Solving of Hacker Challenge 2007 Phase 1

Author: Omega Red (omegared@o2.pl)

Background

Participants will receive a protected Windows binary that produces certain output when run. The goal of the contest is to achieve the following two objectives:

1. Reverse engineer the mathematical formula that results in the value **10.9319** of the output.
2. Remove the limitation on an input data field of the code so that values greater than **210.5** are treated the same as values less than **210.5**.

The binary is a standard win32 executable. It uses text file **data.txt** as input. It also requires correct **password.txt** file to run, which is not provided. Completing these objectives required a number of steps:

- Decrypting the binary to allow its static analysis in disassembler.
- Generating correct password.txt file.
- Finding the formula for objective 1.
- Removing the input limitations for objective 2.

Various anti-debugging/anti-tampering methods were analyzed and disabled during all of these stages. None of them was particularly difficult and the author succeeded in achieving both objectives in about two days. During analysis author didn't find any surprises or tricks that he wasn't familiar with already.

Attack Narrative

First analysis

When we first run the executable, it produces following output:

```
Missing password.txt - We apologize for the inconvenience.
```

Right, so the executable uses some form of key file protection (or just makes us think it does ;). Let's make sure.

We will use *FileMon* - a free utility that can list all file system activity on Windows system. After setting filter to the name of our executable to not be flooded by logs produced, we can see:

```
11:06:16   final.exe:2428   IRP_MJ_CREATE   C:\hackerchallenge\password.txt   FILE NOT
FOUND   Attributes: N   Options: Open
11:06:16   final.exe:2428   IRP_MJ_CREATE   C:\hackerchallenge\password.txt   FILE NOT
FOUND   Attributes: N   Options: Open
```

OK - it seems that our target really uses key file protection. So, we need to do either of:

- Find out what *should* be in **password.txt** file by reverse engineering and create such file that will pass the check.
- Just modify the binary to disable the file check.

Anyway, we will need to locate and analyze code that performs the check. Let's look at the file in the direct meaning of this word. We will use any hex editor or just *Total Commander*'s internal viewer.

We see normal section names **(.text .data .rdata .rsrc**). No suspicious sections that would indicate well known executable modifiers. Then most of the file seems to be encrypted - there is very little 0 bytes, rather uncommon. Encryption must be very weak however, as there are easily spottable patterns. That indicates some kind of substitution cipher for single bytes, most likely simple arithmetic operation being used. We'll check that later.

After code section we see some strings from Visual C CRT, unencrypted strings from our target (`Incorrect password - We apologize for the inconvenience`.). Then there are some imports - most notably `IsDebuggerPresent`. At the end of file we see indication of anti-SoftICE routines (meltICE) - strings "\\.\SICE" and "\\.\NTICE".

Next step is analyzing executable structure a bit more in detail using *PeID*. Section viewer reveals additional section named **JR** that was not spotted by us earlier. Entry point is located there, so our hypothesis is that this section decrypts the real code. Quick disassembly of entry point shows some code obfuscation used:

```
00428288: EB 00            JMP 0042828A
0042828A: BD FA A4 FD FF   MOV EBP,FFFDA4FA
0042828F: E8 00 00 00 00   CALL 00428294
00428294: E8 68 00 00 00   CALL 00428301
00428299: 90               NOP
0042829A: 90               NOP
```

We'll disassemble it properly later.

Sections have unusual attributes: all are read/write data - clear indication of self-modifying code. PE header seems to not contain anything unusual except sections, especially there are no TLS callbacks which could be used to make debugging harder. *PEiD*'s Crypto analyzer shows no signs of known crypto/hash algorithms, but this can be wrong since code section is encrypted.

Summary

Executable is written in Visual C++. Decryptor and possibly other parts were most likely hand-coded in assembly. Code section is protected by some weak encryption; data section is most likely not encrypted. Executable uses various anti-debugging methods that will be analyzed later. No well-known protectors were used.

Decrypting the executable

We will need to analyze the decryption routine and create unencrypted executable, if possible, to make later analysis easier. Author used *IDA Pro* freeware version for this and all static disassembly analysis.

Entry point of the binary indicates on-purpose obfuscation:

```
JR:00428288 start          proc near
JR:00428288                jmp     short $+2
JR:0042828A                mov     ebp, 0FFFDA4FAh ; EBP set
```

```
JR:0042828F                call    $+5
JR:00428294                call    sub_428301
```

After a few jumps we arrive here:

```
JR:0042842B loc_42842B:                                 ; CODE XREF: sub_428301+ED□j
JR:0042842B                mov     edx, ebp
JR:0042842D                add     edx, 44E508h         ; EDX = 00428A02
JR:00428433                mov     eax, [edx]           ; EAX = 00400000 - image base of the
executable
JR:00428435                call    sub_4284B9           ; main decryption routine as we see
later
JR:0042843A                jmp     loc_428890
JR:0042843A sub_428301     endp
```

The decryption routine looks like this:

```
JR:004284B9 sub_4284B9     proc near                    ; CODE XREF: sub_428301+134□p
JR:004284B9                mov     edi, eax             ; edi = 00400000
JR:004284BB                add     edi, [edi+3Ch]       ; PE header offset
JR:004284BE                mov     esi, edi
JR:004284C0                add     esi, 0F8h            ; start of section table
JR:004284C6                xor     edx, edx             ; section counter
JR:004284C8 loc_4284C8:                                 ; CODE XREF: sub_4284B9+22C□j
JR:004284C8                push    edx
JR:004284C9                push    eax                  ; eax = image base
JR:004284CA                db      3Eh                  ; DS segment override, can be hidden
in IDA analysis options
JR:004284CA                cmp     dword ptr [esi], 7865742Eh  ; 'xet.'
JR:004284D1                jz      loc_428598           ; jr_decrypt_code_stub
JR:004284D7                db      3Eh
JR:004284D7                cmp     dword ptr [esi], 45444F43h  ; 'EDOC'
JR:004284DE                jz      loc_428598           ; jr_decrypt_code_stub
```

We see simple "switch" construct to invoke specific functions for various PE sections like ".tex" and "CODE". The code only compares first 4 characters of section name, so we could say it's buggy. Let's take a look at the actual decryption routine.

```
JR:00428598 jr_decrypt_code_stub:                       ; CODE XREF: jr_decrypt+18□j
JR:00428598                                             ; jr_decrypt+25□j
JR:00428598                cmp     dword ptr [esi+14h], 0 ; section RVA
JR:0042859D                jz      jr_decrypt_nextsection
JR:004285A3                cmp     dword ptr [esi+10h], 0 ; section VSize
JR:004285A8                jz      jr_decrypt_nextsection
JR:004285AE                push    esi                  ; esi & edi are popped after this
'procedure'
JR:004285AF                push    edi
JR:004285B0                push    ecx
JR:004285B1                push    ebx
JR:004285B2                mov     ecx, [esi+10h]       ; ecx = section VSize
JR:004285B6                xor     ebx, ebx             ; ebx = 0
JR:004285B8                mov     esi, [esi+0Ch]       ; esi = section RVA
JR:004285BC                add     esi, eax             ; add image base, esi = section VA
JR:004285BE                call    jr_decrypt_code      ; actual decryption takes place there
JR:004285C3                pop     ebx
JR:004285C4                pop     ecx
JR:004285C5                mov     edx, ebp             ; FFFDA4FA
JR:004285C7                add     edx, 44E1DEh         ; edx = 4286D8
JR:004285CD                lea     eax, [edx]
JR:004285CF                push    eax                  ; obfuscated jmp 4286d8
JR:004285CF                                             ; (process next section)
JR:004285D0                retn
```

Right. The real decryption algorithm can be seen at **0042847C** (junk jumps omitted):

```
JR:0042847C jr_decrypt_code proc near              ; CODE XREF: jr_decrypt+105□p
JR:0042847C                 mov     edi, esi        ; esi = data pointer
JR:0042847C                                         ; ecx = data size
JR:00428484                 lodsb                   ; al = data byte
JR:00428485                 clc
JR:00428486                 add     al, 10h
JR:00428488                 stc
JR:00428492                 xor     al, 53h
JR:00428494                 ror     al, 0BDh
JR:00428497                 add     al, 0AFh
JR:00428499                 sub     al, 1Fh
JR:004284B0                 add     al, 0A0h
JR:004284B2                 add     al, 0Fh
JR:004284B4                 nop
JR:004284B5                 stosb
JR:004284B6                 loop    loc_428484
JR:004284B8                 retn
JR:004284B8 jr_decrypt_code endp
```

It can be simplified to (all numbers in hex):

x' = (((x+10) xor 53) ror 5) + 3f

We can see that it's indeed very simple algorithm. Our assumption that it's single byte substitution was correct.

Procedure that decrypts data section is very similar, only the actual algorithm is different, involving value of **CL** register (which is part of the loop counter). A bit more complex, but it's still very easy to decrypt.

Procedure for 'BSS' section seems to be incomplete:

```
JR:00428447                 lodsb
JR:00428448                 add     [eax], al
JR:0042844B                 add     [eax], al
JR:0042844E                 add     [eax], al
JR:00428451                 add     [eax], al
JR:00428454                 add     [eax], al
```

...but that's OK - there is no BSS section in our executable. The decryption stub is just a little more generic. ;)

Same goes for '.ida' and '.eda' decryptors - they are not working/unused. There is also decryption stub for '.rsr' (resource) section. It seems to parse PE resource directory, but there are no resources in the executable except the manifest, so it does nothing. It's also written in C/++, unlike most of the decryptor which seems to be hand-coded assembly.

We can observe decryptor in action under debugger - there is no anti-debugging code there. Author used *OllyDbg* for this. We can set breakpoint at **0042843A** to have all sections decrypted (it's the next instruction after decryption routine call). Then it's just a matter of writing them to the binary and altering PE entry point to **004094B8**, where the 'real' execution begins (a few junk jumps later). We can also use *PEiD* generic unpacker (using mentioned entry point) - this method was used by the author as it's most convenient. Decrypted executable is uploaded as **final_decrypted.exe**. It also has input limits removed, since this patch was done last.

Passing the password file check

Our target won't run without **password.txt** with correct content. As mentioned before, we have two main choices: patching executable to bypass the check, or finding out the correct password. We will test both approaches.

Finding the check in the code is easiest with *IDA* - we can find references to "password.txt" (the file name) or error messages and follow them. We find this:

```
.text:00406F67                  push    1
.text:00406F69                  push    40h
.text:00406F6B                  push    1
.text:00406F6D                  push    offset aPassword_txt ; "password.txt"
.text:00406F72                  lea     ecx, [ebp+68h+var_220]
.text:00406F78                  mov     [ebp+68h+var_3B0], offset off_41E204
.text:00406F82                  call    sub_4065B0
.text:00406F87                  cmp     [ebp+68h+var_1CC], 0
.text:00406F8E                  mov     [ebp+68h+var_6C], 0
.text:00406F95                  jz      pwd_open_error
```

Doesn't it look like a call to "fopen"-type function? Actually it's *ifstream* constructor or similar - we see **ECX** being loaded before function call (object pointer, *thiscall* convention), and some strings in the code indicate that it uses C++ streams. But the real deal is just below:

```
.text:00406F9B                  push    20h
.text:00406F9D                  push    3                   ; buffer size
.text:00406F9F                  lea     eax, [ebp+68h+buf]
.text:00406FA2                  push    eax
.text:00406FA3                  lea     ecx, [ebp+68h+fs_password]
.text:00406FA9                  call    sub_406410          ; read from file
...
.text:00406FE8                  lea     edx, [ebp+68h+var_28]
.text:00406FEB                  push    edx                 ; char *
.text:00406FEC                  call    j__atol             ; string to dword
.text:00406FF1                  mov     ecx, eax            ; ecx = x
.text:00406FF3                  mov     eax, 30C30C31h ;
.text:00406FF8                  imul    ecx                 ; edx:eax = (x * 0x30C30C31)
.text:00406FFA                  sar     edx, 3              ; edx = (x * 0x30C30C31) shr
0x23
.text:00406FFD                  mov     eax, edx
.text:00406FFF                  shr     eax, 1Fh            ; eax = 0
.text:00407002                  add     eax, edx            ; eax = (x * 0x30C30C31) shr
0x23
.text:00407004                  imul    eax, 2Ah            ; eax = 0x2a * ((x *
0x30C30C31) shr 0x23)
.text:00407007                  mov     edx, ecx            ; edx = x
.text:00407009                  add     esp, 4
.text:0040700C                  sub     edx, eax            ; x = 0x2a * ((x *
0x30C30C31) shr 0x23)
; 0x2a * 0x30C30C31 = 80000000A, so x = 0x2a * (x shr 5)
.text:0040700E                  jnz     short loc_40705E    ; "bad boy" jump
.text:00407010                  test    ecx, ecx
.text:00407012                  jz      short loc_40705E    ; "bad boy" jump
.text:00407014                  push    offset aThankYou_ ; "Thank you. \n"
.text:00407019                  push    offset dword_4254F8
.text:0040701E                  call    sub_405F70          ; print-type function
```

We see a char buffer being converted to number, then some calculations being performed on it, and finally the "good/bad" jump. So **password.txt** should contain an integer number in ASCII. From the calculations performed we can deduct that the final equation being evaluated is **x = 0x2a ***

(x shr 5), where x is the number read from password.txt. Decomposing right-hand as "0x2a * 1" gives us first solution: **x = 0x2a or 42 decimal**. Oh, The Answer to the Ultimate Question of Life, the Universe, and Everything! Well, other possible solutions are multiples of 42, but the executable only reads two decimal digits from password.txt (what can be observed under debugger) - so the set of correct passwords is just 42 and 84. Trivial solution of 0 is deemed false by the comparison at **00407010**.

There is another method to find correct password, after knowing that it's only 2 digits: brute force. Simple .bat script can test all possible passwords in a second:

```
@echo off
for /l %%a in (1,1,99) do call :test %%a
goto end

:test
echo %1 > password.txt
final.exe > %1.txt

:end
```

After browsing generated output files we can see that indeed only **42** and **84** were correct. This method was used by the author at first.

What about patching? "For educational purposes" author tried to just patch the whole check by inserting `jmp 407014` at **00406F67** (after disabling integrity checks which will be described later). That didn't work as expected, however - output looked like this:

```
Thank you.
 1 3 10.9319
 33 17 10 5 6 10 8 4
 21.8638 178.136 1
 1 7 9.02697
 33 17 10 5 6 10 8 4
 18.0539 181.946 1
 9 3 14.8862
 32 14 5 8 12 12 13 8
 17.8634 102.137 2
 11 3 0.
 45 22 6 7 5 12 3 33
 0. 220. 1
```

After closer inspection of patched code it was clear what went wrong:

```
.text:00406F5C                  call    calc_init
.text:00406F61                  mov     esi, global1
.text:00406F67                  push    1                           ; jmp 407014
.text:00406F69                  push    40h
.text:00406F6B                  push    1
.text:00406F6D                  push    offset aPassword_txt        ; "password.txt"
.text:00406F72                  lea     ecx, [ebp+68h+fs_password] ; stream object
.text:00406F78                  mov     [ebp+68h+var_3B0], offset off_41E204     ; <- this
instruction was omitted after patching
.text:00406F82                  call    fsopen
```

After moving instruction from **00406F78** to **00406F67** and adding "`jmp 407014`" after, executable still crashes after printing data. Tracing over with *OllyDbg* reveals the call that is responsible for it:

```
.text:00407649                 lea     ecx, [ebp-0F4h]
.text:0040764F                 mov     byte ptr [ebp-4], 0
.text:00407653                 call    sub_404A50
.text:00407658                 lea     ecx, [ebp-1B8h]    ; object pointer
.text:0040765E                 mov     dword ptr [ebp-4], 0FFFFFFFFh
.text:00407665                 call    sub_404A50   ; this call causes access violation
```

It's part of the cleanup code, this call is actually a destructor for a stream object that was used to read **password.txt**. And since we skipped constructor by our patch:

```
.text:00406F6D                 push    offset aPassword_txt ; "password.txt"
.text:00406F72                 lea     ecx, [ebp-1B8h]      ; object pointer
```

...then the destructor tries to delete null object. If we NOP the call at **00407665**, executable runs fine without password.txt. Patched binary that doesn't require password file to run is uploaded as **final_nopasswd.exe**.

Finding algorithm for output calculation (objective 1)

We need to find where all the calculation is taking place. We know that program output depends on input: contents of **data.txt**. That's the first attack vector: open up the disassembly in *IDA* and search for code that opens data.txt. Here comes the first obstruction: there is no "**data.txt**" string found by *IDA*. Well, we have several other options. We can fire up debugger and trap **`CreateFile`** or **`ReadFile`**. But *IDA* will be sufficient - we already observed at least one instance of opening and reading file, so we'll search for other references to these functions.

```
.text:00406F67                 push    1
.text:00406F69                 push    40h
.text:00406F6B                 push    1
.text:00406F6D                 push    offset aPassword_txt ; "password.txt"
.text:00406F72                 lea     ecx, [ebp+68h+obj_stream]
.text:00406F78                 mov     [ebp+68h+var_3B0], offset off_41E204
.text:00406F82                 call    fsopen
.text:00406F87                 cmp     [ebp+68h+var_1CC], 0
.text:00406F8E                 mov     [ebp+68h+var_6C], 0
.text:00406F95                 jz      pwd_open_error
.text:00406F9B                 push    20h
.text:00406F9D                 push    3                    ; buffer size
.text:00406F9F                 lea     eax, [ebp+68h+buf]
.text:00406FA2                 push    eax
.text:00406FA3                 lea     ecx, [ebp+68h+obj_stream]
.text:00406FA9                 call    fsread
.text:00406FAE                 lea     ecx, [ebp+68h+var_218]
.text:00406FB4                 call    fsclose
```

There are only 2 recognized references to **`fsopen`**: one above (password file check) and one just a bit after that:

```
.text:0040718A                 push    1
.text:0040718C                 push    40h
.text:0040718E                 push    1
.text:00407190                 push    offset dword_41E4E0
.text:00407195                 lea     ecx, [ebp+68h+var_15C]
.text:0040719B                 call    fsopen
```

There is no plain-text file name here, instead some DWORD reference. *IDA* must've misinterpreted it, because after changing interpretation of this "DWORD" to a string all becomes clear:

```
.text:00407190                  push    offset aData_txt ; "data.txt"
.text:00407195                  lea     ecx, [ebp+68h+var_15C]
.text:0040719B                  call    fsopen
.text:004071A0                  cmp     [ebp+68h+var_108], 0
.text:004071A7                  mov     byte ptr [ebp+68h+var_6C], 1
.text:004071AB                  jz      loc_407309
.text:004071B1                  push    20h
.text:004071B3                  push    3               ; buffer size
.text:004071B5                  lea     eax, [ebp+68h+var_20]
.text:004071B8                  push    eax
.text:004071B9                  lea     ecx, [ebp+68h+var_15C]
.text:004071BF                  mov     [ebp+68h+var_1], 1
.text:004071C3                  call    fsread
```

Right, we have it. There is a block of file reads and **`atol`**/**`atof`**-s. Some calculations, some prints as well - seems we're in the right place. So all the password checking and calculations seem to be in one monolithic **`main`** function. Let's see what happens after successful password check.

```
.text:00407014                  push    offset aThankYou_ ; "Thank you. \n"
.text:00407019                  push    offset dword_4254F8
.text:0040701E                  call    print
.text:00407023                  add     esi, 0FFFFFFFBh ; there is
.text:00407023                                          ; mov     esi, dword_423068
.text:00407023                                          ; before
.text:00407026                  mov     dword_423068, esi
.text:0040702C                  mov     esi, ds:GetTickCount
.text:00407032                  add     esp, 8
.text:00407035                  call    esi ; GetTickCount
.text:00407037                  mov     edi, eax         ; EDI = tick count at the start
```

Here we see one of the anti-debug tricks, or the start of it. Current tick count (millisecond counter) is stored in **`EDI`**. It will be later compared to current tick count, and if elapsed time is greater than some threshold, code assumes that it's run under debugger (manual tracing/single stepping is much slower than normal execution).

Countermeasures: patching **`GetTickCount`** to always return 0 or another small number; changing the comparison code; using specialized *OllyDbg* plugin (like *OllyAdvanced*).

```
.text:00407039                  mov     eax, large fs:30h ; PEB
.text:0040703F                  movzx   eax, byte ptr [eax+2] ; BOOL BeingDebugged
.text:00407043                  or      al, al
.text:00407045                  jz      short loc_407050
```

Another anti-debug trick. **`FS:30`** is a **Thread Environment Block** field that holds pointer to **Process Environment Block**. And **`PEB:3`** is a boolean flag that indicates if a process is being debugged.

Countermeasures: patching **`PEB:BeinDebugged`** field to 0; changing the comparison code; using specialized *OllyDbg* plugin (like *OllyAdvanced*).

```
.text:00407047                  jmp     short $+2
.text:00407049                  mov     eax, 1
.text:0040704E                  jmp     short loc_407052
.text:00407050 ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:00407050
.text:00407050 loc_407050:                              ; CODE XREF: _main+115□j
```

```
.text:00407050                 xor     eax, eax
.text:00407052
.text:00407052 loc_407052:                             ; CODE XREF: _main+11E□j
.text:00407052                 nop
.text:00407053                 test    al, al
.text:00407055                 jz      short loc_407077
.text:00407057                 push    0FFFFFFFFh      ; uExitCode
.text:00407059                 call    exit
.text:0040705E ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0040705E
.text:0040705E bad_boy:                                ; CODE XREF: _main+DE□j
.text:0040705E                                         ; _main+E2□j
.text:0040705E                 push    offset aIncorrectPassw ; "Incorrect password - We apologize for t"...
.text:00407063                 push    offset dword_4254F8
.text:00407068                 call    print
.text:0040706D                 add     esp, 8
.text:00407070                 push    0               ; uExitCode
.text:00407072                 call    exit
.text:00407077 ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:00407077
.text:00407077 loc_407077:                             ; CODE XREF: _main+125□j
.text:00407077                 call    ds:IsDebuggerPresent
.text:0040707D                 test    eax, eax
.text:0040707F                 jz      short loc_407088
.text:00407081                 push    0FFFFFFFEh      ; uExitCode
.text:00407083                 call    exit
```

Another trick: this is essentially the same as the previous one; it just uses API function to get the **`BeingDebugged`** flag.
Countermeasures: patching **`PEB:BeinDebugged`** field to 0; patching **`IsDebuggerPresent`** to always return 0; changing the comparison code; using specialized *OllyDbg* plugin (like *OllyAdvanced*).

...some calculations...

```
.text:0040711B                 call    esi                       ; GetTickCount
.text:0040711D                 sub     eax, edi
.text:0040711F                 cmp     eax, 7D0h
.text:00407124                 jbe     short loc_40712D
.text:00407126                 push    0FFFFFFFCh                ; uExitCode
.text:00407128                 call    exit
.text:0040712D ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0040712D
.text:0040712D loc_40712D:                             ; CODE XREF: _main+1F4□j
.text:0040712D                 lea     eax, [ebp+68h+var_68]
```

That's the second part of **`GetTickCount`** trick. We can see exit being called if elapsed time is too long.

Well, we have found the approximate location of code that does all the calculations, but we need the exact algorithm. Probably the easiest method will be "reverse engineering" in the literal meaning of the phrase: pinpoint the moment when the values are printed and then "go backwards" in code flow.

When looking at the code in *IDA* we see a bunch or prints as noted earlier:

```
.text:0040744D                 push    eax
.text:0040744E                 call    print
```

```
.text:00407453                 add     esp, 8
.text:00407456                 push    eax
.text:00407457                 call    print           ; This prints 10.9319
.text:0040745C                 add     esp, 8
```

"This prints 10.9319" note can be verified under debugger. Backtracking a bit more we see:

```
.text:0040737E                 call    sub_401740
.text:00407383                 cmp     eax, 0D81DB55Ch
.text:00407388                 jz      short loc_4073C4
```

Does this ring a bell? Well, it should - `sub_401740` is a simple integrity check returning a checksum that is compared to "good" value just after the call. It can be subverted by making it to always return good value; modifying compared value so that it matches modified image; or just eliminating the call and compare altogether. We will patch the jump at **00407388** to be unconditional. On failed check we see some cleanup and return from main:

```
.text:0040738A                 lea     ecx, [ebp+68h+fs_data]
.text:00407390                 mov     byte ptr [ebp+68h+var_6C], 0
.text:00407394                 call    fsdelete
.text:00407399                 lea     ecx, [ebp+68h+fs_password]
.text:0040739F                 mov     [ebp+68h+var_6C], 0FFFFFFFFh
.text:004073A6                 call    fsdelete
.text:004073AB                 mov     eax, 1
.text:004073B0                 mov     ecx, [ebp+68h+var_74]
.text:004073B3                 mov     large fs:0, ecx
.text:004073BA                 pop     edi
.text:004073BB                 pop     esi
.text:004073BC                 pop     ebx
.text:004073BD                 add     ebp, 68h
.text:004073C0                 mov     esp, ebp
.text:004073C2                 pop     ebp
.text:004073C3                 retn
.text:004073C4 ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
```

If the integrity check succeeds, our mysterious value is calculated; it can be easily spotted by tracing under debugger and observing FPU registers.

```
.text:004073C4 loc_4073C4:                             ; CODE XREF: _main+458j
.text:004073C4                 mov     eax, [ebp+68h+obj_calc]
.text:004073CA                 mov     edx, [eax]
.text:004073CC                 lea     ecx, [ebp+68h+obj_calc] ; "this" pointer
.text:004073D2                 call    edx
.text:004073D4                 fld     [ebp+68h+var_250] ; loads 10.9319
.text:004073DA                 lea     eax, [ebp+68h+print_buf]
.text:004073E0                 push    eax             ; char *
.text:004073E1                 push    6               ; int
.text:004073E3                 sub     esp, 8          ; double
.text:004073E6                 fstp    qword ptr [esp]
.text:004073E9                 call    __gcvt          ; float to string
```

We're closer now. Let's see where the `call edx` goes. It's a method of some object and we see no parameters passed on stack. The method uses only global variables and object data members.

```
.text:00401290 calc            proc near               ; DATA XREF: .rdata:off_41E204o
.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 ; "tick count" trick again...
.text:0040129A                 mov     esi, ecx                ; object pointer
.text:0040129C                 call    edi ; GetTickCount
.text:0040129E                 mov     ebx, eax
.text:004012A0                 call    DebuggerCheck ; this is just copy of
IsDebuggerPresent
.text:004012A5                 test    al, al
.text:004012A7                 jz      short loc_4012B0
.text:004012A9                 sub     global9, 1      ; corrupt data if debugger
detected
.text:004012B0
.text:004012B0 loc_4012B0:                             ; CODE XREF: calc+17j
.text:004012B0                 call    ds:IsDebuggerPresent
.text:004012B6                 test    eax, eax
.text:004012B8                 jz      short loc_4012C1
.text:004012BA                 add     global8, 1      ; corrupt data if debugger
detected
.text:004012C1
.text:004012C1 loc_4012C1:                             ; CODE XREF: calc+28j
.text:004012C1                 call    edi ; GetTickCount
.text:004012C3                 sub     eax, ebx
.text:004012C5                 cmp     eax, 7D0h        ; tick count check
.text:004012CA                 jbe     short loc_4012D8
.text:004012CC                 fld     ds:dbl_41E228   ; corrupt data if debugger
detected
.text:004012D2                 fstp    global6
.text:004012D8
.text:004012D8 loc_4012D8:                             ; CODE XREF: calc+3Aj
                                                ; the real calculations begin
.text:004012D8                 mov     eax, [esi+0C0h] ; 8 (data1)
.text:004012DE                 fild    global1         ; 495
; This is interesting - the value starts as 500, but it's 495 at runtime. By looking at
cross references in IDA we can find where it is modified - at 00407023, just after "Thank
you" message and successful key file check.
.text:004012E4                 add     eax, [esi+0BCh] ; 17 (data2)
.text:004012EA                 pop     edi
.text:004012EB                 add     eax, [esi+0B8h] ; 10 (data3)
.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] ; 35
.text:004012FE                 mov     [esp+0Ch+var_4], ecx
.text:00401302                 fmul    ds:global2      ; 8.267e-4
.text:00401308                 fsubr   ds:global3      ; 1.10938
.text:0040130E                 fild    [esp+0Ch+var_4]
.text:00401312                 fmul    ds:global4      ; 1.6e-6
.text:00401318                 faddp   st(1), st
.text:0040131A                 fild    dword ptr [esi+30h] ; 33 (data 4)
.text:0040131D                 fmul    ds:global5      ; 2.574e-4
.text:00401323                 fsubp   st(1), st
.text:00401325                 fdivp   st(1), st
.text:00401327                 fadd    global6         ; 0.0
.text:0040132D                 fsub    ds:global7      ; 4.5e2
.text:00401333                 fst     qword ptr [esi+98h] ; result (data5)
; some calculations not directly related to our value follow
.text:00401339                 mov     edx, dword_423070 ; 10
.text:0040133F                 imul    edx, dword_42306C ; 10
.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 calc             endp
```

So, the final formula that produces given value is:

10.9319224036473 = g1 / (x*x*g4 + g3 - x*g2 - d4*g5) + g6 - g7
where
x = d1+d2+d3

(**d** means object data, **g** means global variable)

d1=8, d2=17, d3=10, d4=33
g1=495, g2=8.267e-4, g3=1.10938, g4=1.6e-6, g5=2.574e-4, g6=0, g7=4.5e2

It can be found in **formula.txt** file.

Removing the input limitations (objective 2)

We need to change one value in data.txt from **210.5** to **220**. This should result in values **24.2433** and **195.757** being printed. We will start with changing **data.txt** to see what happens when binary is unmodified. Result: values printed are unchanged. Right, let's go down to the code and find where the binary reads input from **data.txt**.

```
.text:00407227                  push    20h
.text:00407229                  push    3
.text:0040722B                  lea     eax, [ebp+68h+inbuf7]
.text:0040722E                  push    eax
.text:0040722F                  lea     ecx, [ebp+68h+fs_data]
.text:00407235                  call    fsread
.text:0040723A                  push    20h
.text:0040723C                  push    6                 ; buffer size
.text:0040723E                  lea     ecx, [ebp+68h+inbuf8] ; this reads "210.5"
.text:00407241                  push    ecx
.text:00407242                  lea     ecx, [ebp+68h+fs_data]
.text:00407248                  call    fsread
.text:0040724D                  push    20h
.text:0040724F                  push    3
.text:00407251                  lea     edx, [ebp+68h+inbuf9]
.text:00407254                  push    edx
.text:00407255                  lea     ecx, [ebp+68h+fs_data]
```

Nothing interesting so far, let's look down at the code that converts strings to numbers.

```
.text:004072CC                  lea     edx, [ebp+68h+inbuf8]
.text:004072CF                  push    edx                     ; char *
.text:004072D0                  mov     [ebp+68h+x7], eax
.text:004072D3                  call    _atof                   ; convert x8
.text:004072D8                  fstp    [ebp+68h+x8]
.text:004072DB                  lea     eax, [ebp+68h+inbuf9]
.text:004072DE                  push    eax                     ; char *
.text:004072DF                  call    j__atol
.text:004072E4                  lea     ecx, [ebp+68h+inbuf10]
.text:004072E7                  push    ecx                     ; char *
.text:004072E8                  mov     [ebp+68h+x9], eax
.text:004072EB                  call    j__atol

.text:004072F0                  fld     ds:dbl_41E4D8     ; 210.5
```

```
.text:004072F6                 fld     [ebp+68h+x8]     ; x8 - the value we need to change
.text:004072F9                 add     esp, 28h
.text:004072FC                 fcom    st(1)            ; compare
.text:004072FE                 fnstsw  ax
.text:00407300                 test    ah, 41h          ; test for c0 & c3 FPU status bits
.text:00407303                 jnz     short loc_40730D ; x8 <= 210.5
                                        ; this jump will be patched to unconditional
.text:00407305                 fstp    st          ; out of range? replace x8 with 210.5
.text:00407305                                          ; 210.5 -> st
.text:00407307                 jmp     short loc_40730F ; continue
.text:00407309 ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:00407309
.text:00407309 loc_407309:                              ; CODE XREF: _main+27B□j
.text:00407309                 xor     bl, bl
.text:0040730B                 jmp     short loc_40737E
.text:0040730D ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0040730D
.text:0040730D loc_40730D:                              ; CODE XREF: _main+3D3□j
.text:0040730D                 fstp    st(1)
.text:0040730F
.text:0040730F loc_40730F:                              ; CODE XREF: _main+3D7□j
```

Bingo. After conversion we see a simple check that compares `x8` to **210.5**, and if it's larger, replaces it with **210.5**. We can skip it by NOP-ing or inserting short `jmp` over the check, or changing `dbl_41E4D8` value to **220** or more. We'll just patch the conditional jump. We also need to patch the checksum routine mentioned earlier to prevent application from detecting our changes.

After removing the limit we'll check how the program behaves. Here's the output of modified binary with modified **data.txt**:

```
Thank you.
 1 3 10.9319
 33 17 10 5 6 10 8 4
 21.8638 178.136 1
 1 7 9.02697
 33 17 10 5 6 10 8 4
 18.0539 181.946 1
 9 3 14.8862
 32 14 5 8 12 12 13 8
 17.8634 102.137 2
 11 3 14.1597
 45 22 6 7 5 12 3 33
 31.1513 188.849 1
```

We see that this time last values changed - but they are incorrect. There must be some other check in the code after. It's enough to just browse disassembly from where we left off to see:

```
.text:004075FA                 lea     edx, [ebp+68h+var_550]
.text:00407600                 push    edx
.text:00407601                 call    calc2
.text:00407606                 lea     eax, [ebp+68h+var_478]
.text:0040760C                 push    eax
.text:0040760D                 call    calc2
.text:00407612                 add     esp, 8
.text:00407615                 call    sub_401700          ; second checksum routine
.text:0040761A                 cmp     eax, 507AB3F7h
.text:0040761F                 jz      short loc_40762D    ; checksum ok?
.text:00407621                 fld     ds:dbl_41E4C8
.text:00407627                 fstp    global6          ; corrupt data if checksum invalid
.text:0040762D
```

```
.text:0040762D loc_40762D:                                  ; CODE XREF: _main+6EF↓j
.text:0040762D                  test    bl, bl
.text:0040762F                  jz      short loc_40763A
.text:00407631                  lea     ecx, [ebp+68h+var_3B0]
.text:00407637                  push    ecx
.text:00407638                  jmp     short loc_407641
.text:0040763A ;
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0040763A
.text:0040763A loc_40763A:                                  ; CODE XREF: _main+6FF↓j
.text:0040763A                  lea     edx, [ebp+68h+obj_calc]
.text:00407640                  push    edx
.text:00407641
.text:00407641 loc_407641:                                  ; CODE XREF: _main+708↓j
.text:00407641                  call    calc2
```

It's pretty obvious that we found it. **`sub_401700`** looks just like the previous one:

```
.text:00401700 checksum2        proc near               ; CODE XREF: _main+6E5↓p
.text:00401700                  push    ebx
.text:00401701                  push    esi
.text:00401702                  mov     eax, ds:40003Ch
.text:00401707                  mov     esi, [eax+400104h]
.text:0040170D                  mov     ecx, [eax+400108h]
.text:00401713                  add     esi, 400000h
.text:00401719                  add     esi, 637Bh
.text:0040171F                  mov     ecx, 10h
.text:00401724                  shr     ecx, 2
.text:00401727                  xor     ebx, ebx
.text:00401729
.text:00401729 loc_401729:                              ; CODE XREF: checksum2+2E↓j
.text:00401729                  lodsd
.text:0040172A                  rol     ebx, cl
.text:0040172C                  xor     ebx, eax
.text:0040172E                  loop    loc_401729
.text:00401730                  mov     eax, ebx
.text:00401732                  pop     esi
.text:00401733                  pop     ebx
.text:00401734                  retn
.text:00401734 checksum2        endp
```

That's the thing. We need to patch it just like the previous one to avoid tampering detection. Jump at **0040761F** was chosen for the simplicity. After applying modification, output of the binary is finally correct:

```
Thank you.
 1 3 10.9319
 33 17 10 5 6 10 8 4
 21.8638 178.136 1
 1 7 9.02697
 33 17 10 5 6 10 8 4
 18.0539 181.946 1
 9 3 14.8862
 32 14 5 8 12 12 13 8
 17.8634 102.137 2
 11 3 11.0197
 45 22 6 7 5 12 3 33
 24.2433 195.757 1
```

In total, 3 bytes were needed to be modified in the unencrypted binary. It is possible to patch encrypted binary by reverse-engineering encryption formulas, but the author didn't have time to do it. Patched binary is uploaded as **final_modified.exe**.

Time to break

In total, achieving both objectives took about two days. All protections used were very easy to bypass, so there haven't been any real problems. Encryption was quite simple and easy to revert. Understanding the formula was more time-consuming, but still wasn't hard. Author didn't have much experience with FPU, so some searching on the Internet was needed to accommodate for this. Removing input limits didn't prove complicated either. "Attack narrative" part of the report was written in parallel with actual reverse engineering, so it reflects actual steps done to defeat protections of the executable and achieve both objectives.

Tools used

All of the tools used were "industry standard" for any win32 reverse engineer. They will be listed in order of importance.

- **IDA Pro** - hands down the best disassembler for Windows. Automatic code flow analysis, cross-references, and of course the ability to hand-tune the disassembly are invaluable. Signatures that allow recognition of compiler-generated code were a great help as well.
- **OllyDbg** - one of the best, if not the best, user-mode debugger for Windows. Chosen for ease od use, auto analysis capabilities, many plugins available (*OllyAdvanced* was used to circumvent `IsDebuggerPresent` and `GetTickCount` tricks).
- **calc** - standard Windows utility, great for quick calculations, verification or dec/hex conversion.
- **PEiD** - popular executable identifier, able to detect many packers/protectors and show information about PE header. Chosen for its built-in generic unpacker.
- **Filemon** - one of many Sysinternals utilities. Great for analyzing any file system activity.
- **Hex Workshop** - pretty good hex editor, used for quick review and patching the binary.

Script written: **brute.bat**, batch file that tries all possible password files.

```
@echo off
for /l %%a in (1,1,99) do call :test %%a
goto end

:test
echo %1 > password.txt
final.exe > %1.txt

:end
```

Conclusion

The executable was successfully reverse engineered, its protections broken and functionality changed. Overall, protection methods used were very week and easy to bypass. It should be noted, though, that choosing C++ as the language and using object-oriented features raised the difficulty a bit. *IDA*, for example, didn't automatically recognize all stream functions used.

Decryption was pretty straightforward - junk jumps/calls were the only obstruction there, and the decryptor was easy to follow. One could just set one breakpoint to get the decrypted image, and then decrypt/dump the binary automatically with a tool like *PEiD* generic unpacker.

Passing the password check was also easy: after finding out that the password is a two-digit number, it's straightforward to brute-force it. The author started with that, writing a batch file that checked all possibilities. Then, it was also easy to follow calculations done on the number and derive a formula that gives correct passwords.

`IsDebuggerPresent` and `GetTickCount` anti-debug tricks were detected and recognized immediately when spotted in *IDA* disassembly or under debugger. Direct checks for `BeingDebugged` flag were generally easy to spot, as they were very close to the other ones. Most difficult to find (but still easy overall) were the checksum comparisons. For the first time program just shut down after making some modifications or setting breakpoints - that was indicating, that there is some integrity check. Method used to track it down was a breakpoint on `ExitProcess` and backtrack from there. This allowed finding the "good/bad" jumps, and then checksum procedure. The second integrity check corrupted data producing incorrect output if the checksum didn't match - it was spotted by manual disassembly browsing.

Anti-SoftICE protection which was mentioned at the beginning of this report was not actually found. On-access breakpoints on the suspicious strings were never triggered. Author didn't use SoftICE and didn't investigate it further, but it seems that there is no real protection of this kind in the binary.

What could be done to improve the protection? Well, many things, but let's focus on protection techniques that are already present in the binary.

Encryption

- Obfuscate the decryptor more
- Use anti-debugging tricks
- Use more sophisticated algorithm
- Don't decrypt the whole image, instead re-encrypt code that is no longer used

Password file

- Use larger password (that was the main weakness)
- Use binary file
- Better protect from totally skipping the check

Formula protection

- Use more obfuscation
- Use more sophisticated anti-debugging techniques
- Use code virtualization ;)

Anti-tamper protection

- Use more sophisticated integrity checks
- Don't do "good/bad" jumps after check since it can be just patched – use the checksum value in data processing instead
- Cross-check the checksum procedures with each other