Set 7: Cryptography

Aritra Biswas

1 Hypothetical Threat Model

Suppose that I'm trying to collaborate with my friend Claire on writing a website. Claire is in Nauru, and for some reason, the Nauruan government has decided that I might be a threat to the people of Nauru (all 10,000 of them).

Suppose that the Nauruan government has the ability and legal authority to monitor Claire's network traffic, and the ability (but not the legal authority) to interfere with it. They also do not have the authority to restrict Claire's communication with me.

I don't want Claire to directly access our web server in the US because I would then have to worry about protecting the web server. Instead, Claire and I resort to exchanging code snippets via encrypted email.

2 RSA

The basic idea behind RSA is to create a cryptographic hash function such that:

1. the function is easy to apply with a public key,
2. the function is easy to invert with a private key, and
3. it is difficult to obtain the private key from the public key.

In RSA, there is a modulus n, a public key (n, e), and a private key (n, d). Requirement 3 is achieved because d is the multiplicative inverse of e modulus $\varphi(n)$; that is, $de = 1 \bmod \varphi(n)$.

Thus, calculating d from e requires knowing $\varphi(n)$, where φ is Euler's totient function. The modulus n is a product of two prime numbers p and q. If p and q are known, calculating $\varphi(n)$ is trivial: $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$ since, for a prime number k, there are $k-1$ totatives (numbers between 0 and k, exclusive, that are coprime to k). However, without already the prime factors p and q, the only efficient way to calculate $\varphi(n)$ is to factorize n (i.e. obtain p and q), which is prohibitively difficult if n is large.

The public key exponent e is chosen to be a totative of $\varphi(n)$. The creators of RSA used Fermat's little theorem to prove that, for any integer m, $(m^e)^d = m \pmod n$. Thus, we can express a message as an integer m using a publicly-known padding scheme, create an encrypted message $c = m^e \bmod n$, and then retrieve the original message by $m = c^d \bmod n$.

2.1 Known Weaknesses

Some known weaknesses of RSA include:

- **Insufficient randomness:** if the random number generation in a machine that generates RSA keys is not well-seeded, it can generate two distinct keys n and n' that have a common prime factor, i.e. $n = pq$ and $n' = pq'$. Then computing the gcd of n and n' yields p, and both keys have been factored and compromised. This is currently addressed by using good sources of entropy (such as sensor noise or keystrokes) to generate good random numbers.

- **Timing attacks:** the decryption process involves computing $c^d \bmod n$. If an attacker can measure the amount of time required to decrypt certain values of c, he can predict d. This is remedied by cryptographic blinding, in which a random number r is chosen and the decryption process computes $(r^e c)^d \bmod n = rc^d \bmod n$. Then, the decryption process can multiply by the modular multiplicative inverse of r to obtain the desired $c^d \bmod n$. This way, the computation time depends on r as well as d and c, and if r is chosen randomly, one cannot infer d from the computation time.

3 Toy Implementation of RSA

```
# toy_rsa
# Aritra Biswas
#
# A demo implementation of RSA.

import random
from math import sqrt

def gen_primes(bound):
    '''Generates a list of prime numbers below bound using a sieve.'''

    # status list: composite[i] = 0 means i is composite
    composite = [0] * (bound + 1)

    for i in xrange(2, int(sqrt(bound)) + 1):
        if composite[i] == 0:
            for n in xrange(2, int(bound/i) + 1):
                composite[n * i] = 1

    # primes list
    primes = []
    for i in xrange(2, bound):
        if composite[i] == 0:
            primes.append(i)

    return primes

def gen_keys():
    '''Generates an RSA public and private key pair, returned as
    (public, private) = ((modulus, public_exp), (modulus, private_exp)).'''

    # pick two primes
    four_bit_primes = gen_primes(2**4) # chosen such that n is 8 bits
    while True:
        p = random.choice(four_bit_primes)
        print p
        q = random.choice(four_bit_primes)
        print q
        if (p != q):
            print 'Ready!'
            break
```

```
    # calculate the product n, and its totient
    n = p * q
    totn = (p - 1) * (q - 1) # since phi(n) = phi(p) * phi(q), both prime

    # public key exponent must be smaller than and coprime with totn, so
    # pick a prime number so that we only have to check if it divides totn
    possible_e = gen_primes(totn)
    for elem in possible_e:
        if totn % elem == 0:
            possible_e.remove(elem)
    e = random.choice(possible_e)

    # private key exponent is the multiplicative inverse mod totn, i.e.:
    # d*e = 1 (mod totn)
    # d*e = k*totn + 1
    #   d = (k*totn + 1) / e,   for any k that makes d an integer
    k = 0
    # increment k until the quotient (k*totn + 1) / e is an integer
    while (k*totn + 1) % e != 0:
        k += 1
    d = (k*totn + 1) / e

    public_key = (n, e)
    private_key = (n, d)

    return (public_key, private_key)

def encrypt(m, public_key):
    '''Encrypts an integer message m.'''

    (n, e) = public_key
    c = (m**e) % n

    return c

def decrypt(c, private_key):
    '''Decrypts an integer message c.'''

    (n, d) = private_key
    m = (c**d) % n

    return m
```

3.1 Collision Attack

The following function is a brute-force attempt at a traditional collision attack (no chosen prefix), where we aim to find two message $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$ where H is our hash function.

```
def brute_collision_attack(public_key):
    '''Attempts to find two messages that result in the same encrypted
```

```
    message given a public key. '''

    hashes = []
    m = 0

    while True:
        new_hash = encrypt(m, public_key)
        if new_hash in hashes:
            return (hashes.index(new_hash), m)
        hashes.append(new_hash)
        m += 1
```

The brute-force method simply tries all possible messages sequentially until it finds one with a hash that matches one that has previously been generated.

After running this attack a few times, it seems to always return $(0, n)$. Upon further investigation, it appears that $H(m_1) = H(m_1 + kn)$ for all integers k. This means that while a collision can be found trivially, it should be difficult to find an useful collision since the decoy message m_2 must be related to the original message m_1 by $m_2 = m_1 + kn$.

4 Test Drive: Tor

Tor is a network allowing anonymous communication through onion routing. Suppose Alice and Bob are communicating, and Alice is using Tor.

The Tor network consists of various nodes, or computers equipped to relay information. Suppose Alice wishes to send a message to Bob. The message must include Bob's address. The Tor client obtains a list of known nodes and then picks, at random, n nodes in order. The client takes the message (including Bob's address) and encrypts it with the nth node's public key. Then it adds the nth node's address and encrypts the result with the $(n-1)$th node's public key, and so on, finally adding the 2nd node's address and encrypting it with the 1st node's public key.

The encrypted "onion" is then sent to the first node, which decrypts the message with its private key, uncovers the 2nd node's address, and sends it to the 2nd node. This chain proceeds with the ith node decrypting the message, discovering the $(i+1)$th node's address, and sending it to the $(i+1)$th node. Finally, the nth node decrypts the message, finds Bob's address, and delivers it to Bob.

Suppose an attacker Mallory, who wants to eavesdrop on the conversation between Alice and Bob, manages to accesses the ith node in this path (which is already unlikely since the chain of nodes is randomly chosen). She can find out the address of the $(i-1)$th node, and the $(i+1)$th node, but nothing else. To fully decrypt the message and know both the source and destination addresses, Mallory would have to access every Tor node involved in the chain between Alice and Bob.

5 Proof of Work

A screenshot of `systemd` log showing a running Tor service is attached.

6 Password Security

If an attacker is trying to guess a password, it is likely that he will (1) use all available information - the user's name, etc. and then (2) brute force methods.

To prevent strategy (1), it is important to not use any of the following in a password:

- names and/or nicknames of the user or relatives

- names of places, pets, etc.
- anything that could be found by finding the user on social media

In a brute force method (2), the attacker is likely to first use a dictionary, so it is important to not use full words. If a dictionary attack fails, the attacker could brute-force individual characters. Suppose a password can be at most n letters long. Let p be the number of characters in the alphabet used. Then there are $p + p^2 + p^3 + \cdots + p^n$ possible passwords.

The higher the number of possible passwords, the less likely a brute-force approach will be to succeed. Thus, the user can increase p (by using special characters, etc.) or increase n (use a longer password). For a fixed k, $f(x) = k^x$ will eventually grow faster than $f(x) = x^k$, so increasing n is probably the more effective choice.

A strong password can be easily compromised if the attacker gains access to tools that the user uses to remember the password (a piece of paper in a wallet with the password written on it, password management software, etc.). Thus, it is also important to strike a balance between password strength and memorability, because a password that needs to be written down is inherently unsafe.

7 Conclusion and Application to Threat Model

I had known before that randomness and prime numbers play a key role in encryption; now I'm aware of the specific need to generate good random numbers such that large keys do not have a common prime factor.

In reference to the threat model, suppose Claire and I each generate our own public/private key pairs and encrypt our emails using them. It is important for Claire to store these keys in a directory that cannot be accessed by any program on her computer that can connect to the internet; otherwise, the Nauruan government might be able obtain her key (if they access her computer).

The government is in a prime position to attempt a man in the middle attack since they control Claire's internet access. Thus, it would be ideal for Claire and I to have exchange keys before she left the country. If this is not possible, it would be necessary to use some authentication mechanism that would be difficult to duplicate. For example, we could agree to exchange keys through voice recordings.