Arthur

The numbers g and p are sent via an unprotected network. So Eve is able to eavesdrop on them, reading them as plain text. In this case, Eve knows that g = 7 and p = 61. According to the Diffie-Helman Algorithm, Alice will then generate a random number a, which only she has. After that, she will use a in an equation alongside g, and p to compute A. The equation is the following:

$A = g^a mod(p)$.

After the computation, the number A will also be sent via an unprotected network, and therefore eavesdropped by Eve. So, in this case, Eve will know that A = 17. Now Eve has to find the value of a.

Before we start finding a, we should analyze the equation $A = g^a mod(p)$. The first thing we must calculate is g^a, and in this case specifically, 7^a. We know that 7 is a positive integer, now let's list all the possible choices for a. Well, a can be negative (a < 0), zero, or positive (a > 0). I will outline what will happen in all three cases.

The value of a cannot be negative. According to Jeff, during office hours, the value of a should be a number between 0 and p. What is more, if we do 7^a and a is negative, then the result of the operation is going to be a floating point number, for example $7^{-2} = \frac{1}{49}$. As explained in class the mod operator behaves weirdly when we work with fractions. Sometimes, just a handful of all the possible values in the codomain of the mod function are also in the image when working exclusively with fractions. What is more, some calculations might have weird results. So we should not consider negative integers.

If a = 0, then $g^a = 1$ regardless of the value of g. If this was the case, then 1 mod p = 1, regardless of the value of p. This happens because if we divide 1 by any integer greater than 1, the remainder is going to be 1. So, this equation would not generate a secure key. Therefore, we can safely ignore this case.

So, it is safe to assume a will be a positive number.

We know that the equation is $A = g^a mod(p)$, and that there is no easy way to break it. Reversing the mod operation would be hard as there is no formula (or algorithm) that given A, we can find g^a. This happens because there are multiple numbers that leave a reminder of A when divided by p, and we don't know which one of these g^a is. In other words, there are multiple different inputs that leave remainder A when divided by p (for example, 30, 91, and 152 all leave remainder 30 when divided by 61). This means that the mod function is not invertible. So all we can do is bruteforce it.

I wrote the following Python code to do so:

Arthur

```
def DiffieHellmanBreaker(g, p, target):
    ab = 1
    while ((g**ab) % p) != target:
        ab += 1
    return ab
print(DiffieHellmanBreaker(7, 61, 30))
```

The code is simple and straightforward. It has variable ab, and this variable is plugged in the equation $g^a mod(p)$ in the position of a. So what the code does is: test for a = 1, if it fails, the code will test the case of a = 2, if it fails again the code will test the case of a = 3, and so on until it finds the correct value of a. This code is guaranteed to succeed as it tries every single value for a until it finds the correct one. But it is super slow.
Using it, I found out that a = 41.
Just like Alice has randomly generated number a, which only she has access to, Bob will randomly generate secret number b, which only he has access to.

In a process analogous to Alice's he will use it in equation $B = g^b mod(p)$. And send B in the unprotected network. Therefore, Eve can eavesdrop on it. In this specific case, B = 17. See how the equation computed by Bob is very similar to the one computed by Alice. However, instead of having g^a we have g^b. What is more, instead of naming the result A, we name it B. Outside that they are exactly the same equation. In other words, only the randomly generated variable (a for Alica and b for Bob) are different between these two equations. So we can use the same Python code to calculate the value of b.
The reasons the Python code works for b are analogous to the reasons it works for a.
As B = 17, running the Python script we get that b = 23.

Well, in the next part of the code, both will perform a computation. Alice will do $B^a modp$ and Bob will do $A^b modp$. Well, $B^a modp = A^b modp = 6$. This is going to be the key they will use to encrypt and decrypt their messages. As Eve has the value of the key she will be able to eavesdrop on them.
We have seen that the only way to get the key is by calculating either b or a, and both of these variables are private to the person who randomly generated them. We have also shown that the most effective way to calculate either one of these variables is through brute force. We were lucky that the numbers used were small and the computation was pretty fast. If we were using big numbers it would take a lot longer to be able to break the key. What is worse, if the numbers were much much larger finding the values of a or b would take longer than the death of the universe. So the security relies entirely on big numbers and the time it takes to brute force them.

Decrypting RSA:
We have Bob's public key, which literally everyone has. It is publicly available online and used to send messages to Bob. Bob's public key P is composed of two parts n, and e. P = (e, n).
We know that n = 170171, and e = 17.

Arthur

Let's crack the code, the first thing done in RSA was picking two prime numbers p and q, and multiplying them together to get n. Let's first try finding the values of p and q. Well, there is no good way to find the prime factors of numbers, it is a slow process and the usual way to find factors of numbers is brute forcing them. We try to see if 2 is a factor, if it is not we try three and so on until we find the factors we want.
As p and q are prime numbers, number n will only have two prime factors, and that is what we are trying to find. I used the following python code to get the two prime factors:

```
def primeFactorsOfInteger(n):
    x = 2
    while n % x != 0:
        x += 1
    return x, n / x
```

This code will work. First of all, for every single number n, we are testing if x evenly divides n. If it does, then by definition x will be a factor of n. If it doesn't we increment x by one and try again, we do this until we have found the correct factors of n. We start with two, as every number is divisible by one and that is not useful for our computation. As the number only has 2 factors once we find the first factor, we divide n by x so we can get the second prime factor of the number.
By running this code for 170171 we can see that the factors of n are 379 and 449.
The very next step of RSA is finding a number e < (p - 1)(q - 1) such that e and (p - 1)(q - 1) have no common divisors among them, except for 1. Luckily we already have e, as it is part of the public key and made available by Bob online.
The last part of the algorithm (which is also the next part) is to find a number d such that $ed \equiv mod(p-1)(q-1)$. We know that p = 379 and q = 449. So, p-1 = 378 and q-1 = 448. We also know that e = 17. Now we need to use Python to find d. To do so, I used the following code:

```
def finddRSA(e, p, q):
    d = 1
    while ((d * e) % ((p-1) * (q - 1))) != 1:
        d += 1
    return d
```

This is also a brute force approach. This is an appropriate way to get d as we don't know ways that are much more efficient than that. For the same reasons explained in the Diffie-Hellman part of this assignment, it is hard to find the value d when we only know that it should leave remainder 1 when divided by (p-1)(q-1). Let's start explaining the code with our choice for d. Well, d can be either negative (d < 0), positive (d > 0) or 0. If d = 0, then $d \times e = 0$, which leaves remainder 0 when divided by any number. As we are not looking for remainder 0, we can safely ignore it.

Arthur

As explained in class, the module operator behaves weirdly when working with negative integers. So the best choice was for d to be positive, the initial value of d in this case is 1. Then for every value of d we do $\frac{de}{(p-1)\,(q-1)}$ if the remainder of that equation is 1, we have found the value of d and return it. Otherwise, we increment the value of d by 1 and try again. As we try this for every value of d, we are guaranteed that we will eventually find the correct value for d. Running this program, Eve found out that the value of d is 119537.
Now, she has the private (secret) key which is S(n, d) = S(170171, 119537). Now Eve can start decrypting the message sent by Alice.
In this scenario Alice will get Bob's public key, which is available to everyone online, encrypt her message, and send it to Bob. Bob will get his private key and use it to decrypt Alice's message. The encrypted message is generated by E(P, M) = M', where P is the public key, M is the message written by Alice and E is the function responsible to encrypt it. Bob will get the message sent by alice and do E(S, M'), where M' is the encrypted message and S is the private (secret) key.

The computation performed by E is $M^{ed} mod(n)$. Observe that the message was encrypted with the private key and the function E and the same function (but now with the public key) will be used to decrypt the message. We can write a script in python where we can input the encrypted message and the code will spit out the decrypted version of the message in binary. I used the following code:

```
def decryptRSA(M, n, d):
    decrypted = []
    for i in M:
        binary = bin((i**d) % n)
        binaryToStr = str(binary)
        binaryToStr = binaryToStr[2:]
        if len(binaryToStr) < 16:
            doConversion = ["0"] * (16 - len(binaryToStr))
            doConversion = "".join(doConversion)
            binaryToStr = doConversion + binaryToStr
        firstByte = binaryToStr[:8]
        lastByte = binaryToStr[8:]
        decrypted.append(firstByte)
        decrypted.append(lastByte)
    return decrypted
```

This code is a bit more complex than the ones above so I will explain it in more detail. This is the decryption function we are using in this assignment. First of all, the input expected is M, which in this case is the encrypted message, n is part of the private (secret) key, and d is also part of the secret key.
M is an array of integers like the one posted on moodle:

```
[65426, 79042, 53889, 42039, 49636, 66493, 41225, 58964, 126715,
67136, 146654, 30668, 159166, 75253, 123703, 138090, 118085,
120912, 117757, 145306, 10450, 135932, 152073, 141695, 42039,
137851, 44057, 16497, 100682, 12397, 92727, 127363, 146760,
```

Arthur

```
5303, 98195, 26070, 110936, 115638, 105827, 152109, 79912,
74036, 26139, 64501, 71977, 128923, 106333, 126715, 111017,
165562, 157545, 149327, 60143, 117253, 21997, 135322, 19408,
36348, 103851, 139973, 35671, 93761, 11423, 41336, 36348, 41336,
156366, 140818, 156366, 93166, 128570, 19681, 26139, 39292,
114290, 19681, 149668, 70117, 163780, 73933, 154421, 156366,
126548, 87726, 41418, 87726, 3486, 151413, 26421, 99611, 157545,
101582, 100345, 60758, 92790, 13012, 100704, 107995]
```

The value for n is 170171 and the value for d will be 119537, we were given n and computed d above. We can see that d is two bytes long, and so the decrypted version of each element of the list consists of two ASCII characters. This was mentioned in class by Jeff, and if we convert 119537 we see that it demands more than one byte.
The results of the message are saved in array decrypted.

For each one of the characters in M we first do the calculation $M^{ed} mod(n)$, which is required to decrypt the message. We will see that the results are 2 bytes long (it will contain at least 8 bits, and sometimes less than 16 bits). After we compute $M^{ed} mod(n)$ we convert the results to binary using the bin() function, which is native to Python. The result is going to be something like: 0b100100001101001. There are quite a few problems with the result of this function, while the number is correctly converted to binary python adds a 0b to indicate the base. What is more, the string above is not 16 characters long, it is 15 characters long.
First of all, we remove the 0b that indicates the base of the number. We do that by converting the number into a string and storing it in the variable binaryToString. We then remove the first two characters of the string using the command binaryToString[2:] this way we get rid of 0b.
Then we can address the next issue, not all numbers are exactly 2 bytes (16 bits) long. To fix this we start by checking the size of the number, if it is less than 16 it has less than 2 bytes. If that is the case, we create an array of 0s containing enough zeros to make the size of the string exactly equal to 16. We convert this array into a string by using join and add it to the left of our original string. This way if our string was less than 16 bits long we add 0s to its left to make it exactly 16 bits long, adding 0s to the left of the number will also not impact the result.
The string contains two bytes, and each character is one byte. So to fix this issue we split the string into two. The first part will contain the 8 first bits and the second part contains the last 8 bits. We add each one of those bits to the decrypted array and return it. After we do this process to every single number in the message we can return the results to the user.
The result will contain the following binary numbers:
[1101001, 1001000, 1000010, 100000, 1100010, 1101111, 100000, 101110, 100111, 1001001, 100000, 1101101, 1100001, 1110111, 1101011, 1101100, 1101110, 1101001, 100000, 1100111, 1110010, 1100110, 1101101, 1101111, 1101110, 100000, 1110111, 1101111, 1101111, 100000, 101110, 1101110, 1011001, 100000, 1110101, 1101111, 100000, 1110010, 1100001, 1110000, 101100, 1101100, 1000001, 100000, 1101001, 1101100, 1100101, 1100011, 100000, 101110, 1110100, 1101000, 1110000, 1110100, 111010, 1110011, 101111, 101111, 1101111, 1100110, 1101110, 1110101, 1100001, 1100100, 1101001, 1110100, 1101110, 1101111, 1101101, 101110, 1111010, 1101111, 1101100, 1101001, 1100001, 1101100, 1101111, 101110, 1100111, 1110010,

Arthur

1100101, 101111, 101111, 1101110, 1110010, 1110000, 1110110, 1101001, 1100011, 1100001, 1101110, 1111001, 1110100, 1101111, 1101110, 1101001, 1101100, 1100011, 1100100, 1110101, 1100100, 1100101, 1100001, 101111, 1110100, 1110010, 1100011, 1101001, 1100101, 1101100, 101111, 1110011, 1110100, 1101001, 101101, 1110011, 1100110, 1101111, 1101001, 1100110, 1101001, 1100011, 1101100, 1100001, 1100011, 101101, 1110010, 1100001, 101101, 1110011, 1110010, 1100001, 101101, 1100101, 1101000, 1110100, 101101, 1100101, 1101111, 1110111, 1110011, 1110010, 101101, 1110100, 1110010, 1110000, 1100100, 1101111, 1100011, 1110101, 101101, 1110100, 1100001, 1100011, 1100101, 1110100, 1101111, 1100111, 1111001, 1110010, 1110111, 101101, 101101, 1100101, 1100001, 1101000, 1100101, 1110110, 1100101, 101101, 1100101, 1110110, 101101, 1110010, 1100101, 1110010, 1101001, 1110110, 1110111, 1100101, 1100100, 1100101, 1100110, 101101, 1110010, 1101111, 1110000, 101101, 1101001, 1110010, 1100001, 1110110, 1111001, 1100011, 0, 101111].
To get the results in ASCII characters we can use an online converter such as this one. Using it we wil see that the message is:
Hi Bob. I'm walking from now on. Your pal, Alice.
https://foundation.mozilla.org/en/privacynotincluded/articles/its-official-cars-are-the-worst-product-category-we-have-ever-reviewed-for-privacy/

There is an issue in the way Bob did RSA because he is using small numbers. Brute forcing the values of p,q, and d took only a few seconds. With these values in hand, all Eve has to do is write a function to decrypt Alice's message. Small numbers are easy to find out, even if the best approach is brute-forcing. The way to make the algorithm more secure is by using larger values. If the values are large enough finding each of the values of p, q, and d might take more time than the time until the heat death of the universe.
Even with the RSA encryption, Alice might still be attacked through an adversary in the middle attack. The problem is that she is relying that she has Bob's public key, while she might have Eve's.
What happens is the following. Eve is between Alice and Bob. So instead of requesting Bob's public key, Alice requests Eve's public key, encrypts the message, and sends it to Eve. Then, Eve will decrypt the message read it, get Bob's public key, encrypt the message again with his key, and send it to Bob. Bob, will read the message sent from Alice, and respond to it. But Eve is in the middle, so he will get Eve's public key, encrypt the message, and send it to Eve. Eve will decrypt it with her private key and read it. Then she will encrypt the message with Alice's public key and send Bob's response to her. In this case, Bob and Alice think they are talking to each other, but there is someone in the middle listening to their conversation, so they might not be safe.