# Flare-On 3: Challenge 1 Solution

**Challenge Author: Alex Rich**

When running `challenge1.exe` we are presented with a password prompt, for which the program will respond to an incorrect response with "Wrong password".


Figure 1: Screenshot of challenge1 wrong password

Since we are looking to find a specific password, we could start off by checking strings on the challenge using the Microsoft Sysinternals "Strings" tool. Two of the most interesting strings that would be revealed are:

```
String1: x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q
String2: ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+/
```

Figure 2: Interesting strings from Strings tool output

Unfortunately, neither of these strings works as the password, so the next step would be to open this challenge up in a disassembler. The freeware version of IDA Pro will work fine in this case. The `main()` function is located at address 0x401420.

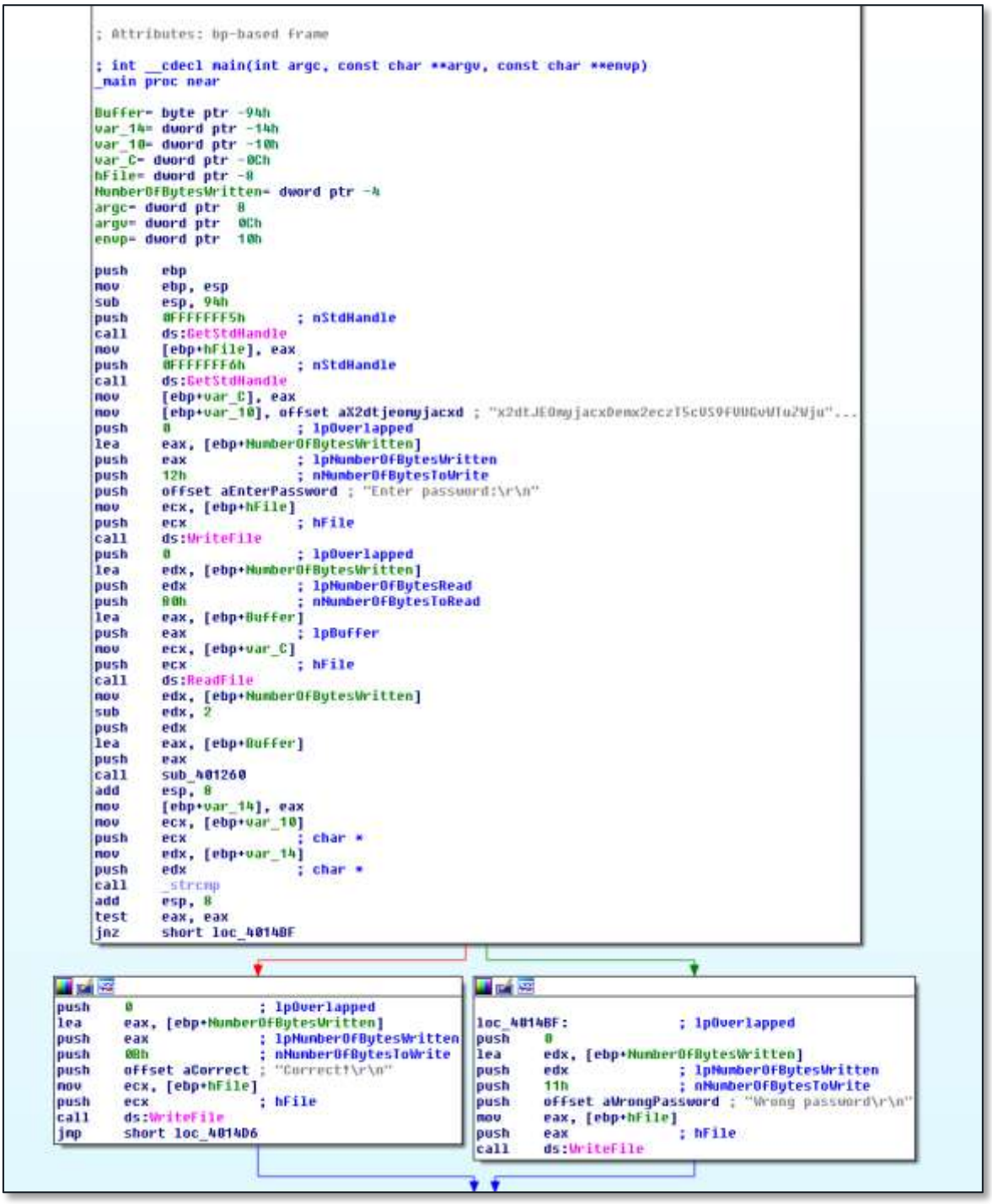


Figure 3: IDA Pro screenshot of challenge1 main()

Upon looking at the control flow of the program (shown in Figure 3), we should see two calls to

`GetStdHandle()`, an API used to retrieve the handles for the Input and Output standard I/O streams.

We can get IDA Pro to show the specific handles being requested by right clicking on the value being used as the parameter to `GetStdHandle()` (eg. 0FFFFFFF5h) and selecting "Use standard symbolic constant", then picking from the next dialog the constant matching STD_*_HANDLE. The return values from these functions, recorded in the `eax` register, are then stored in stack variables that we can rename appropriately for greater clarity.

```
push    STD_OUTPUT_HANDLE ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+stdout], eax
push    STD_INPUT_HANDLE ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+stdin], eax
```

Figure 4: Marked up calls to GetStdHandle() for challenge1

Next in `main()`, the first of the strings that we found originally from running the strings tool is shown getting assigned to a variable labeled by IDA as stack variable `var_10`, which we can rename to "`important_string`".

```
mov     [ebp+important_string], offset aX2dtjeomyjacxd ; "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWju"...
```

Figure 5: Interesting string in challenge1

Then "`Enter Password:`" text is written to the console's standard output handle using `WriteFile()`, and user input is read from the console standard input handle via `ReadFile()`. The input password is stored in the stack variable that IDA labels as Buffer and can be renamed to "`input_password`".

```
push    0               ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax             ; lpNumberOfBytesWritten
push    12h             ; nNumberOfBytesToWrite
push    offset aEnterPassword ; "Enter password:\r\n"
mov     ecx, [ebp+stdout]
push    ecx             ; hFile
call    ds:WriteFile
push    0               ; lpOverlapped
lea     edx, [ebp+NumberOfBytesWritten]
push    edx             ; lpNumberOfBytesRead
push    80h             ; nNumberOfBytesToRead
lea     eax, [ebp+input_password]
push    eax             ; lpBuffer
mov     ecx, [ebp+stdin]
push    ecx             ; hFile
call    ds:ReadFile
```

Figure 6: Console output and input in challenge1

This password input is then used as a parameter to an unknown function `sub_401260`, along with the password length. The result from this function is compared with the interesting string in the variable we named "`important_string`", to determine whether the password is correct. This comparison happens in the `_strcmp()` function (at address 0x402c30). If the return value (in register eax) from `_strcmp` is zero, this means the strings were equal, and we will follow the path to writing "`Correct!`" to the console.

Function sub_401260 is therefore responsible for massaging the password input into a form that can be compared with "`important_string`" and is the key to solving this challenge. We can name this function "`modify_password`", and its return value "`modified_password`."

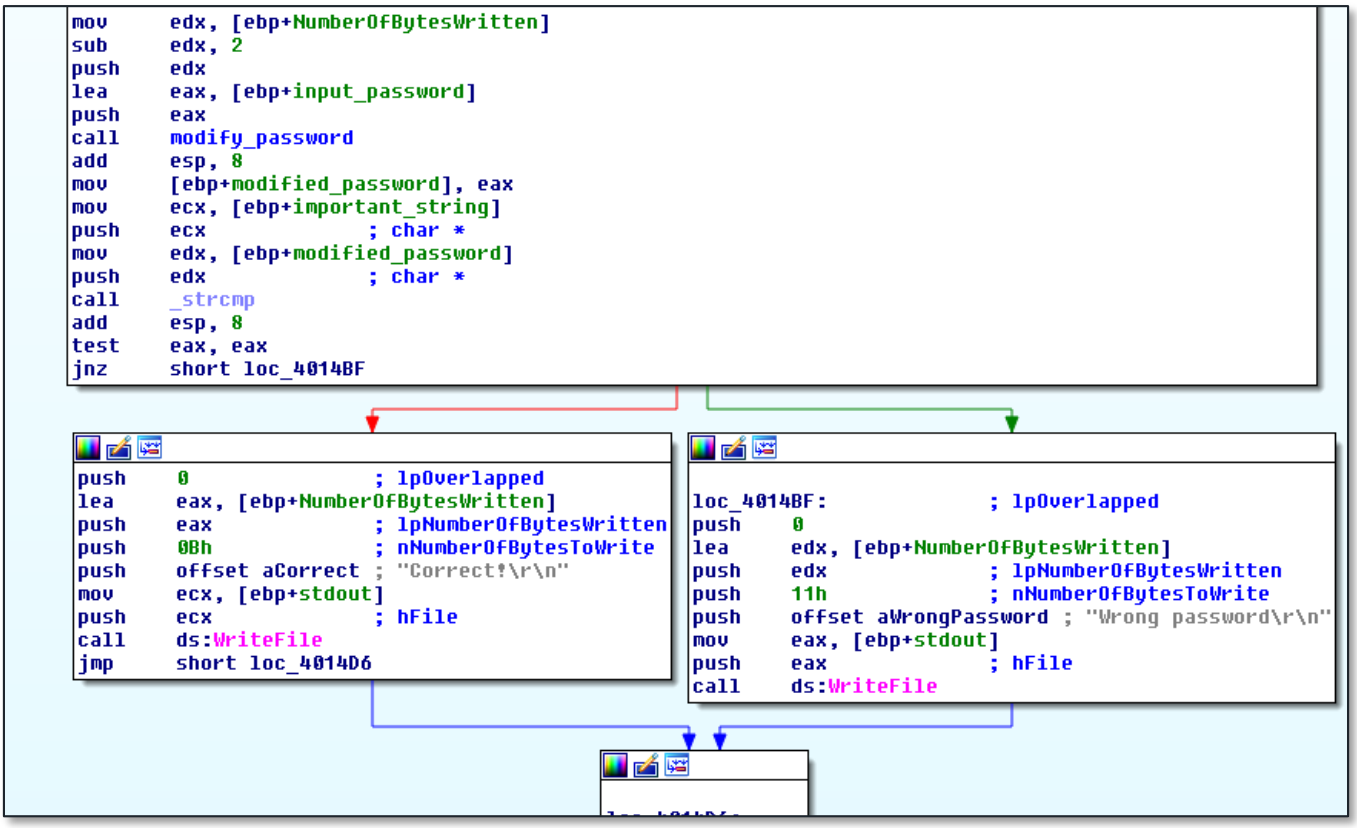


Figure 7: Modify password and compare with important string

The `modify_password` function will initially appear complicated. However, there are a couple of clues we can look for to figure out a standard function is being used here. First, the reference to constant 0x3d, which is the ASCII character code for '='.

Second, references to a 64 character array at 0x413000 (which points to the second interesting string that was identified earlier with the strings tool).

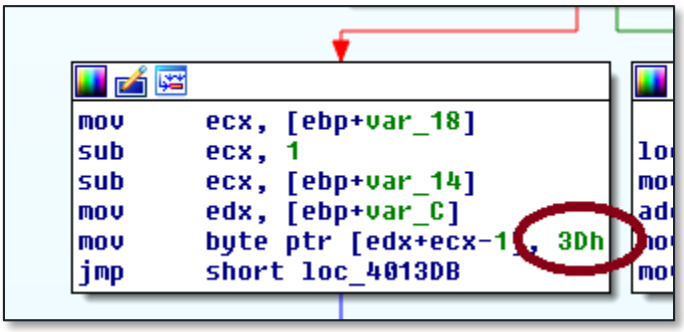


Figure 8: Reference to 0x3d (=) padding character in the modify password function

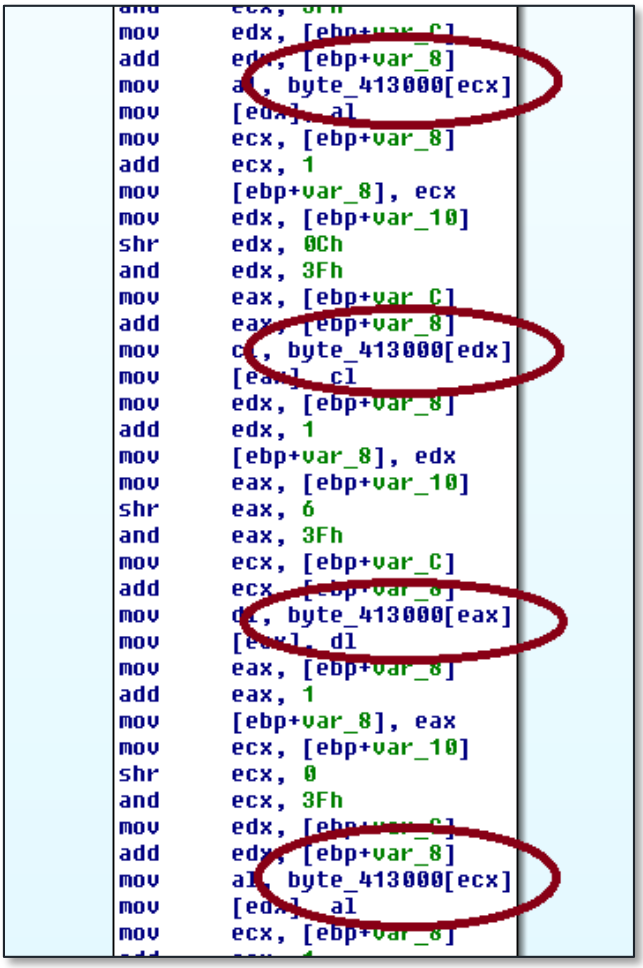


Figure 9: References to 64 character index string in the modify password function

These observations would suggest that this is the Base64 algorithm, which typically uses "=" as a padding character and uses a 64 character array index.

An experienced malware analyst may also have noticed that the first interesting string,

`x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q`, is made up of a pattern of both lowercase and uppercase letters as well as numerals and guessed that it was a Base64 encoded string.

Base64 decoding can be implemented fairly easily in python with the following script:

```
import base64
encoded_string = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q"

print base64.b64decode(encoded_string)
```

Figure 10: Regular Base64 Script

However the output from this is unfortunately nonsensical. The clue to solving this issue is in the second interesting string:

`ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+/`

This is the 64 character byte array at 0x413000. A normal Base64 algorithm uses the following indexing string in the encoding process:

`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`

This challenge has modified that indexing string by moving a few letters around, effectively making it into a substitution cipher, so we will have to account for this in the python script by adjusting our input string to compensate.

```
import base64,string
encoded_string = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q"

translated_encoded_string = encoded_string.translate(

string.maketrans("ZYXABCDEFGHIJKLMNOPQRSTUVWzyxabcdefghijklmnopqrstuvw0123456789+
/",

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/")
)
```

Figure 11: Base64 Script with modified alphabet

This second script produces the correct output: `sh00ting_phish_in_a_barrel@flare-on.com`

We can then test this on the challenge to verify its accuracy:


Figure 12: Screenshot of challenge1 correct password