File: cryptography/being-eve.pdf
File owner: Lysander Miller
Collaborated with: Kiri Salij – asked her a question about whether or not she got a specific error with her code. More details are in the document.

Being Eve

Note:

In class on 9/25, we realized that, at some point before 9/23, Jeff had uploaded a newer version of this assignment on his website. However, when I did this assignment on the evening of 9/23, like a couple others in the class, I found an older version of the assignment on Jeff's website.

As of class on 9/25, Jeff decided that whatever version we had worked on was alright and he wouldn't make us redo our work if we had done the older version of this assignment.

Diffie-Hellman:

The shared secret between Alice and Bob is the integer 6. I figured this out by following these steps:

1. I took the fact that g = 7, p = 61, A = 30, and B = 17 and proceeded to plug x = 0, x = 1, x = 2, x = 3… etc into

   $7^x \bmod 61$

   Until the above outputted either 30 (in which case I knew that a was equal to x's current value) or 17 (in which case I knew that b was equal to x's current value).

2. With the above step, I figured out that a = 41 and b = 23. From there, I computed $B^a \bmod p$ and $A^b \bmod p$. Both were equal to 6 and thus, I was confident that I had found the shared secret.

If the integers involved were much larger, I wouldn't have been able to efficiently try different x's (from 0 up) until I got my a and b.

RSA:

This is the encrypted message sent from Alice to Bob:

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

I was able to figure this out by following these steps:

1. I first wrote and ran this Java code:

```
public class findPQD
{
    /*
    * In this function, I have hardcoded in n and e.
    * The function finds two primes p and q that multiply together
to form n.
    * Then, it finds a d such that (e * d) % ((p - 1)*(q - 1)) == 1.
    * Finally, it prints out p, q, and d.
    */
       public static void main (String[] args)
       {
            int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}; //While I
could create a function to generate prime numbers as needed, I
figured that the integers involved in creating this key weren't very
large. Thus, I could start with a list of the first 25 prime numbers
and, if need be, add more prime numbers to the list or write a
function to generate prime numbers as needed.
            int n = 5561;
            int possibleP = 0;
            int possibleQ = 0;
            int p = 0;
            int q = 0;
            for (int i = (primes.length -1); i > 0; i--) //Started
from the end of the list of primes because n is pretty large and
thus, I figured it was likely that n is the multiple of two larger
primes (larger primes compared to 2, 3, 5, etc.). While starting
from the end of the list of primes is not necessary, it is a bit
more time-efficient.
            {
                possibleP = primes[i];
```

```
                for (int k = i; k > 0; k--) //Started from the end
of the list of primes because n is pretty large and thus, I figured
it was likely that n is the multiple of two larger primes (larger
primes compared to 2, 3, 5, etc.). While starting from the end of
the list of primes is not necessary, it is a bit more
time-efficient.
                {
                    possibleQ = primes[k];
                    if (possibleP * possibleQ == n)
                    {
                        p = possibleP;
                        q = possibleQ;
                        break;
                    }
                }
                if (p != 0) //Not necessary but makes the function a
bit more time efficient.
                {
                    break;
                }
            }
            int e = 13;
            int d = 0;
            while ((e * d) % ((p - 1)*(q - 1)) != 1)
            {
                d = d+1;
            }

            System.out.println("p = " + p + " q = " + q + " d = " +
d);

        }

}
```

2. After step one, I found myself with p = 83, q = 67, and d = 1249. Then, when I tried to use the Java library Math.pow to raise every integer in the ciphertext to the power of d (1249), Math.pow returned “Infinity.” I figured this was related to types and sizing (as something like 3860^{1249} is an incredibly large number) so I

asked my friend Kiri Salij if she had encountered a similar issue. She said she had not encountered this issue but she was using Python. Thus, she suggested that switching to Python might fix my problem. Since I never took CS111, I'm not very good at Python. Additionally, I had already found p, q, and d. Thus, I figured that, while it's not the most elegant solution, I would keep the Java code I had written and write a short program in Python to convert the ciphertext to plaintext.

Here is the Python code I wrote and subsequently ran:

```
#I have hardcoded d, n, and the ciphertext into the following
function.
#The function decrypts the ciphertext and prints out the plain
text.
def main():
    d = 1249
    n = 5561
    cipherText = [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]
```

```
    plainText = ""

    for c in cipherText:
        plainText += chr(pow(c, d) % n)

    print(plainText)

if __name__ == "__main__":
    main()
```

This Python code worked and printed out the message from Alice to Bob.

In order to encode this message, Alice would have had to go through every character in the message and convert the character to the ASCII decimal representation. Then, she'd raise each of those decimal representations to the power of e (e would be taken from Bob's public key) before modding it by n (n would also be taken from Bob's public key). How do we know she went through every character in the message, though? Well, we can notice that there are 174 integers in the ciphertext and 174 characters in the plaintext string. Additionally, if we look at the integers in the ciphertext, we can see that 570 appears at indexes 3 and 8 (among other appearances at other indexes) and, in the plaintext, spaces appear at indexes 3 and 8 (among other appearances at other indexes).

Be sure to note that, if the integers involved were much larger, I would have been unable to calculate p and q. After all, it would have been incredibly inefficient to test so many different combinations of prime numbers. Similarly, if d was much larger, calculating it would also be incredibly inefficient; after all, in my code, I just tested random d's until one worked.

However, even if Bob's keys involved larger integers, the message encoding Alice used would still be insecure. After all, as we mentioned earlier, Alice encoded each individual character in her message rather than encoding the message itself. This led to the repetition of characters in the ciphertext. Because some characters, such as vowels or spaces, are used more often than other characters, when Eve sees the repetition of an integer in the ciphertext, she might think that the repeated integer is an encrypted vowel or space. This gives Eve too much information and, thus, the message is insecure.