Hacker Challenge 2008
Phase 3 Report

by ReWolf
rewolf@rewolf.pl
http://rewolf.pl

Table of Contents:

Background:

After 3 months of waiting, I had finally occasion to solve phase 3 binary of *Hacker Challenge 2008*. I have to say that this time it was pretty interesting (of course it doesn't mean that previous phase, and the last year challenge were not interesting), especially mathematical formula. Except reverse engineering the formula, I had to remove some limitations from the given application. Those two objectives were mandatory, in addition I had to bypass password protection for proper working of the application. Further research lead me to the finding that this time I have to deal not with only one password, but with two (small aggravation). Like in previous challenges, binary was partially encrypted and protected with few anti-debug and anti-patching tricks. After removing all limits and circumventing both passwords, protected application should generate file *data.out* identical to the given file *final.results* and print on the screen three sinusoidal curves.

Attack Narrative:

Decrypting encrypted blocks:

At first I decided to decrypt encrypted parts of code. Function responsible for decryption can be easily found by looking (in **IDA**) at the list of references to any of the encrypted blocks (except few blocks that aren't referenced at first glance).

Address of block	Size of Block	References
0x00401070	0x2D0	Not referenced
0x00401340	0x140	0x00404183, 0x0040418F, 0x0040425F, 0x0040426B
0x00401B80	0x5F0	0x0040412D, 0x00404139
0x004027E0	0x090	0x0040414B, 0x00404156
0x004030C0	0x7A0	0x0040411C, 0x004041A8, 0x004041B4, 0x004041D0, 0x00404241, 0x0040424D
0x00403860	0x030	0x0040431D, 0x00404326
0x00403890	0x4A0	0x00404116, 0x004042E4, 0x004042F0
0x00403D30	0x070	0x00404302, 0x0040430B

Interesting (from decryption point of view) are blocks placed at *0x00401B80*, *0x004027E0*, *0x00403860* and *0x00403D30*, because they have only two references. Looking at those references reveals the typical call to a decryption function:

```
mov    r32, offset_to_encrypted_buffer   ; first reference
add    r32, size_of_encrypted_buffer
push   r32
push   offset_to_encrypted_buffer        ; second reference
push   offset_to_key_buffer
call   rijndael_decrypt                  ; 0x00402750
```

In the C/C++ declaration of **rijndael_decrypt()** would look like this:

```
void __cdecl rijndael_decrypt(BYTE* key, BYTE* beginAddress, BYTE* endAddress);
```

Function **rijndael_decrypt** is placed at *0x00402750* and it is referenced seven times (for every block in the table that has references), sizes in the table are a bit different than arguments passed to the decrypt function, because they contain also alignment bytes **0xCC**, but I'll show the correct sizes at the and of this paragraph. The easiest way (at least for me) of decrypting those blocks is run executable under debugger, break on **WinMain** function and modify execution in that manner, that application will execute only parts of code responsible for decryption of the code. **WinMain** function is placed at *0x00403DA0*, to avoid exceptions on writing to the code section, I had to modify memory access under debugger, it can be also done through modification of **IMAGE_SECTION_HEADER** of code section in **PE** header in the executable. So called „execution modification" is nothing more than just changing EIP to point to the start of the call to the decryption routine, so I have to do it seven times:

Execution	
Start	**End**
0x0040412D	0x00404148
0x0040414B	0x00404165
0x00404183	0x0040419E
0x004041A8	0x004041C3
0x004042E4	0x004042FF
0x00404302	0x0040431A
0x0040431D	0x00404335

After execution of those blocks I've saved code section to the new executable file (*'Copy to executable'* option in **OllyDbg**). New file loaded to **IDA** showed me, that there is still one encrypted block, at address *0x00401070*. This block can be decrypted in similar way as previous blocks, but with one small exception, *'endAddress'* value is set some instructions before the proper call:

```
mov    r32, offset_to_encrypted_buffer   ; first reference
...
add    r32, size_of_encrypted_buffer
...
push   r32
push   offset_to_encrypted_buffer        ; second reference
push   offset_to_key_buffer
call   rijndael_decrypt                  ; 0x00402750
```

So, the execution table will be:

Execution	
Start	**End**
0x00401350	0x00401355

0x0040135F	0x00401365
0x004013D8	0x004013E8

After this I can save changes to my new executable and admire clean code without any encryption under **IDA** (of course at this time my executable will not work).

During inspection of encrypted blocks I noticed in a few places, code very similar to decryption, that references encrypted blocks. Further research showed, that few blocks are re-encrypted at runtime, probably to avoid dumping code from the application executed without debugger. Encryption function is placed at *0x004027E0* and has identical arguments as decryption:

```
void __cdecl rijndael_encrypt(BYTE* key, BYTE* beginAddress, BYTE* endAddress);
```

Blocks that can be re-encrypted:

- *0x004030C0*
- *0x00401340*
- *0x00401070*

To avoid decryption and encryption of already dumped blocks I've patched those two functions (**rijndael_decrypt()** and **rijndael_encrypt()**) to return immediately without doing anything.

Decrypting blocks summary		
Block address	**Block size**	**Decryption key**
0x00401070	0x2C3	59 1F 1B 77 77 0A 4B 8E DC B4 0C 32 E2 2F 59 AE D2 82 03 BB 89 B9 02 D2 6E AD AF 70 9D 81 6A B6
0x004030C0	0x79A	2E 60 71 0C 84 44 86 75 51 F7 E9 42 A2 56 08 25 59 3D 7F 06 D0 68 C4 C6 2C 73 C3 98 D0 2B 2E BF
0x00401B80	0x5E2	EC 41 35 CF 5C AA A1 13 20 60 44 D2 ED C6 65 28 08 70 E8 8A A9 74 02 B4 E2 CE F3 7B C2 7C A6 6C
0x004027E0	0x081	EC 41 35 CF 5C AA A1 13 20 60 44 D2 ED C6 65 28 08 70 E8 8A A9 74 02 B4 E2 CE F3 7B C2 7C A6 6C
0x00401340	0x137	EC 41 35 CF 5C AA A1 13 20 60 44 D2 ED C6 65 28 08 70 E8 8A A9 74 02 B4 E2 CE F3 7B C2 7C A6 6C
0x00403890	0x497	2E 60 71 0C 84 44 86 75 51 F7 E9 42 A2 56 08 25 59 3D 7F 06 D0 68 C4 C6 2C 73 C3 98 D0 2B 2E BF
0x00403D30	0x06F	EC 41 35 CF 5C AA A1 13 20 60 44 D2 ED C6 65 28 08 70 E8 8A A9 74 02 B4 E2 CE F3 7B C2 7C A6 6C
0x00403860	0x027	EC 41 35 CF 5C AA A1 13 20 60 44 D2 ED C6 65 28 08 70 E8 8A A9 74 02 B4 E2 CE F3 7B C2 7C A6 6C

Defeating first password:

During defeating first password I decided to patch given binary in the way that it will allow me to run it under debugger, so in this chapter I'll also describe some

of anti-debug and anti-patch tricks (more tricks will be described in *'Objective 2: Anti-tamper'* paragraph).

First trick calculates modified **SHA-256** hash from **WinMain** function:

```
BYTE specHash[] =
{
    0xA3, 0x52, 0x48, 0xFF, 0xD1, 0x61, 0xC6, 0x5B,
    0xA4, 0xDD, 0xF9, 0xB5, 0xCC, 0xB6, 0x35, 0xBE,
    0xC1, 0xDD, 0x99, 0x28, 0x0F, 0xF6, 0x72, 0x16,
    0x13, 0x9F, 0xC4, 0x68, 0x5B, 0x63, 0xAA, 0x49
};
BYTE* hashCtx = initHash(WinMain, 0x727, “”, 0);
BYTE* outHash = hash(hashCtx);
int i = 0;
int sum = 0;
while (i < 32)
{
    sum += specHash[i] ^ outHash[i];
    i++;
}
if (sum)
    MessageBoxA(0, "Corrupted binary.", 0, 0);
```

This trick can detect on-disk modification of **WinMain** function, or breakpoints set in **WinMain** during debugging. The simplest solution is to patch conditional jump at *0x00403F52* to unconditional.

After this check I've encountered very similar code that checks first password. First password should be passed to the application through command line:

```
final.exe secretpass
```

“secretpass” is concatenated with string *“drpepper”* and hashed with the previously mentioned modified **SHA-256** function. The result should be equal to:

```
FB 7B 2B 75 55 28 A6 81 38 59 37 EB 16 65 F2 38
CA 44 41 E6 57 C1 EA 0A A5 45 DF 6F 2E 24 47 38
```

Removing this check is as simple as previous, I've patched conditional jump at *0x004040A2* to unconditional. Successfully removed password protection lead me to the call to **OutputDebugStringA** with *"Hello there!\n"* as a parameter. After those modifications, executable refused to work, further research showed that there is a little problem with **OutputDebugStringA** and **GetLastError**. I didn't knew this trick before, but it looks like this trick works only on ***Windows XP x86***:

```
    push   offset_to_some_string
    call   OutputDebugStringA
    call   GetLastError
    cmp    eax, 2
    jz     _everything_ok
    ;
    ;code executed if debugger detected
    _everything_ok:
```

On ***Windows XP x64*** and ***Vista x86*** this trick will always detect debugger, even if we don't have such evil thing. Solution for this problem will be small patch (as always), I've changed this conditional jump (jz) to unconditional. This trick is used two times, so I need to patch this jump at *0x004040DF* and *0x0040143F*. First usage is placed in **WinMain** function and in case of detecting debugger it overwrites body of the function (*0x004030C0*) responsible for reading file *'data.in'* with body of the function from *0x00403890* (function draws graph on the screen). Second usage is in function at address *0x00401340* and exit from application. Now application prints the graph identical to the one from *'instructions-phase3.pdf'* and produces empty file *'data.out'*.

Objective 2: Anti-tamper

Further tracing of the binary showed few more anti-debug tricks. Due to improper handling of **Int 2D** instruction under debugger, I've patched it to UD2 instruction (*0x0F0B*). **Int 2D** was used two times, first in **WinMain** function at address *0x0040416F*, and second at *0x004010A8* (**function F**). Under debugger **Int 2D** will not cause exception. In **WinMain** function, if exception handler is not called, function at *0x401340* will not be decrypted. In **function F** exception handler is responsible for all calculations related to the mathematical formula that have to be reverse engineered.

Another anti-debug is placed at *0x004041C6*:

```
004041C6   CALL    KERNEL32.IsDebuggerPresent
004041CC   TEST    EAX, EAX
004041CE   JE      SHORT final4.004041D8
004041D0   MOV     EAX, final4.004030C0
004041D5   MOV     BYTE PTR DS:[EAX], 0C3
```

This is rather desperate check, because probably everyone has patched **IsDebuggerPresent** function (or field in **PEB**), but if not, then in case of detecting debugger application will put ret instruction (**0xC3**) at the beginning of the function at *0x004030C0*. Checking **IsDebuggerPresent** return value is used once more, during processing data collected from *'data.in'* file.

After **IsDebuggerPresent** there is one more timing-based anti-debug trick:

```
004041D8   CALL    KERNEL32.GetTickCount
004041DE   SUB     EAX, DWORD PTR SS:[EBP-4C0]
004041E4   MOV     DWORD PTR SS:[EBP-42C], EAX
004041EA   CMP     DWORD PTR SS:[EBP-42C], 0C8       ; 200
004041F4   JB      SHORT final4.00404202
004041F6   CMP     DWORD PTR SS:[EBP-42C], 7D0       ; 2000
00404200   JBE     SHORT final4.00404209
00404202   XOR     EAX,EAX
00404204   JMP     final4.004044B4
```

It measures execution time of the block of code and if value is outside 200-2000 ticks range application will exit. For the ease of debugging I've patched conditional jump at *0x004041F4* to unconditional jump and changed destination of this jump from

0x00404202 to *0x00404209*.

Before call to the function that will read *'data.in'* file there is one more anti-debug trick. Setting of this trick is done at the beginning of the **WinMain** function:

```
BYTE* addr = VirtualAlloc(0,
                          SystemInfo.dwPageSize,
                          MEM_RESERVE|MEM_COMMIT,
                          PAGE_EXECUTE_READWRITE);
addr[0] = 0xC3u;
VirtualProtect(addr,
               SystemInfo.dwPageSize,
               PAGE_GUARD|PAGE_EXECUTE_READWRITE,
               &flOldProtect);
```

This code allocates one page of virtual memory with the execution rights, puts ret instruction (**0xC3**) in this memory, and sets **PAGE_GUARD** protection on this memory page. In the middle of the **WinMain** function there is jump to that memory, which should trigger exception handler, but under debugger this exception is handled by debugger (actually I didn't bother myself if **OllyDbg** can pass this exception to the application). If exception handler is not called given binary skips call to the function that loads 'data.in' file. My first solution was rather ugly, but it works. I've patched instruction at *0x00404216*:

Original	Patched
`push offset loc_404241`	`jmp short loc_40422F`

It solves all problems without throwing exception. After few days I 'googled' that it can be done easier. During setting this trick, I could change **ret** instruction to **int3** or any other code that will generate exception:

```
addr[0] = 0xCCu;
```

Finally I'm now on the call to the function that reads data from *'data.in'* (address **0x004030C0**). At first, function opens file *'data.in'*, then it counts all occurrences of **0xCC** in the function body:

```
BYTE* funcAddr = 0x004030C0;
int i = 0;
int occurrs = 0;
do
{
  BYTE cByte = funcAddr[i] ^ 0xDE;
  if ( cByte == 0x12 )
      occurrs++;
}
while ( i < 0x79A );
if (occurrs != 0x10)
    goto _end_of_function;
```

If there is more than **0x10** occurrences of **0xCC** byte it means that someone set breakpoint on the checked code. For me it was very comfortable to 'nop' this detection,

so I've 'nopped' conditional jump at address *0x00403132*. File *'data.in'* is processed by sequential calls to **fgets()** function, gathered lines are converted through **atoi()** or **atof()** functions, but there is one exception to this. For the value from fourth line application calls function **isdigit()** on the first character from line. If the first character is not digit, whole line is concatenated with *"mrsdash"* string and modified **SHA-256** is calculated. Calculated hash should be equal to:

```
D2 F1 EB 1B C3 FF 5B 72 76 7D 51 0A D0 41 39 3B
B3 0D 06 36 5E D2 81 18 5D 68 8D 2B 4A 97 9B 7B
```

If this password is not set properly, application generates empty *'data.out'* file, to avoid this I've patched binary at address *0x004032E7*:

Original	Patched
`lea esi, [ebx+eax]`	`xor esi, esi` `nop`

Now I can describe ho to remove limits mentioned in instructions:

1. The first value is a real number and is limited to a minimum of around 1.4:

```
00403379   FLD     QWORD PTR DS:[4051D0] ; equal 2.0
0040337F   FLD     ST
00403386   FSQRT
0040338B   FCOM    QWORD PTR DS:[409150] ; value from the first line
00403391   FSTSW   AX
00403393   TEST    AH,41
00403396   JNZ     SHORT final6.004033A0 ; jump if value higher than sqrt(2.0)
00403398   FSTP    QWORD PTR DS:[409150] ; else store minimal value = sqrt(2.0)
0040339E   JMP     SHORT final6.004033A2
004033A0   FSTP    ST
004033A2
```

The exact limit is equal to sqrt(2.0). To remove this limit I've patched conditional jump at *0x00403396* to unconditional jump.

2. The second value is a real number and is limited to a maximum of around 4.9:

```
004033A2   FLD     QWORD PTR DS:[405270] ; equal 3.14
004033A8   FXCH    ST(1)                 ; ST0 = 2.0 from the previous operation
004033AA   CALL    <JMP.&MSVCR80._CIpow> ; pow(3.14, 2.0)
004033AF   FMUL    QWORD PTR DS:[4051B8] ; equal 0.5
004033B5   FCOM    QWORD PTR DS:[409158] ; value from the second line
004033BB   FSTSW   AX
004033BD   TEST    AH,5
004033C0   JPE     SHORT final6.004033CA ; jump if value lower than 4.9...
004033C2   FSTP    QWORD PTR DS:[409158] ; else set minimal value to 4.9...
004033C8   JMP     SHORT final6.004033CC
004033CA   FSTP    ST
004033CC
```

The exact limit is equal to **pow(3.14, 2.0)/0.5**. To remove this limit I've patched conditional jump at *0x004033C0* to unconditional jump.

3. The third value is an integer and is limited to being less than 64:

```
00403491   MOV     EAX,DWORD PTR DS:[409160] ; value from the third line
0040349F   AND     EAX,3F                    ; 63 decimal
```

To remove this limit I've 'nopped' an and instruction at *0x0040349F*.

Before removing the last limit I had to bypass one more self-checking code. At *0x004034F4* there is code that calculates modified **SHA-256** from the current function, calculated hash is used then in some calculations and if it is not equal to values below it changes some initial values of the future calculations. Proper hash:

```
CC 7C 9B 8E FF 3C 2B 55 27 23 6C 2E 6F 84 09 26
70 80 0D 50 02 08 24 EF 76 77 55 17 59 75 EE 25
```

Patch to support this checksum will be showed in the table with summarized all patches (at the end of this paragraph).

4. The sixth through eighth values are real and are limited to being greater than around 0.2:

```
00401355   FLD     QWORD PTR DS:[4051F8] ; equal 0.2058008
0040135C   FLD     QWORD PTR SS:[EBP+8]  ; input value to check
00401365   FCOM    ST(1)
00401368   FSTSW   AX
0040136A   TEST    AH,5
0040136D   JPE     SHORT final6.00401376 ; jump if value greater than 0.2058008
0040136F   FSTP    ST
00401371   FST     QWORD PTR SS:[EBP+8]  ; else store 0.2058008
00401374   JMP     SHORT final6.00401378
00401376   FSTP    ST(1)
00401378
```

To remove this limit, I've patched conditional jump at *0x0040136D* to unconditional jump.

After removing all limits, application generated file *'data.out'*, but with wrong values, also graph printed on the screen don't look so good (it is green line at the top of the window). It looks that I'm still missing few self-checks.

Those missed checks are placed in **F function**. First check calculates modified **SHA-256** hash from the body of **F function**. Two double-words are taken from hash and used to initialize one of quad-word used in further calculations. The solution is rather simple, I've gathered those two values from hash from the original executable and patched code at *0x004010FB* to:

```
004010FB     MOV     DWORD PTR SS:[EBP+54],E8584CAA
00401102     NOP
00401103     NOP
00401104     NOP
00401105     NOP
```

```
00401106     NOP
00401107     MOV     DWORD PTR SS:[EBP+58],400BB67A
0040110E     NOP
0040110F     NOP
00401110     NOP
00401111     NOP
00401112     NOP
```

Second missed check is placed at *0x004012F3* and counts how many times byte **0xCC** occurs in the **F function**:

```
BYTE* funcAddr = 0x00401070;
int i = 0;
int occurrs = 0;
do
{
  BYTE cByte = funcAddr[i] ^ 0x64;
  if ( cByte == 0xA8 )
      occurrs++;
}
while ( i < 0x2C3 );
if (occurrs != 2)
    goto _exit;
```

Patching conditional jump at *0x00401322* to unconditional solves the problem. Now I've fully working executable with proper output (graph and file).

Below table summarizes patches that were done to the application, except patches for encrypted blocks, which were discussed earlier.

Address	Size	New code	Reason
0x4010A8	0x02	UD2	Int 2D changed to UD2 for better exception handling under debugger.
0x4010FB	0x18	MOV DWORD [EBP+54],E8584CAA 5xNOP MOV DWORD [EBP+58],400BB67A 5xNOP	Setting proper initialization values in function F, to avoid using wrong generated hash.
0x401322	0x06	JMP 004010B2	Removes detection of 0xCC breakpoints in F function.
0x40136D	0x02	JMP 00401376	Removes limits from the sixth through eighth values in 'data.in' file.
0x40143F	0x02	JMP 00401449	Patch for OutputDebugStringA/ GetLastError trick. It is mandatory to run binary on systems other than Windows XP x86.
0x403132	0x06	6xNOP	Removes detection of 0xCC breakpoints in function at 0x004030C0.
0x4032E7	0x03	XOR ESI,ESI NOP	Patch for second password check.
0x403396	0x02	JMP 004033A0	Removes limit from first value

			in 'data.in' file.
0x4033C0	0x02	JMP 004033CA	Removes limit from second value in 'data.in' file.
0x40349F	0x03	3xNOP	Removes limit from third value in 'data.in' file.

0x403528	0x06	XOR EAX,EAX
		INC EAX
		MOV ECX,ESI
		MOV ESI,004088D8
			Setting offset to correct hash of function at 0x004030C0 instead of using hash generated at runtime.

0x40353C	0x02	JMP 00403544	
0x4035A1	0x01	PUSH ECX	
0x4035AB	0x01	PUSH EDI	
0x40378B	0x06	6xNOP	Removes one of the time-based checks.
0x403E0F	0x03	MOV BYTE [EAX],CC	Putting int3 instead of ret instruction in the PAGE_GUARD memory.
0x403F52	0x02	JMP 00403F70	Removes hash-based self-check from WinMain function.
0x4040A2	0x02	JMP 004040CB	Removes first password check.
0x4040DF	0x02	JMP 0040412D	Patch for OutputDebugStringA/ GetLastError trick.
0x40416F	0x02	UD2	Int 2D changed to UD2 for better exception handling under debugger.
0x4041F4	0x02	JMP 00404209	Patch for another time-based check.

Objective 1: Reverse Engineering a Fromula:

Reverse engineering a formula was as usual very challenging task. Locating **function F()** wasn't hard, searching for **fyl2x FPU** instruction gave me only one result:

```
0040140D   call    _F_      ;call 0x00401070
00401412   fldlg2
00401414   add     esp, 18h
00401417   fxch    st(1)
0040141E   fyl2x
```

The main part of formula is placed in the **SEH** handler that should be triggered by **Int 2D** command. **Function F()** takes three double parameters: **p1**, **p2** and **p3**. There are three global values:

Address	Name	Value
0x004051C0	G1	1000000000.0
0x004051E8	G2	299792458.0
0x004051D8	G3	0.25

There is also one value constructed from the hash of the body of **F function**:

```
d1 = 3.464101615137754
```

Function F() calls two other functions which are well defined operations on complex numbers: **complex_multiply** at *0x00401000* and **complex_divide** at *0x00401030*. The initial calculations are rather easy, at first I've defined complex **value A**:

$$A=\left(\frac{1}{\frac{G2}{p2*G1}}*d1*p1*p3;\ 0\right)=\left(\frac{p2*G1*d1*p1*p3}{G2};\ 0\right)$$

Next I've defined sequence of complex numbers:

$$a_n=\begin{Bmatrix} a_0=(0,1) \\ a_1=(1,0) \\ a_n=\frac{(2*n-1)*a_{n-1}}{A}-a_{n-2},\ for\ n\geqslant 2 \end{Bmatrix}$$

Using sequence a_n I've defined sequence b_n :

$$b_n=\frac{(-1)^n*(2*n+1)}{n*a_{n+1}^2-A*a_{n+1}*a_n}$$

Now I can define **function f**:

$$f(A)=\sum_{n=0,\,n\in\mathbb{N}}^{\infty} b_n=(x,y)$$

In the given application, summation of b_n is done until precision will reach 1.0e-12. f**(A)** produces complex value that is converted to real **value C** through equation:

$$C=(x^2+y^2)*p1^2*p3*G3*d1$$

Value C is returned from **function F**.

Time to break:

- Overall time: **2 days**
- Breaking password and objective 2: **3 to 4 hour**s
- Reverse engineering a formula: **8 to 10 hours**

Tools used:

- ***IDA Pro Advanced*** – overall static analysis of executable
- ***OllyDbg 1.10*** with ***Olly Advanced*** plug-in – dynamic analysis and patching executable

- ***Notepad*** – quick notes and ideas to check
- ***Calc*** – irreplaceable tool for any calculations

Conclusion:

I must say that complication of mathematical formula surprised me, but at all everything can be reversed. Anti-debug tricks were rather usual and well known, except this OutputDebugStringA/GetLastError, which wasn't good idea, because it not worked as it should. Partial encryption and decryption of blocks of code at runtime is a step in good direction, but commercial packers used such technique few years ago, so it is nothing new, and nothing hard to bypass. Actually the best solution to protect code from reverse engineering is morphing or virtualizing code like for example in Themida. Calculating checksums and searching for breakpoints at runtime is good idea, especially if application don't report anything to the attacker, but silently modify execution of the program. Such tricks are usually very hard to track in big commercial applications. Using passwords to run application is also good choice, but it has sense only if those passwords are crucial for application execution (or decryption), it is of course understandable that it doesn't make sense in challenge that is supposed to be beaten. Anyhow, I'm greatly appreciated that I've occasion to solve Hacker Challenge once again and I'm waiting for the next challenge.