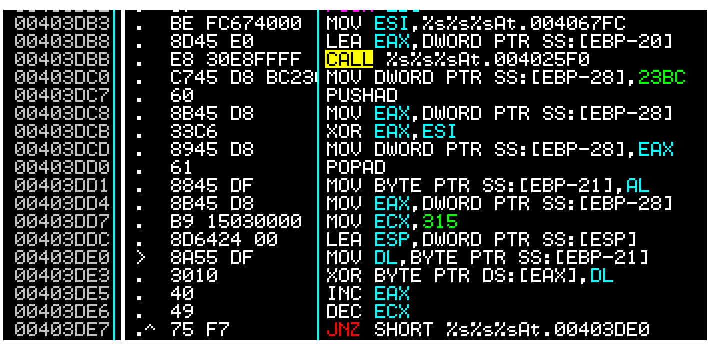


Figure 37 403DA0 Decrypt_Function2

The next call inside WinMain (from *Figure 35*) decrypts the function located at 00403A50.

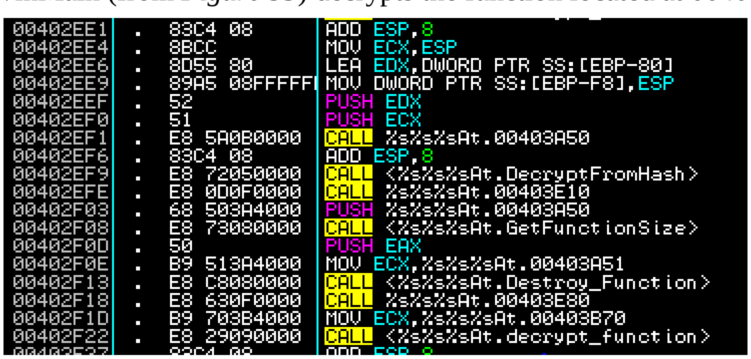


Figure 38 WinMain Second Code Analysis Block

The second code analysis block (see *Figure 38*) calls the newly decrypted function at 00403A50. The result of that function is (at this moment) assumed to be a hash-string value due to the immediate call at DecryptFromHash.

This function contains two interesting CALLs (see *Figure 39*):

1. A call to 00404440 (labeled "Parenting") at address 00403AEE.
2. A call to EDX (at 00403B25) which, through dynamic analysis, revealed itself to be an SHA-512 hashing function.

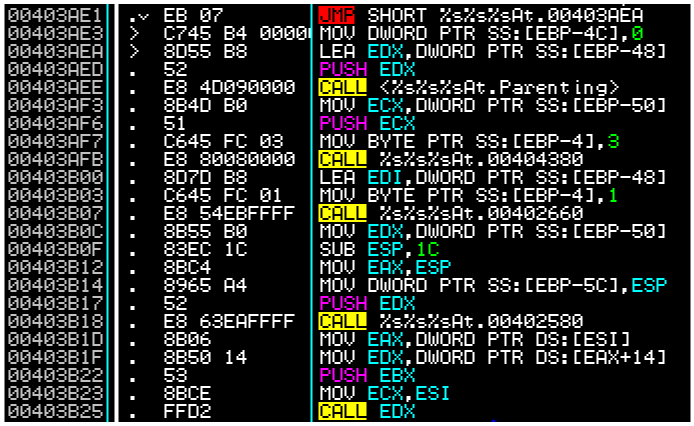


Figure 39 00403A50 Function, POI

3.2.1 Analysis of Parenting Function 1

The "parenting" function located at 00404440 uses a number of API calls to enumerate information about the currently running processes on the host system. That information is then used to retrieve and later assess (outside this function) the parent process id of AthCon_2011.

The APIs invoked are:

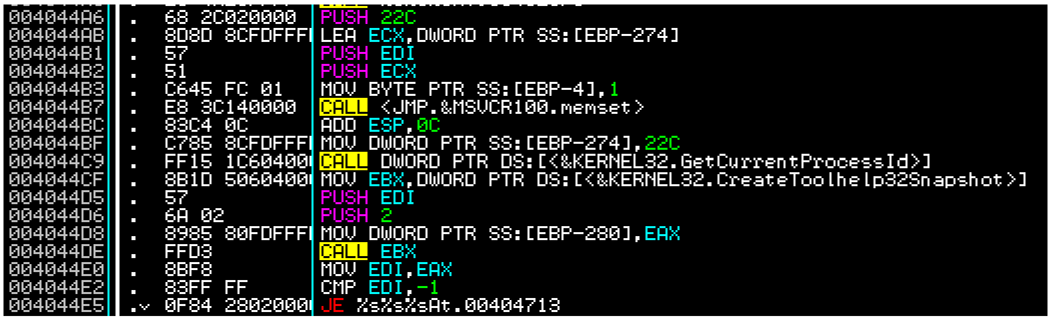
- CreateToolhelp32Snapshot
  http://msdn.microsoft.com/en-us/library/ms682489%28v=vs.85%29.aspx
- Process32FirstW
  http://msdn.microsoft.com/en-us/library/ms684834%28v=vs.85%29.aspx
- Process32NextW
  http://msdn.microsoft.com/en-us/library/ms684836%28v=vs.85%29.aspx

The latter APIs take as argument a pointer to a PROCESSENTRY32 structure containing the required information about the process.

Initially, the function's header (see *Figure* 40) establishes it's stack frame with 0x274 (=628) bytes allocated for local variables (see 00404451). It then set's up the GS stack cookie (see 0040444A to 0040445E) and a local structured exception handler (see 0040445E and 00404465 to 00404468).

Figure 40 00404440 Parenting Function First Code Analysis Block

Next (see *Figure 41*), the memset function is called with ECX being the buffer, located at EBP-274 (local variable), EDI containing the setting value which is equal to zero (see XOR EDI, EDI at 00404477 in *Figure 40*) and the value 0x22C (=556) as the number of bytes to set the value. This reveals that EBP-274 is a buffer or structure with a size equal to 0x22C (=556) and that its first DWORD element is set to that size. This also hints that the structure we are investigating is actually a PROCESSENTRY32 structure which has to have its first element (dwSize) equal to sizeof(PROCESSENTRY32). Additionally, the GetCurrentProcessId function is called and the result is stored in EBP-280 (at address 004044D8). Finally, CreateToolhelp32Snapshot is called with th32ProcessId argument equal to 0 (for current process) and dwFlags equal to 2 (TH32CS_SNAPPROCESS); if the result is equal to -1 (INVALID_HANDLE_VALUE) then the function exits.



```
004044A6 .   68 2C020000  PUSH 22C
004044AB .   8D8D 8CFDFFF LEA ECX,DWORD PTR SS:[EBP-274]
004044B1 .   57           PUSH EDI
004044B2 .   51           PUSH ECX
004044B3 .   C645 FC 01   MOV BYTE PTR SS:[EBP-4],1
004044B7 .   E8 3C140000  CALL <JMP.&MSVCR100.memset>
004044BC .   83C4 0C      ADD ESP,0C
004044BF .   C785 8CFDFFF MOV DWORD PTR SS:[EBP-274],22C
004044C9 .   FF15 1C60400 CALL DWORD PTR DS:[<&KERNEL32.GetCurrentProcessId>]
004044CF .   8B1D 5060400 MOV EBX,DWORD PTR DS:[<&KERNEL32.CreateToolhelp32Snapshot>]
004044D5 .   57           PUSH EDI
004044D6 .   6A 02        PUSH 2
004044D8 .   8985 80FDFFF MOV DWORD PTR SS:[EBP-280],EAX
004044DE .   FFD3         CALL EBX
004044E0 .   8BF8         MOV EDI,EAX
004044E2 .   83FF FF      CMP EDI,-1
004044E5 .v  0F84 2802000 JE %s%s%sAt.00404713
```

Figure 41 00404440 Parenting Function Second Code Analysis Block

If the process snapshot was successful, the execution flow continues (see *Figure 42*) by retrieving the first process from it using the Process32FirstW API call with EDI (the snapshot handle) and EDX (PROCESSENTRY32 structure address) as arguments.

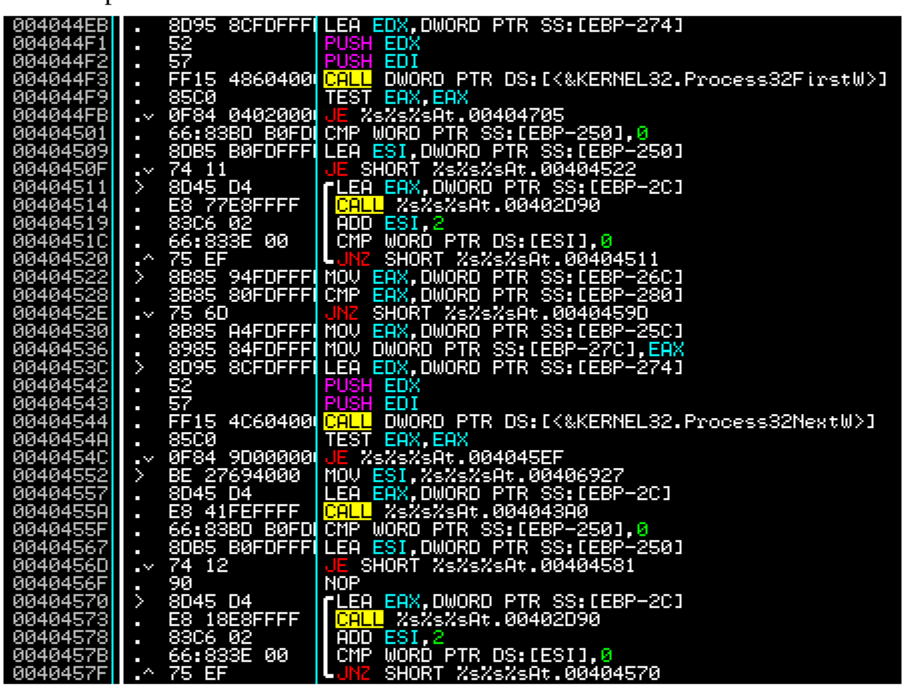
Process32FirstW(

hSnapshot = EDI (assigned to at 004044E0 from resulting EAX),

lppe = EDX (assigned to as the address of EBP-274 at 004044EB)

);

If unsuccessful, the jump located at 004044FB will branch to a call at CloseHandle(hSnapshot = EDI) and then exit the function. If successful, the WORD value at [EBP-250] which falls inside the PROCESSENTRY32 structure at offset +24 (0x274-0x250) from its base address and contains the first wide character of szExeFile element will be compared with 0. In essence, the program validates that the executable file name is present.



```
004044EB .   8D95 8CFDFFF LEA EDX,DWORD PTR SS:[EBP-274]
004044F1 .   52           PUSH EDX
004044F2 .   57           PUSH EDI
004044F3 .   FF15 4860400 CALL DWORD PTR DS:[<&KERNEL32.Process32FirstW>]
004044F9 .   85C0         TEST EAX,EAX
004044FB .v  0F84 0402000 JE %s%s%sAt.00404705
00404501 .   66:83BD B0FD CMP WORD PTR SS:[EBP-250],0
00404509 .   8DB5 B0FDFFF LEA ESI,DWORD PTR SS:[EBP-250]
0040450F .v  74 11        JE SHORT %s%s%sAt.00404522
00404511 >   8D45 D4      LEA EAX,DWORD PTR SS:[EBP-2C]
00404514 .   E8 77E8FFFF  CALL %s%s%sAt.00402D90
00404519 .   83C6 02      ADD ESI,2
0040451C .   66:833E 00   CMP WORD PTR DS:[ESI],0
00404520 .^  75 EF        JNZ SHORT %s%s%sAt.00404511
00404522 >   8B85 94FDFFF MOV EAX,DWORD PTR SS:[EBP-26C]
00404528 .   3B85 80FDFFF CMP EAX,DWORD PTR SS:[EBP-280]
0040452E .v  75 6D        JNZ SHORT %s%s%sAt.0040459D
00404530 .   8B85 A4FDFFF MOV EAX,DWORD PTR SS:[EBP-25C]
00404536 .   8985 84FDFFF MOV DWORD PTR SS:[EBP-27C],EAX
0040453C >   8D95 8CFDFFF LEA EDX,DWORD PTR SS:[EBP-274]
00404542 .   52           PUSH EDX
00404543 .   57           PUSH EDI
00404544 .   FF15 4C60400 CALL DWORD PTR DS:[<&KERNEL32.Process32NextW>]
0040454A .   85C0         TEST EAX,EAX
0040454C .v  0F84 9D00000 JE %s%s%sAt.004045EF
00404552 >   BE 27694000  MOV ESI,%s%s%sAt.00406927
00404557 .   8D45 D4      LEA EAX,DWORD PTR SS:[EBP-2C]
0040455A .   E8 41FEFFFF  CALL %s%s%sAt.004043A0
0040455F .   66:83BD B0FD CMP WORD PTR SS:[EBP-250],0
00404567 .   8DB5 B0FDFFF LEA ESI,DWORD PTR SS:[EBP-250]
0040456D .v  74 12        JE SHORT %s%s%sAt.00404581
0040456F .   90           NOP
00404570 >   8D45 D4      LEA EAX,DWORD PTR SS:[EBP-2C]
00404573 .   E8 18E8FFFF  CALL %s%s%sAt.00402D90
00404578 .   83C6 02      ADD ESI,2
0040457B .   66:833E 00   CMP WORD PTR DS:[ESI],0
0040457F .^  75 EF        JNZ SHORT %s%s%sAt.00404570
```

Figure 42 00404440 Parenting Function Third Code Analysis Block

If so, execution moves into a loop block (from 00404509 to 00404520) which translates the wide characters in szExeFile to their ASCII equivalent. Next the DWORD value in [EBP-26C], that falls inside the PROCESSENTRY32 structure and contains the th32ProcessId element (at offset +8 = 0x274 - 0x26C), is loaded into EAX and compared against the current process Id stored in a local variable inside [EBP-280]. This reveals that the program is looking for the process snapshot of itself. If the two values match (thus we are "looking" at the PROCESSENTRY32 snapshot of the current process) then the program retrieves the value in [EBP-25C], which is in offset +18 (0x274 - 0x25C) of the structure (which contains the th32ParentProcessID element) and stores it in a local variable located at EBP-27C. The whole process loops again and again using the Process32NextW API, until the parent thread Id is found.

In order to bypass this protection one could simply alter the structure object passed to the function. However a much simpler and more generic way of doing so is to set a breakpoint on Process32Next and patch the appropriate parent process id number inside the PROCESSENTRY32 structure. We've drafted an ODBGScript (see ODBGScript AthCon_2011 module Parent ID Hook in Appendix) that:

1. Requests the parent PID we wish to patch
2. Retrieves the current process id
3. Hooks the Process32Next function

Whenever the hook is hit:

1. Compare the current PROCESSENTRY32 structure's th32ProcessId element with the current process id
2. If equal then patch the th32ParentProcessId given by the user
3. Continue execution

Once all processes are accounted for, the same GetToolhelp32Snapshot + Process32Next/First is executed again to verify that the given results are the same. Once the appropriate parent process is located, the function returns to function 00403A50 where the name of the parent process is hashed using SHA-512 to retrieve the next decryption signature which is:

```
"7433bffcd1b34f1b61d9d304f5a9e6f4b4a88281c7db6e3826a0534c0212c559447a1fbcea4a56f3908be173
b8d75baaeb571b63301d01db2b0e55f2a3b80cfa"
```

3.2.2 Analysis of Parenting Function 2

The second code analysis block for WinMain function (see *Figure 43*) contains nothing more than a number of function decryptions and destructions as well as the remaining DecryptFromHash functions. The most interesting part is a call to 00403B70 which contains yet another "Parenting" function slightly different than the first.

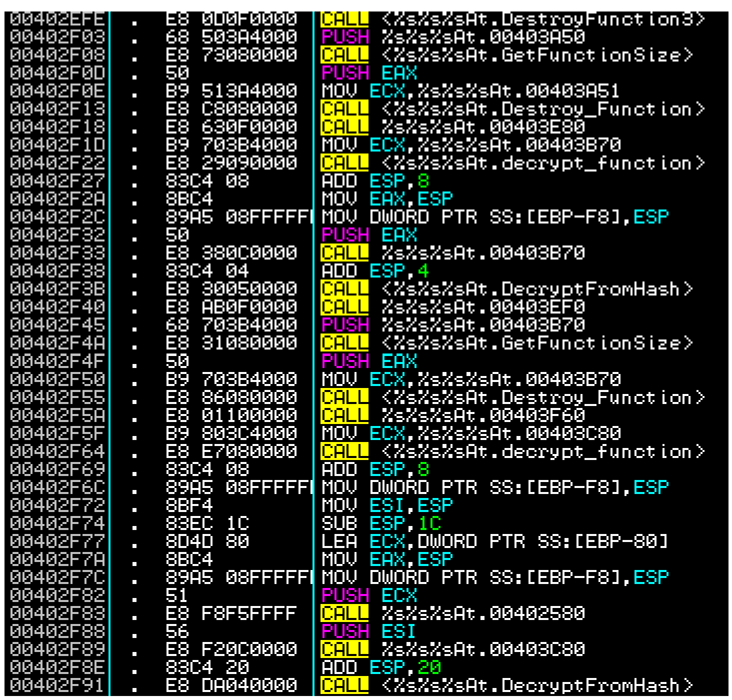


Figure 43 WinMain Second Code Analysis Block

Within the 00403B70 function's body (see *Figure 44*), we discovered two interesting calls. First, a call to the parenting function (at address 00403C01) and then a call to a file handling function (at address 00403C25). As mentioned above, the parenting function is somewhat similar to the one in "Analysis of Parenting Function 1" subsection. However, the only difference is that this function retrieves the parent module's file name by invoking the GetModuleFileNameExA Windows API. This protection can be bypassed by using the same script as before.

Figure 44 00403B70 Parenting and File Handling Functions

Next, the module's file name is passed to the file handling function which opens the parent module file as shown in *Figure 45*, with the "rb" flag.

fopen(

filename = parentPathname,

mode = "rb"

);

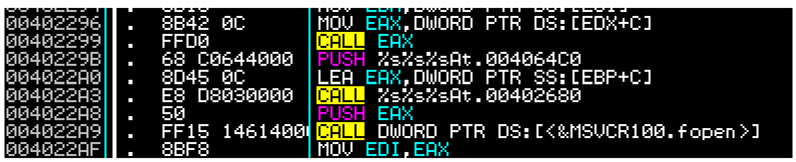


Figure 45 00402250 fopen Call

Next in *Figure 46*, the file handle is send to the fread function (at 0040231C), retrieving 0x400 (=1024) bytes from the beginning of the file.

fread(

ptr = EAX,

size = 1,

count = 0x400,

stream = EDI

);

Once the first fread call is successful, the program enters a looping state in which the function located in EDX (a hashing function) is called, with the fread buffer as an argument. Next, the rest 0x400 (=1024) bytes of the same file stream are read (see CALL EBX at 00402345) and the program loops until the entire file is read and hashed.

On exit, the next SHA-512 digest hash which is send to the DecryptFromHash function chain, is equal to:

"5acae9beaa8b8e99d01849c654ad16770f5ea0c5ff085aac7d4614eb056ab1d84e0a3020bc6e38493f4c7f0c32b8e32e1777215e0b95c87d5c42a85558dee4ba"

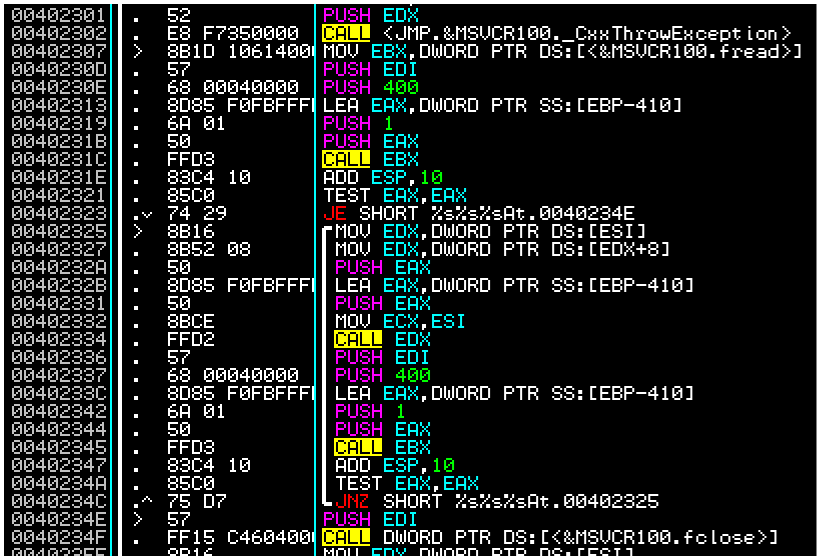


Figure 46 00402250 fread and Hash

3.2.3 Analysis of Parenting Function 3

Moving back to WinMain from *Figure 43*, we identify the final DecryptFromHash function located at 00402F91. Above it, is (yet again) the last Parenting function located in 00403C80. In a similar way, this function compares the parent name retrieved from a previous parenting function against each process in the system (see *Figure 47*). If the two strings match then the parent id is stored and compared to in a loop below, whose goal is to retrieve the parent's parent process name (the parent process of %s%s%sGet_It_All%s%s%s.exe). This is an effective technique against debugging the GIA module while Athcon_2011 is running.

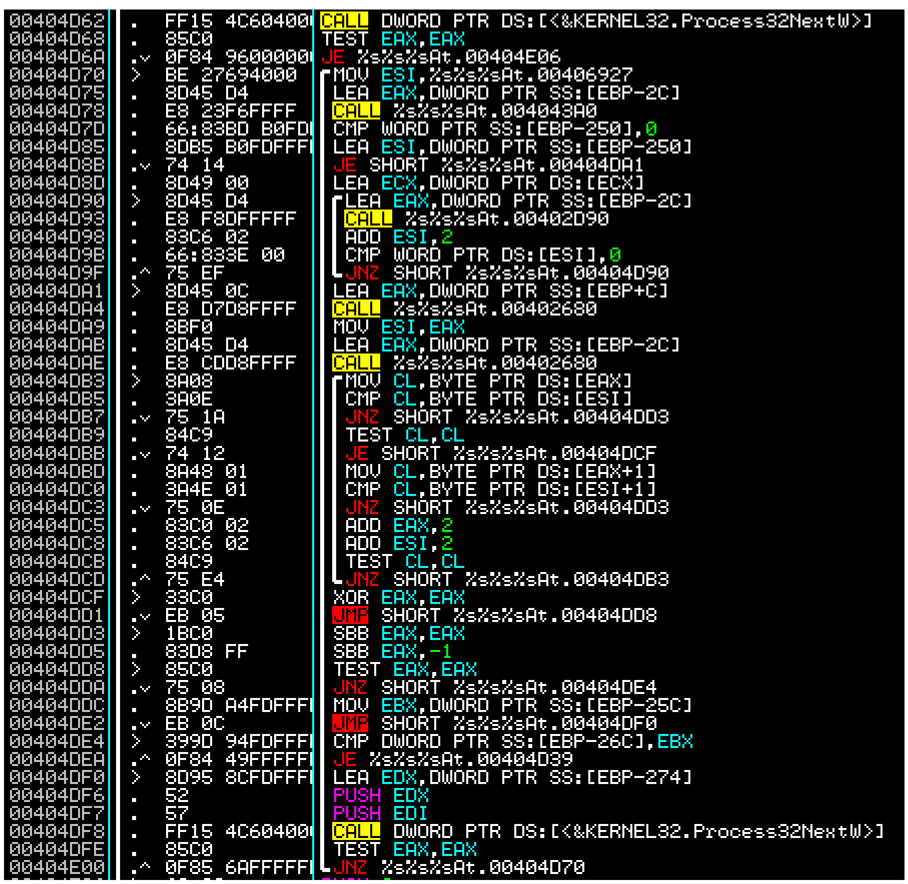


Figure 47 00404c10 Parenting Function 3 Loop 1

Once the parent's parent module name is retrieved, the function exits and the result is send to the SHA-512 hashing function at the CALL instruction located at 00403D5A. The expected argument to the hashing function is "explorer.exe" and the resulting hash digest must be equal to:

"0c3dc6a9d88ac98ee08a6aac028a1cf72e6d736227d36904a9daec84b30c2fccfd57a41daa4d73384bb9133 9482e98e226578eb0d87c958c2bfd2353181b680b"

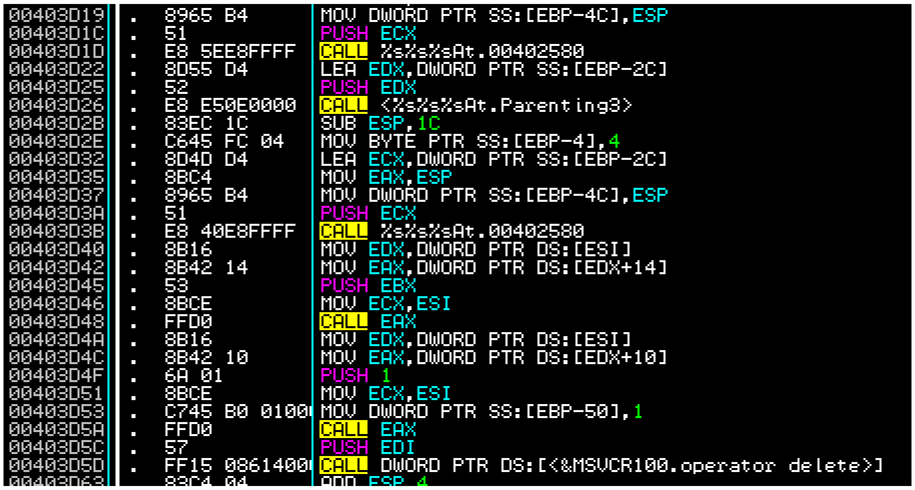


Figure 48 00403C80 Parent and Hash

3.3 Analysis of WinMain Decrypted Body

After the final DecryptFromHash function we come across the same INT3 exception inline instructions previously described in section 3.1. We abuse "ODBGScript AthCon_2011 module INT3 block Patcher" once more to clear out and patch all remaining traces of this annoying protection. The script should finish with a message box reporting that 0x2A CALLs were patched.

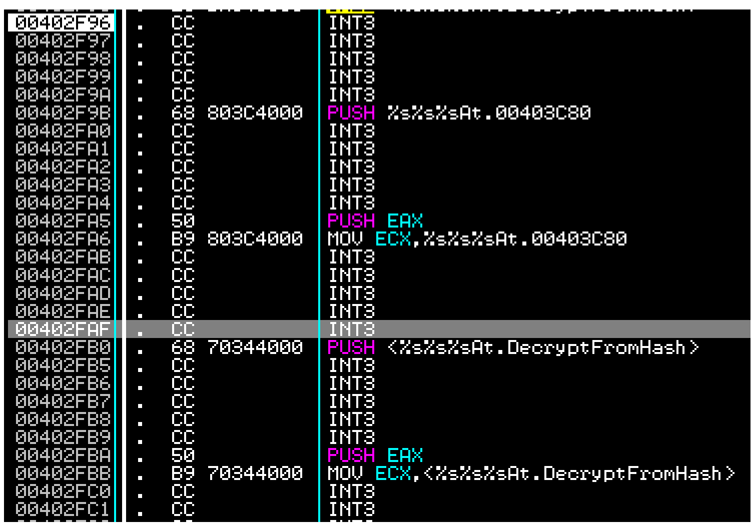


Figure 49 WinMain Decrypted Body

After repairing the rest of the code, we come across a number of functions (see *Figure 50*) that mainly destroy the DecryptFromHash function (see 00402FC0) and decrypt the rest of the anti-debugging functions.

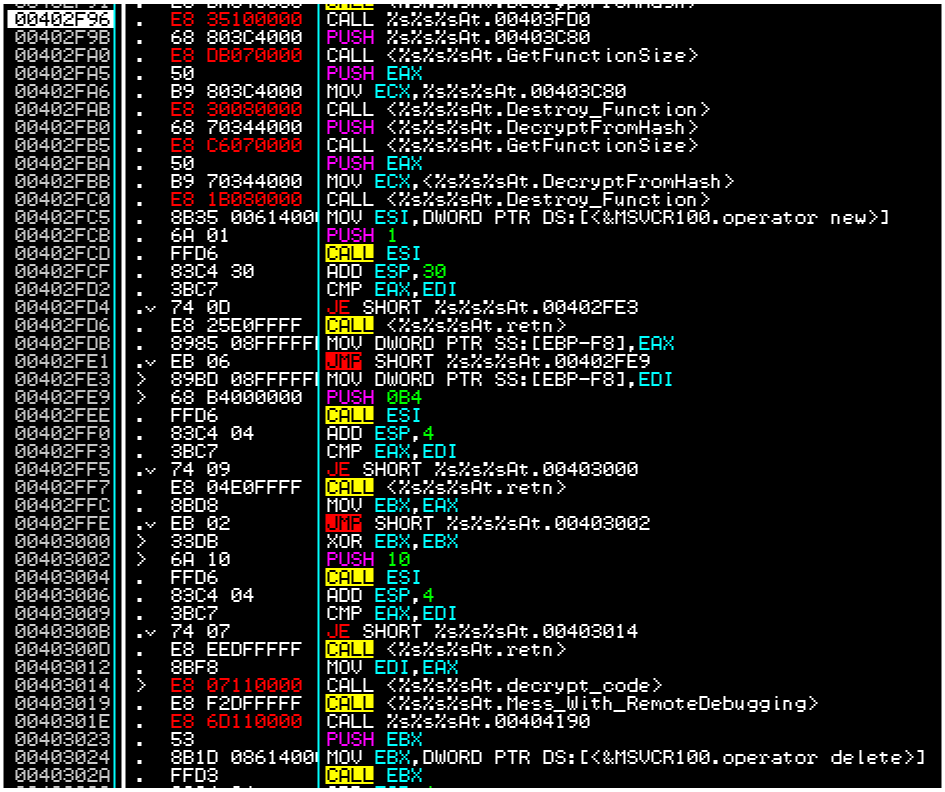


Figure 50 WinMain Third Code Analysis Block

The function labeled "Mess_With_RemoteDebugging" is rather interesting. It effectively disables the standard remote debugging capabilities of the current process making it impossible for a number of debuggers (such as OllyDBG) to attach and debug the process. A simple analysis with OllyDBG reveals that the functions "DbgUiRemoteBreakin" and "DbgBreakPoint" from within the ntdll module have their first byte replaced with a 0x0C3 (RETN) instruction (see assignment at 0040101E and usage at 0040107F).

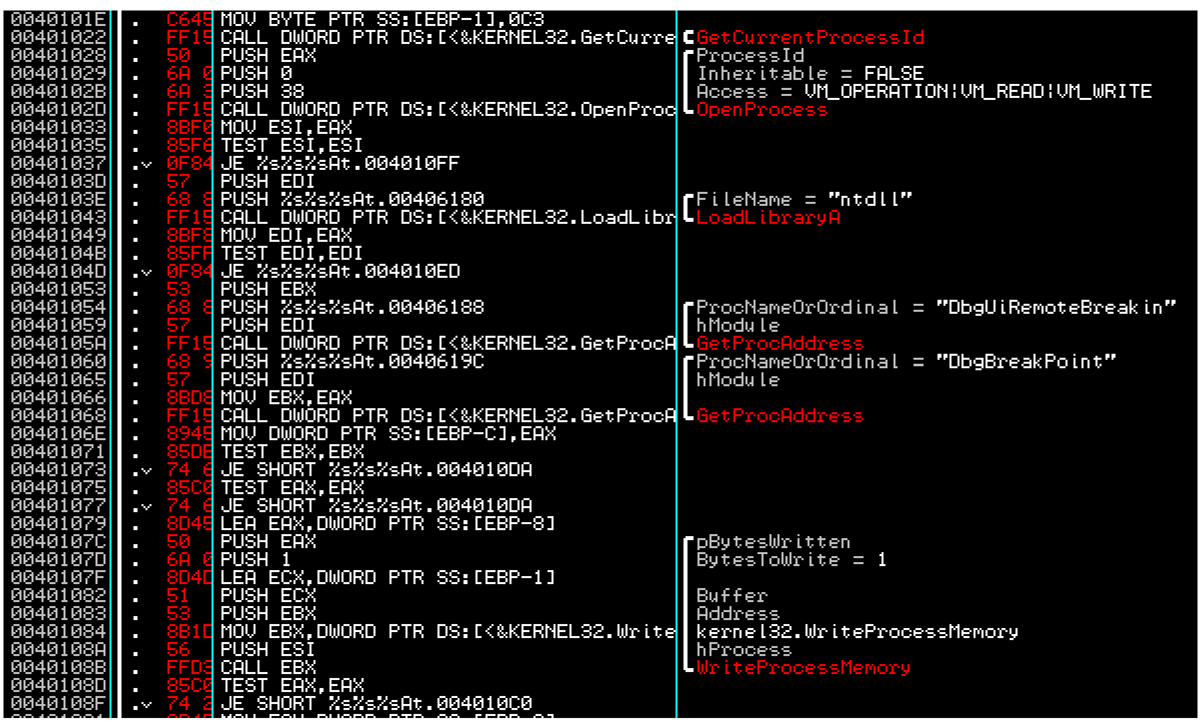


Figure 51 00401010 Mess_With_RemoteDebugging Function

Next, the immediate function called right after Mess_With_RemoteDebugging (see CALL instruction at 0040301E in *Figure 50*) is yet another Destroy_Function function for the previous CALL.

```
0040308A  .  C645 FC 06     MOV BYTE PTR SS:[EBP-4],6
0040308E  .  E8 7DFDFFFF    CALL <%s%s%sAt.GetTickCount Stub>
00403093  .  B9 C0384000    MOV ECX,<%s%s%sAt.openprocess>
00403098  .  E8 B3070000    CALL <%s%s%sAt.decrypt_function>
0040309D  .  FF15 1C604000  CALL DWORD PTR DS:[<&KERNEL32.GetCurrentProcessId>]
004030A3  .  50             PUSH EAX
004030A4  .  E8 17080000    CALL <%s%s%sAt.openprocess>
004030A9  .  68 C0384000    PUSH <%s%s%sAt.openprocess>
004030AE  .  8BF0           MOV ESI,EAX
004030B0  .  E8 CB060000    CALL <%s%s%sAt.GetFunctionSize>
004030B5  .  50             PUSH EAX
004030B6  .  B9 C0384000    MOV ECX,<%s%s%sAt.openprocess>
004030BB  .  E8 20070000    CALL <%s%s%sAt.Destroy_Function>
004030C0  .  83C4 0C        ADD ESP,0C
004030C3  .  E8 780F0000    CALL %s%s%sAt.00404040
004030C8  .  E8 B3E0FFFF    CALL <%s%s%sAt.CheckForHWBreakpoints>
004030CD  .  83F8 01        CMP EAX,1
004030D0  .v 0F84 DF020000  JE %s%s%sAt.004033B5
004030D6  .  B9 10394000    MOV ECX,<%s%s%sAt.Init_Table>
004030DB  .  E8 70070000    CALL <%s%s%sAt.decrypt_function>
004030E0  .  E8 2B080000    CALL <%s%s%sAt.Init_Table>
004030E5  .  68 10394000    PUSH <%s%s%sAt.Init_Table>
004030EA  .  E8 91060000    CALL <%s%s%sAt.GetFunctionSize>
004030EF  .  50             PUSH EAX
004030F0  .  B9 10394000    MOV ECX,<%s%s%sAt.Init_Table>
004030F5  .  E8 E6060000    CALL <%s%s%sAt.Destroy_Function>
004030FA  .  83C4 08        ADD ESP,8
004030FD  .  B9 E0394000    MOV ECX,<%s%s%sAt.ConstructKey>
00403102  .  E8 49070000    CALL <%s%s%sAt.decrypt_function>
00403107  .  85F6           TEST ESI,ESI
00403109  .v 74 09          JE SHORT %s%s%sAt.00403114
0040310B  .  56             PUSH ESI
0040310C  .  E8 CF080000    CALL <%s%s%sAt.ConstructKey>
00403111  .  83C4 04        ADD ESP,4
00403114  >  68 E0394000    PUSH <%s%s%sAt.ConstructKey>
00403119  .  E8 62060000    CALL <%s%s%sAt.GetFunctionSize>
0040311E  .  50             PUSH EAX
0040311F  .  B9 E0394000    MOV ECX,<%s%s%sAt.ConstructKey>
00403124  .  E8 B7060000    CALL <%s%s%sAt.Destroy_Function>
00403129  .  83C4 08        ADD ESP,8
0040312C  .  E8 4FE0FFFF    CALL <%s%s%sAt.CheckForHWBreakpoints>
00403131  .  83F8 01        CMP EAX,1
00403134  .v 0F84 7B020000  JE %s%s%sAt.004033B5
```

Figure 52 WinMain Fourth Code Analysis Block

The next code analysis block begins with a call to a GetTickCount stub function which invokes the GetTickCount API function to retrieve a counter that indicates the number of seconds elapsed since the system was booted. This is a common technique to detect code debugging by invoking the same API twice within a code block, taking the difference in seconds from both resulting values, thus retrieving the number of seconds needed to execute that code block and finally, comparing that number with the expected number of seconds required to execute that block. If time difference between the two calls are greater than the expected number then something or someone paused the execution of the program during analysis. Indeed, in later analysis of the code we've identified a secondary GetTickCount stub (see *Figure 53*) and a comparing function that compares the time required to execute the block with the number 0x3E8 (=1000). In order to bypass this, one could hook GetTickCount and control the number of seconds returned in EAX or even patch the conditional branch located at 0040317E with NOP instructions.

```
00403170  .  E8 BBFCFFFF    CALL <%s%s%sAt.GetTickCount Stub2>
00403175  .  8BC7           MOV EAX,EDI
00403177  .  E8 E4DFFFFF    CALL <%s%s%sAt.Test GetTickCount>
0040317C  .  84C0           TEST AL,AL
0040317E  .v 0F85 31020000  JNZ %s%s%sAt.004033B5
```

Figure 53 GetTickCount Check

Next, the GetCurrentProcessId and OpenProcess windows API functions are invoked. Their purpose is to provide information for the function labeled CheckForHWBreakpoints which calls the GetThreadContext function (see *Figure 54*) with a locally allocated (in the stack) CONTEXT structure located at EBP-2D0 and check if the elements at offsets +4 (Dr0), +8 (Dr1), +C (Dr2), +10 (Dr3) are equal to zero. These "elements" control hardware breakpoints (actually Dr0-Dr3 contain the breakpoint addresses) and are special debug registers within the processor. If one of these registers is not equal to zero, the conditional branch is taken and the function returns 1 in EAX which is then checked at 004030D0 in WinMain and results to the unsuccessful message box if a hardware breakpoint is detected.

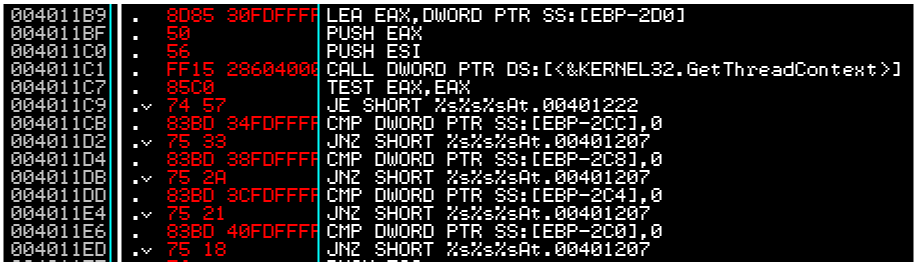


Figure 54 00401180 CheckForHWBreakpoints

The next function labeled "Init_Table" takes the first step towards generating the required archive key to unlock the text files containing the system passwords. Within it, an array of global DWORD values inside the .data section, beginning at address 00408040 and containing 0x210 / 0x04 = 0x84 (=132) entries, is XORed with the value 004030C0.

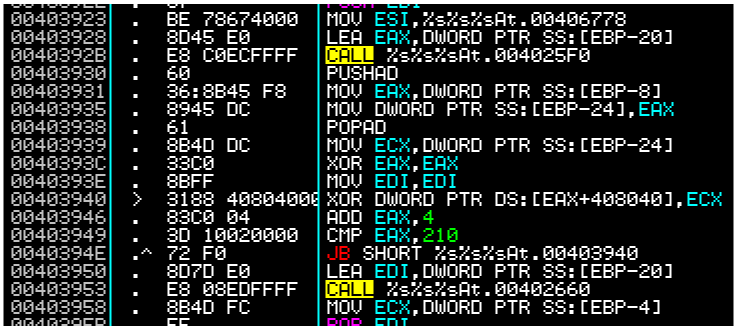


Figure 55 00403910 Init Table Function

The original table values along with the resulting XORed values are listed at "Init_Table Values (Addresses)" section within the Appendix. Notice that the patch values refer to address locations within the program's code section.

Next, the function labeled ConstructKey is called which uses the newly created table to finally construct the key solution to this challenge. Inside, the function makes use of the ReadProcessMemory API to read a single byte from the locations inside the table we've just seen.

```
ReadProcessMemory(
        hProcess = EDI (Given as an argument and is located at EBP+8),
        lpBaseAddress = Patched_Table[i] (i = 0; i < 0x84; i++),
        lpBuffer = ECX (Local buffer located at EBP-21),
        nSize = 1,
        lpNumberOfBytesRead = EAX (Local value)
);
```

Each time the loop iterates, the byte value located at address ESI+00408630 (where ESI = 0 and is incremented each time) is set to the byte read using ReadProcessMemory from the program's code.

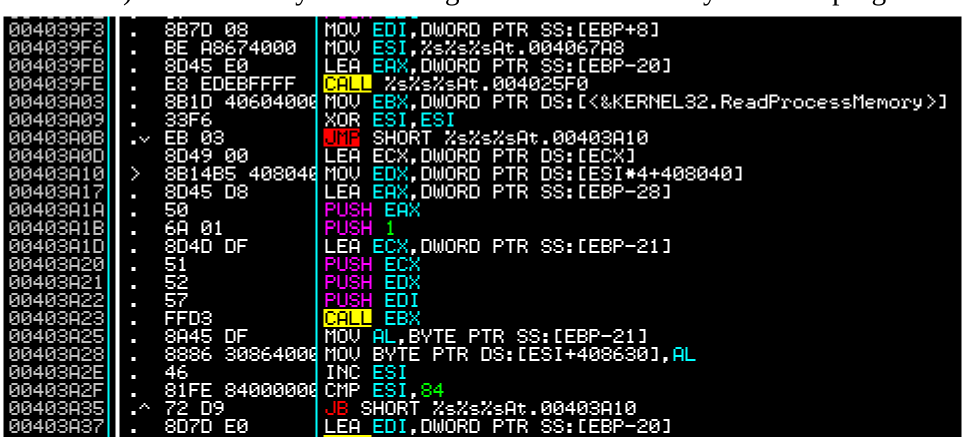


Figure 56 004039E0 Key Construction Function

It is worth noting that the addresses within the table point to locations inside the code that were patched during the execution of the script in Appendix 4.4. The byte values retrieved using the ReadProcessMemory function are therefore the changed bytes and not the original CC (INT3) instructions. However, during a normal program execution (eg, no debuggers attached) those bytes remain unchanged thus creating a small paradoxical scenario; this is because the key required to open the archive is only valid when the INT3 instructions are replaced with the appropriate CALL instructions. Failure to replace them will result in an invalid key filled with 0xCC bytes.

For example, when remote debugging functionality is restored and AthCon_2011 can be attached to, the resulting key is illustrated in *Figure 57*.

```
00408630  cccccccc cccccccc cccccccc cccccccc
00408640  cccccccc cccccccc cccccccc cccccccc
00408650  cccccccc cccccccc cccccccc cccccccc
00408660  cccccccc cccccccc cccccccc cccccccc
00408670  cccccccc cccccccc cccccccc cccccccc
00408680  cccccccc cccccccc cccccccc cccccccc
00408690  cccccccc cccccccc cccccccc cccccccc
004086a0  cccccccc cccccccc cccccccc cccccccc
```

Figure 57 Key When AthCon_2011 Not Patched

The correct key for the archive is illustrated in *Figure 57*. To unlock it you would have to binary copy the ASCII representation of each byte in the long binary string:

9005B30572053005DA041402350271038003B90873083108DB07C607CB06910662061F06A50468042
804FB030E09C808860830081B082007E606B7067406FA04BD047D045004B609AC096F092909E708
B30770074907FE06B40560052305E00417082B081908CF085A0B380CF20CC90E0D0F630FAB0F011
03510780F710F07116D114B00

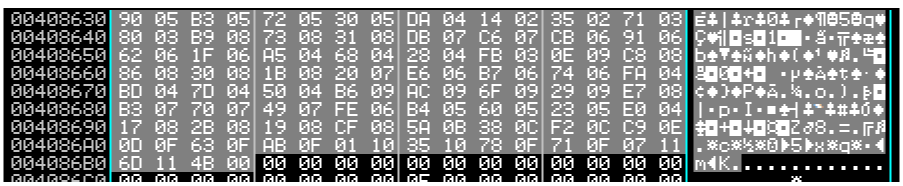


Figure 58 Archive Key

This concludes the analysis of Athcon_2011 module. If you wish to continue the program's execution to the end you just need to bypass the GetTickCount check illustrated in *Figure 53* and patch the conditional branch at address 0040328C (see *Figure 58*) thus allowing the program to generate a file named "Athcon.ctf" containing the key.

Figure 59 Final Conditional Branch

3.4 Conclusions

The analysis of Athcon_2011 module provided quite the challenge due to its polymorphic features that forced us to revert to dynamic analysis, since a static analysis approach would have been inefficient and time consuming. However, the archive key "paradox" we've just seen raises a number of questions about the programmers intensions in regards to the approach vector for successful completion of this challenge.

We'd like to extend our thanks to Kyriakos Economou for his amazing job on creating this challenge and for his contributions to the community. Hopefully, the three of us will meet next year for the completion of a personal challenge involving an unspecified number of beer pints.

-

Chapter 4. Appendix

4.1 Get_It_All De-obfuscation Script for ODBGScript

```
var cnt
var loop
var obf
var nob
var ret_to

mov loop,0
mov cnt,0
redo:
	an eip //Analyze current module
	mov obf, 00401ee0 //Obfuscation Handler 1
	mov nob, 6
	call deobfuscate
	mov obf, 00401ef0 //Obfuscation Handler 2
	mov nob,7
	call deobfuscate
	mov obf, 00401f00 //Obfuscation Handler 3
	mov nob,8
	call deobfuscate
	mov obf, 004010f0 //Obfuscation Handler 4
	mov nob,9
	call deobfuscate
	mov obf, 00401f10 //Obfuscation Handler 5
	mov nob,A
	call deobfuscate
	inc loop
	cmp loop,4
	je exit
	jmp redo


deobfuscate:
	REF obf //Find references to address
	cmp $RESULT, 0 //Check if we have a valid reference
	je out
	inc cnt //Increase counter
	fill $RESULT, nob, 90 //Fill refferer with nob * 0x90 (NOP)
	jmp deobfuscate //Continue
out:
	ret
exit:
	eval "Patched {cnt} calls to obfsc functions"
	MSG $RESULT
	ret
```

4.2 GIA Unknown Object structure

Address Displacement (Decimal)	Type	Comments
+0x00 (0)	DWORD	HANDLE hOfProcess
+0x04 (4)	HANDLE hThread	004017E5 OpenThread
	struct { //PROCESS_INFORMATION	
+0x24 (36)	DWORD	HANDLE hProcess;
+0x28 (40)	DWORD	HANDLE hThread;
+0x2C (44)	DWORD	DWORD dwProcessId;
+0x30 (48)	DWORD	DWORD dwThreadId;
	}	
	struct { //STARTUPINFO	
+0x34 (52)	DWORD[17]	
	}	
+0x60 (96)	DWORD	
+0x64 (100)	WORD	
	DEBUG_EVENT	
+0x78 (120)	DWORD	DWORD dwDebugEventCode
+0x7C (124)	DWORD	DWORD dwProcessId
+0x80 (128)	DWORD	DWORD dwThreadId
	struct { // EXCEPTION_DEBUG_INFO struct { // EXCEPTION_RECORD	
+0x84 (132)	DWORD ExceptionCode	
+0x88 (136)	DWORD ExceptionFlags	
+0x8C (140)	struct EXCEPTION_RECORD *ExceptionRecord	
+0x90 (144)	PVOID ExceptionAddress	
	struct { //CONTEXT	
+D8 (216)	DWORD ContextFlags	
+190 (400)	DWORD Eip	
+19C (412)	DWORD Esp	

4.3 INT3 Address Lookup Program

```
unsigned long SomeVars[66] = {
	0x00000595, 0x000005B8, 0x00000577, 0x00000535, 0x000004DF, 0x00000219,
	0x0000023A, 0x00000376, 0x00000385, 0x000008BE, 0x00000878, 0x00000836,
	0x000007E0, 0x000007CB, 0x000006D0, 0x00000696, 0x00000667, 0x00000624,
	0x000004AA, 0x0000046D, 0x0000042D, 0x00000400, 0x00000913, 0x000008CD,
	0x0000088B, 0x00000835, 0x00000820, 0x00000725, 0x000006EB, 0x000006BC,
	0x00000679, 0x000004FF, 0x000004C2, 0x00000482, 0x00000455, 0x000009BB,
	0x000009B1, 0x00000974, 0x0000092E, 0x000008EC, 0x000007B8, 0x00000775,
	0x0000074E, 0x00000703, 0x000005B9, 0x00000565, 0x00000528, 0x000004E5,
	0x0000081C, 0x00000830, 0x0000081E, 0x000008D4, 0x00000B5F, 0x00000C3D,
	0x00000CF7, 0x00000ECE, 0x00000F12, 0x00000F68, 0x00000FB0, 0x00001006,
	0x0000103A, 0x00000F7D, 0x00000F76, 0x0000110C, 0x00001172, 0x00001350
};
unsigned long EIPTable[66] = {
	0x00402E8B, 0x00402EB8, 0x00402EF9, 0x00402F3B, 0x00402F91, 0x00403307,
	0x00403376, 0x004032AA, 0x0040333B, 0x00402EC2, 0x00402F08, 0x00402F4A,
	0x00402FA0, 0x00402FB5, 0x004030B0, 0x004030EA, 0x00403119, 0x0040315C,
	0x004032D6, 0x00403313, 0x00403353, 0x00403380, 0x00402ECD, 0x00402F13,
	0x00402F55, 0x00402FAB, 0x00402FC0, 0x004030BB, 0x004030F5, 0x00403124,
	0x00403167, 0x004032E1, 0x0040331E, 0x0040335E, 0x0040338B, 0x00402E95,
	0x00402E9F, 0x00402EDC, 0x00402F22, 0x00402F64, 0x00403098, 0x004030DB,
	0x00403102, 0x0040314D, 0x00403297, 0x004032EB, 0x00403328, 0x0040336B,
	0x004030A4, 0x004030E0, 0x00403152, 0x0040310C, 0x00402EF1, 0x00402F33,
	0x00402F89, 0x00402ED2, 0x00402EFE, 0x00402F18, 0x00402F40, 0x00402F5A,
	0x00402F96, 0x004030C3, 0x0040313A, 0x00403014, 0x0040301E, 0x00402EB0
};
int main()
{
	char number[11] = {0};
	unsigned long n, raddr;
	int i;
	printf("Address?: ");
	scanf("%10s",number);
	n = (unsigned long)strtol(number, NULL, 16);
	for(i=0;i<sizeof(EIPTable);i++) {
		if(n == EIPTable[i]) {
			raddr = EIPTable[i] + 5;
			printf("EIP: 0x%08X\nStack WPM: 0x%08X", EIPTable[i]+SomeVars[i],raddr);
			break;
		}
	}
}
```

4.4 ODBGScript AthCon_2011 module INT3 block Patcher

```
push ebp
mov ebp,esp
sub esp,214
mov [ebp-10C],0402E8B
mov [ebp-108],0402EB8
mov [ebp-0104],0402EF9
mov [ebp-0100],0402F3B
mov [ebp-0FC],0402F91
mov [ebp-0F8],0403307
mov [ebp-0F4],0403376
mov [ebp-0F0],04032AA
mov [ebp-0EC],040333B
mov [ebp-0E8],0402EC2
mov [ebp-0E4],0402F08
mov [ebp-0E0],0402F4A
mov [ebp-0DC],0402FA0
mov [ebp-0D8],0402FB5
mov [ebp-0D4],04030B0
mov [ebp-0D0],04030EA
mov [ebp-0CC],0403119
mov [ebp-0C8],040315C
mov [ebp-0C4],04032D6
mov [ebp-0C0],0403313
mov [ebp-0BC],0403353
mov [ebp-0B8],0403380
mov [ebp-0B4],0402ECD
mov [ebp-0B0],0402F13
mov [ebp-0AC],0402F55
mov [ebp-0A8],0402FAB
mov [ebp-0A4],0402FC0
mov [ebp-0A0],04030BB
mov [ebp-09C],04030F5
mov [ebp-098],0403124
mov [ebp-094],0403167
mov [ebp-090],04032E1
mov [ebp-08C],040331E
mov [ebp-088],040335E
mov [ebp-084],040338B
mov [ebp-080],0402E95
mov [ebp-07C],0402E9F
mov [ebp-078],0402EDC
mov [ebp-074],0402F22
mov [ebp-070],0402F64
mov [ebp-06C],0403098
mov [ebp-068],04030DB
mov [ebp-064],0403102
mov [ebp-060],040314D
mov [ebp-05C],0403297
mov [ebp-058],04032EB
mov [ebp-054],0403328
mov [ebp-050],040336B
mov [ebp-04C],04030A4
mov [ebp-048],04030E0
mov [ebp-044],0403152
mov [ebp-040],040310C
mov [ebp-03C],0402EF1
```

```
mov [ebp-038],0402F33
mov [ebp-034],0402F89
mov [ebp-030],0402ED2
mov [ebp-02C],0402EFE
mov [ebp-028],0402F18
mov [ebp-024],0402F40
mov [ebp-020],0402F5A
mov [ebp-01C],0402F96
mov [ebp-018],04030C3
mov [ebp-014],040313A
mov [ebp-010],0403014
mov [ebp-0C],040301E
mov [ebp-8],0402EB0

mov [ebp-0214],0595
mov [ebp-0210],05B8
mov [ebp-020C],0577
mov [ebp-0208],0535
mov [ebp-0204],04DF
mov [ebp-0200],0219
mov [ebp-01FC],023A
mov [ebp-01F8],0376
mov [ebp-01F4],0385
mov [ebp-01F0],08BE
mov [ebp-01EC],0878
mov [ebp-01E8],0836
mov [ebp-01E4],07E0
mov [ebp-01E0],07CB
mov [ebp-01DC],06D0
mov [ebp-01D8],0696
mov [ebp-01D4],0667
mov [ebp-01D0],0624
mov [ebp-01CC],04AA
mov [ebp-01C8],046D
mov [ebp-01C4],042D
mov [ebp-01C0],0400
mov [ebp-01BC],0913
mov [ebp-01B8],08CD
mov [ebp-01B4],088B
mov [ebp-01B0],0835
mov [ebp-01AC],0820
mov [ebp-01A8],0725
mov [ebp-01A4],06EB
mov [ebp-01A0],06BC
mov [ebp-019C],0679
mov [ebp-0198],04FF
mov [ebp-0194],04C2
mov [ebp-0190],0482
mov [ebp-018C],0455
mov [ebp-0188],09BB
mov [ebp-0184],09B1
mov [ebp-0180],0974
mov [ebp-017C],092E
mov [ebp-0178],08EC
mov [ebp-0174],07B8
mov [ebp-0170],0775
mov [ebp-016C],074E
```

```
mov [ebp-0168],0703
mov [ebp-0164],05B9
mov [ebp-0160],0565
mov [ebp-015C],0528
mov [ebp-0158],04E5
mov [ebp-0154],081C
mov [ebp-0150],0830
mov [ebp-014C],081E
mov [ebp-0148],08D4
mov [ebp-0144],0B5F
mov [ebp-0140],0C3D
mov [ebp-013C],0CF7
mov [ebp-0138],0ECE
mov [ebp-0134],0F12
mov [ebp-0130],0F68
mov [ebp-012C],0FB0
mov [ebp-0128],01006
mov [ebp-0124],0103A
mov [ebp-0120],0F7D
mov [ebp-011C],0F76
mov [ebp-0118],0110C
mov [ebp-0114],01172
mov [ebp-0110],01350

mov cnt,0
pusha

main_loop:
        mov ebx, eip ; ebx = current position we want to patch

        next:
                call findCC
                cmp $RESULT,0
                je next
                mov eax, $RESULT
                mov ecx,eax
                call find_index
                inc cnt
                cmp eax,0
                jne continue
                eval "Patched {cnt} CALLs, continue?"
                msgyn $RESULT
                cmp $RESULT,0
                je exit
        continue:
                jmp next
exit:
                popa
                mov esp,ebp
                pop ebp
                ;add esp,214
                ret
findCC:
        cmp [ebx+4], CC, 1
        je l1
        add ebx,5
        jmp findCC_exit
```

```
        l1:
        cmp [ebx+3], CC, 1
        je l2
        add ebx,4
        jmp findCC_exit
        l2:
        cmp [ebx+2], CC, 1
        je l3
        add ebx,3
        jmp findCC_exit
        l3:
        cmp [ebx+1], CC, 1
        je l4
        add ebx,2
        jmp findCC_exit
        l4:
        cmp [ebx], CC, 1
        je l5
        add ebx,1
        jmp findCC_exit
        l5:
        mov $RESULT,ebx
        ret

findCC_exit:
        mov $RESULT,0
        ret

find_index:

        xor edi,edi
        find_index_loop:
                mov edx, edi*4
                add edx, ebp
                sub edx, 10c
                cmp eax, [edx] ; [ebp+edi*4-10c]
                je out
                inc edi
                cmp edi, 42
                je fail
                jmp find_index_loop

out:
        mov edx, edi*4
        add edx, ebp
        sub edx, 214
        add eax, [edx]
        eval "call {eax}"
        mov tmp, $RESULT
        asm ecx, $RESULT
        eval "{ecx} -> {tmp}"
        log $RESULT, ""Patched: "
        ret
fail:
        xor eax,eax
        ret
```

4.5 ODBGScript AthCon_2011 module Parent ID Hook

```
var pid
ask "Parent PID (in hex)?"
mov ppid, $RESULT

jmp main

Pr32Next:
	mov pentry, [esp+8]
	rtr
	cmp [pentry+8], pid
	je patch_parent
	run

patch_parent:
	eval "{pentry->th32ProcessID (= {pentry+8} )"
	log $RESULT, "For: "
	eval "{pentry}->th32ParentProcessID = {ppid}"
	log $RESULT, "Patched: "
	mov [pentry+18], ppid
	run

main:
gpa "GetCurrentProcessId", "kernel32.dll"
mov gcpid, $RESULT
exec
	push eax
	call GetCurrentProcessId
ende

;call gcpid
mov pid,eax
pop eax

gpa "Process32NextW", "kernel32.dll"
mov p32n, $RESULT
bp p32n
bpgoto p32n, Pr32Next
```

4.6 Init_Table Values (Addresses)

```
const
  unsigned long Original_Values[84] = {
    0x00001E4C, 0x00001E4D, 0x00001E79, 0x00001E7A, 0x00001E3A, 0x00001E3B,
    0x00001FFC, 0x00001FFD, 0x00001F52, 0x00001F53, 0x000003C8, 0x000003C9,
    0x000003B7, 0x000003B8, 0x0000026B, 0x0000026C, 0x000003FC, 0x000003FD,
    0x00001E03, 0x00001E04, 0x00001FC9, 0x00001FCA, 0x00001F8B, 0x00001F8C,
    0x00001F61, 0x00001F62, 0x00001F76, 0x00001F77, 0x00000071, 0x00000072,
    0x0000002B, 0x0000002C, 0x000001DA, 0x000001DB, 0x0000019D, 0x0000019E,
    0x00000217, 0x00000218, 0x000003D4, 0x000003D5, 0x00000394, 0x00000395,
    0x00000341, 0x00000342, 0x00001E0E, 0x00001E0F, 0x00001FD4, 0x00001FD5,
    0x00001F96, 0x00001F97, 0x00001F6C, 0x00001F6D, 0x00001F01, 0x00001F02,
    0x0000007C, 0x0000007D, 0x00000036, 0x00000037, 0x000001E5, 0x000001E6,
    0x000001A8, 0x000001A9, 0x00000222, 0x00000223, 0x000003DF, 0x000003E0,
    0x0000039F, 0x000003A0, 0x0000034C, 0x0000034D, 0x00001E56, 0x00001E57,
    0x00001E60, 0x00001E61, 0x00001E1D, 0x00001E1E, 0x00001FE3, 0x00001FE4,
    0x00001FA5, 0x00001FA6, 0x00000059, 0x0000005A, 0x0000001C, 0x0000001D
  };

const
  unsigned long PatchedValues[84] = {
    0x00402E8C, 0x00402E8D, 0x00402EB9, 0x00402EBA, 0x00402EFA, 0x00402EFB,
    0x00402F3C, 0x00402F3D, 0x00402F92, 0x00402F93, 0x00403308, 0x00403309,
    0x00403377, 0x00403378, 0x004032AB, 0x004032AC, 0x0040333C, 0x0040333D,
    0x00402EC3, 0x00402EC4, 0x00402F09, 0x00402F0A, 0x00402F4B, 0x00402F4C,
    0x00402FA1, 0x00402FA2, 0x00402FB6, 0x00402FB7, 0x004030B1, 0x004030B2,
    0x004030EB, 0x004030EC, 0x0040311A, 0x0040311B, 0x0040315D, 0x0040315E,
    0x004032D7, 0x004032D8, 0x00403314, 0x00403315, 0x00403354, 0x00403355,
    0x00403381, 0x00403382, 0x00402ECE, 0x00402ECF, 0x00402F14, 0x00402F15,
    0x00402F56, 0x00402F57, 0x00402FAC, 0x00402FAD, 0x00402FC1, 0x00402FC2,
    0x004030BC, 0x004030BD, 0x004030F6, 0x004030F7, 0x00403125, 0x00403126,
    0x00403168, 0x00403169, 0x004032E2, 0x004032E3, 0x0040331F, 0x00403320,
    0x0040335F, 0x00403360, 0x0040338C, 0x0040338D, 0x00402E96, 0x00402E97,
    0x00402EA0, 0x00402EA1, 0x00402EDD, 0x00402EDE, 0x00402F23, 0x00402F24,
    0x00402F65, 0x00402F66, 0x00403099, 0x0040309A, 0x004030DC, 0x004030DD
  };
```

Original Values	Patched Values
00001E4C	00402E8C
00001E4D	00402E8D
00001E79	00402EB9
00001E7A	00402EBA
00001E3A	00402EFA
00001E3B	00402EFB
00001FFC	00402F3C
00001FFD	00402F3D
00001F52	00402F92
00001F53	00402F93
000003C8	00403308
000003C9	00403309
000003B7	00403377
000003B8	00403378
0000026B	004032AB
0000026C	004032AC
000003FC	0040333C
000003FD	0040333D
00001E03	00402EC3
00001E04	00402EC4
00001FC9	00402F09
00001FCA	00402F0A
00001F8B	00402F4B
00001F8C	00402F4C
00001F61	00402FA1
00001F62	00402FA2
00001F76	00402FB6
00001F77	00402FB7
00000071	004030B1
00000072	004030B2
0000002B	004030EB
0000002C	004030EC
000001DA	0040311A
000001DB	0040311B
0000019D	0040315D
0000019E	0040315E
00000217	004032D7
00000218	004032D8
000003D4	00403314
000003D5	00403315
00000394	00403354
00000395	00403355
00000341	00403381
00000342	00403382
00001E0E	00402ECE
00001E0F	00402ECF
00001FD4	00402F14
00001FD5	00402F15
00001F96	00402F56
00001F97	00402F57
00001F6C	00402FAC
00001F6D	00402FAD
00001F01	00402FC1
00001F02	00402FC2
0000007C	004030BC
0000007D	004030BD
00000036	004030F6
00000037	004030F7
000001E5	00403125
000001E6	00403126
000001A8	00403168
000001A9	00403169
00000222	004032E2
00000223	004032E3
000003DF	0040331F
000003E0	00403320
0000039F	0040335F
000003A0	00403360
0000034C	0040338C
0000034D	0040338D
00001E56	00402E96
00001E57	00402E97
00001E60	00402EA0
00001E61	00402EA1
00001E1D	00402EDD
00001E1E	00402EDE
00001FE3	00402F23
00001FE4	00402F24
00001FA5	00402F65
00001FA6	00402F66
00000059	00403099
0000005A	0040309A
0000001C	004030DC
0000001D	004030DD