Varun Saini

Being Eve

Diffie–Hellman

Python Script

```python
# --- Diffie–Hellman --- #

# Variable initialization
g, p = 7, 61
A, B = 30, 17

# Finding a and b using a brute-force method
# We check p values since the order of g in Z/pZ is p
for i in range(p):
    num = (g ** i) % p
    if num == A:
        a = i
    if num == B:
        b = i

K_a = (B ** a) % p
K_b = (A ** b) % p

# Results
print("a:", a)  # 41
print("b:", b)  # 23
print("Alice's shared secret:", K_a)   # 6
print("Bob's shared secret:", K_b)     # 6
```

I used the Python script above to calculate the values of *a* and *b*, which I then used to calculate *K_a* and *K_b*, which are the shared keys from Alice's and Bob's perspectives, respectively.

As we can see, we can use a simple brute-force method to find *a* and *b*. Some algebraic reasoning lets us conclude that we only need to check at most *p* values, so we check *p* consecutive values of *i* to see if *i* was the random *a*/*b* used by Alice/Bob to generate *A*/*B*.

The Python script is my work for this problem. If the integers involved were much larger (specifically *p*, as the other numbers—*g*, *A*, *B*—can be represented with a congruent number (modulo *p*) that is at most *p*), the `for` loop in the Python script above would take significantly longer to run. If *p* is enormous, it might simply take too long to do the calculations shown above.

Side note: calculating quantities of the form `(x ** y) % z` wouldn't significantly slow this down. Due to some interesting properties of modular arithmetic, we can keep this computation relatively fast, even if *y* is large.

Varun Saini

RSA

Python Script

```python
# --- RSA --- #

e_Bob, n_Bob = (13, 5561)

encrypted = [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]

# https://factorization.info/prime-factors/0/prime-factors-of-5561.html
p, q = 67, 83

# Find d_Bob such that e_Bob * d_Bob = 1 (mod (p - 1)(q - 1))
mod = (p - 1) * (q - 1)

for i in range(mod):
    if (e_Bob * i) % mod == 1:
        d_Bob = i
        break

print("d_Bob:", d_Bob) # 1249

# Decrypt the encrypted message
decrypted = []
for e in encrypted:
    d = (e ** d_Bob) % n_Bob
    decrypted.append(d)

# Decode the decrypted message
decoded = []
for d in decrypted:
    decoded.append(chr(d))

# Print message
decoded_msg = "".join(decoded)
print(decoded_msg)

# 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
```

Varun Saini

The encrypted message sent from Alice to Bob was the following:

- 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 Python script above demonstrates how I found this.

- First, I used an online resource to find *p* and *q*, the prime factors of *n_Bob* that were used to generate Bob's secret key and public key. This alone could be very difficult for large values of *n_Bob*, as factoring can be fairly computationally intensive.
- Then, I used a brute-force algorithm to find *d_Bob*. I knew that *d_Bob* * *e_Bob* had to equal 1 (mod (*p* - 1)(*q* - 1)), so I checked all values of d_Bob from 0 to (*p* - 1)(*q* - 1) - 1. This step could also be extremely time-intensive for large values of *p* and *q*.
- Once I found *d_Bob* (1249), I had Bob's secret key. I used this to decrypt the encrypted buffer character-by-character.
- I noticed that every value in the decrypted buffer was less than 128, so I guessed that this was encoded using ASCII. To decode this, I simply used Python's *chr* function.
- Then, I printed the decoded buffer to get the original message.

To summarize, the most time-intensive steps if we had larger integers would be factoring *n_Bob* into primes *p*, *q* and using a brute-force algorithm to find *d_Bob*.

Even if Bob's keys involved larger integers, this encryption would still be insecure. By encrypting the message character-by-character, Alice has basically created a complicated-looking substitution cipher. Analyzing character frequencies could allow Eve to crack the message.

For example, this sequence could be cracked somewhat easily, as it "looks" like the beginning of a URL, even when we can only see character frequencies. (Repeated characters underlined.)

- 1511, <u>3433, 3433</u>, 3000, 653, 3269, <u>4951, 4951</u>, <u>2187, 2187, 2187</u>, 299
- Actual text: `https://www.`