Being Eve

Ankit Sanghi, Eric Neidhart, Ginnie White

Diffie Hellman:

Alice and Bob agree on g = 17 and p = 61.
Alice sent Bob the number 46 (A).
Bob sent Alice the number 5 (B).

We know that in Diffie Hellman, A is calculated using A = g^a mod p and B is calculated using B = g^b mod p. So we need to calculate a and b from A and B. This can be done by trying all the different values for a and seeing which one works for this equation. So trying all possible combinations until we see the number 46, we get the following output:

tried 0 but failed. Value of A is: 1
tried 1 but failed. Value of A is: 17
tried 2 but failed. Value of A is: 45
tried 3 but failed. Value of A is: 33
tried 4 but failed. Value of A is: 12
tried 5 but failed. Value of A is: 21
tried 6 but failed. Value of A is: 52
tried 7 but failed. Value of A is: 30
tried 8 but failed. Value of A is: 22
tried 9 but failed. Value of A is: 8
tried 10 but failed. Value of A is: 14
tried 11 but failed. Value of A is: 55
tried 12 but failed. Value of A is: 20
tried 13 but failed. Value of A is: 35
the answer is: 14 this means A is: 46

As a result, we know that the answer for a is: 14, as 17^14 mod 61 = 46.

After we have found the value of a, we can use that combined with Bob's public message to Alice (5), calculating 5^14 mod 61 to get a value of 12 for their shared 'secret' key. This calculation is the same way that Alice would get the value of their key, but the only reason that we can do this is that we used brute force to get her secret value a.

We would have failed in the sequential attack on A searching for the value of a if we had needed to work with larger integers. We would have had a much larger space that we

would have had to sequentially search, and we were essentially trying one number at a time and going up the list. As a result, this attack would not scale well and wouldn't be very efficient if Alice and Bob had chosen larger integers.

RSA:

Here's Bob's public key:
(e_Bob, n_Bob) = (31, 4661)
Here's the encrypted data sent from Alice to Bob:
[2677, 4254, 1152, 4645, 4227, 1583, 2252, 426, 3492, 4227, 3889, 1789, 4254, 1704, 1301, 4227, 1420, 1789, 1821, 1466, 4227, 2252, 3303, 1420, 2234, 4227, 4227, 1789, 1420, 1420, 4402, 1466, 4070, 3278, 3278, 414, 414, 414, 2234, 1466, 1704, 1789, 2955, 4254, 1821, 4254, 4645, 2234, 1704, 2252, 3282, 3278, 426, 2991, 2252, 1604, 3278, 1152, 4645, 1704, 1789, 1821, 4484, 4254, 1466, 3278, 1512, 3602, 1221, 1872, 3278, 1221, 1512, 3278, 4254, 1435, 3282, 1152, 1821, 2991, 1945, 1420, 4645, 1152, 1704, 1301, 1821, 2955, 1604, 1945, 1221, 2234, 1789, 1420, 3282, 2991, 4227, 4410, 1821, 1301, 4254, 1466, 3454, 4227, 4410, 2252, 3303, 4645, 4227, 3815, 4645, 1821, 4254, 2955, 2566, 3492, 4227, 3563, 2991, 1821, 1704, 4254]

First, in order to crack the message efficiently, we need to determine what the factors are for n_Bob, which are p = 59 and q = 79. These values are used to calculate the e_Bob and d_Bob values using the formula e_Bob * d_Bob mod(p-1)(q-1) = 1. (p-1)(q-1) happens to be 4524 in this case. From here, we will brute force the value of d_Bob, since we already have the value of e_Bob. To do this, We looked at multiples of 4525 and tried to figure out which ones could be factored by 31. 2335 satisfies this condition, as 31*2335 mod 4524 = 1. So, now we can decrypt the message as Bob would, using 2335 as the value of d_Bob. This means that we raise each character to that power and then use mod n_Bob.

We have the following decrypted numbers after we do that:
[874, 1868, 3487, 2309, 1895, 3888, 467, 2229, 1211, 1895, 1582, 4066, 1868, 4023, 3082, 1895, 3219, 4066, 3586, 3237, 1895, 467, 1528, 3219, 421, 1895, 1895, 4066, 3219, 3219, 1528, 3237, 1785, 1489, 1489, 2177, 2177, 2177, 421, 3237, 4023, 4066, 660, 1868, 3586, 1868, 2309, 421, 4023, 467, 1485, 1489, 2229, 688, 467, 3931, 1489, 3487, 2309, 4023, 4066, 3586, 1638, 1868, 3237, 1489, 3319, 1805, 3546, 3663, 1489, 3546, 3319, 1489, 1868, 3204, 1485, 3487, 3586, 688, 3718, 3219, 2309, 3487, 4023, 3082, 3586, 660, 3931, 3718, 3546, 421, 4066, 3219, 1485, 688, 1895, 1520, 3586, 3082, 1868, 3237, 1121, 1895, 1520, 467, 1528, 2309, 1895, 2040, 2309, 3586, 1868, 660, 793, 1211, 1895, 1268, 688, 3586, 4023, 1868]

When we look at These numbers translate to:
"Dear Bob, Check this out.
https://www.schneier.com/blog/archives/2017/12/e-mail_tracking_1.html Yikes! Your friend, Alice"

The method we used is not time efficient for bigger numbers. We essentially used brute force to find both the multiples of 4661 in order to get p and q. Later on, in order to get d_bob, we had to find all the multiples of 4525 then factor them to check which could be factored by 31. Factoring is a time intensive operation, which only becomes worse as the integers involved become larger.

Below is the python code we used to automate our testing:

```
def diffie_hellman():
  A = 46
  g = 17
  p = 61
  for i in range(100):
    if A == pow(g, i) % p:
      print('the answer is: ', i, ' this means A is: ', pow(g, i) % p)
    else:
      print('tried ', i, 'but failed. Value of A is: ', pow(g, i) % p)

def rsa_cracking():
  e_bob, n_bob = 31, 4661
  d_bob = 0
  ciphertext = [2677, 4254, 1152, 4645, 4227, 1583, 2252, 426, 3492, 4227, 3889,
1789, 4254, 1704, 1301, 4227, 1420, 1789, 1821, 1466, 4227, 2252, 3303, 1420, 2234,
4227, 4227, 1789, 1420, 1420, 4402, 1466, 4070, 3278, 3278, 414, 414, 414, 2234,
1466, 1704, 1789, 2955, 4254, 1821, 4254, 4645, 2234, 1704, 2252, 3282, 3278, 426,
2991, 2252, 1604, 3278, 1152, 4645, 1704,
            1789, 1821, 4484, 4254, 1466, 3278, 1512, 3602, 1221, 1872, 3278, 1221,
1512, 3278, 4254, 1435, 3282, 1152, 1821, 2991, 1945, 1420, 4645, 1152, 1704, 1301,
1821, 2955, 1604, 1945, 1221, 2234, 1789, 1420, 3282, 2991, 4227, 4410, 1821, 1301,
4254, 1466, 3454, 4227, 4410, 2252, 3303, 4645, 4227, 3815, 4645, 1821, 4254, 2955,
2566, 3492, 4227, 3563, 2991, 1821, 1704, 4254]
  for d in range(10000):
```

```
    if (e_bob * d) % 4524 == 1:
        print('Yay the value of d is: ', d)
        d_bob = d
        break

if d_bob != 0:
    plaintext = []
    for i in ciphertext:
        plain = pow(i, d_bob) % n_bob
        plaintext.append(chr(plain))
    print("".join(plaintext))
```