Flare-On 11 Challenge 2: checksum

By Chuong Dong (@cPeterr)

Overview

The file **`checksum.exe`** is a 64-bit Windows executable. When executed, the sample prompts the user for an answer to a random number of math problems.

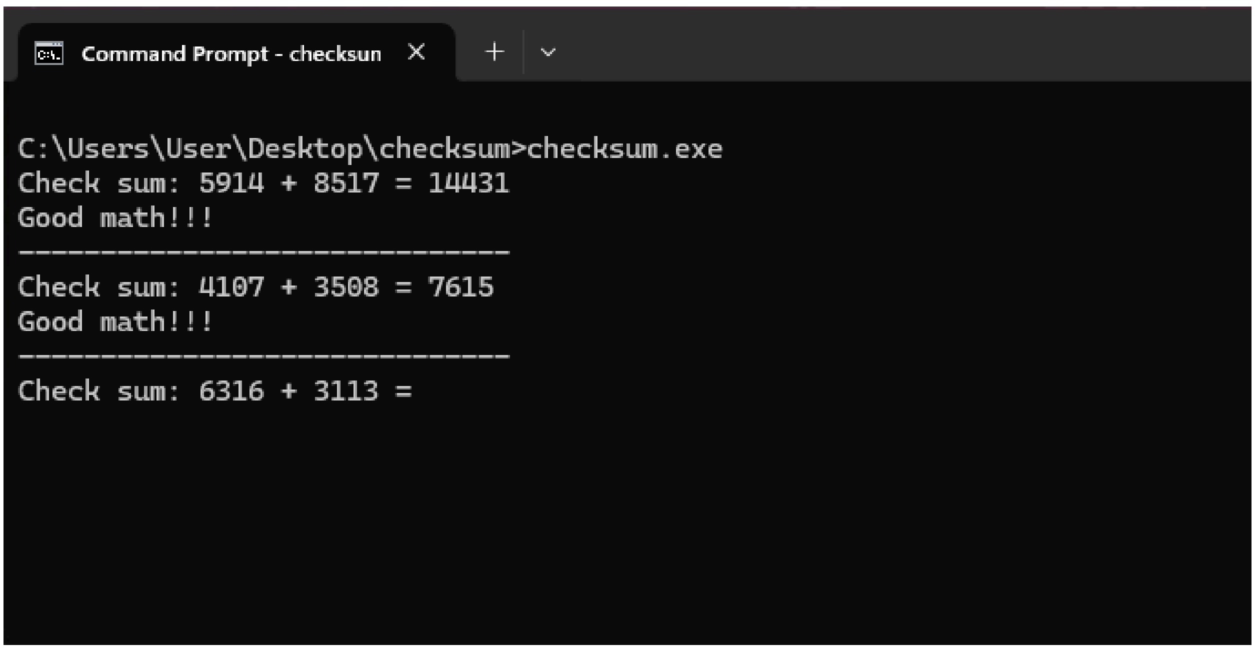

Figure 1: *Program initial execution*

If the user answers the question correctly, the program moves on to the next. If the answer is incorrect, the program simply terminates.

```
C:\Users\User\Desktop\checksum>checksum.exe
Check sum: 5914 + 8517 = 14431
Good math!!!
------------------------------
Check sum: 4107 + 3508 = 7615
Good math!!!
------------------------------
Check sum: 6316 + 3113 = -1
Try again! ;)

C:\Users\User\Desktop\checksum>
```

Figure 2: *Program terminating when incorrect input is provided*

However, if the user gets through enough questions, the program will display a new prompt asking for a checksum input.

```
C:\Users\User\Desktop\checksum>checksum.exe
Check sum: 9750 + 2773 = 12523
Good math!!!
------------------------------
Check sum: 3221 + 3457 = 6678
Good math!!!
------------------------------
Check sum: 6529 + 5310 = 11839
Good math!!!
------------------------------
Check sum: 37 + 9070 = 9107
Good math!!!
------------------------------
Check sum: 8523 + 6443 = 14966
Good math!!!
------------------------------
Check sum: 5774 + 7982 = 13756
Good math!!!
------------------------------
Check sum: 69 + 9724 = 9793
Good math!!!
------------------------------
Checksum: aabbccdd
Maybe it's time to analyze the binary! ;)

C:\Users\User\Desktop\checksum>
```

Figure 3: Program prompting for a "checksum" input

Nothing is provided to the user at this stage beside the checksum prompt, which might indicate that the program is expecting a specific hash/checksum. With this, we can safely assume that this is a crackme challenge that requires the user to reverse engineer the binary to find the correct answer to retrieve the flag.

Challenge Static Analysis

Upon opening the program in a binary analysis tool such as IDA Pro, we can quickly tell that the program is written in Golang and compiled with full debug symbols. We can also see that the program only contains 3 non-library functions: **`main`**, **`a`**, and **`b`**. This tells us that the main functionality of the program will be inside of these three functions, and that the program itself might not be too large or complicated.

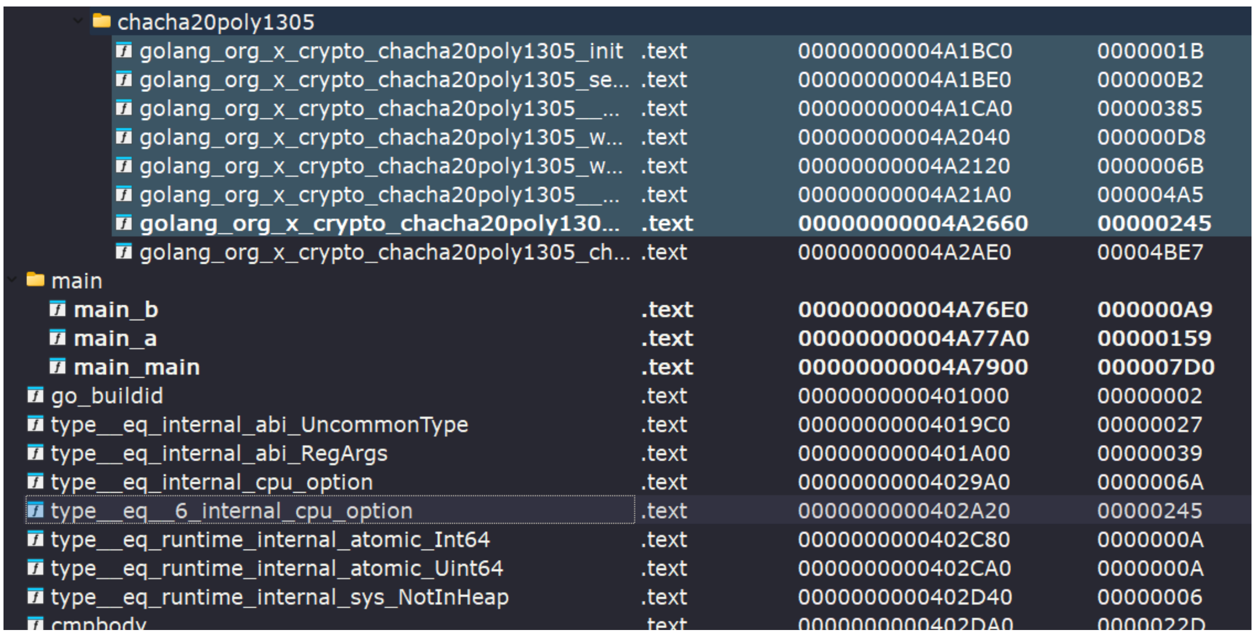

Figure 4: IDA Functions subview showing the program's Go symbols

Examining the decompiled code of the program's entrypoint **`main`** function, we can see the functionality to prompt the user to answer a series of math questions.

```
random_number_of_round = math_rand_v2__ptr_Rand_uint64n(math_rand_v2_globalRand, 5uLL);
check_sum_input_1 = runtime_newobject(&RTYPE_int);
for ( i = 0LL; ; i = v52 + 1 )
{
  v4 = random_number_of_round + 3;
  if ( i ≥ (random_number_of_round + 3) )     // between 3 and 8 rounds
    break;
  v52 = i;
  first_number_1 = math_rand_v2__ptr_Rand_uint64n(math_rand_v2_globalRand, 10000uLL);// random number ≥ 0 & < 10000
  second_number_1 = math_rand_v2__ptr_Rand_uint64n(math_rand_v2_globalRand, 10000uLL);// random number ≥ 0 & < 10000
  first_number = v1;
  second_number = v1;
  v6 = runtime_convT64(first_number_1, 10000LL, v5, v0, v4);
  *&first_number = &RTYPE_int;
  *(&first_number + 1) = v6;
  v7 = runtime_convT64(second_number_1, 10000LL, &RTYPE_int, v0, v4);
  *&second_number = &RTYPE_int;
  *(&second_number + 1) = v7;
  v57 = first_number_1 + second_number_1;
  fmt_Fprintf(
    &go_itab__os_File_io_Writer,
    *&os_Stdout,
    "Check sum: %d + %d = ",
    21,
    &first_number,
    2,
    2);
  *check_sum_input = &RTYPE__ptr_int;
  *&check_sum_input[2] = check_sum_input_1;
  v8 = os_Stdin;                              // prompt user for a number input
  fmt_Fscanf(&go_itab__os_File_io_Reader, os_Stdin, "%d\n", 3, check_sum_input);
  v0 = 21;
  v10 = main_b(v8, v9, "Not a valid answer...", 21);
  if ( *check_sum_input_1 ≠ first_number_1 + second_number_1 )// math check
  {
    runtime_printlock(v10);
    v11 = runtime_printstring("Try again! ;)\n", 14);
    runtime_printunlock(v11);
```

Figure 5: *Generating random summation questions*

Here, a random number between 0 and 5 is generated, and the number of questions is derived by adding 3 to that. As a result, there will be randomly 3 to 8 math questions every time the program is run.

The program also generates 2 random numbers between 0 and 10000, prompts the user for an integer input, and compares the input with the sum of the two generated numbers. This also confirms what we have seen when running the program in the earlier stage.

After all the math questions are answered correctly, the program moves to prompting the user for a "checksum" string. It uses the Golang API hex.Decode to decode the input string, which tells us that the program only accepts a valid hex string for this prompt.

```
checksum_string = runtime_newobject(&RTYPE_string);
checksum_string->ptr = 0LL;
*v71 = &RTYPE_string;
*&v71[2] = &off_4EDAD0;                          // "Checksum: "
fmt_Fprint(&go_itab__os_File_io_Writer, os_Stdout, v71, 1, 1);
*checksum_input = &RTYPE__ptr_string;
*&checksum_input[2] = checksum_string;
v12 = os_Stdin;
fmt_Fscanf(&go_itab__os_File_io_Reader, os_Stdin, "%s\n", 3, checksum_input);
main_b(v12, v13, "Fail to read checksum input ...", 30);// printing error
checksum_string_ptr = checksum_string->ptr;
v63 = runtime_stringtoslicebyte(
        &v48,
        checksum_string->ptr,
        checksum_string->len,
        30,
        checksum_input);
v50 = v15;                                       // hex decode checksum input
decoded_checksum_input_len = encoding_hex_Decode(v63, checksum_string_ptr);
if ( decoded_checksum_input_len > v50 )
  runtime_panicSliceAcap(
    decoded_checksum_input_len,
    checksum_string_ptr,
    decoded_checksum_input_len);
decoded_checksum_input_len_1 = decoded_checksum_input_len;
main_b(checksum_string_ptr, v17, "Not a valid checksum ...", 23);// fail if hex decode fails
nonce = runtime_makeslice(&RTYPE_uint8, 24, 24, 23, checksum_string_ptr);
```

Figure 6: *Prompting for a "Checksum" input hex string*

We also see the program's **`b`** function is called with what appears to be a debug string as a parameter. Upon examining the subroutine closer in IDA, it can be confirmed that this **`b`** function simply prints the debug string before exiting with the status code **`0xDEADBEEF`**.

```
// main.b
__int64 __golang main_b(__int64 result, int a2, int debug_string, int a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  if ( result )
  {
    debug_string_1[0] = &RTYPE_string;
    debug_string_1[1] = runtime_convTstring(debug_string, a4, debug_string, a4);
    fmt_Fprintln(&go_itab__os_File_io_Writer, *&os_Stdout, debug_string_1, 1, 1);
    return os_Exit(0xDEADBEEF);
  }
  return result;
}
```

Figure 7: *Printing error message*

In the next part of the code in **`main`**, we can see that the program allocates a slice buffer of 24 bytes in memory and copies the first 24 bytes of the hex-decoded input into it.

```
data_buffer = runtime_makeslice(&RTYPE_uint8, 24);// alloc 24 bytes
decoded_checksum_input_len_2 = decoded_checksum_input_len_1;
decoded_checksum_input = v63;
for ( j = 0LL; decoded_checksum_input_len_2 > j && j ≠ 24; ++j )
{
  if ( j ≥ 0x18 )
    runtime_panicIndex(j, 24LL, 24LL, 23LL);
  data_buffer[j] = decoded_checksum_input[j]; // data_buffer = first 24 bytes in decoded input
}
```

Figure 8: *Populating a 24-byte buffer with the hex-decoded user input*

Next, the program checks if the length of the decoded input is 32 bytes and throws the error **"`chacha20poly1305: bad key length`"** if the check fails.

This part of code appears to be from the Golang chacha20poly1305 library. It indicates that the "checksum" input is a 32-byte hex string that will be used as a ChaCha20-Poly1305 key.

```
decrypted_flag_data.cap = data_buffer;
if ( decoded_checksum_input_len_2 == 32 )     // decoded checksum input length must be 32 bytes
{
  p_chacha20poly1305_xchacha20poly1305 = runtime_newobject(&RTYPE_chacha20poly1305_xchacha20poly1305);
  if ( p_chacha20poly1305_xchacha20poly1305 ≠ v63 )
  {
    v64 = p_chacha20poly1305_xchacha20poly1305;// chacha20poly1305.NewX(key)
    runtime_memmove(p_chacha20poly1305_xchacha20poly1305, v63, 32LL, 23LL, decoded_checksum_input);
    p_chacha20poly1305_xchacha20poly1305 = v64;
  }
  v22 = go_itab__golang_org_x_crypto_chacha20poly1305_xchacha20poly1305_crypto_cipher_AEAD;
  v23 = 0LL;
  chacha20poly1305_cipher = p_chacha20poly1305_xchacha20poly1305;
  LODWORD(v25) = 0;
}
else
{
  *&v69[2] = 32LL;
  *v69 = "chacha20poly1305: bad key length";
  v22 = 0LL;
  v23 = &go_itab__errors_errorString_error;
  chacha20poly1305_cipher = 0LL;
  v25 = v69;
}
```

Figure 9: *Decompiled code setting up ChaCha20-Poly1305 context*

Looking a bit deeper into the library's documentation, we see that the crypto algorithm's key size must be 32-bytes with 2 different nonce sizes. The 12-byte nonce is used for the standard ChaCha20-Poly1305 variant, while the 24-byte nonce is used for the XChaCha20-Poly1305 algorithm.

From this, it is a valid assumption that the 24-byte slice buffer allocated in **Figure 8** will be used as the nonce for the program to encrypt/decrypt with XChaCha20-Poly1305.

```
https://pkg.go.dev/golang.org/x/crypto/chacha20poly1305

chacha20poly1305 package

Constants

const (
    // KeySize is the size of the key used by this AEAD, in bytes.
    KeySize = 32

    // NonceSize is the size of the nonce used with the standard variant of this
    // AEAD, in bytes.
    //
    // Note that this is too short to be safely generated at random if the same
    // key is reused more than 2³² times.
    NonceSize = 12

    // NonceSizeX is the size of the nonce used with the XChaCha20-Poly1305
    // variant of this AEAD, in bytes.
    NonceSizeX = 24

    // Overhead is the size of the Poly1305 authentication tag, and the
    // difference between a ciphertext length and its plaintext.
    Overhead = 16
)
```

Figure 10: *Nonce sizes in ChaCha20-Poly1305 library documentation*

In the next part, we see that the program is calling a function in the **`chacha20poly1305`** library, passing in a data buffer with the name **`main_encryptedFlagData`** along with its size of 181548 bytes. This size is stored in memory at address **`0x59A4F8`**. The 24-byte nonce buffer is also passed as a parameter.

Without doing more digging, we can make another educated guess that the program is calling a function to decrypt the encrypted flag data using XChaCha20-Poly1305 with the key and nonce from the checksum input.

```
00000000004A7D6B mov     rdx, cs:main_encryptedFlagData
00000000004A7D72 mov     rsi, cs:qword_59A4F8            ; encrypted flag data length = 0x2C52C bytes
00000000004A7D79 mov     r8, cs:qword_59A500
00000000004A7D80 mov     r9, [rsp+248h+go_itab__golang_org_x_crypto_chacha20poly1305_xchacha20poly1305_crypto_cipher_AEAD]
00000000004A7D88 mov     r9, [r9+20h]                    ; r9 = function to decrypt ChaCha20-Poly1305
00000000004A7D8C mov     [rsp+248h+var_248], rdx         ; encrypted flag data
00000000004A7D90 mov     [rsp+248h+var_240], rsi         ; encrypted flag data length
00000000004A7D95 mov     [rsp+248h+var_238], r8          ; encrypted flag data cap
00000000004A7D9A movups  [rsp+248h+var_230], xmm15       ; nonce
00000000004A7DA0 mov     [rsp+248h+var_220], 0
00000000004A7DA9 mov     rax, [rsp+248h+chacha20poly1305_cipher_1]
00000000004A7DB1 xor     ebx, ebx
00000000004A7DB3 xor     ecx, ecx
00000000004A7DB5 mov     rdi, rcx
00000000004A7DB8 mov     rsi, qword ptr [rsp+248h+decrypted_flag_data+10h] ; _DWORD
00000000004A7DC0 mov     r8d, 18h
00000000004A7DC6 mov     rdx, r9
00000000004A7DC9 mov     r9, r8
00000000004A7DCC call    rdx
```

__Figure 11:__ Decrypting flag data with XChaCha20-Poly1305

After the decryption finishes successfully, the program calculates the SHA256 checksum of the decrypted data.

```
decrypted_flag_data_1.ptr = *&decrypted_flag_data;
decrypted_flag_data_1.len = *decrypted_flag_data_len;
decrypted_flag_data_1.cap = *decrypted_flag_data_cap;
crypto_sha256__ptr_digest_Write(&v61, decrypted_flag_data_1);
decrypted_flag_data_1.ptr = 0LL;
decrypted_flag_data_1.len = 0LL;
v29 = 0LL;
decrypted_flag_data_1 = crypto_sha256__ptr_digest_Sum(&v61, *(&v29 - 2));// SHA256 hashing the decrypted flag data
ptr = decrypted_flag_data_1.ptr;
v65 = v30;
v57 = 2 * decrypted_flag_data_1.ptr;
decrypted_flag_SHA256_hash_len = runtime_makeslice(
                                   &RTYPE_uint8,
                                   2 * LODWORD(decrypted_flag_data_1.ptr));
```

__Figure 12:__ Generating the SHA256 hash of the decrypted data

Finally, the program compares the user's input for the "checksum" prompt with the SHA256 checksum of the decrypted flag data.

If they do not match, the program prints "**`Maybe it's time to analyze the binary! ;)`**" and terminates.

```
if ( decrypted_flag_SHA256_hash_len_1 == checksum_string->len )
{
  decrypted_flag_SHA256_hash_len_1 = checksum_string->ptr;
  if ( runtime_memequal(decrypted_flag_SHA256_hash_string, checksum_string->ptr) )// check if the input == SHA256(decrypted_flag)
  {
    decrypted_flag_SHA256_hash_len_1 = checksum_string->len;
    v42 = main_a(checksum_string->ptr, decrypted_flag_SHA256_hash_len_1);
  }
  else
  {
    v42 = 0;
  }
}
else
{
  v42 = 0;
}
if ( !v42 )
{
  v70[0] = &RTYPE_string;
  v70[1] = &off_4EDAE0;                        // Maybe it's time to analyze the binary! ;)
  decrypted_flag_SHA256_hash_len_1 = *&os_Stdout;
  LODWORD(v29) = 1;
  fmt_Fprintln(&go_itab__os_File_io_Writer, *&os_Stdout, v70, 1, 1);
}
```

__Figure 13:__ Checking if the user's input is the SHA256 hash of the decrypted flag

This tells us that the user's input must be the SHA256 hash of the decrypted flag data, and that the SHA256 checksum is used as the 32-byte XChaCha20-Poly1305 key to decrypt the flag. We should also note that the first 24 bytes of the SHA256 checksum is also used as the nonce in the algorithm.

However, knowing this is not enough to recover and decrypt the flag. We would need the decrypted data to get its SHA256 hash, but we can not get the decrypted data without having the checksum to use as the key.

Fortunately, there is also a second condition where the program is calling the function **a** and checking for the return value. When calling the function below, the program passes in the decrypted data's SHA256 checksum with its length as the parameters.

```
// main.a
__int64 __golang main_a(_BYTE *checksum_string, __int64 checksum_string_len)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  if ( !checksum_string )
    checksum_string = &runtime_noptrbss;
  checksum_string_1 = checksum_string;
  XOR_encoded_checksum_string_1 = runtime_makeslice(&RTYPE_uint8, checksum_string_len, checksum_string_len);
  checksum_string_2 = checksum_string_1;
  for ( i = 0LL; checksum_string_len > i; ++i )
  {
    XOR_encoded_checksum_string = XOR_encoded_checksum_string_1;
    v7 = checksum_string_2;
    v8 = i - 11 * (((i * 0x5D1745D1745D1746LL) >> 64) >> 2);
    v9 = checksum_string_2[i];                  // weird math for doing modulus
    if ( v8 >= 0xB )
      runtime_panicIndex(v8, i, 11LL, v7, XOR_encoded_checksum_string);
    XOR_encoded_checksum_string[i] = aFlareon2024[v8] ^ v9;// FlareOn2024
    XOR_encoded_checksum_string_1 = XOR_encoded_checksum_string;
    checksum_string_2 = v7;
  }
```

Figure 14: *XOR-ing the SHA256 hash with* ***"FlareOn2024"***

In this subroutine **a**, the program compares the SHA256 checksum to ensure that the data is decrypted properly. First, it converts the SHA256 checksum string into a slice of bytes and XOR-encodes that with the XOR key "**FlareOn2024**".

Then, the XOR-encoded result is Base64-encoded and compared to the string "**cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNAxBSe15QCVRVJ1pQEwd/WFBUAlElCFBFUnlaB1ULByRdBEFdfVtWVA==**".

```
v17 = XOR_encoded_checksum_string;
v10 = XOR_encoded_checksum_string;
v11 = checksum_string_len_1;
v12 = encoding_base64__ptr_Encoding_EncodeToString(
        runtime_bss,
        XOR_encoded_checksum_string,
        checksum_string_len_1);
v18[0] = &RTYPE_string;
v18[1] = runtime_convTstring(v12, v10, v13, v11);
fmt_Fprintln(&go_itab__os_File_io_Writer, *&os_Stdout, v18, 1, 1);
Base64_XOR_encoded_checksum_string = encoding_base64__ptr_Encoding_EncodeToString(
                                       runtime_bss,
                                       v17,
                                       checksum_string_len);// Base64-encode
if ( v17 == 88 )
  return runtime_memequal(
           Base64_XOR_encoded_checksum_string,
           "cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNAxBSe15QCVRVJ1pQEwd/WFBUAlElCFBFUnlaB1ULByRdBEFdfVtWVA==");
else
  return 0LL;
```

Figure 15: *Checking the SHA256 hash to make sure the flag is decrypted properly*

Since both Base64 and XOR encodings are reversible, the correct SHA256 hash can be derived by Base64-decoding the string **`“cQoFRQErX1YAVw1zVQdFUSxfAQNRBXUNAxBSe15QCVRVJ1pQEwd/WFBUAlElCFBFUnlaB1ULByRdBEFdfVtWVA==”`** and XOR-decoding the result with the key **`“FlareOn2024”`**.

This can quickly be done in CyberChef, which results in the original SHA256 hash **`“7fd7dd1d0e959f74c133c13abb740b9faa61ab06bd0ecd177645e93b1e3825dd”`**.

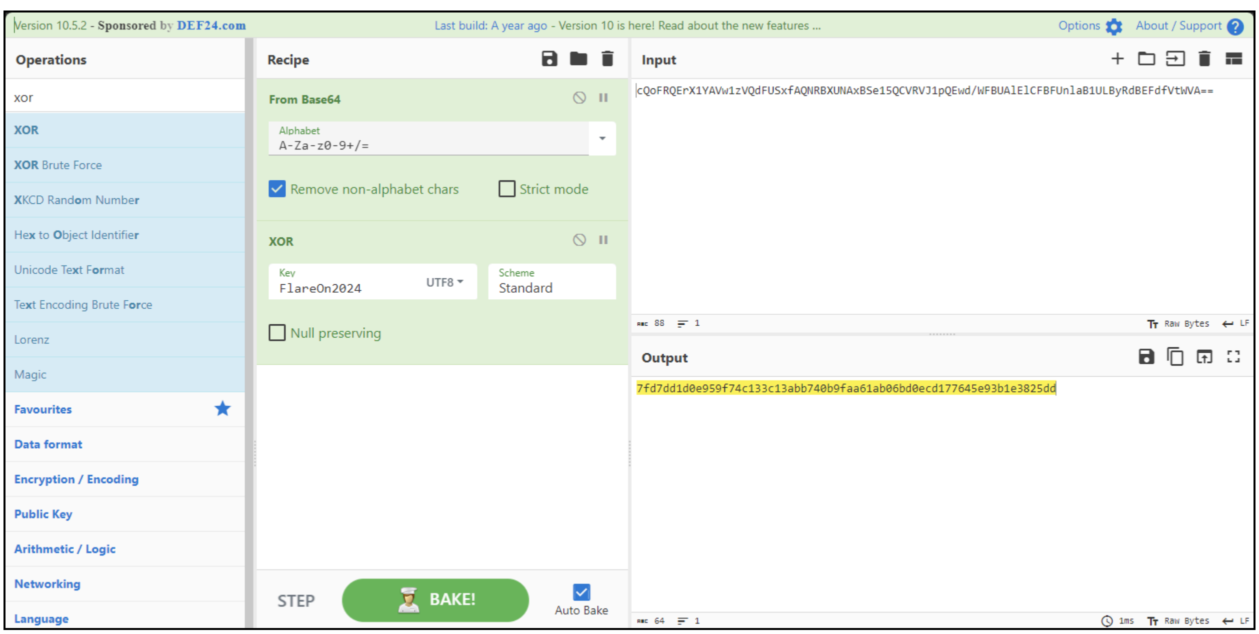

Figure 16: *Retrieving the correct SHA256 hash using CyberChef*

Retrieving the Flag

As the correct checksum has been recovered through static analysis in our binary analysis tool, we can provide the correct hash when prompted.

```
PS C:\Users\User\Desktop\checksum> .\checksum.exe
Check sum: 7125 + 5382 = 12507
Good math!!!
------------------------------
Check sum: 8055 + 823 = 8878
Good math!!!
------------------------------
Check sum: 3193 + 2077 = 5270
Good math!!!
------------------------------
Check sum: 6622 + 9090 = 15712
Good math!!!
------------------------------
Checksum: 7fd7dd1d0e959f74c133c13abb740b9faa61ab06bd0ecd177645e93b1e3825dd
Noice!!
PS C:\Users\User\Desktop\checksum>
```

Figure 17: *Providing the correct checksum input to the program*

When the correct input is provided, nothing much happens. The program simply prints the message **“Noice!!”** before terminating. With this, additional analysis must be performed to find the challenge flag.

In the final part of the program's **main** function, the malware calls the API os.UserCacheDir to retrieve the default root directory that stores user-specific cached data. On Windows, this path is the **%LocalAppData%** path. The program then proceeds to write the decrypted data content to the file **REAL_FLAREON_FLAG.JPG** in the **%LocalAppData%** folder.

```
APPDATA_PATH_1 = os_UserCacheDir();
*v56 = v2;
main_b(v43, v29, "Fail to get path ... ", 19);
APPDATA_PATH = APPDATA_PATH_1;
v45 = runtime_concatstring2(0, APPDATA_PATH_1, v56[0], "\\REAL_FLAREON_FLAG.JPG", 22);
v46 = os_WriteFile(
        v45,
        APPDATA_PATH,
        decrypted_flag_data,
        decrypted_flag_data_len[0],
        decrypted_flag_data_cap[0],
        420);
main_b(v46, APPDATA_PATH, "Fail to write file ... ", 21);
v69[0] = &RTYPE_string;
v69[1] = &off_4EDAF0;
fmt_Fprintln(&go_itab__os_File_io_Writer, *&os_Stdout, v69, 1, 1);
```

Figure 18: *Writing the flag file to disk*

Beside reading the decompiled Golang code , the flag's location can also be retrieved by setting up ProcMon and monitoring the program's file operations.

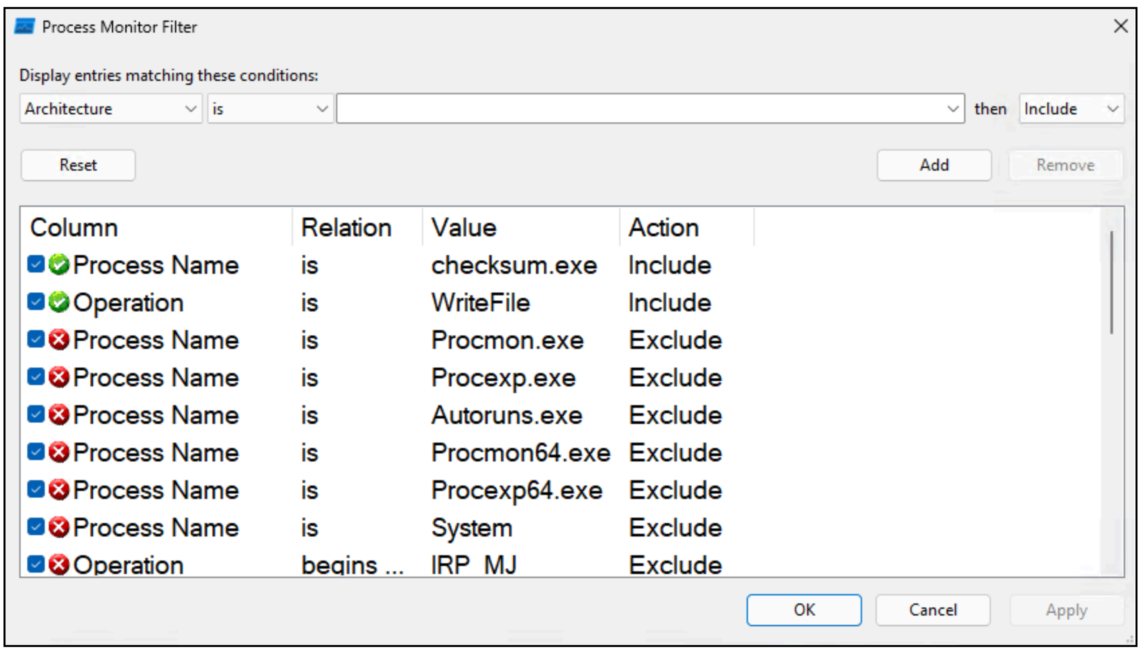

Figure 19: *ProcMon filter to find the flag's destination*

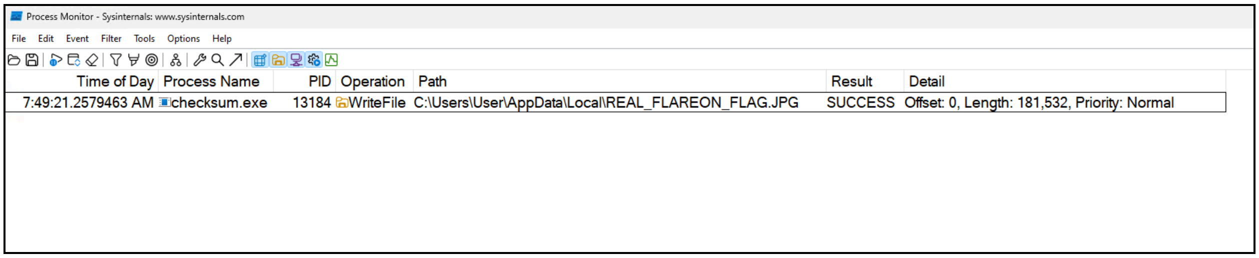

Figure 20: *ProcMon view showing where the flag is written*

Below is the image **REAL_FLAREON_FLAG.JPG** in the **%LocalAppData%** directory, which gives us the flag of **Th3_M4tH_Do_b3_mAth1ng@flare-on.com**.

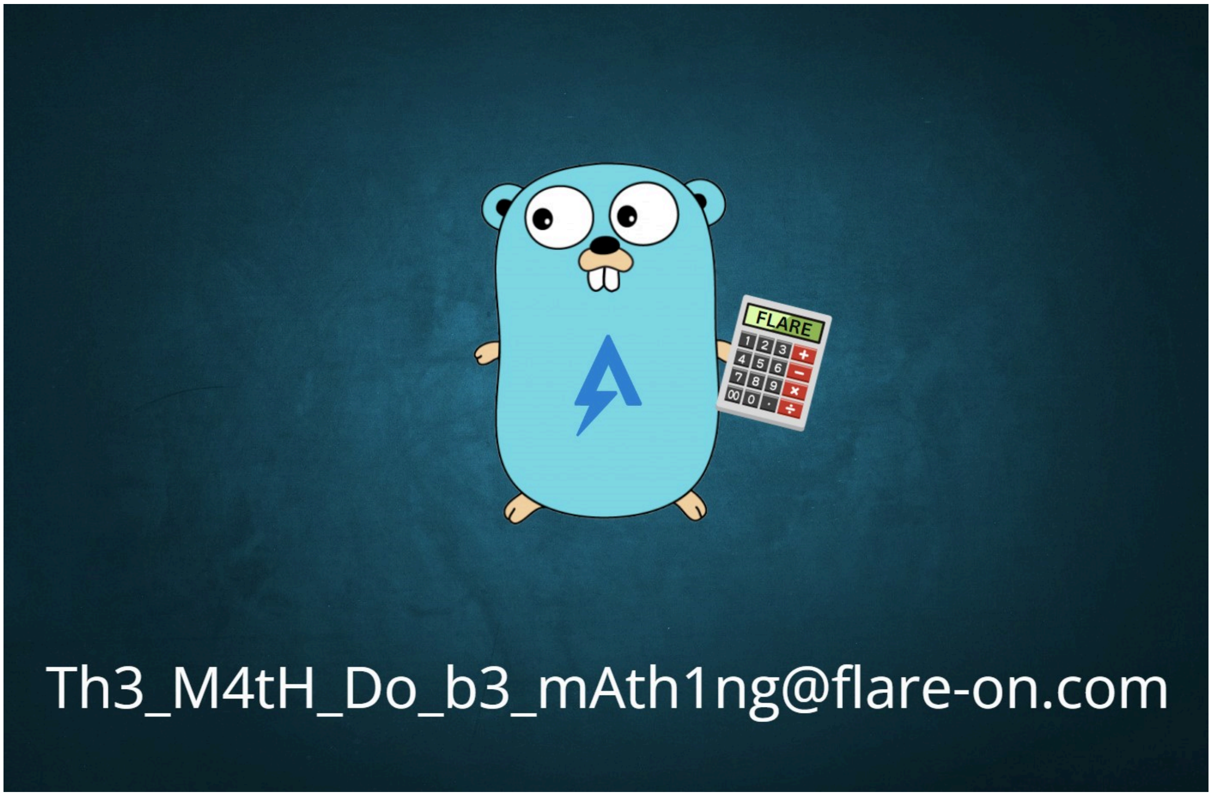

Figure 21: *The challenge's flag*

Final Flag

```
Th3_M4tH_Do_b3_mAth1ng@flare-on.com
```