Reverse Engineering Exercise Writeup

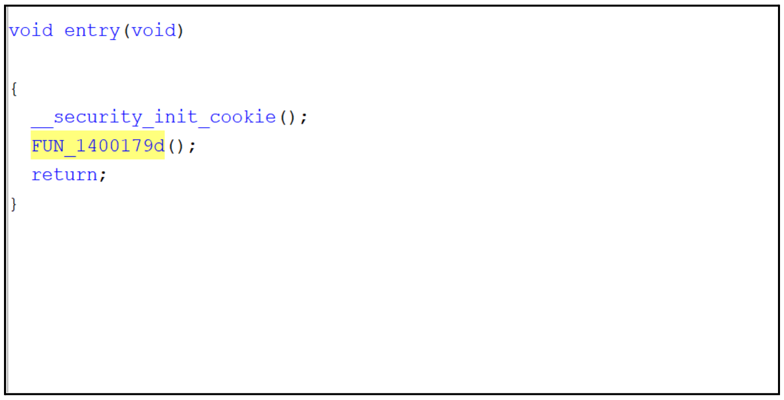
Basic_Password_challenge

Sample: Basic_password_challenge.exe

Link: https://crackmes.one/crackme/65bfedcaeef082e477ff66aa

Ghidra Static Analysis

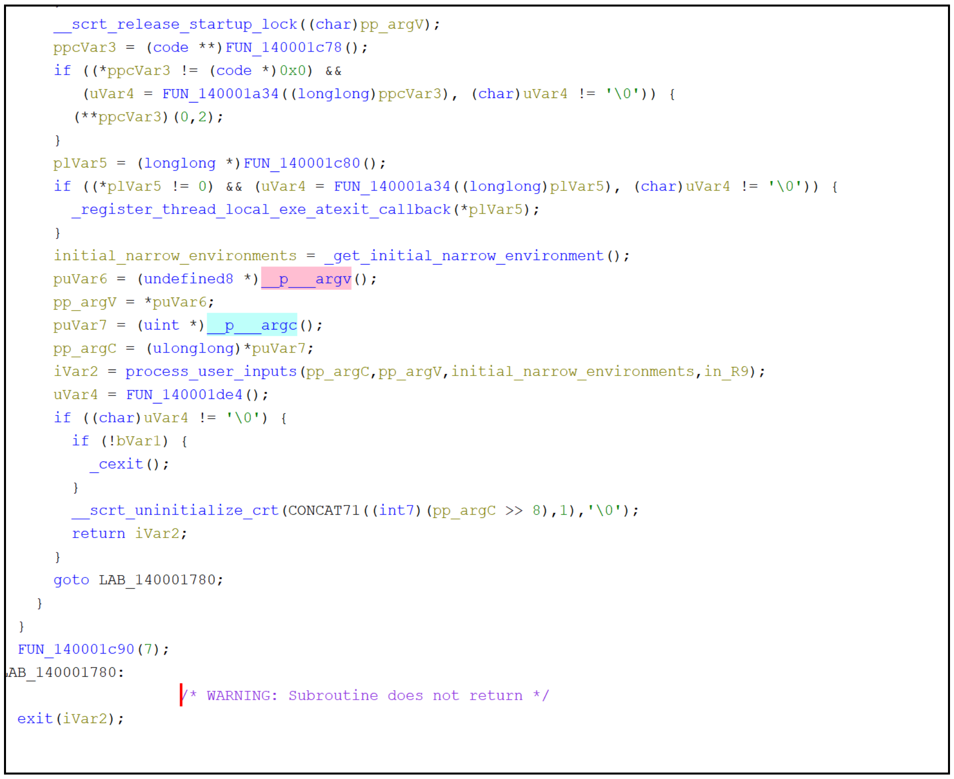
We have a **Windows Executable** which calls to a single function from entry.



```
void entry(void)

{
  __security_init_cookie();
  FUN_1400179d();
  return;
}
```

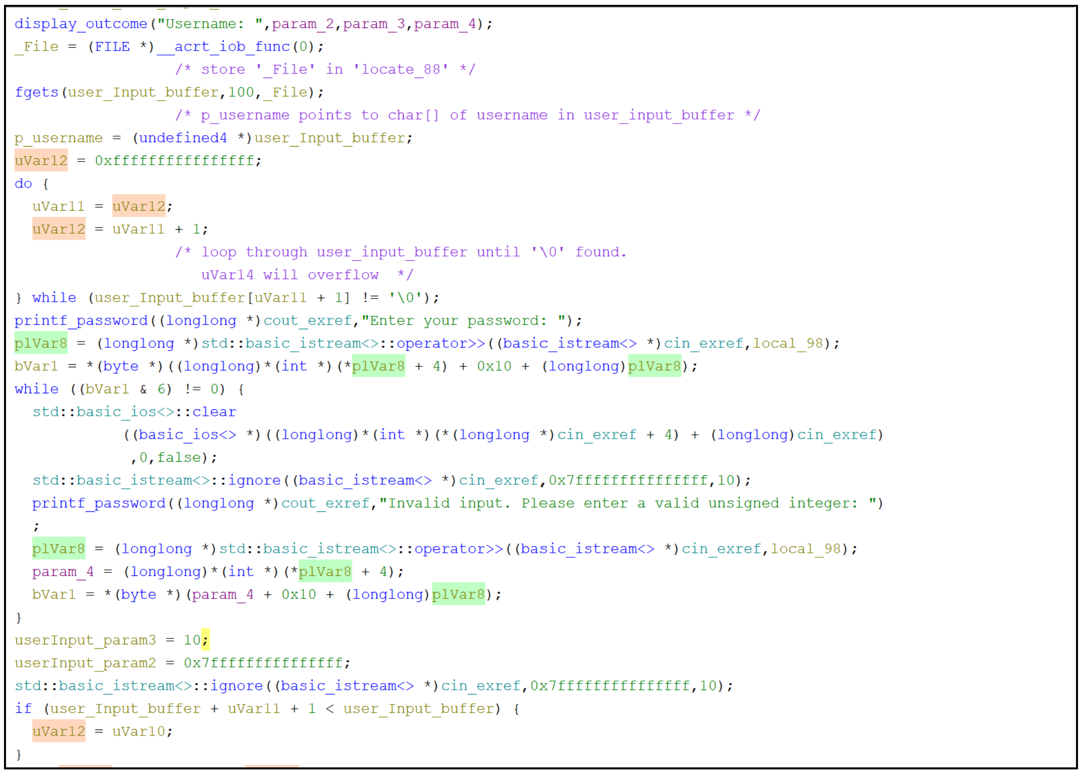
From a closer look, we can infer that this is the **main()** function as it directly calls for argc() and argv(), and exits upon execution instead of returning.



```
    __scrt_release_startup_lock((char)pp_argV);
    ppcVar3 = (code **)FUN_140001c78();
    if ((*ppcVar3 != (code *)0x0) &&
       (uVar4 = FUN_140001a34((longlong)ppcVar3), (char)uVar4 != '\0')) {
      (**ppcVar3)(0,2);
    }
    plVar5 = (longlong *)FUN_140001c80();
    if ((*plVar5 != 0) && (uVar4 = FUN_140001a34((longlong)plVar5), (char)uVar4 != '\0')) {
      _register_thread_local_exe_atexit_callback(*plVar5);
    }
    initial_narrow_environments = _get_initial_narrow_environment();
    puVar6 = (undefined8 *)__p___argv();
    pp_argV = *puVar6;
    puVar7 = (uint *)__p___argc();
    pp_argC = (ulonglong)*puVar7;
    iVar2 = process_user_inputs(pp_argC,pp_argV,initial_narrow_environments,in_R9);
    uVar4 = FUN_140001de4();
    if ((char)uVar4 != '\0') {
      if (!bVar1) {
        _cexit();
      }
      __scrt_uninitialize_crt(CONCAT71((int7)(pp_argC >> 8),1),'\0');
      return iVar2;
    }
    goto LAB_140001780;
  }
}
FUN_140001c90(7);
AB_140001780:
                    /* WARNING: Subroutine does not return */
exit(iVar2);
```

We can also see that the arguments are set to (renamed)variables ‘pp_argV’ and ‘pp_argC’, which are then passed as parameters to the (also renamed) function **‘process_user_inputs()’.**

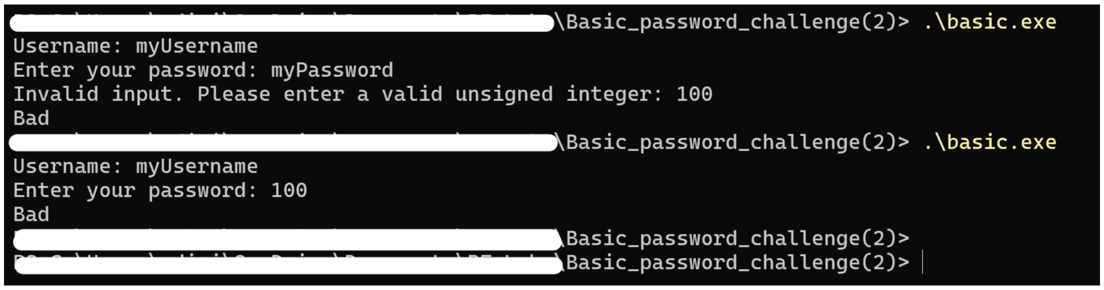
Moving into the function, we see string outputs through the code related to username/password inputs. The program can be seen asking for a username and password, to which in variables ‘uVar12’ and ‘p1Var8’.



```
display_outcome("Username: ",param_2,param_3,param_4);
_File = (FILE *)__acrt_iob_func(0);
                  /* store '_File' in 'locate_88' */
fgets(user_Input_buffer,100,_File);
                  /* p_username points to char[] of username in user_input_buffer */
p_username = (undefined4 *)user_Input_buffer;
uVar12 = 0xffffffffffffffff;
do {
  uVar11 = uVar12;
  uVar12 = uVar11 + 1;
                  /* loop through user_input_buffer until '\0' found.
                     uVar14 will overflow  */
} while (user_Input_buffer[uVar11 + 1] != '\0');
printf_password((longlong *)cout_exref,"Enter your password: ");
plVar8 = (longlong *)std::basic_istream<>::operator>>((basic_istream<> *)cin_exref,local_98);
bVar1 = *(byte *)((longlong)*(int *)(*plVar8 + 4) + 0x10 + (longlong)plVar8);
while ((bVar1 & 6) != 0) {
  std::basic_ios<>::clear
            ((basic_ios<> *)((longlong)*(int *)(*(longlong *)cin_exref + 4) + (longlong)cin_exref)
             ,0,false);
  std::basic_istream<>::ignore((basic_istream<> *)cin_exref,0x7fffffffffffffff,10);
  printf_password((longlong *)cout_exref,"Invalid input. Please enter a valid unsigned integer: ")
  ;
  plVar8 = (longlong *)std::basic_istream<>::operator>>((basic_istream<> *)cin_exref,local_98);
  param_4 = (longlong)*(int *)(*plVar8 + 4);
  bVar1 = *(byte *)(param_4 + 0x10 + (longlong)plVar8);
}
userInput_param3 = 10;
userInput_param2 = 0x7fffffffffffffff;
std::basic_istream<>::ignore((basic_istream<> *)cin_exref,0x7fffffffffffffff,10);
if (user_Input_buffer + uVar11 + 1 < user_Input_buffer) {
  uVar12 = uVar10;
}
```

Upon closer examination of the section above, it becomes evident that there's a third input field. In the event of an invalid password, this field prompts for a "valid unsigned integer," which is subsequently stored in the variable password. This clarifies why, unlike the username input variable, the password input is accessed by a pointer, allowing for potential modification by the third user input. More importantly, this also tells me that the password is expected to be stored and/or processed into an unsigned integer.

We can verify the arguments and there types by running the code on terminal:

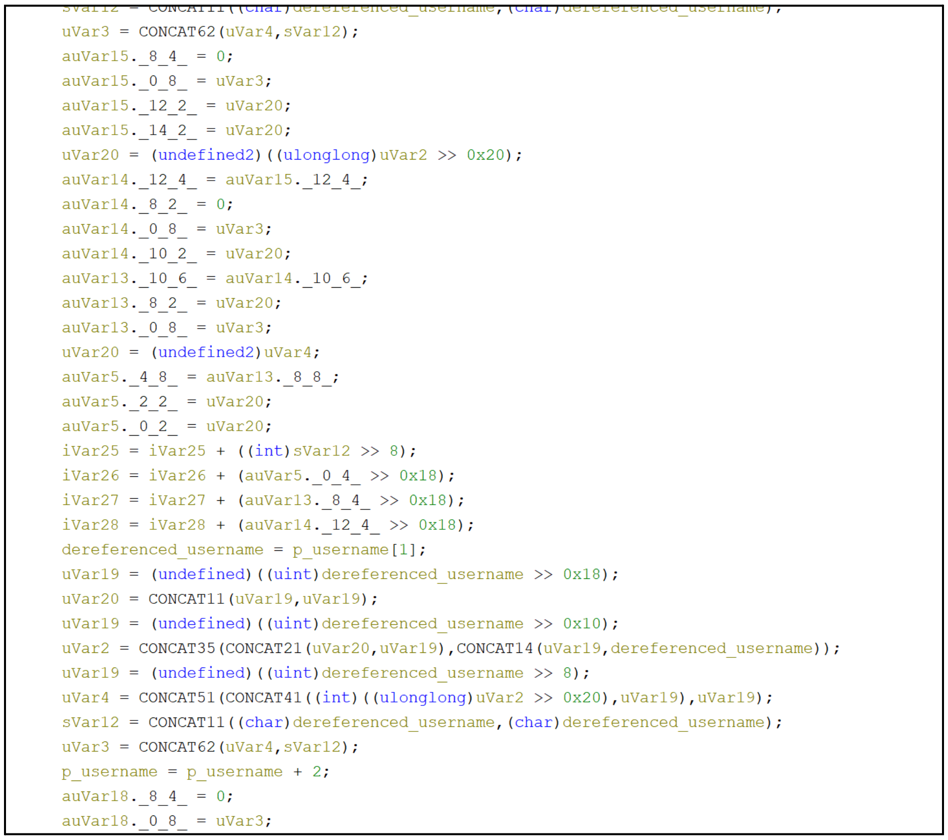


```
[redacted]\Basic_password_challenge(2)> .\basic.exe
Username: myUsername
Enter your password: myPassword
Invalid input. Please enter a valid unsigned integer: 100
Bad
[redacted]\Basic_password_challenge(2)> .\basic.exe
Username: myUsername
Enter your password: 100
Bad
[redacted]\Basic_password_challenge(2)>
[redacted]\Basic_password_challenge(2)>
```

As shown above, when a password is provided as a string, it shows an error and asks for a 'valid unsigned signature'. If we input an unsigned integer into the password, no error is provided and supposedly moves on to verify the credentials.

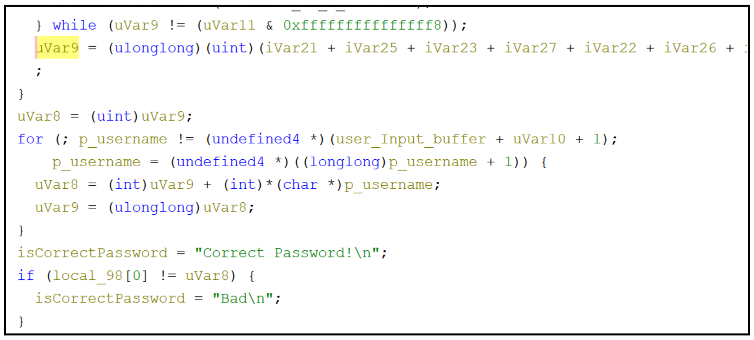
Here's where things get complicated. What we see in the process_user_input() function is that, after the function takes in the username and password, it puts the username through a do-while loop, where it undergoes a series of concatenations and bitwise operations. The resulting integer value is then compared with the password for validation.

A .cpp copy of the process_user_input() function should be included in the repo for reference. The transformation code itself spans 70 lines of code, a portion of it the code is shown below:



```
sVar12 = CONCAT11((char)dereferenced_username,(char)dereferenced_username);
uVar3 = CONCAT62(uVar4,sVar12);
auVar15._8_4_ = 0;
auVar15._0_8_ = uVar3;
auVar15._12_2_ = uVar20;
auVar15._14_2_ = uVar20;
uVar20 = (undefined2)((ulonglong)uVar2 >> 0x20);
auVar14._12_4_ = auVar15._12_4_;
auVar14._8_2_ = 0;
auVar14._0_8_ = uVar3;
auVar14._10_2_ = uVar20;
auVar13._10_6_ = auVar14._10_6_;
auVar13._8_2_ = uVar20;
auVar13._0_8_ = uVar3;
uVar20 = (undefined2)uVar4;
auVar5._4_8_ = auVar13._8_8_;
auVar5._2_2_ = uVar20;
auVar5._0_2_ = uVar20;
iVar25 = iVar25 + ((int)sVar12 >> 8);
iVar26 = iVar26 + (auVar5._0_4_ >> 0x18);
iVar27 = iVar27 + (auVar13._8_4_ >> 0x18);
iVar28 = iVar28 + (auVar14._12_4_ >> 0x18);
dereferenced_username = p_username[1];
uVar19 = (undefined)((uint)dereferenced_username >> 0x18);
uVar20 = CONCAT11(uVar19,uVar19);
uVar19 = (undefined)((uint)dereferenced_username >> 0x10);
uVar2 = CONCAT35(CONCAT21(uVar20,uVar19),CONCAT14(uVar19,dereferenced_username));
uVar19 = (undefined)((uint)dereferenced_username >> 8);
uVar4 = CONCAT51(CONCAT41((int)((ulonglong)uVar2 >> 0x20),uVar19),uVar19);
sVar12 = CONCAT11((char)dereferenced_username,(char)dereferenced_username);
uVar3 = CONCAT62(uVar4,sVar12);
p_username = p_username + 2;
auVar18._8_4_ = 0;
auVar18._0_8_ = uVar3;
```

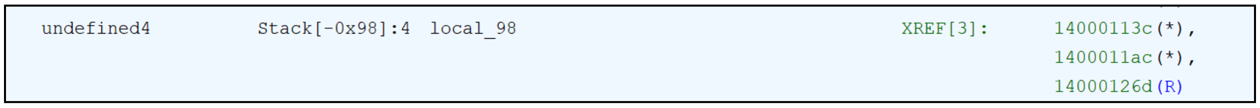
Towards the end, we see this section of code that handles validation:



```
  } while (uVar9 != (uVar11 & 0xfffffffffffffff8));
  uVar9 = (ulonglong)(uint)(iVar21 + iVar25 + iVar23 + iVar27 + iVar22 + iVar26 + 
  ;
}
uVar8 = (uint)uVar9;
for (; p_username != (undefined4 *)(user_Input_buffer + uVar10 + 1);
    p_username = (undefined4 *)((longlong)p_username + 1)) {
  uVar8 = (int)uVar9 + (int)*(char *)p_username;
  uVar9 = (ulonglong)uVar8;
}
isCorrectPassword = "Correct Password!\n";
if (local_98[0] != uVar8) {
  isCorrectPassword = "Bad\n";
}
```

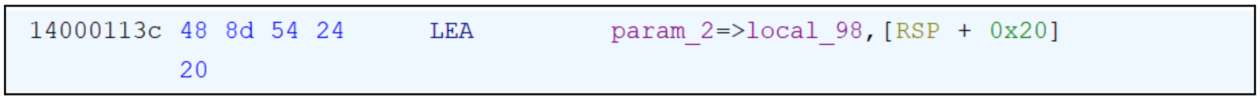
uVar9 seems to be the result of the do-while loop. In the validation section, we see that uVar9 is being casted as an unsigned-integer and set to uVar8. It then compares uVar8 to local_98[0] to determine if the password is valid.

So let's figure out what local_98 could be. In the function prologue, we can see local_98 is located on Stack[9-0x980], and its RSP is referenced in three locations within the disassembly.

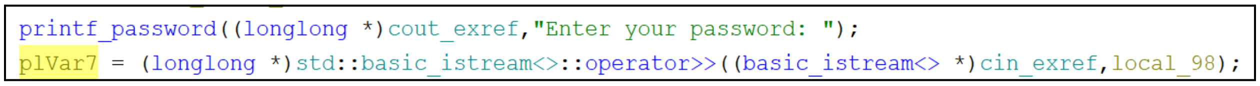


```
undefined4        Stack[-0x98]:4  local_98                        XREF[3]:     14000113c(*),
                                                                               1400011ac(*),
                                                                               14000126d(R)
```

We can move into 13000113c to look at it's disassembly and C code:



```
14000113c 48 8d 54 24      LEA        param_2=>local_98,[RSP + 0x20]
          20
```



```
printf_password((longlong *)cout_exref,"Enter your password: ");
plVar7 = (longlong *)std::basic_istream<>::operator>>((basic_istream<> *)cin_exref,local_98);
```

This instruction is loading the effective address of local_98 into the register param_2 calculated as [RSP + 0x20] which is likely intended to be used as a pointer. This address could be pointing to the location of local_98 in memory or to some other data structure that local_98 is a part of. LEA is commonly used to pass addresses or pointers between functions in assembly language.

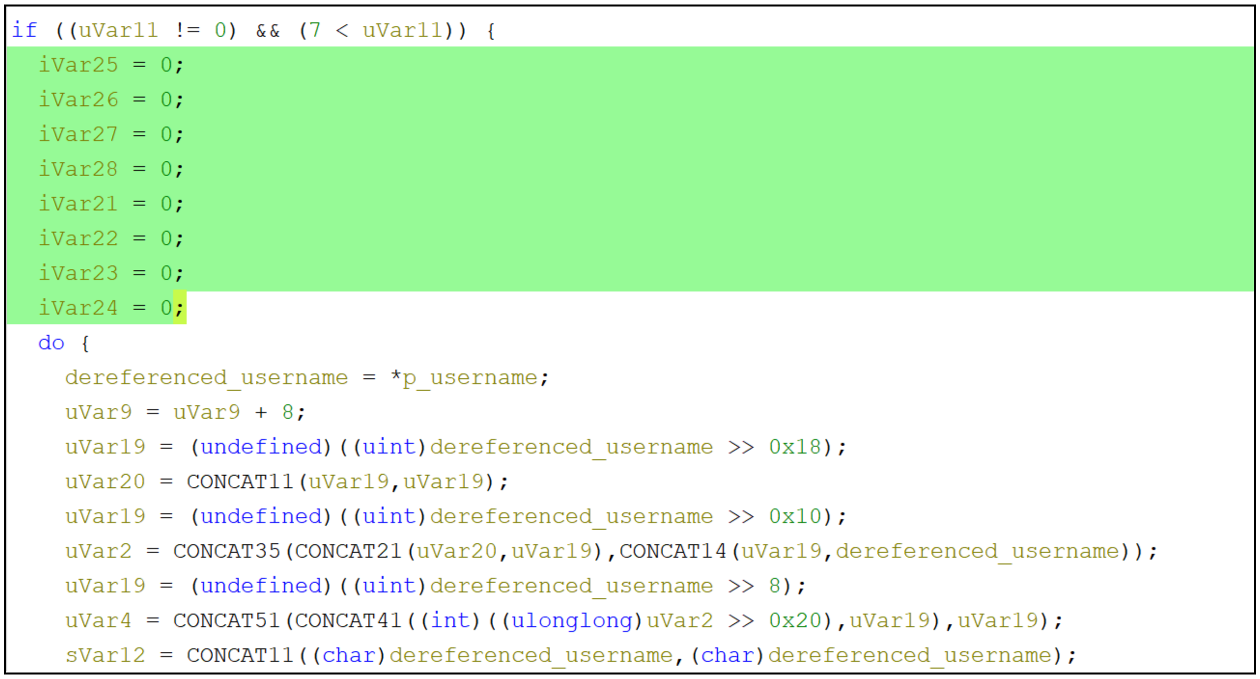
The C code shows that 13000113c is a pointer variable to the password input. Considering both the disassembly and C code, we can infer that local_98 is likely the inputted password with an unsigned-integer type.

Here's what we know so far: Username and password are passed as parameters to process_user_input(). The 'username' then undergoes a long series of concatenation and bitwise operations, whose integer output is compared with the 'password'. If they're equal, the function should print "correct password!", and otherwise print "bad".

Knowing this, we should then figure out what exactly the "series of transformations" are doing to the username. We can replicate the transformation code to create a keygen program.

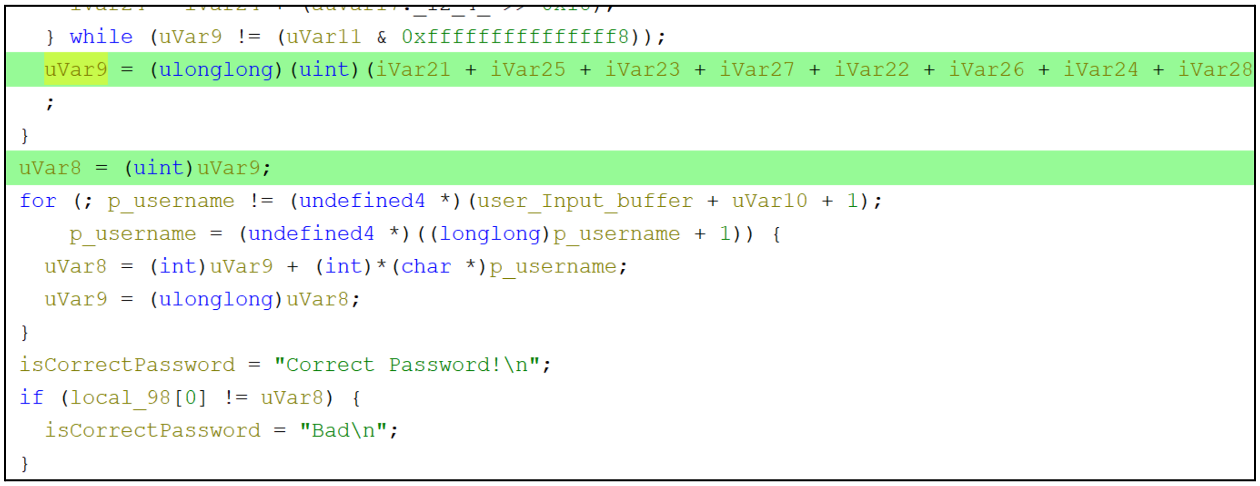
Now let's figure out what uVar8 could be.

At the start of the do-while loop, we see 8 integer variables being initialized as 0.



```
if ((uVar11 != 0) && (7 < uVar11)) {
  iVar25 = 0;
  iVar26 = 0;
  iVar27 = 0;
  iVar28 = 0;
  iVar21 = 0;
  iVar22 = 0;
  iVar23 = 0;
  iVar24 = 0;
  do {
    dereferenced_username = *p_username;
    uVar9 = uVar9 + 8;
    uVar19 = (undefined)((uint)dereferenced_username >> 0x18);
    uVar20 = CONCAT11(uVar19,uVar19);
    uVar19 = (undefined)((uint)dereferenced_username >> 0x10);
    uVar2 = CONCAT35(CONCAT21(uVar20,uVar19),CONCAT14(uVar19,dereferenced_username));
    uVar19 = (undefined)((uint)dereferenced_username >> 8);
    uVar4 = CONCAT51(CONCAT41((int)((ulonglong)uVar2 >> 0x20),uVar19),uVar19);
    sVar12 = CONCAT11((char)dereferenced_username,(char)dereferenced_username);
```

Then towards the end of the loop inside the while(), we see the post-processed concatenation of these integer variables being set to uVar9. In the validation section, uVar9 is casted as an unsigned-integer and set to uVar8.



```
  } while (uVar9 != (uVar11 & 0xfffffffffffffff8));
  uVar9 = (ulonglong)(uint)(iVar21 + iVar25 + iVar23 + iVar27 + iVar22 + iVar26 + iVar24 + iVar28
  ;
}
uVar8 = (uint)uVar9;
for (; p_username != (undefined4 *)(user_Input_buffer + uVar10 + 1);
    p_username = (undefined4 *)((longlong)p_username + 1)) {
  uVar8 = (int)uVar9 + (int)*(char *)p_username;
  uVar9 = (ulonglong)uVar8;
}
isCorrectPassword = "Correct Password!\n";
if (local_98[0] != uVar8) {
  isCorrectPassword = "Bad\n";
}
```

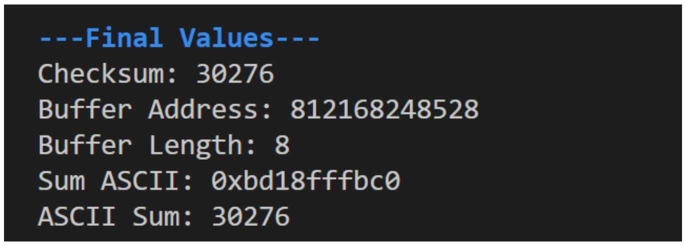
While also considering that uVar11 - which seams to keep count of either bytes of chars in the inputted password - has to be greater than 7 for the do-while loop to run, we can infer that the function splits the inputted password between variables uVar21 through uVar28, process them through various bitwise and concatenation operations, then finally concatenate the variables in the following order: iVar21 + iVar25 + iVar23 + iVar27 + iVar22 + iVar26 + iVar24 + iVar28.

Writing the Keygen

From what's been uncovered from the process_user_input() function, we can move onto building a keygen. The most important part was understanding the exact processes behind splitting the inputted password between variables uVar21 through uVar28 and their subsequent processing. I took the liberty to dissect the code and rewrite it more concisely in C++ to better understand the processes, the code can be found in the initial_keygen_attempt.cpp file.

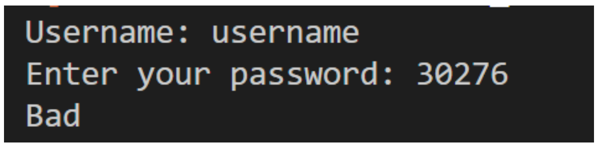
During the translation, I figured out that many of the unknown variables were initializations for output fields, such as buffer_size, buffer_length, ASCII_sum, etc. I also found out the uVar11 was actually a count for the bytes, meaning that the do-while loop checks if the password is greater than 8 bytes before it begins. Eventually, I figured that the inputted password is split into 8-byte blocks, which are then stored between variables uVar21 through uVar28, which are then processed and concatenated.

My initial_keygen_attempt.cpp code seems to produce the incorrect output, but gave me insight into how the code works. For example, I will input username as 'username' and password as 'password' into the code. Here's the output:



```
---Final Values---
Checksum: 30276
Buffer Address: 812168248528
Buffer Length: 8
Sum ASCII: 0xbd18fffbc0
ASCII Sum: 30276
```

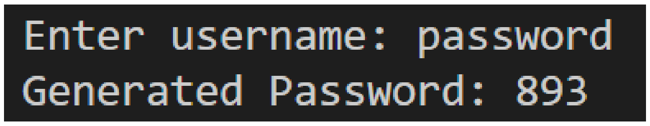
Focus on the checksum, which comes out as 30276. From all that we've seen, we can assume that this should be the password generated from the username. Let's try running it on the original executable:



```
Username: username
Enter your password: 30276
Bad
```

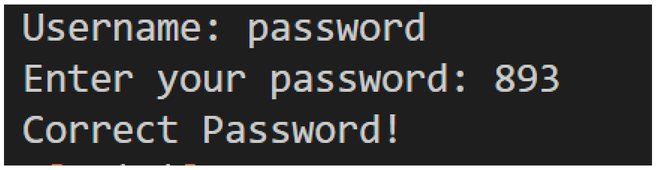
Due to the complexity of the translated code, I'm assuming two things: My translated code failed to precisely translate the decompiled code, and that the code is simply just concatenating the bytes of each character in the inputted password.

A second keygen code attempt - simply named keygen.cpp - was made that just concatenates the bytes blocks in the given order. In short, we know that the password is a concatenation of the bytes in the username. This code outputs just that. Let's run it:



```
Enter username: password
Generated Password: 893
```

Instead of 30276, our checksum is 893. Not let's verify it against the original executable:



```
Username: password
Enter your password: 893
Correct Password!
```

It outputs 'Correct Password!'. In summary, the generated password is just the concatenated bytes of the username.

---- END ----