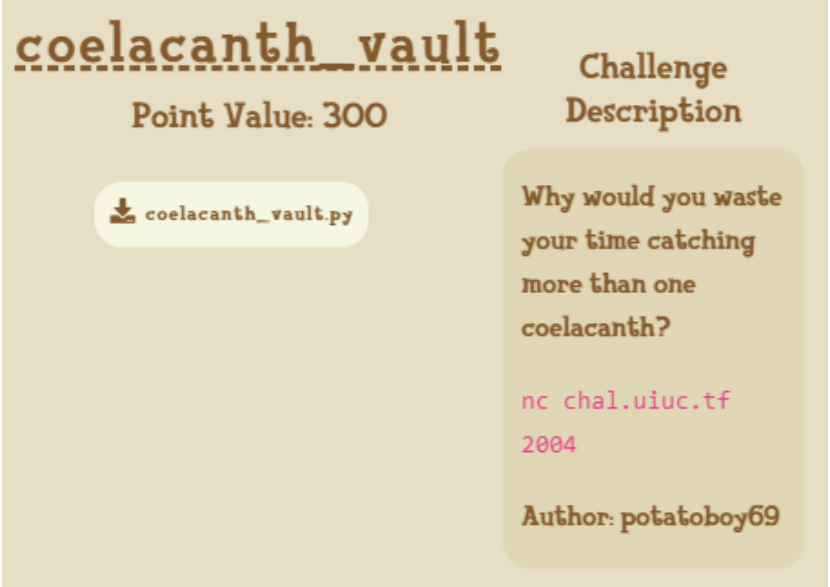
Coelacanth Vault Challenge Write-Up



Summary

Solving the challenge required a deep analysis of the output received from the server and the python script that was provided. This write-up could be summed up with these key steps:

1. Thoroughly reviewing the output from the server and python code provided.
2. Identifying the key functions that generate the secret key.
3. Understanding the create_key function well and figuring out the importance of prime numbers stored in the seq variable.
4. Concluding that it is necessary to figure out all possible prime numbers that could be generated.
5. Computing all possible prime numbers and identifying the smallest one not provided by the program.
6. Using the construct_key function to create hundreds of possible secret key guesses that contain the correct one.
7. Optionally, automating the entire process of grabbing the key portions and sending each guess until it finds the correct one.

Getting Started - Time to Analyze The Script and Server Output

At first, I connected to the server to see what information the challenge provided. I noticed the hint saying it is a waste of time catching one coelacanth. So when I noticed the prompt asking how many I caught, I put in 2. I was prompted for a key that I had no idea what it was. So I then gave it a random number to see what would happen. Apparently, you have 251 tries to figure out the password to the lock.

```
kali@kali:~/Desktop$ nc chal.uiuc.tf 2004
Hi, and welcome to the virtual Animal Crossing: New Horizon coelacanth vault!
There are 5 different cryptolocks that must be unlocked in order to open the vault.
You get one portion of each code for each coelacanth you have caught, and will need to us
e them to reconstruct the key for each lock.
Unfortunately, it appears you have not caught enough coelacanths, and thus will need to f
ind another way into the vault.
Be warned, these locks have protection against brute force; too many wrong attempts and y
ou will have to start over!

How many coelacanth have you caught? 2
Generating key for lock 0, please wait...
Generated!
Here are your key portions:
[(148, 173), (55, 167)]
Please input the key: 1
Key incorrect. You have 250 tries remaining for this lock.
Please input the key: ^C
kali@kali:~/Desktop$
```

Next, I took a look at the code provided to see what the python script was doing. At first glance, I could tell the script had a lot going on. At the top of the script, you can see the libraries it was using. The one that stood out to me was the Crypto library so I had a hunch that cryptography was involved.

Beneath the import section, you will see some global variables that are probably constant values since they were in all uppercase (a common programming style). The MAX_COELACANTH = 9 was helpful because it meant that 9 was some threshold. Beneath that you see a lambda function with a comment saying it was a substitute for the built-in math prod function. If you look up that function, you can quickly tell that it is just multiplying all of the values that call the "prod" lambda function.

```
 1 #!/usr/bin/python
 2 import random
 3 from Crypto.Util import number
 4 from functools import reduce
 5
 6 TOTAL = 15
 7 THRESHOLD = 10
 8 MAX_COELACANTH = 9
 9
10 NUM_LOCKS = 5
11 NUM_TRIES = 250
12
13 # substitute for math.prod
14 prod = lambda n: reduce(lambda x, y: x*y, n)
15
```

Next, you will see the function call create_key with two required inputs t and n and an optional input for size which defaults to the value of 8. If you recall, the server requested that you input the key, so this function must be important for understanding. Beneath it, there is a function that

constructs a key with an input of shares. Without having any experience in cryptography, I moved on to read the rest of the script.

```
16 def create_key(t, n, size=8):
17     while True:
18         seq = sorted([number.getPrime(size) for _ in range(TOTAL)])
19         if len(set(seq)) != len(seq):
20             continue
21 
22         alpha = prod(seq[:t])
23         beta = prod(seq[-t + 1:])
24         if alpha > beta:
25             secret = random.randint(beta, alpha)
26             shares = [(secret % num, num) for num in seq]
27             return secret, shares
28 
29 def construct_key(shares):
30     glue = lambda A, n, s=1, t=0, N=0: (n < 2 and t % N or glue(n, A % n, t, s - A//n
    * t, N or n), -1)[n < 1]
31     mod = prod([m for s, m in shares])
32     secret = sum([s * glue(mod//m, m) * mod//m for s, m in shares]) % mod
33     return secret
34 
```

Finally, you get to the "main" section of the python script that has the familiar and also strange if __name__ == "__main__": statement. This is a python way of running the main code only if the script was executed normally instead of being imported. So this should be the first code executed if the script was designed well. It is quite lengthy too, but that is because of a lot of print statements. Fortunately, the print statements are hints as to what should cause the code to reach that point, so they help understand the logic of the script. The first chunk of print statements is the text you get when you connect to the server. It then gets your input on how many of the fish you caught and stores it in the num_shares.

```
35 if __name__ == "__main__":
36     print("Hi, and welcome to the virtual Animal Crossing: New Horizon coelacanth vau
   lt!")
37     print("There are {} different cryptolocks that must be unlocked in order to open
   the vault.".format(NUM_LOCKS))
38     print("You get one portion of each code for each coelacanth you have caught, and
   will need to use them to reconstruct the key for each lock.")
39     print("Unfortunately, it appears you have not caught enough coelacanths, and thus
    will need to find another way into the vault.")
40     print("Be warned, these locks have protection against brute force; too many wrong
    attempts and you will have to start over!")
41     print("")
42 
43     num_shares = abs(int(input("How many coelacanth have you caught? ")))
```

The next chunk is under an if statement. It states that the number of shares you typed in cannot be more than the MAX_COELACANTH defined earlier. So you can only have a number as large as 9.

```
44      if num_shares > MAX_COELACANTH:
45          print("Don't be silly! You definitely haven't caught more than {} coelacanth!
    ".format(MAX_COELACANTH))
46          print("Come back when you decide to stop lying.")
47          quit()
48
```

The next section is a for loop iterating over the number of locks defined earlier (5). This implies that 5 locks will need to be solved. The first code says it is generating a key for the lock. It calls the create_key function with the THRESHOLD and TOTAL constants as inputs and it sets the returned values to secret and shares. I scrolled up and looked at the create_key function and my eyes went crossed eyed so I read on.

Next, there is a construc_key function call using the random.sample built-in function. This will return a new list from a larger list (in this case shares) with random unique values chosen. It only returns the THRESHOLD amount (10). So the construct_key is being sent 10 random "shares" and it returns the value to a r_secret variable. Once again I looked at the functions above and I started getting a headache. Someone wrote some fancy code that is probably cryptographically secure, but it was over my head. So I continued reading.

Next, you will have the assert function being called. It will throw an error if something returns false. So the value of secret must equal r_secret of the program will display an error before saying "Generated!." I noticed in the original output when I connected to the server that I never received an error or something about an assert failing, so these values must have been equal. This is interesting because the shares variable that comes from the create_key variable is used to create the key again. But more interestingly it only needs 10 random samples to create the key. But since I was near the end of the script, I kept reading on.

The next part is it prints out key portions from the shares variable. Sweet! That means it is giving you the output necessary to generate the key. I was beginning to wonder why so few people solved the challenge so far. Then I noticed that the num_shares variable is the value that I provided when I connected to the program (which was 2). I want 10 so I can generate the secret.

```
49      for lock_num in range(NUM_LOCKS):
50          print("Generating key for lock {}, please wait...".format(lock_num))
51          secret, shares = create_key(THRESHOLD, TOTAL)
52          r_secret = construct_key(random.sample(shares, THRESHOLD))
53          assert(secret == r_secret)
54          print("Generated!")
55
56          print("Here are your key portions:")
57          print(random.sample(shares, num_shares))
58
```

So I ran the program again trying to get the 10 key portions so I could just generate the key.

```
kali@kali:~/Desktop$ nc chal.uiuc.tf 2004
Hi, and welcome to the virtual Animal Crossing: New Horizon coelacanth vault!
There are 5 different cryptolocks that must be unlocked in order to open the vault.
You get one portion of each code for each coelacanth you have caught, and will need to us
e them to reconstruct the key for each lock.
Unfortunately, it appears you have not caught enough coelacanths, and thus will need to f
ind another way into the vault.
Be warned, these locks have protection against brute force; too many wrong attempts and y
ou will have to start over!

How many coelacanth have you caught? 10
Don't be silly! You definitely haven't caught more than 9 coelacanth!
Come back when you decide to stop lying.
kali@kali:~/Desktop$
```

However, I quickly learned that I forgot about the if statement preventing me from saying I caught 10 fish. The largest number I could type in was 9. At this point, I was sad since the challenge wasn't as easy as I hoped, so I went back to the source code. Under the key portions, there is another for loop nested under the first (ya nested for loops!). This loops NUM_TRIES (250) which stores each try in the num_attemps. So it was clear I only had 250 attempts. I immediately concluded that there was going to be some kind of guessing involved for the 10 key portions since you can't make the program just print it out.

The first thing it does is ask you for the key. It then checks to see if it matches the secret key created earlier and tells you that you did a good job or you have x number attempts left to guess the key depending on your guessing skills. If you were correct, it breaks out of the nested for loop and appears to ask you for the key for a new lock. I also noticed that if you exceed the 250 guesses the program would quit due to some brute force detection. Bummer, that is usually the easiest solution given enough time.

```
59          for num_attempts in range(NUM_TRIES):
60              n_secret = abs(int(input("Please input the key: ")))
61              if secret == n_secret:
62                  print("Lock {} unlocked with {} failed attempts!".format(lock_num, nu
    m_attempts))
63                  break
64              else:
65                  print("Key incorrect. You have {} tries remaining for this lock.".for
    mat(NUM_TRIES - num_attempts))
66          else:
67              print("BRUTE FORCE DETECTED. LOCKS SHUT DOWN.")
68              print("Get out. You don't deserve what's in this vault.")
69              quit()
70
```

Finally, at the end of the file, you will notice that it will print the contents of a flag.txt file if you managed to break all five locks. Phew, that was a lot to read and understand before even trying to figure out how to get the flag!

```
71      print("Opening vault...")
72      with open("flag.txt", "r") as f:
73          print(f.read())
```

Tl;dr after thoroughly reading the non-cryptographic code, it was clear that you needed to guess (up to 250 times) some of the key portions to generate the key.

Debugging Time!
Remember those headache-inducing create_key and contstruct_key functions? Time to figure out what they are doing! Since I had no cryptology experience before this challenge, I made a copy of the script and added a lot of print statements to aid in debugging.

```
16 def create_key(t, n, size=8):
17      while True:
18          seq = sorted([number.getPrime(size) for _ in range(TOTAL)])
19          print("debug 1:")
20          print(seq)
21          if len(set(seq)) != len(seq):
22              print("debug 2:")
23              print(len(set(seq)))
24              print(len(seq))
25              continue
26
27          alpha = prod(seq[:t])
28          print("debug 3:")
29          print(alpha)
30          beta = prod(seq[-t + 1:])
31          print("debug 4:")
32          print(beta)
33          if alpha > beta:
34              secret = random.randint(beta, alpha)
35              print("debug 5:")
36              print(secret)
37              shares = [(secret % num, num) for num in seq]
38              print("debug 6:")
39              print(shares)
40              return secret, shares
```

The output I got from my debug version was very verbose. I omitted it for the sake of brevity, but there were hundreds and hundreds of debug1 and debug2 prints before it moved on to debug 3. When analyzing the difference between the two debugs, I noticed that the lengths were not the same (the values printed out in debug 2). I also noticed that the only one that seemed to move on to debug 3 had no duplicates. I underlined all of the duplicates in other outputs to show this.

```
debug 1:
[139L, 149L, 149L, 163L, 173L, 181L, 181L, 191L, 211L, 211L, 223L, 223L, 227L, 229L, 233L
]
debug 2:
11
15
debug 1:
[137L, 137L, 139L, 149L, 149L, 151L, 163L, 163L, 191L, 191L, 191L, 193L, 223L, 223L, 241L
]
debug 2:
9
15
debug 1:
[131L, 157L, 163L, 167L, 179L, 191L, 193L, 197L, 211L, 223L, 227L, 229L, 239L, 241L, 257L
]
debug 3:
34243037020354006974599
```

So it was safe to assume that the code was checking to make sure that the values stored in the seq variable did not have any duplicates. Looking at the code, it looks like it contains prime numbers based on some size 8 (remember that the create_key function never passes the third variable so the default is used) and it stores 15 of them (value of TOTAL).

At this point, I ran the program and was still confused about the super large numbers for debugging 3, 4, and 5. However, I noticed that the debug 5 value is the secret. So I wanted to see if I pasted this value into the prompt, it would continue. Bingo! Too bad we can't update the script on the server-side because that would make getting this flag much easier.

```
How many coelacanth have you caught? 9
Generating key for lock 0, please wait...
debug 3:
27537187327405791666307
debug 4:
1452478888553936392049
debug 5:
6702438057876536992944
debug 6:
[(52L, 139L), (53L, 149L), (142L, 157L), (26L, 163L), (88L, 173L), (7L, 179L), (91L, 181L
), (57L, 197L), (149L, 211L), (179L, 223L), (206L, 233L), (9L, 239L), (218L, 241L), (22L,
 251L), (204L, 257L)]
Generated!
Here are your key portions:
[(142L, 157L), (206L, 233L), (9L, 239L), (218L, 241L), (179L, 223L), (52L, 139L), (91L, 1
81L), (7L, 179L), (26L, 163L)]
Please input the key: 670243805787653699 2944
Lock 0 unlocked with 0 failed attempts!
Generating key for lock 1, please wait...
debug 3:
13301179395137992002847
```

So next I tried to get an understanding of the debug 3 and 4 values since they are used to generate the secret.

```
16 def create_key(t, n, size=8):
17     while True:
18         # gets 15 prime numbers and stores it in seq
19         seq = sorted([number.getPrime(size) for _ in range(TOTAL)])
20 
21         #print("debug 1:")
22         #print(seq)
23 
24         # if seq does not have unique numbers, start the loop over
25         if len(set(seq)) != len(seq):
26 
27             #print("debug 2:")
28             #print(len(set(seq)))
29             #print(len(seq))
30 
31             continue
32 
33         alpha = prod(seq[:t])
34         print("debug 3:")
35         print(alpha)
36         beta = prod(seq[-t + 1:])
37         print("debug 4:")
38         print(beta)
39         if alpha > beta:
40             secret = random.randint(beta, alpha)
41             print("debug 5:")
42             print(secret)
43             shares = [(secret % num, num) for num in seq]
44             print("debug 6:")
45             print(shares)
46             return secret, shares
```

Both alpha and beta call the prod function which just multiples all of the numbers provided and returns the product. There is some slicing going on for the seq variable, so it looks like alpha and beta are the products of some, but not all of the prime numbers. I tried to understand why this was done, but it was not clear. So now that I knew what debug 3 and 4 were for, I moved on to the if statement right above where secret is set. It makes sure that the alpha variable is greater than the beta variable.

Next, the secret is set to the return value of the randint function which is >= beta and less than or equal to alpha. Looking back at my output, I saw that alpha and beta were extremely huge numbers so it was impossible to guess what the random number was with only 250 attempts per lock! I learned a lot so far, but I still had a long way to go.

Now it was time to figure out what the shares were. This was the variable set to the construct_key function that magically creates the secret from it! So understanding this seemed critical, especially since the program prints out up to 9 of them. This means you only need to guess 1 correctly. The code for setting the shares variable is using the cryptic python shortcut to make a very compact for loop. This loop iterates over each prime number in the seq variable and assigns it to num. Then, in the parentheses, the modulus is computed for the secret by

dividing it by num and storing its remainder. If that makes no sense, check out khan academy to learn more about it.

Next, it stores the value of num which happens to be the prime number! This is awesome because the shares that are given to us contain the remainder of dividing the secret by the prime number and the prime number itself. At this point, we "reverse engineered" how the create_key works, but the problem is how do we get the missing key portion?

```
33          # gets the product of some but not all of the prime numbers in seq
34          alpha = prod(seq[:t])
35
36          #print("debug 3:")
37          #print(alpha)
38
39          # gets the product of some but not all of the prime numbers in seq
40          beta = prod(seq[-t + 1:])
41
42          #print("debug 4:")
43          #print(beta)
44
45          if alpha > beta:
46
47              # random number between beta and alpha. Example: 6702438057876536992944
48              secret = random.randint(beta, alpha)
49
50              #print("debug 5:")
51              #print(secret)
52
53              # stores the remainder of the secret number divided by the prime number
54              # and the prime number itself as a pair in the shares variable
55              shares = [(secret % num, num) for num in seq]
56
57              #print("debug 6:")
58              #print(shares)
59
60              return secret, shares
```

Tl;dr after understanding everything in the create_key function, it is clear that the goal is to guess a single key portion and use the other nine given to figure out the secret key.

Ready To Start Coding?

Now that we have our objective of guessing the remaining key, we need to understand all of the possible values that seq can have so we know what to try. To do this, I created yet another copy of the script to start using their functions in a way that generates helpful output to crack the key.

So next I got rid of unnecessary prints, set the number of coelacanths to be 9, and made it easy to run without being bothered by unhelpful data and annoying prompts. At this point, I wanted to get every possible prime number out there. So I updated the code for seq in the script to generate 10,000 prime numbers and use the set() function (which I found out returns unique values) and it looked like this:

```python
def create_key(t, n, size=8):
    while True:
        #seq = sorted([number.getPrime(size) for _ in range(15)])
        seq = sorted([number.getPrime(size) for _ in range(10000)])
        print(sorted(set(seq)))
        quit()
        if len(set(seq)) != len(seq):
            continue
```

After running it a few times I discovered that I found all of the 24 possible values!

```
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py
[131L, 137L, 139L, 149L, 151L, 157L, 163L, 167L, 173L, 179L, 181L, 191L, 193L, 197L, 199L
, 211L, 223L, 227L, 229L, 233L, 239L, 241L, 251L, 257L]
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py
[131L, 137L, 139L, 149L, 151L, 157L, 163L, 167L, 173L, 179L, 181L, 191L, 193L, 197L, 199L
, 211L, 223L, 227L, 229L, 233L, 239L, 241L, 251L, 257L]
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py
[131L, 137L, 139L, 149L, 151L, 157L, 163L, 167L, 173L, 179L, 181L, 191L, 193L, 197L, 199L
, 211L, 223L, 227L, 229L, 233L, 239L, 241L, 251L, 257L]
kali@kali:~/Desktop$
```

With this information, I could try and guess the single prime value not provided by the program running on the server. So it was time to strip away basically all of the code except for the construct_key function and the prod function used by it. I then created a placeholder for the values given (keyPortions) that I planned on manually copying and pasting in to run my hacked version of the script when trying to get the flag. For the time being, I put in one of the values from a previous run so I could see the type of data and understand it better.

```python
#!/usr/bin/python
import random
from Crypto.Util import number
from functools import reduce

# substitute for math.prod
prod = lambda n: reduce(lambda x, y: x*y, n)

# all possible prime numbers
allPrimes = [131L, 137L, 139L, 149L, 151L, 157L, 163L, 167L, 173L, 179L, 181L, 191L, 193L, 197L, 199L, 211L, 223L, 227L, 229L, 233L, 239L, 241L, 251L, 257L]

keyPortions = [(172L, 239L), (12L, 197L), (1L, 233L), (123L, 227L), (13L, 131L), (37L, 163L), (15L, 251L), (87L, 211L), (22L, 151L)]

def construct_key(shares):
    glue = lambda A, n, s=1, t=0, N=0: (n < 2 and t % N or glue(n, A % n, t, s - A//n * t, N or n), -1)[n < 1]
    mod = prod([m for s, m in shares])
    secret = sum([s * glue(mod//m, m) * mod//m for s, m in shares]) % mod
    return secret
```

Next, I needed to write code to separate the prime numbers given from the possible prime numbers. To do that I extracted the primer numbers from the key portions. Next, I checked every

prime number and added it to the possiblePrimes if it was not in the givenPrimes. At this point, I thought about how I only needed to guess one of the possible prime numbers to get the secret code and I remembered I only have 250 attempts. The code for constructing the key needs to know the modulus (the remainder) which means I would have to guess the secret as many times as the value of the prime number. So if the prime number was 233, it would take 233 guesses!

So next I found the smallest prime number out of the possible ones to use that for guessing. I did this by sorting the list and selecting the first item. At this point, I was ready to start guessing possible secrets. Using the smallest prime number, I created a for loop for all possible modulus values of that prime number (0 to prime number - 1).

I then made a temporary copy of the givenPrimes using deepcopy (requires importing copy) so I didn't need to keep track of which ones were added that I would have to remove later. In the temporary copy, I added the guess and sent it to the construct_key function from the original script. I made sure to add data using the same structure as the key portions. I then printed out the possible secret keys returned from the function. That way I could copy and paste the guesses into the netcat session to hopefully get the flag later (if everything went according to plan that is).

```
21 if __name__ == "__main__":
22     possiblePrimes = []
23     givenPrimes = []
24 
25     for keyData in keyPortions:
26         # keyData is (modulus, prime number)
27         givenPrimes.append(keyData[1])
28 
29     for prime in allPrimes:
30         if prime not in givenPrimes:
31             possiblePrimes.append(prime)
32 
33     smallestPrime = sorted(possiblePrimes)[0]
34 
35     for modulus in range(smallestPrime):
36         guess = copy.deepcopy(keyPortions)
37         guess.append([modulus, smallestPrime])
38         print(construct_key(guess))
```

When I ran the tool, I was overwhelmed by the output. There were 137 different possible answers for the set of key portions I was given during one of my tests!

```
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py | head
6364641405413848628743
18271466135407811219915
30178290865401773811087
42085115595395736402259
53991940325389698993431
7640372626484630334940
19547197356478592926112
31454022086472555517284
43360846816466518108456
55267671546460480699628
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py | wc -l
137
kali@kali:~/Desktop$
```

My original plan called for copying and pasting these one at a time, but that was not going to do. So I found netcat code on StackOverflow and updated my script to use that instead. All I changed was returning the output instead of printing it.

Tl;dr after finding all 24 of the possible prime values, python code was written to generate all possible secrets based on the smallest prime number not provided in the key portions.

Optional: Automating the Flag Retrieval Process

At this point, I could tell that I was very close to getting the flag. If I wasn't so lazy, I could copy and paste in the guessed keys or, if I was mad, hand-jam them in until I got the correct secret for all five locks. Once I realized that could involve up to 1,250 guesses and I didn't think that was a good use of my time. So I needed to re-write my script to store the possible keys in a list and automate the input for the guessing. After a lot of trial and error (and arguably more time wasted than manually inputting each guess), I got it to work. Here is the code I created:

```
#!/usr/bin/python
import random
from Crypto.Util import number
from functools import reduce
import copy, socket, time, ast

# substitute for math.prod
prod = lambda n: reduce(lambda x, y: x*y, n)

# all possible prime numbers
allPrimes = [131L, 137L, 139L, 149L, 151L, 157L, 163L, 167L, 173L, 179L, 181L,
191L, 193L, 197L, 199L, 211L, 223L, 227L, 229L, 233L, 239L, 241L, 251L, 257L]
```

```
def construct_key(shares):
    glue = lambda A, n, s=1, t=0, N=0: (n < 2 and t % N or glue(n, A % n, t, s
- A//n * t, N or n), -1)[n < 1]
    mod = prod([m for s, m in shares])
    secret = sum([s * glue(mod//m, m) * mod//m for s, m in shares]) % mod
    return secret

def getSecrets(keyPortions):
    possiblePrimes = []
    givenPrimes = []

    for keyData in keyPortions:
        # keyData is (modulus, prime number)
        givenPrimes.append(keyData[1])

    for prime in allPrimes:
        if prime not in givenPrimes:
            possiblePrimes.append(prime)

    smallestPrime = sorted(possiblePrimes)[0]

    secrets = []

    for modulus in range(smallestPrime):
        guess = copy.deepcopy(keyPortions)
        guess.append([modulus, smallestPrime])
        secrets.append(construct_key(guess))

    return secrets

if __name__ == "__main__":
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("chal.uiuc.tf", 2004))

    # sleep a second before sending the value 9
    time.sleep(1)
    s.sendall("9\n".encode())
```

```
data = s.recv(4096)
# the 9 in the print is to display it where it would have been typed
print(data + "9")
time.sleep(0.5)

# get the next set of data (which contains the key portions)
data = s.recv(4096)
print(data)
returnData = data.split("\n")

keyPortionString = ""

# get the specific string with the key portions
for index, data in enumerate(returnData):
    if data == "Here are your key portions:":
        keyPortionString = returnData[index + 1]
        break

# this magic function converts the string into a list of tuples!
keyPortions = ast.literal_eval(keyPortionString)

# go through all 5 locks to open them
for i in range(5):
    # using the keyPortions, generate the possible secrets
    secrets = getSecrets(keyPortions)

    # reset returnData
    returnData = []

    for secret in secrets:
        # send each secret to the server
        s.sendall(str(secret) + "\n")
        print(secret)

        # get the returned data
        data = s.recv(4096)
        print(data)
        returnData = data.split("\n")
```

```
            # if a key worked, it will either print the next key portions or
the flag
            if "Here are your key portions:" in returnData or "Opening
vault..." in returnData:
                # exit out the secrets loop, these secrets are for the wrong
key portions
                break

        # check to make sure this isn't the end, if so continue
        if "Opening vault..." not in returnData:
            for index, data in enumerate(returnData):
                if data == "Here are your key portions:":
                    keyPortionString = returnData[index + 1]
                    break

            # once again, update the keyPortions with the new ones then repeat
the loop
            keyPortions = ast.literal_eval(keyPortionString)

    # once done, close the socket and connection
    s.shutdown(socket.SHUT_WR)
    s.close()
```

On the next page, there are snippets of the output of the python script. It shows the expected output when it successfully gets the flag.

```
kali@kali:~/Desktop$ ./coelacanth_vault_hack.py
Hi, and welcome to the virtual Animal Crossing: New Horizon coelacanth vault!
There are 5 different cryptolocks that must be unlocked in order to open the vault.
You get one portion of each code for each coelacanth you have caught, and will need to us
e them to reconstruct the key for each lock.
Unfortunately, it appears you have not caught enough coelacanths, and thus will need to f
ind another way into the vault.
Be warned, these locks have protection against brute force; too many wrong attempts and y
ou will have to start over!

How many coelacanth have you caught? 9
Generating key for lock 0, please wait...
Generated!
Here are your key portions:
[(29, 151), (157, 223), (159, 229), (51, 149), (192, 193), (33, 139), (109, 173), (232, 2
57), (6, 163)]
Please input the key:
27360618652413197387632
Key incorrect. You have 250 tries remaining for this lock.
Please input the key:
4575952320264727798306
Key incorrect. You have 249 tries remaining for this lock.
Please input the key:
11053945689208900524683
Key incorrect. You have 248 tries remaining for this lock.
Please input the key:
17531939058153073251060
Key incorrect. You have 247 tries remaining for this lock.
Please input the key:
24009932427097245977437
Key incorrect. You have 246 tries remaining for this lock.
Please input the key:
1225266094948776388111
Key incorrect. You have 245 tries remaining for this lock.
Please input the key:
7703259463892949114488
Key incorrect. You have 244 tries remaining for this lock.
Please input the key:
14181252832837121840865
```

```
Key incorrect. You have 182 tries remaining for this lock.
Please input the key:
6139605892078838456397
Lock 0 unlocked with 69 failed attempts!
Generating key for lock 1, please wait...
Generated!
Here are your key portions:
[(14, 191), (29, 223), (145, 257), (133, 211), (15, 179), (91, 149), (79, 139), (160, 229
), (75, 239)]
Please input the key:
32879901105273200955014
Key incorrect. You have 250 tries remaining for this lock.
Please input the key:
59123858187399404203038
Key incorrect. You have 249 tries remaining for this lock.
Please input the key:
23975701380980381995863
Key incorrect. You have 248 tries remaining for this lock.
Please input the key:
502196584631065852438887
Key incorrect. You have 247 tries remaining for this lock.
Please input the key:
15071501656687563036712
Key incorrect. You have 246 tries remaining for this lock.
```

```
Key incorrect. You have 195 tries remaining for this lock.
Please input the key:
7371195390689873094213
Key incorrect. You have 194 tries remaining for this lock.
Please input the key:
17238898738308299261797
Key incorrect. You have 193 tries remaining for this lock.
Please input the key:
27106602085926725429381
Key incorrect. You have 192 tries remaining for this lock.
Please input the key:
3697430543354515l596965
Key incorrect. You have 191 tries remaining for this lock.
Please input the key:
6446098201850645641002
Lock 4 unlocked with 60 failed attempts!
Opening vault...
Looks like the vault has already been emptied :( however, you can have this flag instead:
 uiuctf{small_oysters_expire_quick}
```