CrackMe #1 Write-Up:

Defeating a Simple Serial Check

Level: Beginner **Tools:** IDA Pro Freeware **Concepts:** Static Analysis, Windows API, Keygen Development

Executive Summary

This crackme presents a classic serial number validation scheme commonly found in beginner-level reverse engineering challenges. The goal is to analyze the binary, understand the algorithm that generates a valid serial from a given name, and then craft a keygen. The protection mechanism is straightforward, making it a perfect exercise for learning fundamental static analysis techniques with IDA Pro.

Static Analysis with IDA Pro

Initial Loadup and Navigation

1. Load the `CrackMe1.exe` into IDA Pro and let it perform its initial auto-analysis. This process generates the disassembly listing and builds the function call graph, which is crucial for understanding the program's flow.
2. The entry point of a Windows application is often boilerplate code that sets up the main window. Our primary interest lies in the **Window Procedure** (`WndProc` function) that handles messages for the main window, including the click event from the "Register" menu item.
3. **Pro Tip:** Press `Shift+F12` to open the **Strings window** in IDA. This often provides quick clues. In this case, you'd immediately find the failure string "No luck there, mate!" and the success message. Double-clicking the failure string would lead you to the critical validation function, bypassing the need to start from the `WinMain`.

The Main Event: WndProc and The Validation Routine

As you correctly identified, the heart of the crackme is the `WndProc` function. Let's break down the key instructions you found:

```
.text:00401228 call    f_num_from_username ; EAX = magic number from username
.text:0040122D mov     [ebp+var_10], eax   ; Store the result
...
.text:0040123B call    f_num_from_serial   ; EAX = number from serial input
.text:00401240 mov     [ebp+var_14], eax   ; Store the result
.text:00401243 mov     ecx, [ebp+var_14]
```

```
.text:00401246 cmp     [ebp+var_10], ecx    ; Compare the two numbers
.text:00401249 jnz     short fail           ; Jump if not equal (FAIL)
.text:0040124B ...                         ; Success code here
```

Translation: The program calculates a magic number based on the `username`. It then takes the entered `serial`, converts it from a string to an integer. The final check is a simple comparison of these two integers. If they match, you win.

Reverse Engineering the Algorithm

f_num_from_username (sub_40137E)

This function is the core of the keygen algorithm. Its logic is as follows:

1. **Input Validation:** It checks that every character in the provided username is an alphabetic character (`[a-zA-Z]`). If it finds a non-letter character, it immediately fails with the iconic message, "No luck there, mate!".
2. **Case Conversion:** It converts the entire username to **uppercase**. This is important! The serial generated for "Alice" will be the same as for "ALICE".
3. **Checksum Calculation:** It iterates through each character of the uppercased name, summing their ASCII values into a register.
4. **XOR Obfuscation:** This sum is then XORed with the hex value `0x5678`. The result of this operation is the "magic number" that a valid serial must match.

f_num_from_serial (sub_4013E2)

This function is simpler. Its job is to take the string from the "Serial" input field and convert it into an integer (e.g., it converts the string "12345" into the integer `12345`). This integer is then passed to the comparison function.

Cracking the Scheme & Keygen Development

The vulnerability is now clear: the serial is just a numeric representation of `(sum(ASCII_values_of_uppercased_username) XOR 0x5678)`. There is no cryptographic strength here; it's a simple checksum.

Armed with this algorithm, writing a keygen is trivial. Here is the Python code to generate a valid serial for any given name:

```
#!/usr/bin/env python3
# Keygen for CrackMe1

def keygen(username):
    """
    Generates a valid serial for the CrackMe1.
    """
    # Convert to uppercase and calculate the sum of ASCII values
    upper_name = username.upper()
    ascii_sum = sum(ord(c) for c in upper_name)

    # Perform the XOR operation to get the valid serial number
    serial = ascii_sum ^ 0x5678
    return serial

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <Username>")
        sys.exit(1)

    username = sys.argv[1]
    serial_num = keygen(username)
    print(f"[+] For username: '{username}'")
    print(f"[+] Generated Serial: {serial_num}")

    # Bonus: Test the algorithm
    upper_name = username.upper()
    print(f"[i] Used uppercase name: '{upper_name}'")
    print(f"[i] Sum of ASCII values: {sum(ord(c) for c in upper_name)}")
    print(f"[i] After XOR with 0x5678: {serial_num} (0x{serial_num:X})")
```

Conclusion

This crackme was a great introduction to the basics of static analysis. The process involved:

- Locating the main validation function by following cross-references from error strings.
- Analyzing the disassembly of two key functions to understand their purpose (`f_num_from_username` and `f_num_from_serial`).
- Identifying a weak checksum algorithm used for "protection."
- Writing a proof-of-concept keygen script to defeat the protection scheme automatically.

The key takeaway is that simply hiding a validation routine in code is insufficient. Without strong cryptography and proper anti-reversing measures, such checks can be reversed and exploited with minimal effort. For anyone starting their reverse engineering journey, mastering this type of crackme is the essential first step towards tackling more complex targets.

Flag: The "flag" is simply the correct serial number generated by the algorithm above for your chosen name. For example, try the name "REVERSE" and see what you get!