2007 Hacker Challenge Report

Background

The challenge presented in this exercise is to reverse engineer a 32-bit Windows binary such that we are able to:

1) Provide and analysis of the software protection mechanisms implemented by the author(s);
2) Determine how to bypass the program's password requirements;
3) Achieve objective 1 – reproduce the formula for calculating "10.9319"; and
4) Achieve objective 2 – break the upper limit on the 210.5 value from data.txt.

The details of how these goals were completed successfully can be found in the remainder of this document, where we attempt to describe the methodology in a thorough, but clear and concise manner.

The most useful skill learned during phase 1 of the challenge was gaining a greater practical familiarity with the Intel x86 FPU and its related instructions, registers, control flags, and status flags.

Attack Narrative

The first half of this section will describe the *generic* software protection mechanisms that were observed throughout the challenge. In particular, this will identify the anti-reverse-engineer and anti-debugging code. Each observation will be accommodated by a brief analysis of the methods used to defeat them.

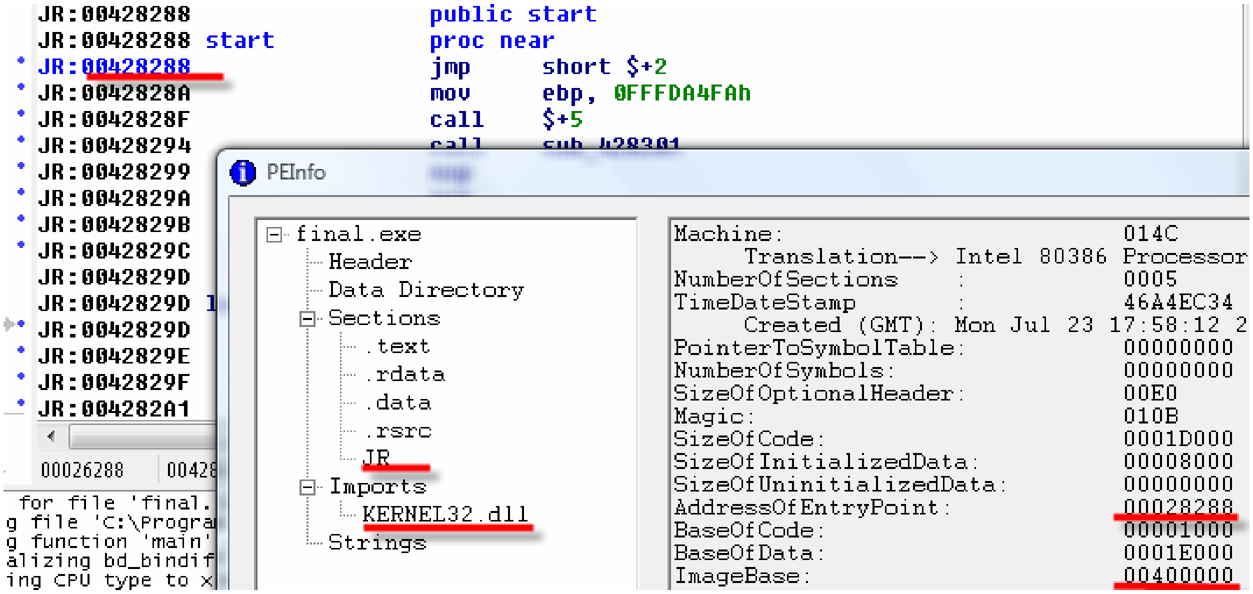
In the second half, the methodologies for achieving the two objectives will be presented, including the appropriate disassembly and script source code. This is where the *specific* software protection mechanisms are discussed, in other words – the code directly relevant to meeting the objectives.

Part I: Generic Software Protections

Software Protection Mechanism: “Packed/Obfuscated Sections”

The final.exe sample is packed with a propritary algorithm that is unknown to both open source and closed source packer detection signatures. We know it is packed due to several indications, which are outlined below and highlighted in the screen shot.

1) The start address (ImageBase + AddressOfEntryPoint) points to a suspicious , out-of-the-ordinary section named “JR” at the end of the binary.
2) The only visible imports are from Kernel32.dll. This is not suspiciuos per se, especially based on the program’s described behavior, but with other correlating evidence, it highly suggests that the binary is packed.
3) There are very few visible strings in the unaltered final.exe program.
4) The instructions at the start address weave and jump around dramatically to properly decode the .text segment, which is otherwise an uninterpretable chunk of data.



```
JR:00428288                     public start
JR:00428288 start               proc near
JR:00428288                     jmp     short $+2
JR:0042828A                     mov     ebp, 0FFFDA4FAh
JR:0042828F                     call    $+5
JR:00428294                     call    sub_428301
JR:00428299
JR:0042829A
JR:0042829B
JR:0042829C
JR:0042829D
JR:0042829D
JR:0042829D
JR:0042829E
JR:0042829F
JR:004282A1
```

```
00026288   00428
```

```
 for file 'final.
g file 'C:\Progra
g function 'main'
alizing bd_bindif
ing CPU type to x
```

```
Machine:                            014C
     Translation--> Intel 80386 Processor
NumberOfSections        :           0005
TimeDateStamp           :           46A4EC34
     Created (GMT): Mon Jul 23 17:58:12 2
PointerToSymbolTable:               00000000
NumberOfSymbols:                    00000000
SizeOfOptionalHeader:               00E0
Magic:                              010B
SizeOfCode:                         0001D000
SizeOfInitializedData:              00008000
SizeOfUninitializedData:            00000000
AddressOfEntryPoint:                00028288
BaseOfCode:                         00001000
BaseOfData:                         0001E000
ImageBase:                          00400000
```

To defeat the custom packing solution, we could engage several methods, such as the following:

1) locating OEP via static analysis or dynamic tracing (debugging) of the algorithm;
2) setting a section-based break-on-execute (see OllyBonE);

3) using generic OEP-locating software such as PEiD or IDA Pro's "Universal Unpacker" plugin;
4) using an API breakpoint within a module such as Kernel32.dll.

In practice, we used the later method, by setting a breakpoint on Kernel32@GetSystemTimeAsFileTime. One might use a tool such as API Monitor to detect which API calls are made early in the program's execution. APIs such as Kernel32@GetVersion(...) are normally safe choices, but in this case, the one we selected is called first.

After letting the code unpack itself, a tool known as OllyDbg PE Dumper v3.01 by FKMA was used to produce a sample of the executable which we could load into IDA Pro for further analysis. In these situations, it is frequently required to fix-up the dumped executable's import tables (see Import REConstructor), however for this challenge, it is not necessary.

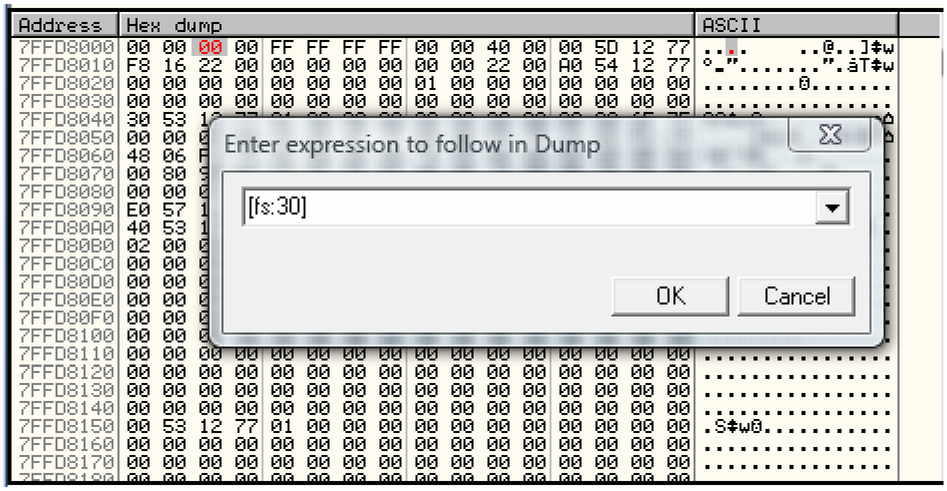
Software Protection Mechanism: "Kernel32@IsDebuggerPresent()"

The program makes several calls to Kernel32@IsDebuggerPresent and behaves differently if the API returns true.

```
.text:00407077                call    ds:IsDebuggerPresent
.text:0040707D                test    eax, eax
.text:0040707F                jz      short not_being_debugged
.text:00407081                push    0FFFFFFFEh      ; uExitCode
.text:00407083                call    exit_sorry
```

To defeat this protection, it would be possible to set an API breakpoint on the call, and manually change the return value to false each time it is called. It would also be possible to install and enable the OllyDbg plug-in for hiding the debugger from Kernel32@IsDebuggerPresent. However, since the final.exe sample also exhibits in-line debugger detection (using the exact same code as the API call), then the more efficient bypass is to simply navigate to [fs:30] of the program and change the 0x1 byte to 0x0. This solves the problem altogether. Below is an example of the in-line detection and a screen shot of how to modify the byte at run-time.

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



```
Address  Hex dump                                          ASCII
7FFD8000 00 00 00 00 FF FF FF FF 00 00 40 00 00 5D 12 77  .. .    ..@..]‡w
7FFD8010 F8 16 22 00 00 00 00 00 00 00 22 00 A0 54 12 77  °▬".......".áT‡w
7FFD8020 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ........θ.......
7FFD8030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8040 30 53 1
7FFD8050 00 00 0
7FFD8060 48 06 F
7FFD8070 00 80 9
7FFD8080 00 00 0
7FFD8090 E0 57 1
7FFD80A0 40 53 1
7FFD80B0 02 00 0
7FFD80C0 00 00 0
7FFD80D0 00 00 0
7FFD80E0 00 00 0
7FFD80F0 00 00 0
7FFD8100 00 00 0
7FFD8110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8150 00 53 12 77 01 00 00 00 00 00 00 00 00 00 00 00  .S‡wθ...........
7FFD8160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7FFD8170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

[fs:30]

Software Protection Mechanism: "Time-based Debugger Detection"

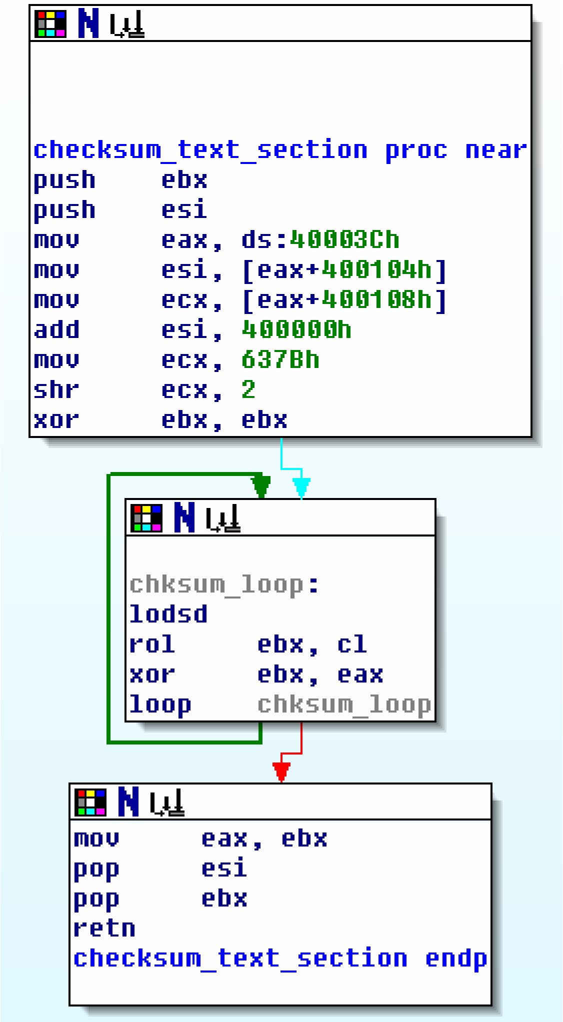
At several points throughout execution, the program calls Kernel32@GetTickCount at point A and again at point B, then performs a subtraction to determine how long it took the processor to get from point A to point B. In these cases, if it takes longer than a specified amount of time, then it is assumed that a debugger is controlling EIP and behaves differently. Below is an example of how this code appears in the disassembly. The time to beat is 200ms.

```
.text:0040129C                  call    edi ; GetTickCount
.text:0040129E                  mov     ebx, eax

.text:004012C1                  call    edi ; GetTickCount
.text:004012C3                  sub     eax, ebx
.text:004012C5                  cmp     eax, 7D0h
.text:004012CA                  jbe     short loc_4012D8
```

To defeat this protection, it is possible to either manually change the return result (in eax) from Kernel32@GetTickCount during a debugging session, or it is possible to temporarily reverse the logic on the jbe condition so that it behaves in the opposite manner. See the next section on why this change to the executable code should be reverted immediately afterward.

Software Protection Mechanism: “Section Checksums and Validation”

On two occasions during the flow of the program, it computes a checksum of data in the .text section and compares it with a hard-coded 32-bit value. If an analyst makes any modifications to the program’s data (including instructions, operands, logic, etc) then the test will fail. Furthermore, the second checksum actually evaluates the state of the first checksum’s code, to make sure the checksum function itself isn’t altered.

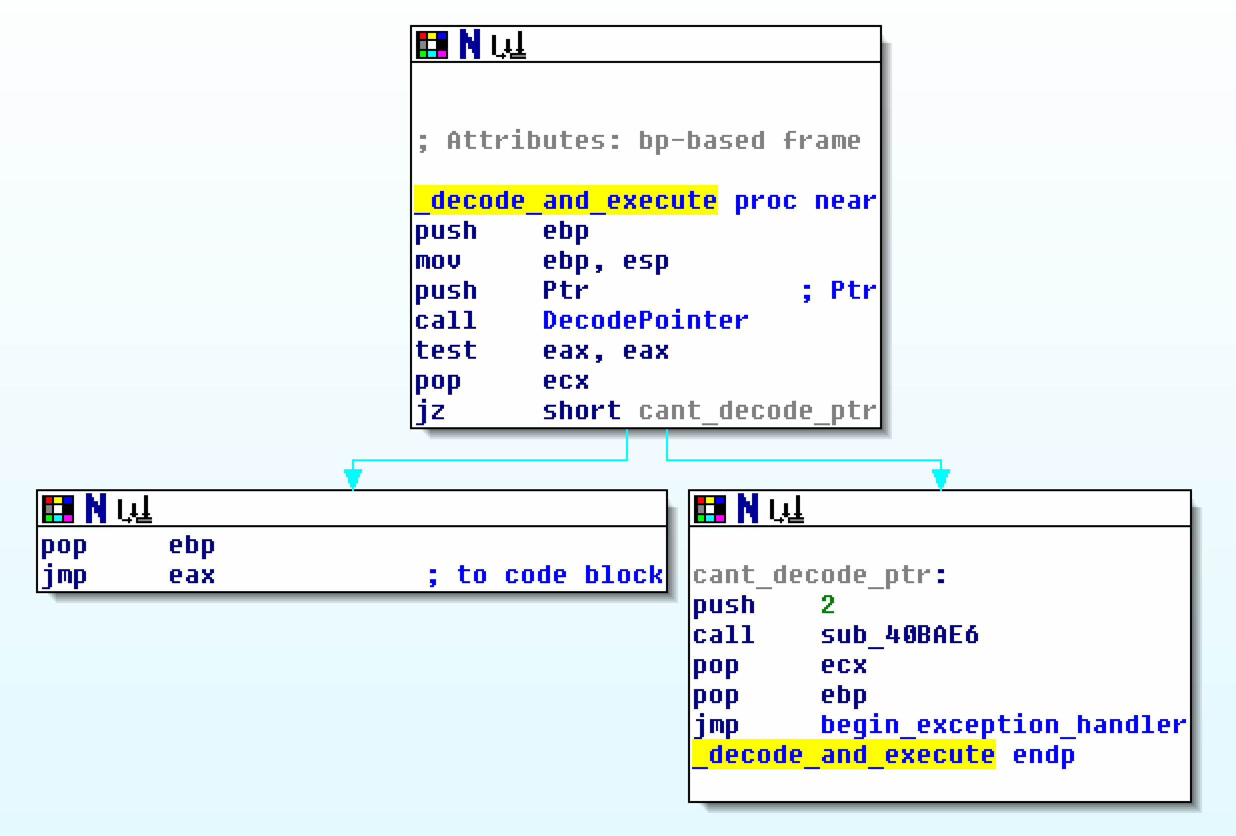


```
checksum_text_section proc near
push    ebx
push    esi
mov     eax, ds:40003Ch
mov     esi, [eax+400104h]
mov     ecx, [eax+400108h]
add     esi, 400000h
mov     ecx, 637Bh
shr     ecx, 2
xor     ebx, ebx

chksum_loop:
lodsd
rol     ebx, cl
xor     ebx, eax
loop    chksum_loop

mov     eax, ebx
pop     esi
pop     ebx
retn
checksum_text_section endp
```

This protection can be defeated by simply being careful and not making any changes to the .text segments without reverting them.

Software Protection Mechanism: "Pointer Encoding"

The program implements the Kernel32@EncodePointer and Kernel32@DecodePointer API calls to obfuscate function pointers at run time. While these APIs are designed for software security (preventing an attacker from gaining control of EIP by overwriting a function pointer) and not specifically anti-reverse-engineering, it often accomplishes both goals. This is especially troublesome for following code execution during static analysis, because without a careful trace of parameters, it is difficult to determine where EIP will land after the next call. An example of this usage by final.exe is shown below.



```
; Attributes: bp-based frame

_decode_and_execute proc near
push    ebp
mov     ebp, esp
push    Ptr             ; Ptr
call    DecodePointer
test    eax, eax
pop     ecx
jz      short cant_decode_ptr
```

```
pop     ebp
jmp     eax                     ; to code block
```

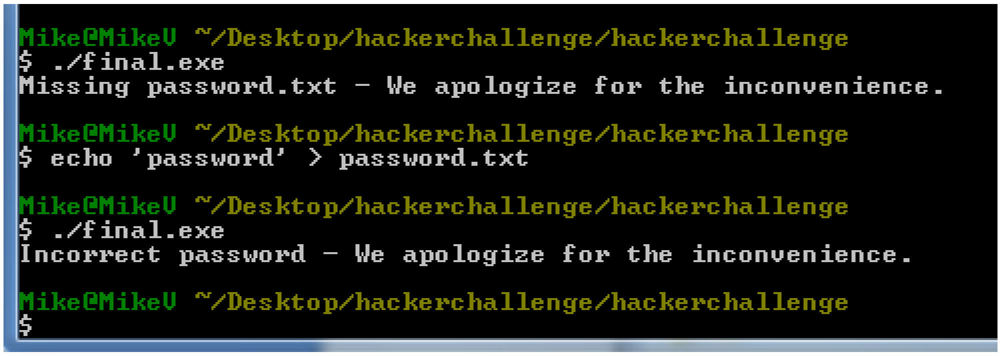
```
cant_decode_ptr:
push    2
call    sub_40BAE6
pop     ecx
pop     ebp
jmp     begin_exception_handler
_decode_and_execute endp
```

The API calls use a pseudo-random XOR key stored in the process information block to encode and decode pointers. It was not necessary to defeat this protection mechanism in order to complete the objectives for the challenge.

Part II: Specific Software Protections

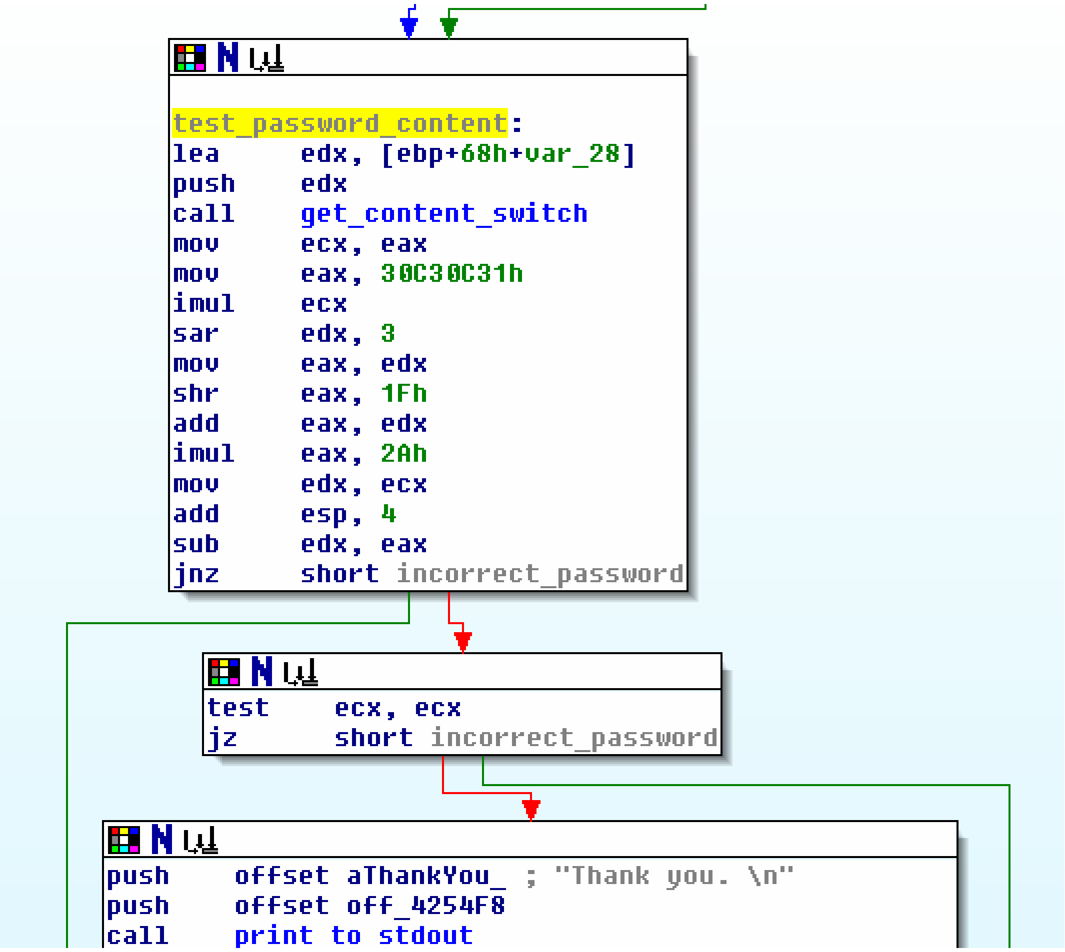
Software Protection Mechanism: "Defeating the Password Requirement"

While generating a password for the program is not one of the two main objectives for phase 1, it is required to make the program execute properly (without modifying the binary). If final.exe is executed for the first time, it will complain that a valid password.txt does not exist. Furthermore, if password.txt does exist, but without a valid password, it will also complain.



```
Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$ ./final.exe
Missing password.txt - We apologize for the inconvenience.

Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$ echo 'password' > password.txt

Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$ ./final.exe
Incorrect password - We apologize for the inconvenience.

Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$
```

Our defeat methodology for this step was rather simple. We navigated to the function in IDA Pro which contains the strings "password.txt," "Incorrect password…," and "Thank you" to examine the surrounding code. It was obvious to see in which direction of the decision tree we need to go in order to encounter the "Thank you" statement instead of the "Incorrect password" statement.



```
test_password_content:
lea     edx, [ebp+68h+var_28]
push    edx
call    get_content_switch
mov     ecx, eax
mov     eax, 30C30C31h
imul    ecx
sar     edx, 3
mov     eax, edx
shr     eax, 1Fh
add     eax, edx
imul    eax, 2Ah
mov     edx, ecx
add     esp, 4
sub     edx, eax
jnz     short incorrect_password
```

```
test    ecx, ecx
jz      short incorrect_password
```

```
push    offset aThankYou_ ; "Thank you. \n"
push    offset off_4254F8
call    print to stdout
```

The function call labeled **get_content_switch** leads to a sub routine that is used throughout the program for multiple purposes. For processing the password content, one of the parameters is a pointer to a NUL-terminated character string composed of the first two bytes in password.txt. We realized this by debugging the program with a password of "abc" and observing the function parameters, one of which was "ab".

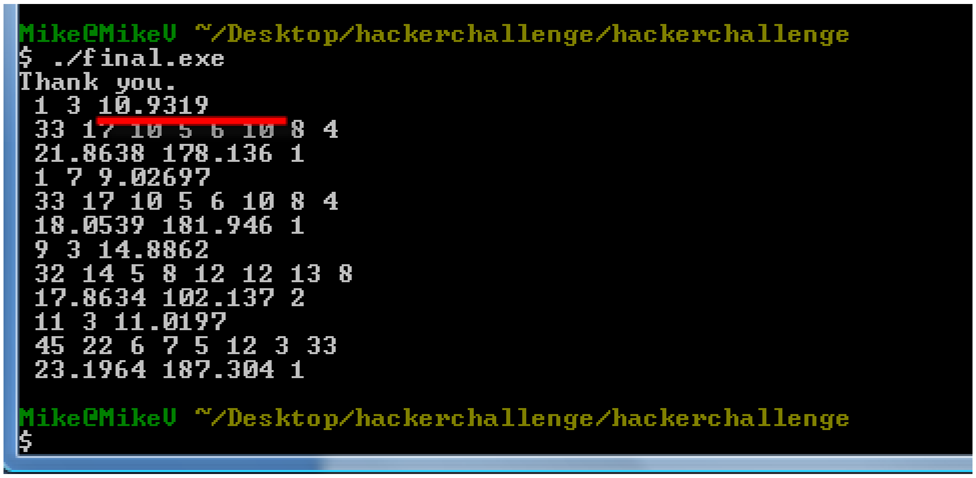
At this point, it was evident that the output of the formula in the node labeled **test_password_content** above is entirely dependent on only the first two bytes of the password. Therefore, a simple program helped brute force a valid password while we focused on the next objectives. The Perl code is shown below.

```
#!/usr/bin/perl
#
# This program brute forces the contents for password.txt that
# we need to gain execution to final.exe. Its based on the idea
# that when the program runs, it only sends the first two bytes
# of password.txt to the validation routine.
#
#.text:00409287   push    0Ah
#.text:00409289   push    0
#.text:0040928B   push    [esp+8+arg_0] ; first two bytes of file
#.text:0040928F   call    sub_4111AF
#
# Therefore, we can know that if this function *ever* returns in
# support of the file's contents, then it will be based on only the
# first two bytes. Move through an array such as:
#
# unsigned char array[0xff][0xff]
#
# As soon as a valid combination is detected through output redir
# and capture, then we break and print the magic bytes. Note:
# output is based in decimal, not hex.
#
# perl brute.pl
# You win: [52][50]
#

sub change_pass {
        my ($i, $j) = @_;
        open(PASS, ">password.txt");
        binmode(PASS);
        print PASS chr($i), chr($j);
        close PASS;
}

sub validate_pass {
        return 1 if (`final.exe` =~ m/thank/i);
        return 0;
}

BRUTE: foreach $i (0..256) {
        foreach $j (0..256) {
                change_pass($i, $j);
                if (validate_pass()) {
                        print "You win: [$i][$j]\n";
                        last BRUTE;
                }
        }
}
```

This program, which is headed by its documentation and sample output, was successful in generating a valid password.txt value (starting with “42”), and allowed us to proceed into the next steps of the challenge.

Software Protection Mechanism: “Reproduce the FP Algorithm”

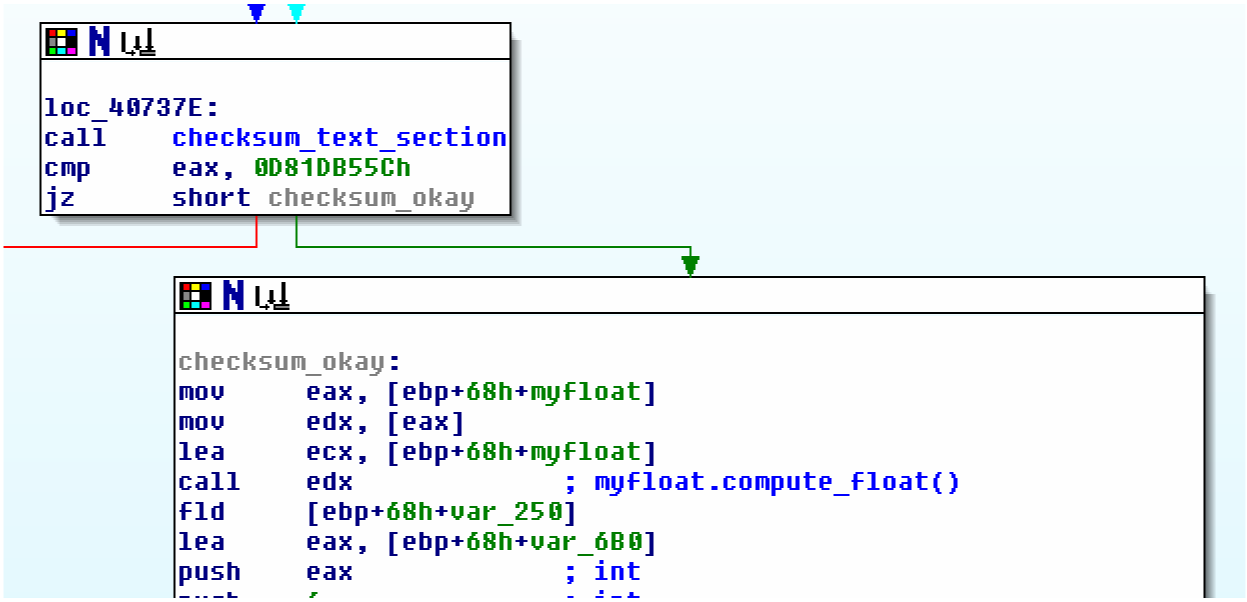
Now that the password is accepted, the final.exe program prints a series of numbers to the terminal. The objective 1 is to reproduce in pseudo code (C/C++) the formula responsible for computing the floating point 10.9319 that is printed as the third element of the first row:



```
Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$ ./final.exe
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
 23.1964 187.304 1

Mike@MikeV ~/Desktop/hackerchallenge/hackerchallenge
$
```

We defeated this objective by first navigating to a point in the program where we were confident about the 10.9319 having already been generated, and then worked backwards. We did it this way because, well, that’s what reverse engineers do sometimes.

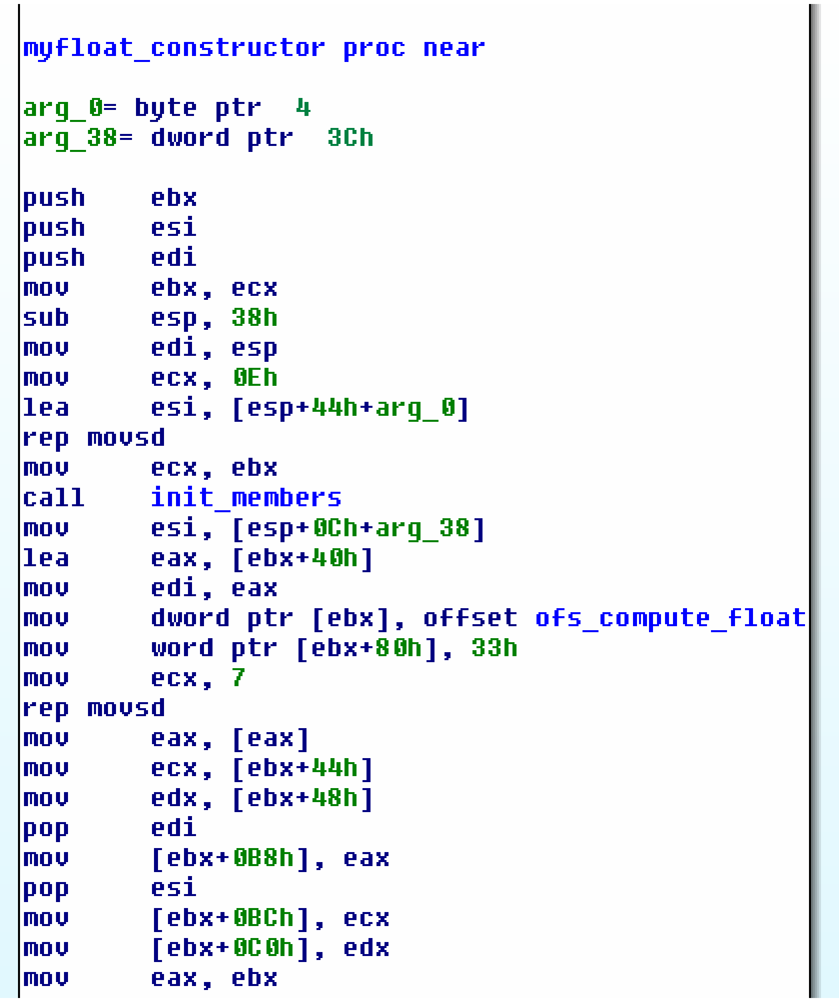
For this case, our confidence point was the print statement used to display the value on screen. Through debugging, we found code of interest just above the print statement, and directly below the first checksum discussed previously. The code is shown below.



```
loc_40737E:
call    checksum_text_section
cmp     eax, 0D81DB55Ch
jz      short checksum_okay
```

```
checksum_okay:
mov     eax, [ebp+68h+myfloat]
mov     edx, [eax]
lea     ecx, [ebp+68h+myfloat]
call    edx             ; myfloat.compute_float()
fld     [ebp+68h+var_250]
lea     eax, [ebp+68h+var_6B0]
push    eax             ; int
```

In the image, we have labeled the local object variable as **myfloat**, and show that the **call edx** instruction is an invocation of the object's only method – **compute_float**. We suspect that the local variable is an object and not a structure based on how it is passed to the function in the **ecx** register, and how the function address is derived by dereferencing the appropriate offset from the base of the local variable. This can further be supported by investigating what appears to be the class constructor, which initializes the members and assigns the proper offset into the function pointer. The code is shown below.



```
myfloat_constructor proc near

arg_0= byte ptr  4
arg_38= dword ptr  3Ch

push    ebx
push    esi
push    edi
mov     ebx, ecx
sub     esp, 38h
mov     edi, esp
mov     ecx, 0Eh
lea     esi, [esp+44h+arg_0]
rep movsd
mov     ecx, ebx
call    init_members
mov     esi, [esp+0Ch+arg_38]
lea     eax, [ebx+40h]
mov     edi, eax
mov     dword ptr [ebx], offset ofs_compute_float
mov     word ptr [ebx+80h], 33h
mov     ecx, 7
rep movsd
mov     eax, [eax]
mov     ecx, [ebx+44h]
mov     edx, [ebx+48h]
pop     edi
mov     [ebx+0B8h], eax
pop     esi
mov     [ebx+0BCh], ecx
mov     [ebx+0C0h], edx
mov     eax, ebx
```

Now, with this background, we are in a nice position to tackle the objective. We need to figure out three main components:

1) which members of the object are used in the computation;
2) the type, size, and exact values of those members that lead to 10.9319; and
3) the formula itself.

All of the components are equally simple, though a bit time consuming to determine. We can learn the information for the first step by examining the **compute_float** function and observing offsets from the base of the object that are used as input. At the same time, we will learn their type and size by examining the instructions that handle them. For example, if we see **add eax, [esi+0xbc]** then we know that the member at offset 0xbc is 32 bits in size. If we see **fadd dbl_4248c0** then we can be relatively certain that the global variable at address 0x4248c0 is 64 bits (a double) in size. We can cross-reference this information with the code from the constructor that initializes the members.

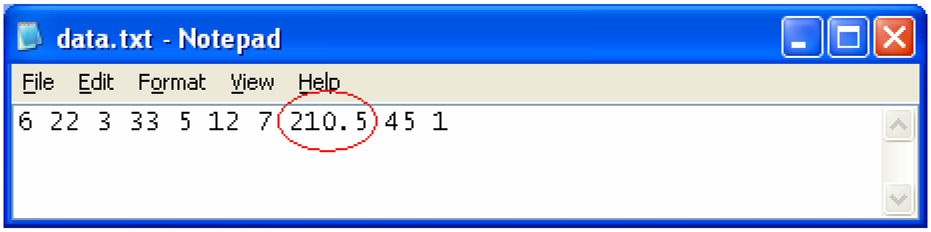
As for component 3, the formula, we reverse engineered it into source code using the disassembly provided by IDA Pro and two references for FPU instructions ("32/64-bit Assembly Language Architecture" by James C Leiterman, and the online NASM manual). Although only pseudo code is required for meeting the objective, we wrote compile-able code to ensure that no mistakes existed in the formula. The program is shown below. Note that this is not an exact reproduction of the code used within final.exe but rather a functional equivalent.

```
#include <stdio.h>
#include <string.h>


// the "m" members are labeled by their offset within the objects as
// indicated in the reverse of the final.exe executable. For instance
// m98 is +0x98 from the start of the class.

// base object
class basefloat
{
public:
      basefloat() { }

      virtual void compute_float(void){ };

      double m98;  // [out] stores 10.9319
      double ma0;  // [out] stores 187.304
      double ma8;  // [out] stores 23.1964
      int mb8;     // [in]  data_txt.m_0        -- parameter
      int mbc;     // [in]  data_txt.m_1        -- parameter
      int mc0;     // [in]  data_txt.m_2        -- parameter

};

// object which contains the equation for objective 1 (derived from the
basefloat class)
class myfloat : basefloat
{
public:

      myfloat() { }

      virtual void compute_float(void);
      virtual void initializeVectors(int a, int b, int c)
            { mb8 = a; mbc = b; mc0 = c; }

      double m28;       // [in]                 -- parameter
      int m30;          // [in]                 -- parameter

};
```

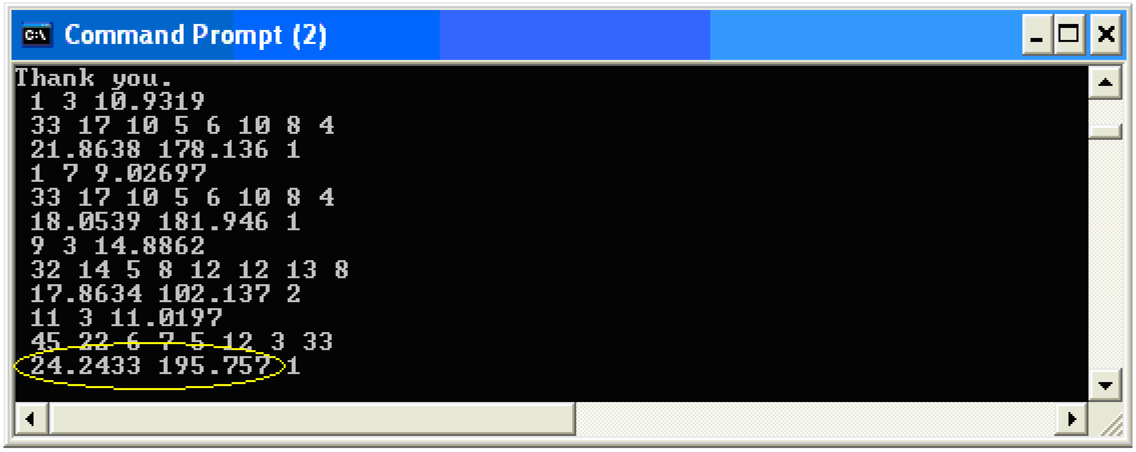
```
// constants (address of constant is indicated in the comment)
const double const_dbl_41E228 = 3.142857142857143;    // 0x41E228
const double c0 = 1.10938;                            // 0x41E218
const double c1 = 8.267e-4;                           // 0x41E220
const double c2 = 1.6e-6;                             // 0x41E210
const double c3 = 2.574e-4;                           // 0x41E208
const double c4 = 4.5e2;                              // 0x41E1B8

// globals (address of global is indicated in the comment)
int g1 = 0x1f4;                                       // 0x423068
double g2 = 0.0;                                      // 0x4248C0
int g4 = 0xa;                                         // 0x423070
int g3 = 0xa;                                         // 0x42306C

void myfloat::compute_float()
{

      // phase I, object I equation
      m98 = ((g1 / (((c0 - ((mbc + mb8 + mc0) * c1))
            + ((mbc + mb8 + mc0)*(mbc + mb8 + mc0) * c2))
            - (c3 * m30))) + g2)
            - c4;

      // rest of the function which contains the P1O1 equation
      ma8 = (m98 / (g4 * g3)) * m28;
      ma0 = m28 - ma8;

}

// for completeness, this will allow the reader to test the accuracy of
// the equation reversed for objective 1.
int main (int argc, char * argv[])
{
      myfloat ns;


      /* configure input for generating the 10.9319 float */
      ns.initializeVectors(0x0a, 0x11, 0x08);
      ns.m30 = 0x21;
      ns.m28 = 200;
      g1 = 0x1ef;
      ns.compute_float();


      return 0;
}
```

Software Protection Mechanism: "Anti-Tamper: Remove the Upper Limit"

The second objective in phase 1 of the challenge is to remove the upper limit on the 8th field from data.txt, such that the program treats values greater than 210.5 in the same manner as it treats values equal to or less than 210.5. This would involve altering either the program itself, or its input, to successfully print the two circled values (of the bottom image) when 210.5 is increased to 220. The following images are taken from README.doc of the challenge instruction sheet:



```
6 22 3 33 5 12 7 210.5 45 1
```



```
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
```

We were admittedly thrown off course a bit in this stage after having reversed the formula that computes these numbers. We know that in this particular call to the **compute_float** function, the object's members which are uses as input in the calculation, are initialized with the first, second, third, eighth (must be 220), and ninth fields from data.txt. It was also evident through both static and dynamic analysis that the global variables which serve as additional input to the computation are the same as those used in other calls to **compute_float**, such as the one that generates 10.9319. Based on this information, we mistakenly jumped to the conclusion that we could simply brute force the 4 variable input fields until they resulted in 24.2433 and 195.7577. This didn't work, and it's what we get for trying to take shortcuts.

Instead, we re-thought the attack method and decided to take a different approach. It involved tracing the input value of 220 from the point the program first receives it from data.txt until **compute_float** is called and then return values are printed on the screen. Theoretically, we should be able to follow the value and determine exactly when it is assigned to the appropriate object member (the double at offset 0x28) and figure out its involvement in the upper limit.

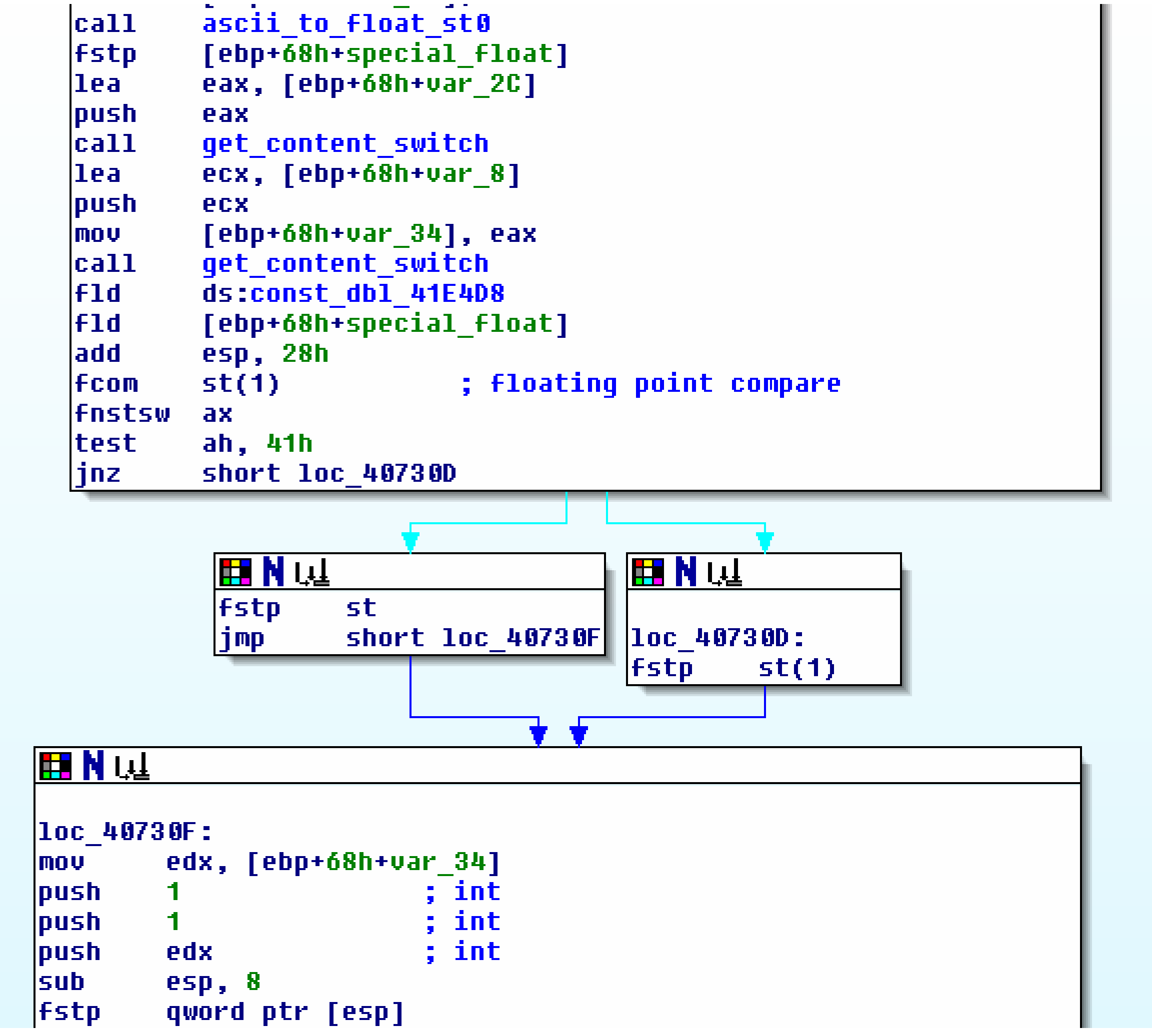
Using the new approach, we located the approximate point where each value from data.txt entered the program. They are gathered during a series of calls to the **get_content_switch** function discussed previously in this report. After acquiring the special 220 value, there is a call to the routine labeled **ascii_to_float_st0** in the image below. The label indicates that the function accepts an ASCII value (e.g. char * asc = "220";), converts it into a float, and loads it into FPU ST0.

The next operation involving the FPU is an **fstp** instruction which pops the value in ST0 into the local variable labeled **special_float**. At this point, **special_float** contains 220, and so tracking its usage is most important to us. The value is reloaded onto the FPU stack in the second of two **fld** instructions, the first of which loads the value at address 0x41e4d8. This is a global (and constant) 64-bit double that resides in the .rdata section and is initialized to equal 210.5 – the very value that represents the upper limit.

The values in ST0 (210.5) and ST1 (220) are then compared with **fcom st(1)**, and based on the value left in the FPU status word after this comparison, the smaller of the two is placed into ST0 by using either the **fstp st** instruction or **fstp st(1)** instruction (see the conditional jump below). Nearing the end, the value that remains in ST0 is saved into the double pointed to by **esp**. It is this value that survives the computations in order to eventually end up the member at 0x28 of the object when **compute_float** is called.

In other words, if the input value is greater than the constant, then the input value is reset to the value of the constant before being used. The observed behavior can be summarized in a manner such as the following:

```
const double dbl_const_val = 210.5;
double dbl_input_val = 220;

if (dbl_input_val > dbl_const_val) {
      dbl_input_val = dbl_const_val;
}
```



```
call    ascii_to_float_st0
fstp    [ebp+68h+special_float]
lea     eax, [ebp+68h+var_2C]
push    eax
call    get_content_switch
lea     ecx, [ebp+68h+var_8]
push    ecx
mov     [ebp+68h+var_34], eax
call    get_content_switch
fld     ds:const_dbl_41E4D8
fld     [ebp+68h+special_float]
add     esp, 28h
fcom    st(1)           ; floating point compare
fnstsw  ax
test    ah, 41h
jnz     short loc_40730D
```

```
fstp    st
jmp     short loc_40730F
```

```
loc_40730D:
fstp    st(1)
```

```
loc_40730F:
mov     edx, [ebp+68h+var_34]
push    1               ; int
push    1               ; int
push    edx             ; int
sub     esp, 8
fstp    qword ptr [esp]
```

To defeat this protection, it crossed our minds to change the flow of execution such that the logic is reversed, allowing the larger number of the two to be selected. However, then we would need to adjust the logic of the two checksum functions also. Furthermore, this would require studying the unpacking algorithm closer, in order to make all three changes to the unpacked binary.

It was a better idea, however, to alter the value of the constant, which as mentioned resides in the .rdata section and is not included in the run-time checksum detections *or* the packing algorithm. Therefore, we can easily make a change to the program with a hex editor such that the constant itself is the maximum value of a double, and therefore there is no upper limit to the 8th value of data.txt that will result in it being treated differently than lower values.

The final solution to this objective would be to replace the constant of 210.5 with the DBL_MAX value from float.h:

```
#define DBL_MAX  1.7976931348623158e+308 /* max value */
```

Time to Break

Phase 1 of the challenge required an estimated 28-36 hours of work, split between two individuals and across 5 days. Although the challenge started on August 27th, unfortunately we didn't begin until September, 3rd. All of the generic software protection mechanisms that required defeat were broken within a very short time period – 1 to 5 minutes each. The password protection was broken in a considerably smaller amount of time (2-3 hours) than it took to reverse the formula and tamper with the upper limit. We spent less than 30 minutes researching topics on the Internet, most of which was related to FPU behaviors.

Tools Used

Our primary tools consisted of a debugger and disassembler. We have summarized the tools below.

1) Immunity Debugger, OllyDbg, and associated plugins, for dynamic analysis of the binary;
2) IDA Pro, for static and dynamic analysis of the binary;
3) PEInfo, for exploration of the binary's sections and headers;
4) Microsoft Visual Studio C++ and GCC, for compilation of the formula;
5) Camtasia Studio SnagIT, for screen captures and manipulation;
6) UltraEdit and FlexHex, for investigation and modification of the binary on disk;
7) The Perl programming language and Cygwin environment;
8) Our eyes, brains, fingers, and sometimes feet.

Conclusion

This phase of the challenge was fairly accommodating to our inquisitive nature and thrill-seeking, problem-solving desires. The generic anti-reverse-engineering protections were clearly not sufficient, and since the ability to defeat the specific protections relies on the ability to bypass the generic ones, this is a weakness in the challenge. There are several additional anti-debugging tricks that could increase the difficulty, such as inspection of other registers (DR0-DR3), usage of other time-telling instructions (rdtsc or Ntdll@NtQueryInformationProcess), and checking for signs of an active debugger throughout the system (memory, file system, registry).

All of these listed options can be defeated as well, but in combination they can help increase the adversary work factor involved in defeating the software protections. We would recommend researching the protections implemented by Oreans Technologies Themida, which provides even more options for protecting software from analysis and/or modifications.

As for the specific protections, the authors of the software that implements these protections should decide for themselves if they are sufficient, based on our most accurate estimations of the time required to defeat them. It would be relative to the worth of the software being protected. The amount of time we spent over the course of 5 days is a lot for two individuals with full-time jobs, wives/girlfriends, and recently – new pets. 5 days is not a lot for a company attempting to secure their proprietary information from software pirates and criminal hackers.