
| ***Author: ReWolf*** |
| ***e-mail: rewolf@rewolf.pl*** |
| ***www   : http://rewolf.pl*** |

HACKER CHALLENGE 2008 Phase 1 Report

1. Background

This year **Hacker Challenge Phase 1** was pretty harder than the last year. It took me **5,5 hours** to remove all limits and reverse engineer the formula. Protected software contain two encrypted code blocks, few self-checks and few anti-debug tricks, all those nuisances can be easily defeated, what you will see during further reading of this report. To start the proper challenge you need to find the password, which is a bit complicated, because it looks like **SHA-256(password + 'salt')** and it is probably irreversible. Successfully patched program should draw graph of the three sinusoidal functions and generate file **data.out** identical to given **final.results**.

2. Attack Narrative

- Decrypting encrypted blocks

Encrypted blocks can be easily found in **IDA**, because encryption is done on the particular block of functions level. **IDA** will not recognize any functions in encrypted block and it will be marked as **'data'**. Encrypted blocks of code are placed at:

- *0x00401180 - 0x00401DC0*
- *0x00403930 - 0x00403F50*

Illustration 1: IDA Navigation Graph (blue color - code; gray color - data)

Now we can search for code that reference those addresses. We should be here:

```
.text:00404029 push    offset _WinMain@16        ; 00403F50
.text:0040402E lea     eax, [ebp+var_44]
.text:00404031 push    offset _windowProc        ; 00403930
.text:00404036 push    eax ; int
.text:00404037 call    _decryptCode              ; 00402B60
.text:0040403C push    offset sub_401DC0         ; int
.text:00404041 push    offset _recur_sub_401180 ; lpAddress
```

```
.text:00404046 push     edi ; int
.text:00404047 call     _decryptCode            ; 00402B60
```

As you can see, function ***_decryptCode*** (function address **0x00402B60**) takes three parameters:

void __cdecl _decryptCode(BYTE* key, BYTE* beginAddress, BYTE* endAddress);

Basically ***_decryptCode*** is the **Rijndael** cipher implementation, we can recognize it be 'magic' values (tables) used to decipher data, or through the **PEiD** plugin called **Krypto ANALyzer** (**KANAL**). **Key** for the first buffer (**0x00403930** - **0x00403F50**) is constant, and **256 bits long**:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Second buffer is encrypted with the key generated as a **SHA-256** hash calculated from the first buffer (before decryption) and it should be equal to (**256 bits long**):

```
7C 1B 8C 42 6D 98 08 15 25 7D 43 BD E4 F8 6F 36
58 DE 12 80 F0 B5 27 D9 50 A7 96 C6 BB ED 95 FA
```

After decryption **IDA** can recognize few more functions:

Function name	Segment	Start	Length	R	F	L	S	B	T	=
sub_401000	.text	00401000	0000017C	R	.	.	.	.	.	.
_recur_sub_401180	.text	00401180	000001DE	R	.	.	.	B	T	.
sub_401360	.text	00401360	00000011	R	.	.	.	.	.	.
_anti	.text	00401380	00000044	R	.	.	.	B	.	.
Function_H?_	.text	004013D0	0000011B	R	.	.	.	B	T	.
Function_G?_	.text	004014F0	000001BE	R	.	.	.	B	T	.
sub_4016B0	.text	004016B0	000000AA	R	.	.	.	B	T	.
sub_401760	.text	00401760	0000008A	R	.	.	.	B	T	.
Function_F_	.text	004017F0	00000592	R	.	.	.	B	T	.
sub_401D90	.text	00401D90	0000002C	R	.	.	.	.	T	.
sub_401DC0	.text	00401DC0	000004BF	R	.	.	.	.	.	.
sub_402280	.text	00402280	000002F7	R	.	.	.	B	.	.
_rijandael	.text	00402580	000005DB	R	.	.	.	.	.	.
_decryptCode	.text	00402B60	000000A7	R	.	.	.	.	T	.
sub_402C10	.text	00402C10	00000027	R	.	.	.	.	.	.
sub_402C40	.text	00402C40	00000497	R	.	.	.	B	T	.
_initEncryptionBuffer	.text	004030E0	000000CA	R	.	.	.	.	T	.
_hash	.text	004031B0	0000077C	R	.	.	.	.	T	.
_windowProc	.text	00403930	0000006F	R	.	.	.	.	T	.
sub_4039A0	.text	004039A0	000005A1	R	.	.	.	B	.	.
WinMain(x,x,x,x)	.text	00403F50	00000272	R	.	.	.	B	T	.
operator delete(void *)	.text	004041D0	00000006	R	.	.	.	.	T	.
_except_handler3	.text	004041D6	00000006	R	.	.	.	.	.	.
__alloca_probe	.text	004041E0	0000002B	R	.	L	.	.	.	.
operator new(uint)	.text	0040420C	00000006	R	.	.	.	.	T	.
__ftol2_sse	.text	00404220	000000AB	R	.	L	.	.	.	.
memset	.text	004042CC	00000006	R	.	.	.	.	T	.
memcpy	.text	004042D2	00000006	R	.	.	.	.	T	.
__allmul	.text	004042E0	00000034	R	.	L	.	.	.	.
___tmainCRTStartup	.text	0040435F	000001DF	R	.	L	S	B	.	.

Illustration 2: IDA Functions list window, selected functions placed in decrypted blocks.

- Removing Anti-Debug and Anti-Tamper Tricks

In this section I will describe all anti-debug and anti-tamper tricks that I found in the executable. I'll do it on the each function basis.

1. Function at 0x00401000:

This function calculates **SHA-256** hash of the **_WinMain@16** (**0x00403F5**, *0x280 bytes long*) function. If hash is different than hash stored in executable, function will overwrite values in table at address **0x00408838**. This table is used later in the **H function** (**0x004013D0**) to calculate final math formula. Original **SHA-256** hash should be equal to:

```
B5 20 B3 11 78 A7 F1 C1 7D B7 EC 5F 04 9F DD 77
C4 A1 FD 0D 26 99 24 88 FA 5E 84 66 2F 7C 49 86
```

Solutions:

- Patch ***conditional jump*** (**jle**) at address **0x00401141** to ***unconditional jump***
- Patch stored in executable hash to the new one. Code responsible for filling table with hash is placed at **0x0040101B**. I've used this solution.

2. Function at 0x00401380:

This function sets **SEH** handler and **Trap Flag**, under debugger **SEH** handler will not be called and function return **1** in **EAX**. **SEH** handler is responsible for setting **EAX** to **0**. This '*anti*' is used in **function G**, if it will detect debugger it will modify (set to **0**) one of the arguments passed to **G** (**0x004014F0**).

Solution:

- Patch function at **0x00401380** to always return **0**.

3. Function at 0x004013D0 (H function):

This function contain very tricky check. On each call it checks if one byte from the code section is equal to **0xCC** (**int3**, ***breakpoint***). In fact it counts all occurrences of **0xCC** byte in the code section, if it is more than **0xE8** it will modify sign (**fchs**) of one of the **H function** arguments.

Solutions:

- Patch ***conditional jump*** (**jbe**) at address **0x00401413** to ***unconditional jump***
- Don't do anything, this check only affects ***software breakpoints***

(**int3**, **0xCC**), so in final executable **0xCC** counter will be ok. Under debugger we can use ***Hardware Breakpoints***.

4. Function at 0x004017F0 (F function)

This function checks **DebugFlag** in ***Process Environment Block*** (**PEB**):

```
.text:00401902 mov     eax, large fs:30h
.text:00401908 movzx   eax, byte ptr [eax+2]
.text:0040190C and     eax, 0FFh
.text:00401911 mov     [ebp+64h+var_4], eax
```

Solution:

- Set **DebugFlag** in **PEB** to 0 or use **OllyAdvanced PlugIn**

5. Function at 0x00402280:

This functions contain two *anti-debug* checks. First is based on **int3** handler, if we have attached debugger, and we will pass **int3** handling to the application, everything will be ok, in other case **AES**(**rijndael**) will use ***Encryption Tables*** instead of ***Decryption Tables***. Second check is based on **GetTickCount** function, if we have patched **GetTickCount** (to return constant value, or just increment on each execution), we will get error, because of too fast execution.

Solutions:

- pass **int3** to the application
- don't patch **GetTickCount** function
- ***trace over*** instead of ***tracing into*** this function

6. Function at 0x00403F50 (_WinMain@16)

This function checks first byte of **SHA-256** hash generated from the first encrypted buffer (described in section *'Decrypting encrypted blocks'*), this byte should be equal to **0x7C**.

Solutions:

- patch ***comparison*** (**cmp**) at **0x00404016** with correct new value
- patch ***conditional jump*** (***jz***) at **0x00404019** to ***unconditional***

- Defeating password protection

Defeating password protection was pretty confusing. I still don't have the proper password, but as long as it is ***not the objective*** I will not bother to find it. Password should be placed on the ***first line*** of the **data.in** file and it can be ***up to 8 characters length***. 8 bytes password buffer is concatenated with string ***'salt'*** and passed to **SHA-256** function. Obvious solution is a **SHA-256** 12-chars ***brute-force***, with four characters constant, but it would take to long to find out correct password. In fact I didn't checked if it is an original **SHA-256**, so I cannot claim that it is irreversible. Hashed password is compared to:

```
09 0A 89 6D 12 27 D0 03 75 0F A2 46 EF F0 2C 1E
92 33 2C 5C 6F FF 36 D8 74 2E 79 B9 E0 EB A0 A9
```

```
.text:004039B5 mov     [ebp+var_28], 6D890A09h
.text:004039BC mov     [ebp+var_24], 3D02712h
.text:004039C3 mov     [ebp+var_20], 46A20F75h
.text:004039CA mov     [ebp+var_1C], 1E2CF0EFh
.text:004039D1 mov     [ebp+var_18], 5C2C3392h
.text:004039D8 mov     [ebp+var_14], 0D836FF6Fh
.text:004039DF mov     [ebp+var_10], 0B9792E74h
.text:004039E6 mov     [ebp+var_C], 0A9A0EBE0h
```

I've changed it to:

```
B6 43 42 83 49 D7 8B 0B E7 B2 A4 51 75 DF 86 34
BA 40 C0 20 E0 7D 5D 77 B2 ED 5D 3D 1B 07 BA E5
```

```
.text:004039B5 mov     [ebp+var_28], 834243B6h
.text:004039BC mov     [ebp+var_24], 0B8BD749h
.text:004039C3 mov     [ebp+var_20], 51A4B2E7h
.text:004039CA mov     [ebp+var_1C], 3486DF75h
.text:004039D1 mov     [ebp+var_18], 20C040BAh
.text:004039D8 mov     [ebp+var_14], 775D7DE0h
.text:004039DF mov     [ebp+var_10], 3D5DEDB2h
.text:004039E6 mov     [ebp+var_C], 0E5BA071Bh
```

So my password in **data.in** is already equal to ***'password'***.

- Reverse engineer the mathematical formula (Objective 1)

Objective 1 was the most time-consuming part this year. Locating **function H()** was pretty easy. At first I searched for **function F()**. My approach relied on searching all ***logarithm*** related **FPU** instructions in disassembly. I searched for phrase ***'fyl'*** and I have found three places where **FPU** instruction **fyl2x** was used:

```
.text:00401162 fyl2x
```

```
.text:00401428 fyl2x
```

```
.text:00401DB3 fyl2x
```

First occurrence is used in one of the *anti-debug* routines, second occurrence is used by **H() function**, finally third occurrence is a place that we are looking for:

```
.text:00401D90 mov      eax, [esp+arg_10]
.text:00401D94 fld      [esp+arg_8]
.text:00401D98 push     eax ; int
```

```
.text:00401D99 sub     esp, 10h
.text:00401D9C fstp    [esp+14h+var_C]
.text:00401DA0 fld     [esp+14h+arg_0]
.text:00401DA4 fstp    [esp+14h+var_14]
.text:00401DA7 call    Function_F_       ; call 0x004017F0
.text:00401DAC fldlg2
.text:00401DAE add     esp, 14h
.text:00401DB1 fxch    st(1)
.text:00401DB3 fyl2x
.text:00401DB5 fmul    ds:dbl_405288     ; dq 10.0
.text:00401DBB retn
```

We can now check all functions called by **F()**, below is the simple graph of functions tree, it lacks all **API** calls from **MSVCP80.dll** and one call to *anti-debug* routine.

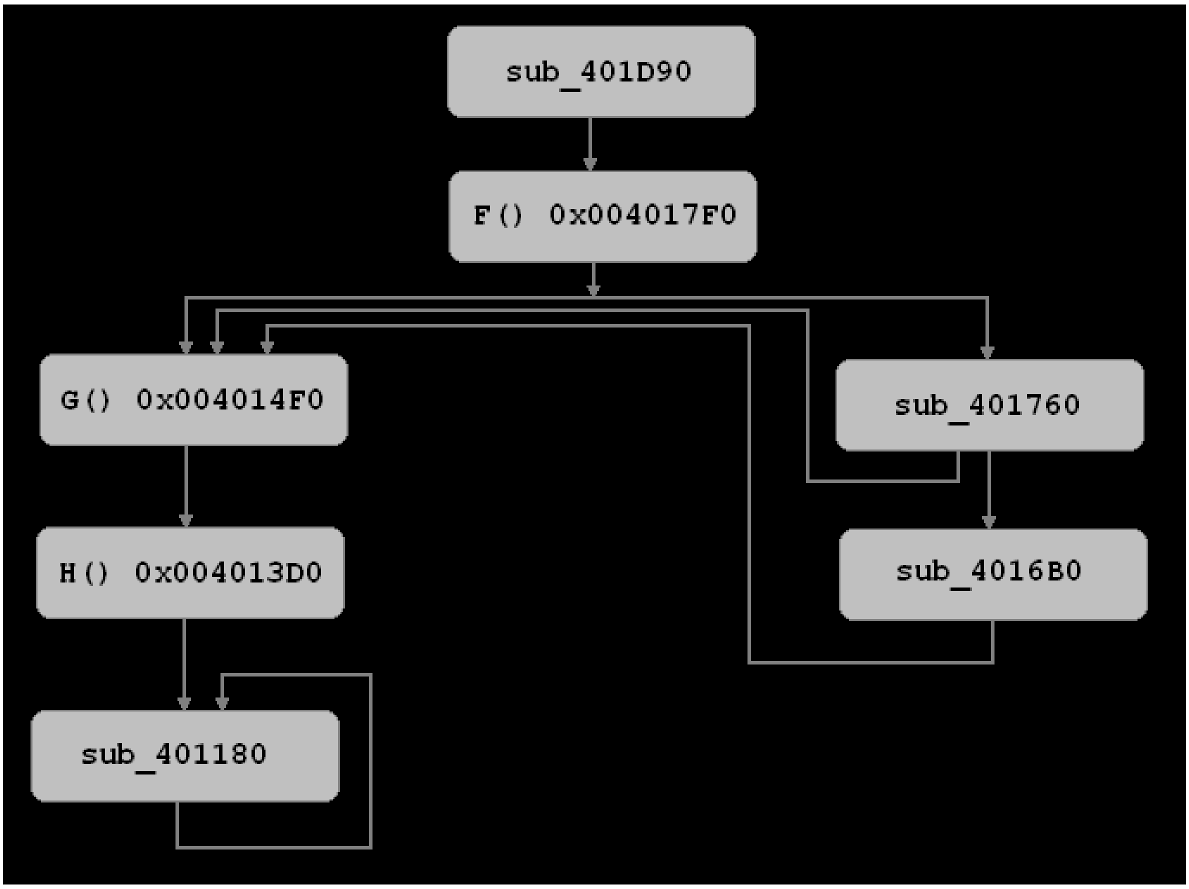

Illustration 3: Call-graph of functions related to mathematical formula.

As we can see on the graph, we have two candidates for **function G()**, first is at **0x004014F0** and second at **0x00401760**. We know that **function G()** have to iteratively call **function H()**, this condition eliminate function at **0x00401760**, because there is no loop inside this function. Final formula looks like this:

result = ((g2)^|p3|) * e^(ln(p2 * p2) * p3 - d1[p3] - ln(f_401180(p3+p1+g1)))

^ -> power
|a| -> abs(a)
ln -> natural logarithm

d1[] it is table with data

g1 = [0x00405238] -> dq 1.0
g2 = [0x00405248] -> dq -0.25

d1 = 0x00408838

Above formula needs a few explanations. First of all, original formula is a bit different, instead of ***ln(p2 * p2)*** there is ***ln(2)*log2(p2 * p2)***. We can change base of the ***log2***:

ln(2) * (ln(p2 * p2) / ln(2) = ln(p2 * p2)

This modification shouldn't affect final calculations. Rest of formula is pretty understandable and don't need further clarifications.

- Patch executable to remove some limits (Objective 2)

The easiest part of challenge, it takes only few minutes to patch all limits, removing self-checks is described in one of the previous chapters.

- ***The first value is a real number and is limited to a minimum of 2.0***: comparison is done at **0x00403B17**, we need to patch conditional jump (**jp**) at **0x00403B21** to unconditional jump.

- ***The second value is a real number and is limited to a maximum of 4.0***: comparison is done at **0x00403B89**, we need to patch conditional jump (**jnz**) at **0x00403B93** to unconditional jump.

- ***The third value is an integer and is limited to being less than 32***: our value is anded with **0x1F** (**and eax, 1Fh**) at address **0x00403BF8**, we can just **nop** this instruction (**0x90 0x90 0x90**).

- ***The fourth value is an integer and is limited to a maximum of 1***: comparison is done at **0x00403C32**, we need to patch conditional jump (**jle**) at **0x00403C35** to unconditional jump.

- ***The fifth value is an integer and is limited to 16***: comparison is done at **0x00403C86**, we need to patch conditional jump (**jle**) at **0x00403C89** to unconditional jump.

All those informations we can get from the simple trace of function **0x004039A0**, which is responsible for parsing **data.in** file.

3. Time to break

- **Removing encryption – 15 minutes**
- **Defeating anti-debug and anti-tamper tricks – 45 minutes**
- **Searching password – 30 minutes**
- **Reverse engineering mathematical formula – 3,5 hours**

- **Removing limits - 20 minutes**

- **Overall time - 5h 20m**

4. Tools used

- **OllyDbg 1.10 + Olly Advanced PlugIn**
- **IDA Pro Advanced 5.3**
- **PEiD + Krypto ANALyzer PlugIn**
- **Notepad**
- **Totalcmd**

5. Conclusions

This year hacker challenge phase 1 was really challenging, it doesn't mean that it was hard (but I'm still confused about few things in the mathematical formula). To get better protection authors should consider developing simple (well, maybe not so simple) obfuscator or code-morpher. It is always harder to reverse engineer obfuscated/morphed code. Executable should be encrypted with some multi-layer protector, with strong import table protection. I would also add more code to the target application just to confuse potential attacker. I really like self-checks used in the target application, but we saw similar tricks in the last year challenge, so it was rather easy to bypass them. Mathematical formula was pretty complicated this year, which is a big plus. I spent 5,5 hours to get all things working, so my final evaluation of the difficulty is medium.