Authors: Avery Hall, Brian Rhee, Robbie Young (Joven)

Diffie-Hellman

Shared secret: 36

Read code comments for explanation of each step in our process

Users > ahall > Desktop > diffie-hellman.py > ...

```python
1  for x in range(0,59): # The range for x is limited to 59 because x must be less than p
2      if ((11**x) % 59 == 57): # Here, we are testing each value in the range to see which values will return the value of A that we intercepted
3          print("x: " + str(x)) # Once we find the value that returns A, we know that we have found Alice's secret number
4
5  for y in range(0,59):
6      if ((11**y) % 59 == 44):
7          print("y: " + str(y))
8
9  # Just to make sure our process was correct, we checked that the 3 values that should be
10 # equal given proper secret values of x and y actually were equal.
11
12 print("K(x): " + str(44**36 % 59))
13 print("K(y): " + str(57**15 % 59))
14 print("K(xy): " + str(11**(15*36) % 59))
15
```

If the numbers were much larger (both Alice and Bob choose a value of p and values for x and y that are much larger than 59), the code would fail on lines 2 and 6. This is because computers take much longer to calculate the modulo operator, particularly with larger numbers. Even when we tried raising the p, x, and y values to numbers near the 10,000 range, we noticed a substantial increase in computational time.

RSA

Shared secret: Hey Bob. It's even worse than we thought! Your pal, Alice.
https://www.schneier.com/blog/archives/2022/04/airtags-are-used-for-stalking-far-more-than-previously-reported.html

The below is used to calculate the p and the q which multiplied make up n. This is where our process would fail if the keys involved were much bigger. Although there most probably exist faster implementations than the one below, there is still to some extent a non-linear relationship, meaning that any implementation is going to be slower than O(n). In our implementation this would be in the ranges, (and as our runtime is O(n^2)), our code would slow down substantially here, and with big enough numbers essentially make this process unfeasible. However, it would still be possible to brute force this, meaning that no matter how big the numbers are this method is not fully secure. However after a certain point, it would take too much time to test all possible numbers.

```python
def is_prime(number): # checks to see if the inputed number is prime
    for i in range(2, int(number / 2) + 1):
        if (number % i) == 0:
            return False
    return True

def find_pq(n): # since n is chosen to be the multiplication of two prime numbers (p and q), this method loops through factoring n until finding the two prime nubmers
    for i in range(100): # the two ranges are of 100 based on the fact that n (5561) is less than 100 * 100. Theoretically this could not find the values of p and q, such as with n being 7063 (p = 7, q = 1009), but the ranges could just be increased
        for j in range(100):
            if is_prime(i) & is_prime(j) & ((i * j) == n): # if both numbers are prime, and if the multiplication of both are equal to the value of n
                print("Values of p and q: ", i, j)
```

Below is the implementation of the modular inverse function:

```python
# The below was taken from a stackoverflow post: https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function-in-python
# and equates to finding d in the "e * d = 1 mod (p - 1) (q - 1)"

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b //a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('DNE')
    else:
        return x % m
```

Once the initial encrypted numbers are decrypted, they are still encoded. Note the method “chr” is used, which just converts the ultimately decoded character into a readable character (ASCII number to character). For example, the first character reads as ‘48’ after encryption and before decoding, and 48 in ASCII translates to capital H.

Independent of RSA, the encoding of this message by itself is not at all secure because ASCII is a simple and well-known encoding that can be instantly translated with no cracking necessary. Below are the values of the encoded message, as well as each calculated/given value needed in this process. Additionally, the decryption and decoding of each character is included, which is then printed out.

```python
encrypted_message = [
    1516, 3860, 2891, 570, 3483, 4022, 3437, 299,
    570, 843, 3433, 5450, 653, 570, 3860, 482,
    3860, 4851, 570, 2187, 4022, 3075, 653, 3860,
    570, 3433, 1511, 2442, 4851, 570, 2187, 3860,
    570, 3433, 1511, 4022, 3411, 5139, 1511, 3433,
    4180, 570, 4169, 4022, 3411, 3075, 570, 3000,
    2442, 2458, 4759, 570, 2863, 2458, 3455, 1106,
    3860, 299, 570, 1511, 3433, 3433, 3000, 653,
    3269, 4951, 4951, 2187, 2187, 2187, 299, 653,
    1106, 1511, 4851, 3860, 3455, 3860, 3075, 299,
    1106, 4022, 3194, 4951, 3437, 2458, 4022, 5139,
    4951, 2442, 3075, 1106, 1511, 3455, 482, 3860,
    653, 4951, 2875, 3668, 2875, 2875, 4951, 3668,
    4063, 4951, 2442, 3455, 3075, 3433, 2442, 5139,
    653, 5077, 2442, 3075, 3860, 5077, 3411, 653,
    3860, 1165, 5077, 2713, 4022, 3075, 5077, 653,
    3433, 2442, 2458, 3409, 3455, 4851, 5139, 5077,
    2713, 2442, 3075, 5077, 3194, 4022, 3075, 3860,
    5077, 3433, 1511, 2442, 4851, 5077, 3000, 3075,
    3860, 482, 3455, 4022, 3411, 653, 2458, 2891,
    5077, 3075, 3860, 3000, 4022, 3075, 3433, 3860,
    1165, 299, 1511, 3433, 3194, 2458
]
n = 5561
e = 13
p = 83
q = 67
d = 1249

message = ''
for encrypted_char in encrypted_message: # loops through each individual character in the encoded message
    message += chr((encrypted_char**(d)) % n) # concatenates each decoded character to the message string. Each character is decoded by exponentiating each block of code
    the power of the public key and then to the power of the private key mod n; in other words, we use the following formula: M = M^(ed) mod n. Therefore, since we have
    intercepted the message after being encoded with M^(Alice's key), by encoding this already encoded message with Bob's private key, we are able to find the original
    message.

print(message)
```

Note: The comment in the code above describes the message of *decrypting, not decoding as it says.

Below is the result in the terminal:

```
robert@Roberto-Jovens-MacBook-Air cs338 % python3 temp.py
Hey Bob. It's even worse than we thought! Your pal, Alice. https://www.schneier.com/blog/archives/2022/04/airtags-are-used-for-stalking-far-more-than-previously-reported.html
robert@Roberto-Jovens-MacBook-Air cs338 %
```