This is a writeup over Cruehead's crackme 2, hopefully providing an intro to IDA and some general thought processes for crackmes and reversing in general. This crackme along with several others can be found at http://www.woodmann.com/RCE-CD-SITES/Quantico/crackme%27.htm.

After downloading and unpacking the zip file we have a RULES file, basically a readme, and a single CRACKME2.EXE.

Loading it up in IDA and poking around for a minute will help to get familiarized with the binary. After a cursory look around, bringing up the strings windows (Shift+F12) shows a wonderful string:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Length</th>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA:004020D6</td>
<td>000000D</td>
<td>C</td>
<td>CrackMe v2.0</td>
</tr>
<tr>
<td>DATA:004020E3</td>
<td>0000001C</td>
<td>C</td>
<td>No need to disasm the code!</td>
</tr>
<tr>
<td>DATA:004020FF</td>
<td>0000005</td>
<td>C</td>
<td>MENU</td>
</tr>
<tr>
<td>DATA:00402104</td>
<td>000000A</td>
<td>C</td>
<td>DLG_REGIS</td>
</tr>
<tr>
<td>DATA:0040210E</td>
<td>000000E</td>
<td>C</td>
<td>DLG_ABOUT</td>
</tr>
<tr>
<td>DATA:00402119</td>
<td>000000B</td>
<td>C</td>
<td>Good work!</td>
</tr>
<tr>
<td>DATA:00402124</td>
<td>0000002C</td>
<td>C</td>
<td>Great work, mate!\rNow try the next CrackMe!</td>
</tr>
</tbody>
</table>
```

Double click the string takes us to the string in the DATA section of the binary, so we press 'x' and follow the string's xref to a function sub_401334. This function just pushes a couple of “Great work!” strings, then makes a call to MessageBoxA, this is the success function we want to hit.
By clicking on the name of function sub_401334 and again pressing the 'x' key we can follow this functions xref to its only caller, WndProc:loc_40124A. This success location is the result of a successful jz call directly before it. If you look at the unsuccessful jz path location right next to the successful one, and click its function sub_401349 you'll notice a couple of pushes of text “No luck” and a call to MessageBoxA. So we know the location where the jz comparison comes from and is doing some final test before popping up a success or failure message box.

You can color, group (by right clicking a node and selecting 'Group Nodes'), and rename these locations accordingly.

You can poke around the WndProc function, but there are just many numeric comparisons taking place, these numbers just represent event codes (say if you clicked the menu button, or the help button, etc.). We notice though the location where the jz stems from pushes a parameter named String to a function (sub_401365) and calls it, then pushes the same parameter to sub_4013B8 and calls it. Afterward the stack is cleaned up (add esp, 4) and cl is checked if it is 0, if it is the jz is true and we hit the success location.

Now the fun begins, we'll check out each of these functions in turn to see what they do, starting with sub_401365. I chose my local Win32 debugger in the IDA drop down menu, set a break point (F2) on the call to sub_401365, and clicked the green arrow to run the program. The program starts and a little crackme window shows up. Clicking Help will show you the Enter Password menu option.
As soon as you click OK, the break point on our first function is hit. Hovering over the String variable we can see that it contains a string 41h, 42h, 43h, 44h – this is our 'ABCD' in hex.

We now know both of these function take a single parameter of a character pointer. We can set the function prototype for both of these functions by clicking the function name and pressing 'y'. Change it from an int (IDA's best guess, which is technically correct since a pointer is an unsigned int) to a character pointer with a name.

This will not only display a comment in the caller location with the new variable name, but will also rename the variable throughout the target function who's prototype we changed.
In the first couple instructions we can see a pointer to user_input is put into esi, a single byte dereferenced into al, and then the byte gets checked to see if it is 0 (or null). The thick blue lines indicate that this is part of a loop and after a loop cycle it begins at this section again. This shows a pointer is getting walked, taking a byte at a time from esi and putting it into al, until the end (a null byte, or 0) is reached.

```
loc_401371:
00401371 mov al, [esi]
00401373 test al, al
00401375 jz short loc_401390
```

Assuming al is not zero, the first check determines if al is less than 0x41 ('A'), if so it jumps to loc_401385 where esi gets incremented and the loop starts over again. If the check is not true though, a second check takes place to see if al is less than 0x5a ('Z'), if al is less than 'Z' execution jumps to the same location where esi is incremented and the loop starts over again. This can be reduced to if the current byte in al is less than 'Z', continue with the next byte in the loop (since 'A' is less than 'Z').

However, with any ASCII character 'Z' or greater, like any lower case letters or some special symbols, we fall into loc_401388 which calls sub_4013B2.

```
sub_4013B2 proc near
sub al, 20h
mov [esi], al
retn
sub_4013B2 endp
```

Subtract 0x20 from the current character, al, and place it back into [esi] before returning and continuing the loop. So know we know:

- We loop through every character of user input
- If the current char is less than 'Z', continue the loop with the next char
- If the current char is equal to or greater than 'Z', replace it with itself minus 0x20

This continues until all characters are looped through, and finally the “test al, al” will be true (meaning the value of al is 0, or null). This drops the program flow into our last location of the function, loc_401390.
There is one last function call, sub_401399, before we return back out of the current function. In the above picture you'll also notice a couple of the locations are just single line statements. As mentioned earlier this is accomplished by selecting a location, right clicking, and grouping it (by itself or with other locations).

The first couple of xors set ebx and edi to 0. Again we see another loop that moves a single byte into cl.
and a single byte into bl, looping for the length of bl (until it tests to be 0). We can see ebx is being treated as a character pointer, having bl (the lower bits of ebx) pointing to a byte at a time from esi. EDI got zeroed out and is being treated as a counter, pointing to an offset of ds:byte_4021A3, incrementing before starting the loop over again. To figure out what these values are we can set a break point (F2) on this function or continue stepping into it from our debugging session a little earlier.

Double clicking ds:byte_4021A3 will show us the representation of it in the data section. It shows this value starting pointing to 'M', and as edi gets incremented it will point to the next byte in the message of “Messing_in_bytes”. Hovering over [esi] will show us:

That ESI is pointing our user input (this could also be seen by following esi in the previous function). The byte in bl and cl then get xored with each other, and the result is placed back into what esi points to. Then the function returns.

So now we know:

- We loop through every character of user input
  - If the current char is less than 'Z', continue the loop with the next char
  - If the current char is equal to or greater than 'Z', replace it with itself minus 0x20
- We again loop through every character of the (possibly) altered user input
  - Every byte of user input gets xored against its corresponding character of the string “Messing_in_bytes”

This last bullet should probably clue us in that our user input needs to be 16 bytes. Sub_401399 returns, sub_401365 returns, and now we're back inside of WndProc. A pointer to our altered user input is pushed on the stack, and now sub_4013B8 is called. This is the last function before the decision is made whether to display the success or failure message, and thankfully its a very short function.
Again, edi and ecx are cleared out and set to 0. A quick google search shows that rpe cmpsb compares 2 strings, esi and edi, for a count of ecx. ECX has 16 in it again, confirming our guess earlier that our input should be 16 bytes long. We see esi has a pointer to our user input, but what about edi? Double clicking offset unk_402150 again takes us to the data section where it shows all 16 bytes of unk_402150.

So after all the previous functions are called and the user input has been altered, its value must equal this 'arbitrary' value of 16 bytes.

So now we know:

- We loop through every character of user input
  - If the current char is less than 'Z', continue the loop with the next char
  - If the current char is equal to or greater than 'Z', replace it with itself minus 0x20
- We again loop through every character of the (possibly) altered user input
  - Every byte of user input gets xored against its corresponding character of the string
“Messing in bytes”

Finally, our altered user input must equal an arbitrary value of 0x1F, 0x2C, … 0x3E

Now xor works both ways, and we know our altered input gets xored against “Messing in bytes” and must equal that specific sequence of bytes above. So we will work backwards and xor “Messing in bytes” against the target sequence of bytes to see what our altered input must equal. Python, as usual, gets the job done:

```
>>> target = [0x1f, 0x2c, 0x37, 0x36, 0x3b, 0x3d, 0x28, 0x19, 0x3d, 0x26, ... 0x1a, 0x31, 0x2d, 0x3b, 0x37, 0x3e]
>>> xor_msg = "Messing in bytes"
>>> xor_msg_ints = map(ord, xor_msg)
>>> xor_msg_ints
[77, 101, 115, 115, 105, 110, 103, 95, 105, 95, 98, 121, 116, 101, 115]
```

We setup our target value as a list, and our message to be xored against. The xor_msg_ints represents the ordinal ASCII values as a list of the same message, because we can't xor an int value from our target against a string value. Now we xor these two:

```
>>> results = []
>>> for i in range(16):
...    results.append(target[i] ^ xor_msg_ints[i])
...
>>> results
[82, 73, 68, 69, 82, 83, 79, 70, 84, 72, 69, 83, 84, 79, 82, 77]
```

These values are a little suspect, they're all in the same range, and as anyone who stares at hex values a lot will know these are the base 10 values of ASCII printable characters.

```
>>> map(chr, results)
```

```
>>> ''.join(map(chr, results))
'RIDERSOFTHESTORM'
```

Well, now we're in business. With this we can update what we know and what we have to do:

- We loop through every character of user input
  - If the current char is less than 'Z', continue the loop with the next char
  - If the current char is equal to or greater than 'Z', replace it with itself minus 0x20
- Now our altered user input must equal “RIDERSOFTHESTORM”.
- We again loop through every character of the (possibly) altered user input
  - Every byte of user input gets xored against its corresponding character of the string “Messing in bytes”
- Finally, our altered user input must equal an arbitrary value of 0x1F, 0x2C, … 0x3E

If our altered user input after the first main bullet equals “ RIDERSOFTHESTORM”, we know the rest will be satisfied because “ RIDERSOFTHESTORM” xored against “Messing in bytes” will equal our target sequence of bytes. Since all the characters from “ RIDERSOFTHESTORM” are less
than 'Z', we don't have to worry about the values being subtracted by 0x20. So, unfortunately this is the end of the story. Entering “RIDERSOFTHESTORM” as the password is successful. It would have been a tad cooler if some of the characters were above 'Z' and made you pick something 0x20 higher than them from the ASCII chart so they would then be subtracted by 0x20 to become the target value.

But regardless, that's it! We picked our target point and worked backwards to see how to hit that piece of code. Then we figured out our target value, figured out the transformations that happen to the password we enter, then worked backwards to determine what password was correct.

Hopefully this provided a helpful introduction to IDA and some processes to think about when doing a crackme or reversing in general.