CS 231: Being Eve

Peter McCrea

April 14, 2021

1 Diffie Hellman:

Consider the following code to brute-force a shared secret for a Diffie Hellman interaction between Alice and Bob:

diffieHellmanBruteForce.py

```
def diffieHellmanBruteForce():
    #All of the defined variables were given in the problem
    #All variables declared but not defined are unknown at the start
    g = 17
    p = 61
    A = 46
    B = 5
    X = None
    Y = None
    sharedSecret = None

    #Brute force Alice and Bob's secret numbers:
    for x in range(p):
        if (g ** x) % (p) == A:
            X = x
    for y in range(p):
        if (g ** y) % (p) == B:
            Y = y

    #If the two shared secrets do not match,
    #something has gone horribly wrong:
    if (((B**X) % p) ==  ((A**Y) % p)):
        sharedSecret = ((B**X) % p)
    else:
        sharedSecret = "Oops, something is very wrong"
    print("The shared secret is: " + str(sharedSecret))
    print("Alice's secret number is: " + str(X))
    print("Bob's secret number is: " + str(Y))

diffieHellmanBruteForce()
```

Output:

```
The shared secret is: 12
Alice's secret number is: 14
Bob's secret number is: 26
```

1.1 Figure out the shared secret agreed upon by Alice and Bob. This will be an integer.

The shared secret is: **12**

1.2 Show your work. Exactly how did you figure out the shared secret?

(See code on previous page) To find the shared secret, you first consider this information which is given in the problem:

Alice sent Bob the number 46.
Bob sent Alice the number 5.

Because we have the numbers Alice and Bob exchanged, we can work backwards. We know that Alice generated her number such that it is equal to $g^X \, mod \, p$ and Bob generated his number such that it is equal to $g^Y \, mod \, p$. For a small value of p (remember that both X and Y are less than p) it is trivial to check all possible values that could satisfy these equations.

Once we have the values of X and Y from the process above, the shared secret is easily calculated by solving either $B^X \, mod \, p$ or $A^Y \, mod \, p$.

These steps are exactly what is done in the python code shown above.

1.3 Show precisely where in your process you would have failed if the integers involved were much larger.

For a sufficiently large p, it would be infeasible to check all possible values of X and Y such that $g^X \, mod \, p = A$ and $g^Y \, mod \, p = B$. Specifically, calculating lots of large exponents and moduli is not fast. In fact, this process would be so slow that it makes up the very reason why Diffie–Hellman key exchange works in the first place.

2 RSA

Consider the following code to reverse RSA:

rsaBruteForce.py

```
import sympy
def rsaBruteForce():
    Dbob = None
    Ebob = 31
    Nbob = 4661
    # Factor Nbob into Pbob and Qbob
    primeFactors = sympy.ntheory.factorint(Nbob)
    Pbob, Qbob = list(primeFactors)[0], list(primeFactors)[1]
    i = 0
    #Brute force the value of Dbob
    while (Dbob is None):
        checkValues = (Ebob * i) % ((Pbob-1) * (Qbob-1))
        if checkValues == 1:
            Dbob = i
        i = i + 1
    ciphertext = [2677, 4254, 1152, 4645, 4227, 1583, 2252, 426, ...]
    plaintext = []
    #Using Bob's private key, go from ciphertext to regular english
    for i in ciphertext:
        newVal = (i**Dbob)%(Nbob)
        plaintext.append(newVal)
    ASCII_string = "".join([chr(value) for value in plaintext])
    print(ASCII_string)

rsaBruteForce()
```

(I added some newlines in this output so it wouldn't go off the page.)

Output:

```
Dear Bob, Check this out.
https://www.schneier.com/blog/archives/2017/12/e-mail_tracking_1.html
Yikes! Your friend, Alice
```

2.1 Figure out the encrypted message sent from Alice to Bob.

Dear Bob, Check this out. `https://www.schneier.com/blog/archives/2017/12/e-mail_tracking_1.html` Yikes! Your friend, Alice

2.2 Show your work. Exactly how did you figure out the message?

Starting off, we know that Bob's public key is $(e_B, n_B) = (31, 4661)$. The very first thing we can do is factor n_B to get p_B and q_B. In our case we factor 4661 to get 59 and 79. After this is completed, we must brute force values of d_B such that $e_B\, d_B\, mod (p_B - 1)(q_B - 1) = 1$. After this is completed, we have all the necessary information to convert the cipher-text Alice sent to Bob into regular ASCII codes. Specifically, we take the text Alice sent to Bob and for each character x apply the function $x^{d_B} mod\, n_B$. Finally, we convert these ASCII character codes to a string, giving us our result.

2.3 Show precisely where in your process you would have failed if the integers involved were much larger.

Given large enough numbers, prime factorization will take too much time to be feasible (exponential run time). In this example, that means we would not be able to go from n_B to p_B and q_B. Additionally, for my specific implementation, brute forcing values of d_B is inefficient because the modulus operation is slow for large numbers $O(n^2)$.