HW2

Problem 1 (15 Points): Hash functions: In class we discussed several desirable properties for hash functions, in particular one-wayness and collision-resistance. In this exercise, we'll show that neither property implies the other. We can do this by counter-example:

1. a) First, define a function that is one-way, but not collision- resistant.

   Hint: The answer will be a trivial function that you would never use as a cryptographic hash function.

   F (x)= x %2

   It is one-way. Because given the output, it is difficult to find the input that produced it, but easy to find a collision input.

1. b) Second, define a function that is collision-resistant, but not one-way.

   Hint: Assume you have a collision-resistant hash-function H. Use that to build a hash function H', which is still collision- resistant but not one-way.

H'(x) = H(x) concate x.

We can get the input at the end of output. So it is not one way.

Because H(x) is collision resistant, so H'(x) is still collision-resistant.

Problem 2 (15 Points): Consider the following C code. What is the problem? Explain.

```
int a, b;
int sum=a+b;
cout<<"Enter two numbers to add: ";
cin>>a;
cin>>b;
cout<<"The sum is: "<<sum;
```

1. Sign mismatch. Two `int-type` number additions may be out of the int range. It can cause overflow. For example, INT_MAX + 1 = INT_MIN
2. Input happens after the sum calculation. So, even if the user enters values, they won't be used in the `sum` calculation because it occurred before the input.

3. if you read a string using `cin` into a fixed-size character array and the input is longer than int length, it can overwrite adjacent memory, leading to undefined behavior and potential security vulnerabilities. To mitigate this, you should use functions like `getline` or consider dynamic memory allocation for input that may vary in size.
4. **Type Mismatch**: `cin` expects input to match the data type of the variable it's reading into. If the user enters data of the wrong type, it can lead to unexpected behavior or errors in the program. For instance, if you're expecting an integer, but the user enters a string, it may result in input failure and possible security risks if the program doesn't handle this gracefully.

Problem 3 (30 Points): Simple Reverse Engineering Course. There's a **Linux** binary file called **doorkeeper**, which has two password checks. Try to crack the two passwords. Show your cracking progress by screenshot step by step. To simplify the process, the debug information is reserved, and the optimization is disabled.

angr

9.2.69

```
601     char v2;  // [bp-0x428]
602     char v3;  // [bp-0x427]
603     char v4;  // [bp-0x426]
604     char v5;  // [bp-0x425]
605     char v6;  // [bp-0x424]
606     char v7;  // [bp-0x423]
607     char v8;  // [bp-0x422]
608     char v9;  // [bp-0x421]
609     char v10;  // [bp-0x420]
610     char v11;  // [bp-0x41f]
611     char v12;  // [bp-0x41e]
612     char v13;  // [bp-0x41d]
613     char v14;  // [bp-0x41c]
614     char v15;  // [bp-0x41b]
615     char v16;  // [bp-0x41a]
616     char v17;  // [bp-0x419]
617     void* v18;  // [bp-0x418], Other Possible Typ
618     char v19;  // [bp-0x218], Other Possible Type
619     unsigned long long v21;  // rcx
620     unsigned long long *v22;  // rdi
621     unsigned long v23;  // d
622     unsigned long long *v24;  // rdi
623     unsigned long long v25;  // rcx
624
625     v0 = "Th1s_1s_PassWord_1";
626     v21 = 64;
627     for (v22 = &v18; v21; v22 = &v22[v23])
628     {
629         v21 -= 1;
630         v18 = 0;
631     }
632     printf("input password 1->");
633     __isoc99_scanf("%s", (unsigned int)&v18);
634     if (check_password(&v18, v0))
635     {
636         puts("password 1 is correct");
637         v2 = 99;
638         v3 = 175;
639         v4 = 86;
640         v5 = 98;
641         v6 = 103;
642         v7 = 81;
```

We can find first password is Th1s_1s_PassWord_1

We should read

```
char* password2
```

```
(gdb) b 40
Breakpoint 1 at 0x2014: file ./test.c, line 40.
(gdb) c
The program is not being run.
(gdb) r
Starting program: /home/ubuntu/linjuyi/share/learningpybind/doorkeeper
input password 1->Th1s_1s_PassWord_1
password 1 is correct

Breakpoint 1, main () at ./test.c:40
40          char input2[AES_ENC_MAX_LEN] = {0};
(gdb) p password2
$1 = 0x555555758a80 "This1sTheS3cret"
(gdb)
```

So the second password is This1sTheS3cret

```
#0  main () at ./test.c:40
40          char input2[AES_ENC_MAX_LEN] = {0};
(gdb) c
Continuing.
input password 2->This1sTheS3cret
password 2 is correct
sh: 1: pause: not found
Nice!!!!!!!!!!!!![Inferior 1 (process 26700) exited normally]
```

Question4

Main course: You are a 1337 H4x0r, after breaching the security perimeter of a high profile target you were able to compromise one of their devices, find some **encrypted** passwords and exfiltrate the **password generation binary.**

First, run the provided **mystery** binary (./mystery32 or ./macMystery) to find the inputs that will produce the following encrypted passwords: (15 points)

We can not brute force 0-99999999 all numbers. It takes too much time.

i. ECEAFHGJ 54379168

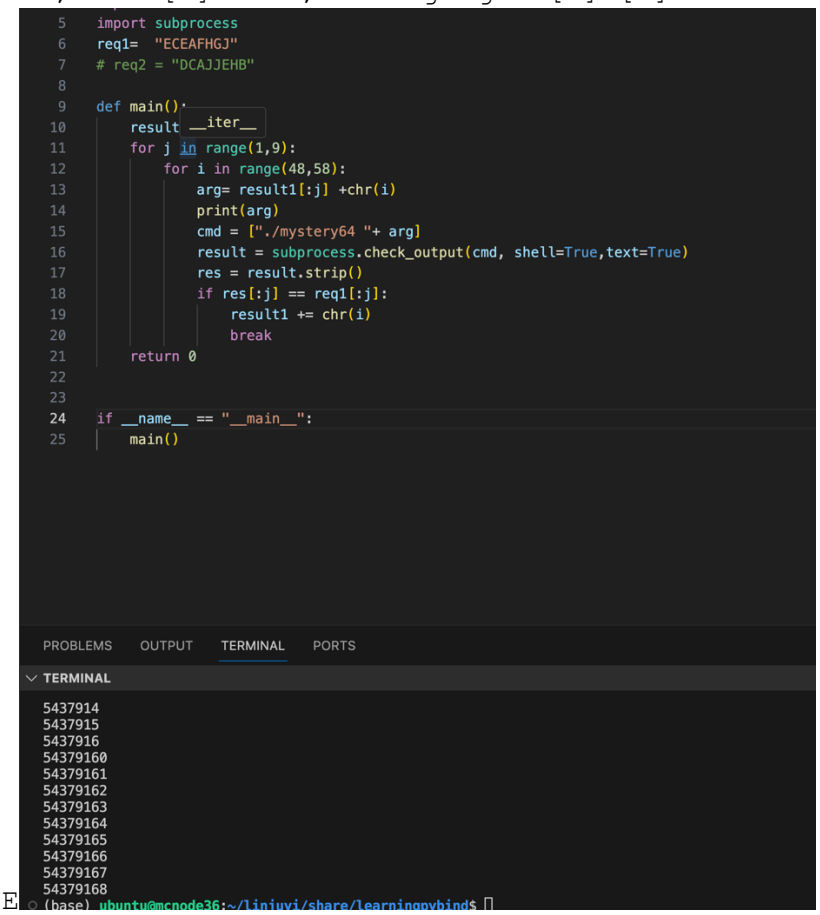
ii. DCAJJEHB 15935728

iii. DEKJJBCI   12345678

```
(base) ubuntu@mcnode36:~/linjuyi/share/learningpybind$ ./mystery64 12345678
DEKJJBCI
```

Hint: Time, automating the process will save

Write a program, if a[0] == E, then judge a[0]a[1] == EC then judge a[0]- 3 == ECE




```python
# write a program, iterate all numbers  from 0 to 99999999.
# For example , we run ./mystery64 0, then ./mystery64 1
import subprocess
# req1=  "ECEAFHGJ"
req1 = "DCAJJEHB"

def main():
    result1= ""
    for j in range(1,9):
        for i in range(48,58):
            arg= result1[:j] +chr(i)
            print(arg)
            cmd = ["./mystery64 "+ arg]
            result = subprocess.check_output(cmd, shell=True,text=True)
            res = result.strip()
            if res[:j] == req1[:j]:
                result1 += chr(i)
                break
    return 0
```

```
if __name__ == "__main__":
    main()
```

2. b) Second, **decompile** the provided binary and **identify the functions** (e.g., names, input-output parameters, etc.) used for the encrypted password generation. (25 points)
   Hint: For function symbols, you shall look

```
220 // 804860A: using guessed type char s[9];
221
222 //----- (08048716) --------------------------------------------------------
223 char *__cdecl blowfish_encryption(int a1)
224 {
225   int i; // [esp+10h] [ebp-28h]
226   int v3; // [esp+14h] [ebp-24h]
227   char v4[9]; // [esp+23h] [ebp-15h] BYREF
228   unsigned int v5; // [esp+2Ch] [ebp-Ch]
229
230   v5 = __readgsdword(0x14u);
231   v3 = 0;
232   for ( i = 0; i <= 7; ++i )
233   {
234     v3 = block_func(*(char *)(i + a1) - 48, v3);
235     v4[i] = v3 + 48;
236   }
237   return v4;
238 }
239 // 8048784: returning address of temporary local variable '%var_15'
240 // 8048716: using guessed type char var_15[9];
241
242 //----- (080487A0) --------------------------------------------------------
```

names: blowfish_encryption

input-output parameters

input: int

output: char*

3. c) Third, identify the steps of the password generation algorithm. Provide clear explanations on how the input is transformed to the output in a step-by-step fashion. (40 points)
   Hint: A chain, everything is

```
132 //----- (080484CB) --------------------------
133 int __cdecl main(int argc, const char **argv,
134 {
135   if ( argc == 2 )
136   {
137     strcpy(&password, argv[1]);
138     pass_len = strlen(&password);
139     if ( pass_len > 8 )
140     {
141       puts("Password too long! Provide **numer
142       return 1;
143     }
144     for ( i = pass_len; i <= 7; ++i )
145       *(&password + i) = 32;
146     enc_output = (int)blowfish_encryption((int
147     convert_to_char((_BYTE *)enc_output);
148   }
149   else if ( argc <= 2 )
150   {
151     puts("Feed me passwords...");
152   }
153   else
154   {
155     puts("Too many inputs...try again");
156   }
157   return 0;
158 }
159 // 804A02C: using guessed type int pass_len;
160 // 804A030: using guessed type int enc_output;
161 // 804A040: using guessed type int i;
```

Input is a numeric password of up to 8 digits.
Then, we do a padding.
for (i = pass_len; i <= 7; ++i)
    *(&password + i) = 32;
means if length <=8, append space to 8 length.

Then, we use blowfish_encryption. From decompiling, we can get some information. See question b

It invokes block_func

```
3 //----- (080485C4) --------------------------
4 int __cdecl block_func(int a1, int a2)
5 {
6   return 3 * (a2 + a1) % 11;
7 }
8
9 //----- (0804860A) --------------------------
```

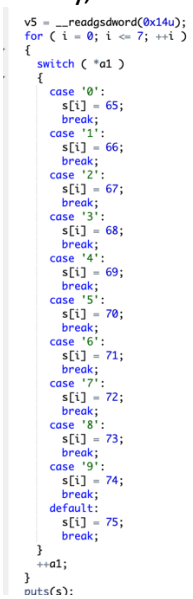
First it calculate the first number, then calculate the second number.

For example, We input `12345678`, get
./mystery64 `12345678`
`DEKJJBCI`

V3 = blockfunc(1, 0)
V3 = 3
V4[0] = 3

V3 = blockfunc(2, 3)
V3 = 4
V4[1] = 4

Finally, it convert number to character.



```
v5 = __readgsdword(0x14u);
for ( i = 0; i <= 7; ++i )
{
  switch ( *a1 )
  {
    case '0':
      s[i] = 65;
      break;
    case '1':
      s[i] = 66;
      break;
    case '2':
      s[i] = 67;
      break;
    case '3':
      s[i] = 68;
      break;
    case '4':
      s[i] = 69;
      break;
    case '5':
      s[i] = 70;
      break;
    case '6':
      s[i] = 71;
      break;
    case '7':
      s[i] = 72;
      break;
    case '8':
      s[i] = 73;
      break;
    case '9':
      s[i] = 74;
      break;
    default:
      s[i] = 75;
      break;
  }
  ++a1;
}
puts(s);
```

so v4[0] = 3 - >E , V4[1] = 4 ->E