Attacks and Vulnerabilities

As discussed above, RSA is considered secure. But this only means that we don't know an efficient decoding algorithm that works for all messages. This does not exclude that certain choices of parameters (the primes p and q, the exponents e and d, the messages to be encoded) are vulnerable. In this section, we will study (and implement!) a few attacks that work well in certain special cases.

Small Space of Messages

Alice needs to send to Bob one of the following two messages: `attack` or `retreat`. This is essentially one bit of information, hence Alice and Bob decide to use $m = 0$ and $m = 1$ for these two possibilities.

Stop and think! Why is this not a good idea?

To see why this is insecure, assume that Alice sends a ciphertext $c \equiv m^e \mod n$ to Bob. Then, Eve can do the following: she encodes both $m = 0$ and $m = 1$ using Bob's public key (n, e) and compares the results to c. One could argue that above we work under the assumption that Eve knows how the two messages (`attack` and `retreat`) are mapped to two integers (0 and 1). This is true, on the one hand. On the other hand, it is typical in cryptography to assume that the way messages are converted to integers is known to all parties. Moreover, even if Eve has decoded an intercepted bit of information, but has no idea whether this bit corresponds to `attack` or `retreat`, this way of using RSA is still insecure. Indeed, if tomorrow Eve intercepts a new ciphertext, she will know whether it is the same as today's ciphertext.

Note that the same attack works also in case when Alice and Bob use any other two small integers instead of 0 and 1. More generally, the attack is applicable when the space of all possible messages is not too large: Eve can then encode all potential messages and compare the resulting encodings to the intercepted ciphertext.

A defense against such an attack is to increase artificially the space of possible messages so that it is computationally infeasible to enumerate all its elements. To achieve this, Alice may use randomness. Namely, use the first 128 bits for the message and append 128 more random bits before encryption. Then, Bob will just drop the last 128 bits after decoding the ciphertext, whereas Eve will not be able to enumerate 2^{128} possibilities in any reasonable time.

Problem

Secret agent Alice has sent one of the following messages to the center:

- `attack`,
- `don't attack`,
- `wait`.

Alice has ciphered her message using public key $(\texttt{modulo}, \texttt{exponent})$ that is available to you, and you have intercepted her ciphertext. You want to know what was the content of her message. You have access to the function

$$\texttt{Encrypt}(\texttt{message}, \texttt{modulo}, \texttt{exponent})$$

that takes in a message as a string and returns a big integer as a ciphertext. It uses RSA encryption with public key $(\texttt{modulo}, \texttt{exponent})$. In the starter code, you have an example usage of the function `Encrypt`. You also have function

$$\texttt{DecipherSimple}(\texttt{ciphertext}, \texttt{modulo}, \texttt{exponent}, \texttt{potential\_messages})$$

implemented in the starter code. You need to fix this implementation to solve the problem. It should take the `ciphertext` sent from Alice to the center, the public key $(\texttt{modulo}, \texttt{exponent})$ and the set of potential messages that Alice could have sent, and return the message that Alice encrypted and sent as a string. For example, if Alice took message "wait", encrypted it with the given `modulo` and `exponent`, and got number 139763215 as the ciphertext, you will need to return the string "wait" given the$\texttt{ciphertext} = 139763215, \texttt{modulo}, \texttt{exponent}$ and

$$\texttt{potential\_messages} = [\texttt{"attack"}, \texttt{"don't attack"}, \texttt{"wait"}]$$

```
def DecipherSimple(ciphertext, modulo, exponent, potential_messages):
    # Fix this implementation
    if ciphertext == Encrypt(potential_messages[0], modulo, exponent):
        return potential_messages[0]
    return "don't know"


modulo = 101
exponent = 12
ciphertext = Encrypt("attack", modulo, exponent)
print(ciphertext)
print(DecipherSimple(ciphertext, modulo, exponent,
                     ["attack", "don't attack", "wait"]))
```

Try it: question 2.

Small Prime

When preparing the keys, Bob generates two random prime numbers, p and q. One of them turns out to be relatively small: say, $p \leq 10^6$. (If Bob generates a random number $p \leq 2^{4096}$ and then checks its primality, then the probability of the event "$p \leq 10^6$" is negligible. Thus, this is just an instructive example rather than a real vulnerability).

Stop and think! Do you see a potential attack?

In this case, Eve can factor n quickly: she goes through all primes up to 10^6 and checks whether the current prime divides n. As discussed above, when n is factored, Eve can decrypt any message the same way Bob does.

There is a natural defense against this attack: Bob needs to ensure that both primes are long enough --- say, 4096 bits long. To achieve this, he should take a random integer $2^{4096} \leq p < 2^{4097}$. This makes the task of enumerating all primes of this length practically infeasible.

Problem

Alice is using RSA encryption with a public key $(\texttt{modulo}, \texttt{exponent})$ such that $modulo = p \cdot q$ with one of the primes p and q being less than $1\ 000\ 000$, and you know about it. You want to break the cipher and decrypt her message.

You can use the function $\texttt{Decrypt}(\texttt{ciphertext}, \texttt{p}, \texttt{q}, \texttt{e})$ which decrypts the `ciphertext` given the private key p, q and the public exponent e.

You are also given the function

$$\texttt{DecipherSmallPrime}(\texttt{ciphertext}, \texttt{modulo}, \texttt{exponent}),$$

and you need to fix its implementation so that it can decipher the `ciphertext` in case when one of the prime factors of the public modulo is smaller than $1\ 000\ 000$.

```
def DecipherSmallPrime(ciphertext, modulo, exponent):
    if modulo % 2 == 0:
        small_prime = 2
        big_prime = modulo // 2
        return Decrypt(ciphertext, small_prime, big_prime, exponent)
    return "don't know"


modulo = 101 * 18298970732541109011012304219376080251334480295537316123696052970419466495220522723330315111017831737980079504337868198011077274303193766040393009648852841770668239779097280026631944319501437547002412556176186750790476901
exponent = 239
ciphertext = Encrypt("attack", modulo, exponent)
print(ciphertext)
print(DecipherSmallPrime(ciphertext, modulo, exponent))
```

Try it: question 3.

Small Difference of Primes

When preparing the keys, Bob generates two random prime numbers, p and q. They turn out to be close to each other: say, $|p - q| < 10^6$. (As with the previous example, the probability of this event is negligible.)

Stop and think! Do you see a potential attack?

In this case, Eve can factor n quickly again. To see why, assume (without loss of generality) that $p < q$. Then,

$$p < \sqrt{n} < q\,.$$

Now, let $r = q - p < 10^6$ be the difference of the primes. Then,

$$\sqrt{n - r} < p < \sqrt{n}\,.$$

Thus, to factor n, it suffices to go through all potential divisors in the range $[\sqrt{n - r}, \sqrt{n}]$ that is short enough by our assumption.

A defense against this attack is simple: after generating the two primes, check whether their difference is small; if it is, regenerate.

Problem

Alice is using RSA encryption with a public key $(\texttt{modulo}, \texttt{exponent})$ such that $modulo = p \cdot q$ with $|p - q| < 5\,000$, and you know about it. You want to break the cipher and decrypt her message.

You can use the function

$$\texttt{Decrypt}(\texttt{ciphertext}, \texttt{p}, \texttt{q}, \texttt{e})$$

which decrypts the `ciphertext` given the private key `p`, `q` and the public exponent `e`. You also have access to the function $\texttt{IntSqrt}(\texttt{n})$which takes integer n and returns the largest integer x such that $x^2 \leq n$. You are also given the function

$$\texttt{DecipherSmallDiff}(\texttt{ciphertext}, \texttt{modulo}, \texttt{exponent})$$

and you need to fix its implementation so that it can decipher the `ciphertext` in case when the difference between prime factors of the public modulo is smaller than $5\,000$.

```
def DecipherSmallDiff(ciphertext, modulo, exponent):
    small_prime = IntSqrt(modulo)
    big_prime = modulo // small_prime
    return Decrypt(ciphertext, small_prime, big_prime, exponent)


p = 1000000007
q = 1000000009
n = p * q
e = 239
ciphertext = Encrypt("attack", n, e)
message = DecipherSmallDiff(ciphertext, n, e)
print(ciphertext)
print(message)
```

Try it: question 5.

Insufficient Randomness

Every day, Bob generates two fresh random prime numbers p and q and then publishes $n = pq$ as part of the public key. To do this, Bob uses a random number generator. Assume that for some two days, the same prime p was used: $n_1 = pq_1$ and $n_2 = pq_2$.

Stop and think! Do you see a potential attack?

In this case it is easy to factor n_1 and n_2 as one can find (efficiently!) their greatest common divisor. This vulnerability occurs when there is not enough randomness to generate the keys: for example, the primes are generated by a router right after startup and there is just no data to extract randomness from. In this case, an attacker can collect public keys from different devices and to try to find a non-trivial common divisor of every pair of them. This is a practical attack: using it, many HTTPS keys have been factored.

A defense against such an attack is to make sure that the pseudorandom generator used is properly seeded. There are