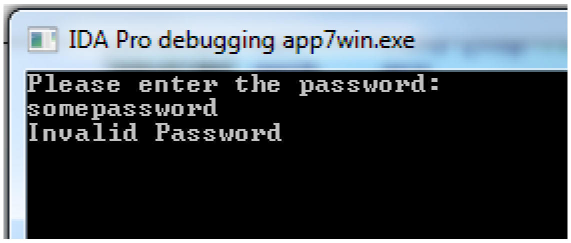
I recently stumbled upon Hack This Site, a site that offers “missions” relating to web security, application reversing, programming, and general hacking topics. I quickly completed all of their application challenges, with the exception of the last two. Along the way, I found an interesting one — application challenge #7. I chose to post about this one since there was an interesting twist in the application which is probably why it was rated as a medium level challenge instead of easy.

The Analysis

This challenge comes with two files, the executable and a file called “encrypted.enc” that the executable uses. From opening up encrypted.enc in a hex editor, it is obvious to see that the contents of encrypted.enc are encrypted or obfuscated in some way. As a result, analyzing it as a standalone file won’t really provide any information. The executable will have to be reverse engineered to see just how it manipulates this file and how it derives the password.

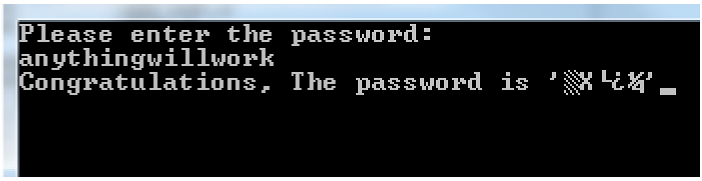


```
Please enter the password:
somepassword
Invalid Password
```

Upon startup, the application simply asks for a password. Providing the wrong password simply causes it to display “Invalid Password” and terminate. Taking a look at how this occurs reveals a lot of information about how the password is stored and derived.

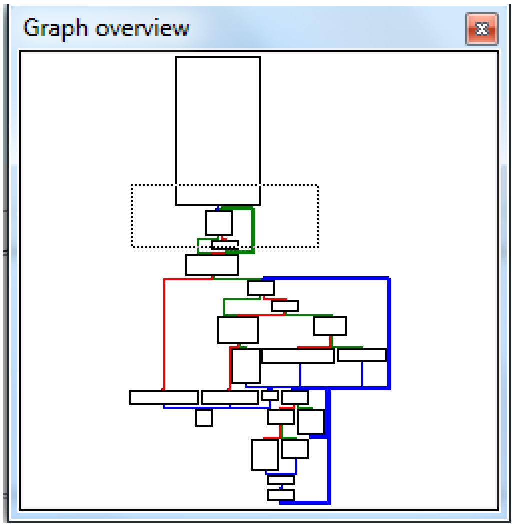
```
.text:0040118C                 cmp     [ebp+var_18], 0DCAh
.text:00401193                 jnz     short loc_4011A8
.text:00401195                 lea     ecx, [ebp+Dst]
.text:00401198                 push    ecx
.text:00401199                 push    offset Format   ; "Congratulations,
The password is '%s'"
.text:0040119E                 call    _printf
.text:004011A3                 add     esp, 8
.text:004011A6                 jmp     short loc_4011B5
.text:004011A8 ; ---------------------------------------------------------------
---------------
.text:004011A8
.text:004011A8 loc_4011A8:                             ; CODE XREF:
_main+193↑j
.text:004011A8                 push    offset aInvalidPasswor ; "Invalid
Password"
.text:004011AD                 call    _puts
```

The application jumps to the “Invalid Password” location at loc_4011A8 if the value stored in [ebp+var_18] does not equal 0DCAh (3530d). Otherwise it continues on to the congratulations message. At first instinct, it might be tempting to simply NOP out the jump or to switch it to whatever [ebp+var_18] is during runtime so the congratulations message is hit. However, this causes some problems as shown below. The comparison was replaced with a NOP instruction and the image below shows what happened as a result.



```
Please enter the password:
anythingwillwork
Congratulations, The password is '▒X└¿¼'_
```

The password becomes unintelligible and the website where you eventually submit the solution rejects it. It is interesting to look at why this occurs. Thus, begins the actual analysis of how this works.



The program flow is simple enough. A few loops and conditionals are the main parts of how this functions.

Identifying the Variables

IDA identifies the following variables used by the application

```
.text:00401000 File                  = dword ptr -2Ch
.text:00401000 var_28                = dword ptr -28h
.text:00401000 var_24                = dword ptr -24h
.text:00401000 DstBuf                = dword ptr -20h
.text:00401000 var_1C                = dword ptr -1Ch
.text:00401000 var_18                = dword ptr -18h
.text:00401000 Dst                   = byte ptr -14h
.text:00401000 var_4                 = byte ptr -4
```

Some were identified in the initial autoanalysis phase by IDA so only five remain to be manually identified. The first ones to occur are [ebp+var_4] and [ebp+var1C]:

```
.text:0040103F loc_40103F:                                    ; CODE XREF:
_main+62↑j
.text:0040103F                 call    j___fgetchar
.text:00401044                 mov     [ebp+var_4], al
.text:00401047                 movsx   ecx, [ebp+var_4]
.text:0040104B                 mov     edx, [ebp+var_1C]
```

```
.text:0040104E                 add     edx, ecx
.text:00401050                 mov     [ebp+var_1C], edx
.text:00401053                 movsx   eax, [ebp+var_4]
.text:00401057                 cmp     eax, 0Ah
.text:0040105A                 jz      short loc_401064
.text:0040105C                 movsx   ecx, [ebp+var_4]
.text:00401060                 test    ecx, ecx
.text:00401062                 jnz     short loc_40103F
```

Looking at this block, it is easy to see that it is a loop, as evidenced by the conditional jnz instruction back to the top at .text:00401062. [ebp+var_4] stores the value returned from fgetchar, which is identified as

```
The next character from the input stream pointed to by stdin. If the stream
is at end-of-file, the end-of-file indicator is set, and the function returns
EOF. If a read error occurs, the error indicator is set, and the function
returns EOF.
```

[ebp+var_1C] then stores the sum of all of the characters that have been read (including the 0Ah line feed character when the enter key is pressed). It is obvious at this point to see that this loop is responsible for reading in the user supplied password. It terminates when the 0Ah line feed character is read from the stream. From here on, [ebp+var_4] will be referenced as [ebp+input_char] and [ebp+var_1C] will be references as [ebp+input_sum]. The application then continues on to open the encrypted.enc file and check for a valid FILE*. Then begins the bulk of the program. The first block does something interesting with the local variable [ebp+var_24]:

```
.text:00401093 loc_401093:                             ; CODE XREF:
_main+7D↑j
.text:00401093                                         ; _main+17B↓j
.text:00401093                 mov     edx, [ebp+var_24]
.text:00401096                 and     edx, 4
.text:00401099                 test    edx, edx
.text:0040109B                 jz      short loc_4010AB
.text:0040109D                 mov     eax, [ebp+var_24]
.text:004010A0                 and     eax, 1
.text:004010A3                 test    eax, eax
.text:004010A5                 jnz     loc_401180
```

[ebp+var_24] has an “and” operation performed with it and checked to see if the zero flag is set. If it is, the program jumps to loc_4010AB to continue execution; otherwise, [ebp+var_24] again has an “and” operation against 1 and jumps out to loc_401180 if the result is 0. IDA gives the helpful hint that this is a loop (as evidenced by the arrows), but it is easy to see without help that this is true by looking a bit further down.

```
.text:00401175                 add     edx, 1
.text:00401178                 mov     [ebp+var_24], edx
.text:0040117B                 jmp     loc_401093
```

[ebp+var_24] is incremented by 1 and a jump is made back to the beginning of the loop. Therefore, it is easy to deduce that the block with the “and” operations is the conditional part of the loop. Studying the structure, it is two statements connected by an “or” operation. The body of the loop will execute as long as (([ebp+var_24] & 4) == 0) || (([ebp+var_24] & 1) == 0). Looking at for what values this works for reveals that the values 0-4 satisfy this conditional. Since [ebp+var_24] is initialized to 0 at the start of the program, this is a loop counter that runs from 0 to 4. From here on, [ebp+var_24] will be referred to as [ebp+dst_index] (shown why later). Assuming normal execution, the program then continues by calling fread at

```
.text:004010B7                 call    sub_401279
```

This is deduced because of the comments noted in IDA, or by simply following the call into until it hits _fread. That block was responsible for reading a character from the encrypted.enc file and storing the character in the [ebp+DstBuf] array. The block that follows at

```
.text:004010D8 loc_4010D8:                             ; CODE XREF:
_main+C2↑j
.text:004010D8                 mov     eax, [ebp+DstBuf]
.text:004010DB                 and     eax, 0FFh
.text:004010E0                 xor     eax, [ebp+input_sum]
.text:004010E3                 mov     ecx, [ebp+var_18]
.text:004010E6                 add     ecx, eax
.text:004010E8                 mov     [ebp+var_18], ecx
.text:004010EB                 mov     edx, [ebp+DstBuf]
.text:004010EE                 and     edx, 0FFh
.text:004010F4                 xor     edx, [ebp+input_sum]
.text:004010F7                 mov     eax, [ebp+dst_index]
.text:004010FA                 mov     [ebp+eax+Dst], dl
.text:004010FE                 mov     [ebp+var_28], 0
.text:00401105                 jmp     short loc_401110
```

is probably the most important block of the entire application. Remembering from earlier, [ebp+var_18] is compared against 0DCAh to see whether the correct password was supplied or not. Looking at what happens in this block, it is shown that a character at [ebp+DstBuf] is moved into eax and has an xor performed against the input sum of the user supplied password. Then its sum is stored in [ebp+var_18] so [ebp+var_18] will be referred to as [ebp+xor_sum] from here on. At this point it is actually possible to deduce how the program works and what steps are required to get a working password. This is because [ebp+xor_sum] is not written to anywhere else for the remainder of the program so anything that happens has no effect on the outcome of the comparison with 0DCAh. Also, if analyzed closely, the number of characters of the password is known (discussed later). This would allow an easy brute-force approach since the way to get the compared sum is known and the number of characters in the password (a very low amount) is known. However, for practice, it is interesting to see how the actual password decoding algorithm works. The analysis won't be as detailed as the required parts, but still provides an overview of how the program behaves. The rest of this block shows that the character as [ebp+eax+Dst] is set to the xor of the input sum and the character that was read. Then a jump is taken to loc_401110. Here [ebp+var_28] makes its appearance in usage (it was set to 0 prior to the jump into loc_401110). It is not obvious at first sight what [ebp+var_28] is used for, just that in this jump it is compared against the the input sum. Ignoring the jump instruction that leaves this block, the code continues on and performs an if-else comparison of [ebp+eax+Dst] "and" 1. Both of these blocks have the same exit point, their last instruction always jumps back in the code to

```
jmp     short loc_401107
```

so this is the beginning of another loop, more specifically, a for loop because of how the instructions are organized (the entry into the loop jumped past the increment instructions at loc_401107). Using this knowledge, it is possible to conclude that [ebp+var_28] is actually a counter for a for loop and will be referred to as [ebp+counter] from here on. Going back to the if-else discovered earlier, the code in the “if” body does an arithmetic shift of the value of the array index to the right by 1 and performs an or with 80h. In the else block, the same thing occurs, except there is no or with 80h. Once this for loop exists, the value in the current index has 3h added to it and the index is increased for the next iteration of the topmost loop. Knowing all this, it is possible to reproduce how this program works. The below programs behaves just like the application. It doesn’t exactly match the diassembly of the application (I took the liberty of making few stylistic changes for readability), but it captures the functionality:

Putting Everything Together

```
#include <stdio.h>

#define CHARS_TO_READ 5

int main(int argc, char* argv[])
{
        unsigned char dst[16] = {0};
        unsigned char enc_char;
        int input_char = 0;
        int input_sum = 0;
        int xor_sum = 0;
        int index = 0;
        printf("Please enter the password:\n");
        while(input_char != 0xA && input_char != EOF)
        {
                input_char = fgetchar();
                input_sum += input_char;
        }
        FILE* enc_file = fopen("encrypted.enc", "rb");
        if(enc_file)
        {
                while(index < CHARS_TO_READ)
                {
                        if(fread(&enc_char, sizeof(unsigned char), 1, enc_file)
!= 1)
                        {
                                printf("An error occured.\n");
                                return 0;
                        }
                        xor_sum += (input_sum ^ enc_char);
                        dst[index] = (input_sum ^ enc_char);
                        for(int i = 0; i < input_sum; ++i)
                        {
                                if(dst[index] & 1)
                                        dst[index] = ((dst[index] >> 1) |
0x80);
                                else
```

```
                                        dst[index] = (dst[index] >> 1);
                        }
                        dst[index] += 3;
                        index++;
                }
                if(xor_sum == 0xDCA)
                        printf("Congratulations, the password is %s\n", dst);
                else
                        printf("Invalid password.\n");
        }
        else
                printf("Failed to open encrypted.enc\n");
        return 0;
}
```

The xor_sum serves as an xor key for the first five characters in the encrypted.enc file. As the algorithm runs through, these five characters are decoded to form the five letter password that solves the solution. In the spirit of the application challenge, the password won’t be disclosed; however, this post is more than enough information to know how to solve the challenge (arithmetically or brute force).

A copy of this post is available as a downloadable PDF here