HW3
Stephen Graham
5/1/2024

Problem 1:

Was unable to guess the secret pin. Retrying the same bit over and over often get indeterminate results as the bit that takes the longest often flips back and forth each time it is tested. Trying to find the average out of the sum of several measurements surpasses the 10 minute deadline of StrangeSystem shut down. And only having it retry bits when the time differences are too close together will either give me inaccurate info if I set the time difference too low, or it will retry too many times and surpass 10 minutes if I set the difference threshold too high. Here is my code used:

```python
from pwn import *
import time

command = '/home/seed/Documents/HW3/StrangeSystem'
wsu_id = "11706998"

proc = process(command)


def login_with_id():
    proc.sendline(wsu_id.encode())
    try:
        response = proc.recvuntil(b"Enter your 32-bits PIN code (in binary form like PIN in pincode.log):")
        print(response.decode())
    except EOFError:
        print("You broke it again, dummy.")
        proc.close()
        return False
    return True

def measure_response_time(pin_guess):
    proc.sendline(pin_guess.encode())
    start_time = time.time()
    proc.recvuntil(b"[***] ERROR: Verification failed!")
    end_time = time.time()
    proc.sendline(b"yes")
    return end_time - start_time

login_with_id()
pin = ''
for i in range(32):
    retries = 0
    max_retries = 3
    time_zero = 0
    time_one = 0

    while (retries < max_retries):
        zero_pin = pin + '0' + '0'*(31-i)
        one_pin = pin + '1' + '0'*(31-i)
        time_zero = measure_response_time(zero_pin)
        time_one = measure_response_time(one_pin)
        time_diff = abs(time_zero - time_one)
        print(f"Bit {i+1}, Time for 0: {time_zero:.4f}, Time for 1:
{time_one:.4f}")
        if (time_diff <= 0.05):
            print("Time difference too small, retrying...")
            retries += 1
        else:
            break

    if (time_zero > time_one):
        pin += '0'
    else:
        pin += '1'

    print(f"Probable Bit {i+1}: {pin[-1]}")

if len(pin) == 32:
    print("THE PIN:", pin)
    with open("/home/seed/Documents/HW3/pincode.log", "a") as f:
        f.write(f"{pin}\n")
    proc.interactive()
```

Problem 2:

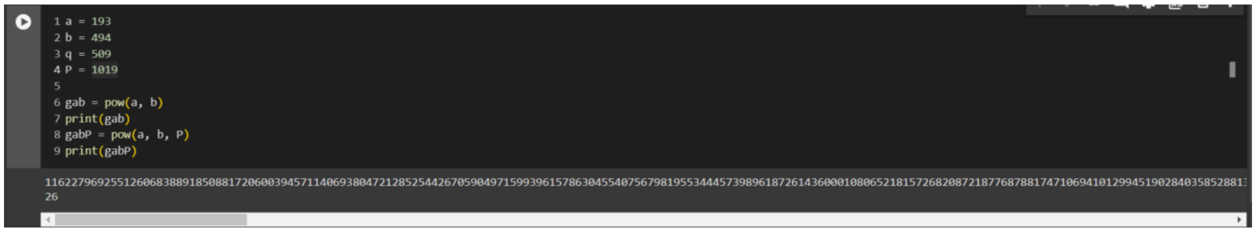
WSU ID: 11706998

g_a = 193

g_b = 494

g_ab = a really big number

g_ab mod 1019 = 26



```
a = 193
b = 494
q = 509
P = 1019

gab = pow(a, b)
print(gab)
gabP = pow(a, b, P)
print(gabP)
```

```
1162279692551260683889185088172060039457114069380472128525442670590497159939615786304554075679819553444573989618726143600010806521815726820872187768788174710694101299451902840358528 81
26
```

Problem 3:

1a.
Keygen(): generates randomized key ‘k’ of a fixed length.
Enc(m, k): Encrypts message ‘m’ to cyphertext ‘c’ by apply a xor ($\oplus$) operation to the message with the key, ie ($c = m \oplus k$),
Dec(c, k): decrypts the cipher text by simply reversing the encryption method, eg ($m = c \oplus k$)

1b.
An adversary can create a ciphertext that appears to be valid without needing the key by requesting a message (m) to be encrypted and he receives back the ciphertext of the message (c). Using c from the message m, a new message (m’) can be given a new valid cypher (c’) without having to make a request and thus not needing to authenticate the origin by reversing the xor operations on the original cipher and messages with the new message the adversary wants to encrypt, eg: ($c' = c \oplus m \oplus m'$).

2a.
Keygen(): generates key (k) of fixed length.
Tag(m, k): generates a tag (c) that is used to authenticate a message (m) using the key (k) using a hash function on the two inputs.
Verify(m, k, c): essentially creates a new tag by hashing m and k, and checking if it matches the input tag.

2b.
Sinch the hash function is not random, an adversary can predict a likely result if they have any knowledge of the message, such as the length, or any predictable parts of the message, like a date stamp, header, etc.

Problem 4:

1.
Yes.

Because the

2.

I’m not sure I understand this one, but couldn’t you just select a single bit at a fixed index of the PRF 128bit output? E.g. always output bit 42? If PRF is truly random (or pseudorandom), then any bit within PRF will also be random.