Bryce Montano
Project 3 – Writeup

Task 2

Private key: 0x16afac61f2d2a01

In Task 2, we are given public key N (the modulus) and a prime value, e. In the first step, we need to factorize N in order to get it's values, p and q. Because N is the product of two primes p and q, we simply need to factorize N. Normally, this would not be possible with 1024, 2048 and 4096 bit integer public keys, but due to the fact that our public key is small (64 bit), we can 'brute force' by attempting to factorize it.

Now, making a 'leap of faith' so to speak, I followed an assumption based on information in this class assignment that I found online that p and q "can't both be larger than the square root of n, or they'd be larger than n when multiplied together" (Bowne 2016) With that in mind, this block of my code:

```
for i in range(int(math.sqrt(n)), int(math.sqrt(n)-300000), -1):
    newvalue = n % i
    if newvalue ==0:
        p = i
        break
q = n / p
```

Brute forces p by looping through a range ($\sqrt{N}$ - $\sqrt{N} - 300000$) and checking to see if a temporary (newValue) == 0. This is becaue n mod(p or q) is alw aaays equal to zero.

Finally, once p is found, we can solve for q.

Next, to find d, I utilized a simple modular inverse code snippet that I found on StackOverflow. (Bakhoff 2012)

In this code, a public prime e and Φn are passed into the modinv() function and are then forwarded to a Euclidian greatest common divisor function that attempts to find the greatest common divisor of two numbers e and n, replace the larger number with the smaller number subtracted from the larger. Rinse and repeat until one of the numbers reaches 1 which means the two numbers are relatively prime. The value returned is d.

Task 3

Note: Much of what I explain below was learned from an excellent write up by Seth David Schoen titled, “Understanding Common Factor Attacks:An RSA-Cracking Puzzle." *Understanding Common Factor Attacks: An RSA-Cracking Puzzle”* (Schoen, n.d.)

In RSA, a public key *n* is composed of two unique, large prime values *p* & *q*. Because this value *n* is made public, anyone can see it and view it. With that said, RSA is based on the fact that no one other than the owner should ever know the values for *p* and *q* since they can be used to generate the private key associated with the public key *n.*

Now, given that *p* and *q* values are generally extremely large, trying to factorize n in order to obtain them is nearly impossible in any short amount of time. Therefore, another technique can be used to determine one of the values (*p* or *q*) of any given public key *n* by simply calculating the greatest common divisor (gcd) between *n1* and *n2.* Ideally, any time a public key is calculated, it uses two completely random prime values such that it would be *highly* improbable (2^{512} for 512bit integers) that any two *p* or *q* values are the same and therefore performing a gcd on *n1* and *n2* would simply yield 1 (an incorrect value). However, in the case that is presented in the P’s and Q’s paper, if randomly generated prime numbers are not generated with proper seeding techniques, there is a chance for some prime numbers to be generated more than once if not numerous times.

Given that some of the offending device manufacturers mentioned in the paper were not generating private keys with enough entropy to make them truly random, this made it easy for attackers to perform simple gcd calculations on public keys within the IP address space (and those they already had obtained) and deriving p and q based on the common divisor.

In order to find d for Task 3, I followed the exact tactics described above. I ran a check to see if a there was a gcd(n1, n2) greater not equal to 1. If there was, I knew I had found my result and returned it to the next function, get_private_key.

In get_prviate_key, I performed the steps mentioned above in order to find p and q by deriving the gcd(n1, n2) which equals p and then solving for q. Now, given that I have p and q, I used the same modular inverse function and greatest common divisor function from Task 2 in order to solve for d.

Citations

1. Bowne, Sam. "Proj RSA2: Cracking a Short RSA Key (15 Pts.)." *Proj RSA2: Cracking a Short RSA Key (15 Pts.)*. N.p., 31 Mar. 2016. Web. 18 Nov. 2016.
2. Bakhoff, Märt. "Modular Multiplicative Inverse Function in Python." *Algorithm - Modular Multiplicative Inverse Function in Python - Stack Overflow*. N.p., 18 Mar. 2012. Web. 18 Nov. 2016.
3. Schoen, Seth David. "Understanding Common Factor Attacks:An RSA-Cracking Puzzle." *Understanding Common Factor Attacks: An RSA-Cracking Puzzle*. N.p., n.d. Web. 18 Nov. 2016.