# Solving of Hacker Challenge 2007 Phase 3

**Author: Omega Red (omegared@o2.pl)**

## *Attack Narrative*

### Finding the password and the formula

When we first run the executable, it produces truncated output. A password is required, but we're not told as in what form it should be delivered to the executable. To find out, author used *Process Monitor* from Sysinternals.

```
69269  21:57:48.8726820    final.exe    2284   CreateFile    C:\hc3\data.txt     SUCCESS
      Desired Access: Generic Read, Disposition: Open, Options: Synchronous IO Non-Alert,
Non-Directory File, Attributes: N, ShareMode: Read, Write, AllocationSize: n/a,
OpenResult: Opened
69270  21:57:48.8727534    final.exe    2284   ReadFile      C:\hc3\data.txt     SUCCESS
       Offset: 0, Length: 152
69272  21:57:48.8728270    final.exe    2284   CloseFile     C:\hc3\data.txt     SUCCESS
```

It seems that our target only reads **data.txt**, it doesn't touch other files or registry or anything else. That would imply that the password should be in data.txt. But in what form should it be? That's the first real problem.

Let's load the exe to IDA. Surprise, it doesn't look like packed/encrypted at first glance. The **main** function is larger and more complex than in phase 1 though. We see `IsDebuggerPresent` calls, `GetTickCount`, `QueryPerformanceCounter` – obvious anti-debug measures. And then in the middle of **main** there is a block of garbage data starting at `00402D54`. Oh, encryption after all ;). Time to load it into Olly.

We place breakpoint at the transition to encrypted code:

```
.text:00402D4D                  jmp     short loc_402D54
```

and trace **main** stepping over all functions. We notice that after `call 00401C20` there is some kind of checksumming performed:

```
00402C46   .  0FB7C8                  movzx ecx,ax
00402C49   .  81F9 80040000           cmp ecx,480
00402C4F   .  74 07                   je short final_or.00402C58
00402C51   .  6A 00                   push 0
00402C53   .  E8 81AC0000             call <final_or.EXIT>
00402C58   >  EB 05                   jmp short final_or.00402C5F
```

The checksum is wrong and program exits. Let's see what's inside the call. Calls to `VirtualProtect` and some number crunching? Decryption routine. It also calls some other functions at the beginning. Since it's important procedure, we examine these functions. First of them, at `00401D80`, is a nice anti-debug trick involving **`int 2d`** – the windows kernel debugger API as described by ReWolf. This function creates SEH frame and issues `int 2d`. In normal conditions, `int` command will cause exception and transition to the SEH handler, which in turn changes faulting thread's context – it moves value **0** to **ebx** register. We see a "`mov ebx, 1`" instruction just before `int 2d` – so under debugger, when no exception is called, `ebx` will contain 1 at the end of the procedure. This is used to alter some variables used during calculations, corrupting results in the process. How to defeat it? Simply change "`mov ebx, 1`" to "`mov ebx, 0`" at `401dc5` and nop out the `int 2d`. Voila, the function now always returns 0 ("not debugged").

Second sub-function called in decryption routine is very similar anti-debug routine, just it uses trap flag to trigger exception:

```
00401BB5  |.  B8 01000000              mov eax,1
00401BBA  |.  9C                       pushfd
00401BBB  |.  813424 54010000          xor dword ptr ss:[esp],154
00401BC2  |.  9D                       popfd
00401BC3  |.  0AC0                     or al,al
```

SEH handler installed at the beginning of this function also changes context, this time modifying `eax` to be 0. As we see, the scheme is almost identical as before. We patch it in the same manner: changing `mov eax, 1` to `mov eax, 0`. Good, both of them should be neutralized.

Now we observe the `VirtualProtect` call and its arguments – it will tell us what region of code is being decrypted. There are 3 decrypt calls in total. First one decrypts code @ **`402d50`** of size 0xDC bytes. That's right where our "garbage bytes" are. Now after we "fixed" anti-debug routines, checksum after decryption is OK. Good. Next we see another checksum calculations @ around `402c98`. For now it returns good value, but we note it in case we need to patch it. Next we have `IsDebuggerPresent` call easily fooled by OllyAdvanced. After it there is a second call to decryption routine (address **`402f20`**, size 0xE0). This area is located a bit further from first encrypted block. Quick glance at it after decryption doesn't give any immediate clues on what it does, but its role will be apparent later.

Next we have `GetTickCount` trick and third, final decryption call (address **`401ef0`**, size 0xF0). Look at the decrypted instructions – oh, some calculations with floating point math and a call to trap-flag anti-debug! Suspicious, maybe this is the function that calculates our needed value? We'll verify it later.

Now we'll dump all decrypted memory sections and patch the executable to make static analysis easier. We'll also nop out the "`mov [esi], edx`" instruction at `00401C73`, this will disable decryptor (since code will be already decrypted by us). Quick test after patching: program outputs wrong results. We recall noticing a checksum calculation at **`402c98`** – now we can see that it returns wrong value ;). We patch jump at **`402c9e`** to unconditional. Result? Wrong values again, but this time different ones. Ouch.

Let's just manually scan **main** disassembly in IDA and look for suspicious instruction blocks. With this approach we can find what follows:

```
.text:004034D6                xor     eax, eax
.text:004034D8                mov     ecx, (offset loc_402C5A+2)
.text:004034DD                mov     edx, 2Ah
.text:004034E2
.text:004034E2  loc_4034E2:                                          ; CODE
XREF: .text:004034ED□j
.text:004034E2                movzx   esi, word ptr [ecx]
.text:004034E5                add     eax, esi
.text:004034E7                add     ecx, 2
.text:004034EA                sub     edx, 1
.text:004034ED                jnz     short loc_4034E2
```

```
.text:004034EF                  mov     ecx, eax
.text:004034F1                  shr     ecx, 10h
.text:004034F4                  jz      short loc_403502
.text:004034F6
.text:004034F6  loc_4034F6:                                                     ; CODE
XREF: .text:00403500□j
.text:004034F6                  movzx   eax, ax
.text:004034F9                  add     eax, ecx
.text:004034FB                  mov     ecx, eax
.text:004034FD                  shr     ecx, 10h
.text:00403500                  jnz     short loc_4034F6
.text:00403502
.text:00403502  loc_403502:                                                     ; CODE
XREF: .text:004034F4□j
.text:00403502                  not     eax
.text:00403504                  movzx   eax, ax
.text:00403507                  movzx   ecx, ax
.text:0040350A                  test    ecx, ecx
.text:0040350C                  jz      short loc_403515
.text:0040350E                  add     dword_42DD50, 1
.text:00403515
.text:00403515  loc_403515:                                                     ; CODE
XREF: .text:0040350C□j
```

A checksum routine similar to what we've seen before. If it fails, a global variable is modified. Let's patch jump at **0040350C** to unconditional. Finally! The executable now produces correct output. Let's continue the analysis of **main**.

After that there is a call pair of `QueryPerformanceFrequency` / `QueryPerformanceCounter`. Another timing trick? Doh. This time OllyAdvanced won't help us, so we'll need to be careful when debugging.

Finally we reach the first decrypted code block. What does it do? Let's trace and see. ;)
At the beginning of it there are 2 calls to some private functions. First one takes 2 arguments, apparently pointers to stack variables. After comparing their (variables) content before and after call, we see that the one pointed by `eax` contains ASCII "1" after call. Hmpf. Let's move on. There is another call with more parameters – constants "0a" (newline?), 0x3e8, and another two stack pointers. And after this call we see that "eax variable" contains more ASCII numbers. Now it's clear that the first call was reading first "word" from **data.txt** (number 1 in case of original file), and the second – rest of the line. Good. What do we have next? Some number crunching that operates on the first read buffer! It reads **12 characters** from it in 4 turns of 3 bytes, produces 4 byte result and compares it with "**4242**" string. Operations are fairly trivial, they can be reduced to sum of 3 chars modulo 0x100 (if byte values don't exceed 0x7f) for every byte of output. Sample string that meets those requirements is "**omega~redG]f**". This must be the password we're looking for ;). Indeed, when we insert such string as first line of data.txt, executable produces full and correct output.

After password check we see second call to `QueryPerformanceCounter`. Then some math that seems to calculate number of seconds elapsed between first call and this one. This number is then compared to 0.1 – if it's less, all is OK and we're not traced ;). We can just flip the flag on comparison during debugging, as it doesn't matter when run without debugger.

```
00402E70    .  DC1D 00854200            fcomp qword ptr ds:[428500]
00402E76    .  DFE0                     fstsw ax
00402E78    .  F6C4 05                  test ah,5
```

```
00402E7B   .  7B 45                     jpo short final_or.00402EC2
```

Manual tracing further gives us headache, there is too much code to swallow. Let's concentrate on the missing formula. Since we suspected that decrypted block at **`401ef0`** might be what we're looking for, let's place a breakpoint there and see what happens. Because we have all code statically decrypted this should present no problem. Executable breaks there without problems, but on the console we can see bogus output. Hmm… there must be more antidebugs somewhere else, but hopefully we will be able to trace this function at least.

It starts with call to familiar "trap flag" anti-debug function. Then a bit of integer number crunching. Some floating point operations and… voila, we have our magic number on top of FP stack.

```
00401EF0   .  55                        push ebp
00401EF1   .  8BEC                      mov ebp,esp
00401EF3   .  83EC 0C                   sub esp,0C
00401EF6   .  56                        push esi
00401EF7   .  8BF1                      mov esi,ecx
00401EF9   .  EB 05                     jmp short final_or.00401F00
00401EFB   .  B8 00FFFFFF               mov eax,-100
00401F00   >  33C0                      xor eax,eax
00401F02   .  8945 F4                   mov dword ptr ss:[ebp-C],eax
00401F05   .  8945 F8                   mov dword ptr ss:[ebp-8],eax
00401F08   .  E8 63FCFFFF               call <final_or.anti_trap>
00401F0D   .  85C0                      test eax,eax
00401F0F   .  74 07                     je short final_or.00401F18
00401F11   .  6A FF                     push -1
00401F13   .  E8 C1B90000               call <final_or.EXIT>
00401F18   >  8B8E D0000000             mov ecx,dword ptr ds:[esi+D0]
00401F1E   .  2B8E C4000000             sub ecx,dword ptr ds:[esi+C4]
00401F24   .  8B86 C0000000             mov eax,dword ptr ds:[esi+C0]
00401F2A   .  038E B8000000             add ecx,dword ptr ds:[esi+B8]
00401F30   .  8D1440                    lea edx,dword ptr ds:[eax+eax*2]
00401F33   .  8D044A                    lea eax,dword ptr ds:[edx+ecx*2]
00401F36   .  2B86 CC000000             sub eax,dword ptr ds:[esi+CC]
00401F3C   .  8B4E 34                   mov ecx,dword ptr ds:[esi+34]
00401F3F   .  2B86 BC000000             sub eax,dword ptr ds:[esi+BC]
00401F45   .  0386 C8000000             add eax,dword ptr ds:[esi+C8]
00401F4B   .  83F9 01                   cmp ecx,1
00401F4E   .  8945 FC                   mov dword ptr ss:[ebp-4],eax
00401F51   .  75 2D                     jnz short final_or.00401F80
00401F53   .  DB45 FC                   fild dword ptr ss:[ebp-4]
00401F56   .  8BC8                      mov ecx,eax
00401F58   .  0FAFC8                    imul ecx,eax
00401F5B   .  DC0D C0844200             fmul qword ptr ds:[4284C0]
00401F61   .  DC2D B8844200             fsubr qword ptr ds:[4284B8]
00401F67   .  894D FC                   mov dword ptr ss:[ebp-4],ecx
00401F6A   .  DB45 FC                   fild dword ptr ss:[ebp-4]
00401F6D   .  DC0D B0844200             fmul qword ptr ds:[4284B0]
00401F73   .  DEC1                      faddp st(1),st
00401F75   .  DB46 30                   fild dword ptr ds:[esi+30]
00401F78   .  DC0D A8844200             fmul qword ptr ds:[4284A8]
00401F7E   .  EB 30                     jmp short final_or.00401FB0
00401F80   >  83F9 02                   cmp ecx,2
00401F83   .  75 30                     jnz short final_or.00401FB5
00401F85   .  DB45 FC                   fild dword ptr ss:[ebp-4]
00401F88   .  8BD0                      mov edx,eax
00401F8A   .  0FAFD0                    imul edx,eax
00401F8D   .  DC0D A0844200             fmul qword ptr ds:[4284A0]
00401F93   .  DC2D 98844200             fsubr qword ptr ds:[428498]
```

```
00401F99   .  8955 FC                    mov dword ptr ss:[ebp-4],edx
00401F9C   .  DB45 FC                    fild dword ptr ss:[ebp-4]
00401F9F   .  DC0D 90844200              fmul qword ptr ds:[428490]
00401FA5   .  DEC1                       faddp st(1),st
00401FA7   .  DB46 30                    fild dword ptr ds:[esi+30]
00401FAA   .  DC0D 88844200              fmul qword ptr ds:[428488]
00401FB0   >  DEE9                       fsubp st(1),st
00401FB2   .  DD5D F4                    fstp qword ptr ss:[ebp-C]
00401FB5   >  DB05 B8EB4200              fild dword ptr ds:[42EBB8]
00401FBB   .  8BCE                       mov ecx,esi
00401FBD   .  DC75 F4                    fdiv qword ptr ss:[ebp-C]
00401FC0   .  DC05 B0EB4200              fadd qword ptr ds:[42EBB0]
00401FC6   .  DC25 40844200              fsub qword ptr ds:[428440]
00401FCC   .  DD9E 98000000              fstp qword ptr ds:[esi+98]
00401FD2   .  E8 49FEFFFF                call final_or.00401E20
00401FD7   .  5E                         pop esi
00401FD8   .  8BE5                       mov esp,ebp
00401FDA   .  5D                         pop ebp
00401FDB   .  C3                         retn
```

This can be translated to:

```
double formula(void)
{
      double v2 = 0;
      int v1 = 3*d3 + 2*(d1 - d2 + d4) - d5 - d7 + d8;

      if (d6 == 1)      // true in this case
            v2 = g2 - v1 * g1 + v1*v1 * g3 - d9 * g4;
      else
      {
            if (d6 == 2)
                  v2 = g9 - v1 * g8 + v1*v1 * g10 - d9 * g11;
      }

      return g5 / v2 + g6 - g7;
}

// these are object data
d1 = 5;
d2 = 4;
d3 = 8;
d4 = 10;
d5 = 10;
d6 = 1;
d7 = 17;
d8 = 6;
d9 = 35;

// these are global variables
g1 = 0.00045719;
g2 = 1.21721;
g3 = 6.7e-07;
g4 = 0.00025696;
g5 = 510;
g6 = 0;
g7 = 485;

// and two local variables
v1 = 25;
```

```
v2 = 1.1972054;

x = 510 / 1.1972054 - 485; // -59.00793548041129784412933653657l;
```

## Removing the input limit

Now we need to remove input limitation. Using some Zen thinking ;) we can come to conclusion, that all encrypted code blocks are vital to the task: password protection, the formula… and, input limit? :) This is indeed true, but author initially took another approach. All global data of application was visually scanned in IDA in search of 200.0 constant that would be used in a comparison. Patched executable with all code decrypted was used. “data.txt” string was used to find rough position of private data. Although “200.0” was not found, another constant was:

```
.rdata:004284E8 dbl_4284E8      dq 1.9999999e2          ; DATA XREF: _main+50A↑r
```

Seems very interesting. Let’s examine that cross reference:

```
.text:00402F6A                  fld     ds:dbl_4284E8
.text:00402F70                  add     esp, 4
.text:00402F73                  fcomp   [ebp+78h+var_30]
.text:00402F76                  fnstsw  ax
.text:00402F78                  test    ah, 5
.text:00402F7B                  jp      short loc_402F8B
.text:00402F7D                  mov     dword ptr [ebp+78h+var_30], 0EB074A77h
.text:00402F84                  mov     dword ptr [ebp+78h+var_30+4], 4068FFFFh
.text:00402F8B
.text:00402F8B loc_402F8B:                             ; CODE XREF: _main+51B↑j
```

Magic constants being loaded to a stack variable are 64bit double representation of **199.99999**. Do we need more? ;) We patch jump at **`402f7b`** to unconditional. This results in input limitation being removed.

**Additional notes.**
There is a third anti-debug function using `int 2d`, but it was left alone. Author also found a few tricks that apparently targeted “cheating” OllyDbg plugins:

- Writing some large value to `BeingDebugged` PEB flag and reading it again later
- Using `Sleep`() to verify if `GetTickCount` returns reasonable values.

Also, initially author used WinDbg in kernel mode on a vmware target to catch all int2d/ trap flag tricks.

## *Time to break*

In total, about a day was required to achieve all objectives. Finding the password was most time-consuming initially, it took one evening (about 4-6 hours). After it has been understood how to enter the password, things got easier. Not all anti-debug protections were found though. Password algorithm was trivial, computing valid password took few minutes. Decrypting executable was easy, maybe half an hour for static patch. Decoding formula took about an hour or two, to not make any mistakes. Finding and patching the input limitation after decryption has been done was also very easy, it took 30 minutes at most. Most of the time was spent on finding “silent” anti-debug checks that corrupted data but didn’t kill the program. There was too few of such checks without debugger though.