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


HACKER CHALLENGE 2007 Phase 1 Report

1. Background

In this report I will describe protection scheme of **Hacker Challenge** first phase binary. To defeat protection we have to pass through four layers:

- unpack custom **PE encrypter** (easy)
- generate **keyfile** "password.txt"
- reverse engineer the **mathematical formula** (objective 1)
- patch executable to extend some functionality (objective 2)

Also we have to patch some **CRC checks**, and disable some **anti-debugging code** (***GetTickCount***, ***IsDebuggerPresent***).

2. Attack Narrative

- Removing cutsom PE encrypter

Target is protected with custom **PE encrypter**, it doesn't encrypt all sections of executable, it not support **Import Table** protection, there is no **stolen bytes** or **code redirection.** Just simple encryption layer. Each byte of protected section is decrypted with algorithm like this:

```
        add     al, 10h
        xor     al, 53h
        ror     al, 0BDh
        add     al, 0AFh
        sub     al, 1Fh
        add     al, 0A0h
        add     al, 0Fh
```

In fact there is six similar decryption routines (generated with some simple **polymorphic engine**). Selection of algorithm depends on section name:

'.tex', 'CODE':

```
          add     al, 10h
          xor     al, 53h
          ror     al, 0BDh
          add     al, 0AFh
          sub     al, 1Fh
          add     al, 0A0h
```

```
          add     al, 0Fh
```

'**.dat**', '**DATA**':

```
          add     al, 6Dh
          add     al, 88h
          rol     al, 0D5h
          add     al, 1Bh
          add     al, cl
          sub     al, 0BBh
          add     al, cl
          rol     al, 5Dh
          sub     al, cl
          ror     al, 0D7h
```

'**BSS**': it's common name for **unitialized data** section, for this section there is fake empty decryption routine, that should never execute

'**.ida**': fake, empty decryption routine
'**.eda**': fake, empty decryption routine

'**.rsr**': this function is a bit different than others. It traversing **resources tree**, and decrypts only specified resources (basically it should skip manifest and first icon group)

```
          sub     al, 0DAh
          sub     al, cl
          add     al, cl
          add     al, 4Ch
```

After all decryptions we're moved to **original enrypoint** of protected application:

```
          mov     edx, original_entry_point
          jmp     edx
```

To aggravate tracing we can see many junks in loader. It's rather simple junks:

```
          jmp     $+XX
```

where XX is the random (I thought) value.

To remove this layer I've traced loader (in **OllyDbg**) to place where it jumps to original entry point. Next I've dumped memory of the process with **LordPE**, and fixed imports with **ImportREC**. All tasks takes about one minute... It's definitely not so hard protection scheme.

- Generating keyfile "password.txt"

When we remove **PE encrypter** we have to generate **password.txt** file. To catch moment when protected executable access **password.txt**, we have to set breakpoint on files access functions:

- **CreateFileA**
- **CreateFileW**
- **ReadFile**

We should break on **CreateFileW**:

```
0041CA36 ┌CALL to CreateFileW from final.0041CA30
0013F36C │FileName = "password.txt"
80000000 │Access = GENERIC_READ
00000003 │ShareMode = FILE_SHARE_READ|FILE_SHARE_WRITE
0013F0E4 │pSecurity = 0013F0E4
00000003 │Mode = OPEN_EXISTING
00000080 │Attributes = NORMAL
00000000 └hTemplateFile = NULL
0013F1C0
```

Now we have to '***step over***' some code until we will back to function that called **CreateFileW**. In fact we have to back to:

```
.text:00405214        call    std::_Fiopen(char const *,int,int)
```

How do I know that ? There is two ways to get to this place:

- It's executable compiled with **VS2k5**, this fact implies that all code written by user is at the beginning of the code section. Rest of the code comes from libraries (exception for this is **STL library**). So we have to trace until we reach quite low address (relative to size of the whole code section). In this case most of all code below **0040769A** is written by user, because **IDA Pro** marked almost all code after that point as 'library' code (**FLIRT**).

- second method is more common (I think). It bases on **IDA Pro** signatures and it's variant of the first method but static.

Next step is to find place where file **password.txt** is read. We should break on **ReadFile** and in the same way as with **CreateFileW** we will reach:

```
.text:0040648D        call    edx             ; read one byte from file
.text:0040648F        nop                     ; <- we should be here after
                                                   ReadFile
```

Bytes from file are read until the end of file, or until **0x20** (space) character.

Read string is converted to integer value:

```
.text:00406FEB        push    edx                     ; char *
.text:00406FEC        call    j__atol
```

and forwarded to this algorithm:

```
.text:00406FF1        mov     ecx, eax        ; coverted value from file
.text:00406FF3        mov     eax, 30C30C31h
```

```
.text:00406FF8          imul    ecx
.text:00406FFA          sar     edx, 3
.text:00406FFD          mov     eax, edx
.text:00406FFF          shr     eax, 1Fh
.text:00407002          add     eax, edx
.text:00407004          imul    eax, 2Ah        ; 42
.text:00407007          mov     edx, ecx
.text:00407009          add     esp, 4
.text:0040700C          sub     edx, eax
.text:0040700E          jnz     short _bad_password
.text:00407010          test    ecx, ecx        ; password cannot be '0'
.text:00407012          jz      short _bad_password
```

I have coded simple brute force to get proper value:

```
//-----------------------------------------------------------------
      #include <windows.h>
      #include <cstdio>

      bool __stdcall _count(DWORD val)
      {
            DWORD _ret = 0;
            __asm
            {
                  MOV     ECX, val
                  MOV     EAX, 0x30C30C31
                  IMUL    ECX
                  SAR     EDX, 3
                  MOV     EAX, EDX
                  SHR     EAX, 0x1F
                  ADD     EAX, EDX
                  IMUL    EAX, EAX, 0x2A
                  MOV     EDX, ECX
                  SUB     EDX, EAX
                  mov     _ret, edx
            }
            return _ret;
      }

      int main()
      {
            DWORD i = 1;
            while (_count(i)) i++;
            printf("%d\n", i);
            return 0;
      }
//-----------------------------------------------------------------
```

So **password.txt** should contain value **"42"**.

- reverse engineer the mathematical formula (objective 1)

To locate code correlated with mathematical formula we have to set breakpoints on all 'user' (located near the beginning of the code section) functions containing **FPU** operations. Actually there are only three possibilities:

* **sub_401090**
* **sub_401150**
* **sub_401290**

When we run program, it will stop at sub_401290 function. It's quite easy piece of code:

```
.text:00401290 sub_401290 proc near
.text:00401290
.text:00401290 var_4 = dword ptr -4
.text:00401290
.text:00401290         push    ecx
.text:00401291         push    ebx
.text:00401292         push    esi
.text:00401293         push    edi
.text:00401294         mov     edi, ds:GetTickCount    ||--------------
.text:0040129A         mov     esi, ecx                || ANTI-DEBUG
.text:0040129C         call    edi ; GetTickCount      ||
.text:0040129E         mov     ebx, eax                ||
.text:004012A0         call    sub_4016E0              ||
.text:004012A5         test    al, al                  || for further
.text:004012A7         jz      short loc_4012B0        || info look
.text:004012A9         sub     dword_42306C, 1         || below
.text:004012B0                                         ||
.text:004012B0 loc_4012B0:                             ||
.text:004012B0         call    ds:IsDebuggerPresent    ||
.text:004012B6         test    eax, eax                ||
.text:004012B8         jz      short loc_4012C1        ||
.text:004012BA         add     dword_423070, 1         ||
.text:004012C1                                         ||
.text:004012C1 loc_4012C1:                             ||
.text:004012C1         call    edi ; GetTickCount      ||
.text:004012C3         sub     eax, ebx                ||
.text:004012C5         cmp     eax, 7D0h               ||
.text:004012CA         jbe     short loc_4012D8        ||
.text:004012CC         fld     ds:dbl_41E228   ; pi    || ANTI-DEBUG
.text:004012D2         fstp    dbl_4248C0              ||--------------
.text:004012D8
.text:004012D8 loc_4012D8:
.text:004012D8         mov     eax, [esi+0C0h]
.text:004012DE         fild    dword_423068    ; 495
.text:004012E4         add     eax, [esi+0BCh]
.text:004012EA         pop     edi
.text:004012EB         add     eax, [esi+0B8h]
.text:004012F1         mov     ecx, eax
.text:004012F3         imul    ecx, eax
.text:004012F6         mov     [esp+0Ch+var_4], eax
.text:004012FA         fild    [esp+0Ch+var_4]
.text:004012FE         mov     [esp+0Ch+var_4], ecx
.text:00401302         fmul    ds:dbl_41E220   ; 0.0008267
.text:00401308         fsubr   ds:dbl_41E218   ; 1.10938
.text:0040130E         fild    [esp+0Ch+var_4]
.text:00401312         fmul    ds:dbl_41E210   ; 0.0000016
.text:00401318         faddp   st(1), st
.text:0040131A         fild    dword ptr [esi+30h]
```

```
.text:0040131D        fmul    ds:dbl_41E208    ; 0.0002574
.text:00401323        fsubp   st(1), st
.text:00401325        fdivp   st(1), st
.text:00401327        fadd    dbl_4248C0       ; 0.0
.text:0040132D        fsub    ds:dbl_41E1B8    ; 450
.text:00401333        fst     qword ptr [esi+98h]
.text:00401339        mov     edx, dword_423070
.text:0040133F        imul    edx, dword_42306C
.text:00401346        mov     [esp+0Ch+var_4], edx
.text:0040134A        fild    [esp+0Ch+var_4]
.text:0040134E        fdivp   st(1), st
.text:00401350        fmul    qword ptr [esi+28h]
.text:00401353        fst     qword ptr [esi+0A8h]
.text:00401359        fsubr   qword ptr [esi+28h]
.text:0040135C        fstp    qword ptr [esi+0A0h]
.text:00401362        pop     esi
.text:00401363        pop     ebx
.text:00401364        pop     ecx
.text:00401365        retn
.text:00401365 sub_401290 endp
```

ANTI-DEBUG:

In this function we have three **anti-debug** methods:

- **GetTickCount** <- actually it is anti-trace method, it count execution time between two **GetTickCount** calls:

```
.text:0040129C        call    edi ; GetTickCount
...
.text:004012C1        call    edi ; GetTickCount
```

  If it takes to long we can suspect that someone tracing our program:

```
.text:004012C3        sub     eax, ebx
.text:004012C5        cmp     eax, 7D0h
```

- **sub_4016E0** <- this function is similar to **IsDebuggerPresent**
- **IsDebuggerPresent** <- standard **Windows API** to detect debugger

If program detects that it is debugged it will not stop execution (it's usual behaviour in commercial applications), instead of nice **MessageBox** with *"Debugger detected"* info program will modify some values used to generate output:

```
.text:004012A9        sub     dword_42306C, 1
...
.text:004012BA        add     dword_423070, 1
...
.text:004012CC        fld     ds:dbl_41E228    ; pi
.text:004012D2        fstp    dbl_4248C0       ; 0.0
```

I will not describe step by step how to get mathematical formula because it is quite easy to do it only by looking on that function. I can only give a few hints:

- `.text:004012DE fild dword_423068 ; 495`
  this is initial instruction

- `.text:00401333 fst qword ptr [esi+98h]`
  this is final instruction (we have **10.9319** in **ST0**)

In my opinion there is one imprecision in the formula, because in one place we have to add **0.0**, it is **global value.** In my formula I have skipped this + **0.0.** So my formula is:

result = (g1 / (g3 - (p1+p2+p3) * g2 + (p1+p2+p3)*(p1+p2+p3)*g4 - p4 * g5)) - g6;

and with that **+0.0** it could look like this:

result = (g1 / (g3 - (p1+p2+p3) * g2 + (p1+p2+p3)*(p1+p2+p3)*g4 - p4 * g5)) + g7 - g6;

where g7 = 0.0

Why I have skipped this value ? I have tracked places where this value is used or changed. If I'm correct this value is changed to **3.14** (**pi**) only if we have attached debugger or when **memory CRC** check fail. At this moment I can mention that in this executable we have two functions that calculates memory checksum (**not standard CRC**):

- **sub_401700**
- **sub_401740**

- patch executable to extend some functionality (objective 2)

This stage is also quite easy. We have to patch binary to remove **210.5** limit on eighth field in **data.txt** file. How we can achieve this? I have found limit value in **.data** section:

```
.rdata:0041E4D8 dbl_41E4D8      dq 2.105e2    ; DATA XREF: _main+3C0#r
```

As we can see, this value is referenced from **_main+3C0**:

```
.text:004072F0         fld     ds:dbl_41E4D8        ; 210.5
.text:004072F6         fld     [ebp+68h+var_98]
.text:004072F9         add     esp, 28h
.text:004072FC         fcom    st(1)
.text:004072FE         fnstsw  ax
.text:00407300         test    ah, 41h
.text:00407303         jnz     short loc_40730D
```

To skip **210.5** limit we should patch **conditional jump** at **.text: 00407303** to **unconditional jump.**

3. Time to break

- unpack custom **PE encrypter** (easy)

As I mentioned earlier this was the easiest part of protection.
time to remove encrypter: about 1 minute
I have worked on unpackers for 1,5 year in AV company, also I have written my own protectors (search at **openrce.org**) so maybe I am not so representative in this area.

- generate keyfile **"password.txt"**

time to generate file: about 1,5 hour
In this 1,5 hour I have also done overall analysis of executable

- reverse engineer the **mathematical formula** (**objective 1**)

time to break: about 1,5 hour
30 minutes to point how to find proper function, and 1 hour to write formula and check it

- patch executable to extend some functionality (**objective 2**)

time to break: 20 minutes

Developed tools:

- brute force for **password.txt**, it was **15 minutes**

Internet research: 0%

4. Tools used

- **OllyDbg** - **x86** assembly level debugger, used to **unpacking** and **analysis**
- **LordPE** - memory dumper, used to **dump** decrypted executable from memory
- **ImportREC** - tool used to **rebuild imports** structure, it was not required this time, because imports were not encrypted, but I didn't even checked ;-)
- **IDA Pro** - most advanced disassembler, used to **overall analysis**
- **Visual Studio** - windows **C/C++** (not only) compiler, used to **compile brute force**
- **notepad** - standard windows notepad, used to write some conclusions
- **total commander** - file manager with nice F3 viewer (Lister)

5. Conclusion

Today writing effective protection is not easy task. First of all to improve this protection we should develop more complicated encryption layer. Executable protector should encrypt imports, move some of the application code to loader, morph parts of application, add some virtualization layer etc... Take a look on commercial protectors like Armadillo, ASProtect, SafeCast or Themida. Of course all of this can be broken, but the effort to do this is sometimes higher than profits. Code responsible for mathematical formula should be at least obfuscated or even virtualized. Overall difficulty of whole protection I'm evaluating as easy.