Friday, Feb. 17
Winter 2017

CME 451 – Lab 4

Network Security [1]

Pre-Lab: While the material provided in this lab is intended to be self-contained, the interested reader is urged to also browse through the online tutorial of Python programming by Python Software Foundation, available here: `https://www.python.org/doc/`.

Verification: Exercises marked with an **asterisk** (∗) are required to be verified and signed off by a lab instructor during your assigned lab time. When you complete a step that requires verification, simply demonstrate the step to a lab instructor. Turn in the completed verification sheet to your TA at the end of the lab.

Code Submission: Exercises marked with a **dagger** (†) can be completed after your assigned lab time. Submit your code and relevant outputs electronically in one zip file (named <NSID>-<Lab#>) through the BlackBoard no later than a week after your lab. Enhance the readability of your code by adding explanatory comments wherever appropriate. If you are unsure about what is expected, ask your TA.

1 Introduction

In this lab, you will learn the principles of security and cryptography in networks, including symmetric encryption, asymmetric encryption (public-key encryption), authentication, and digital signature algorithms. Upon completion of this lab, you will be able to write Python programs to perform basic network encryption and network authentication.

2 Network Encryption

Basically there are two categories of network attacks, the passive attack and the active attack. Passive attacks are intended to obtain the information that being transmitted. They do not involve any alteration of data and are therefore difficult to detect. The most feasible way to prevent this kind of attack is encryption. With encryption, the message is transformed to a set of meaningless characters. Although the attackers might obtain the message, however, without the key and decryption algorithm, they cannot figure out the actual content of the message.

Symmetric encryption is the universal technique to provide confidentiality for the transmitted data. In symmetric encryption, the sender and the receiver share the same encryption key. In the sender side, the sender performs a specific encryption algorithm with an encryption key on the original plaintext to generated the scrambled ciphertext. The ciphertext is transmitted through the network. In the receiver side, the receiver will perform a decryption algorithm (which is the reverse of the encryption algorithm) with the same encryption key that the sender used on the ciphertext to recover the original plaintext. In order to perform symmetric encryption, both the sender and receiver must share the same encryption key in a secure manner. In addition, a strong encryption algorithm is required. Two popular encryption algorithms for symmetric encryption is data encryption standard (DES) and advanced encryption standard (AES).

In Python, symmetric encryption can be efficiently implemented with the `pycrypto` module. `pycrypto` module is a collection of various encryption algorithms and secure hash functions. If you have installed Anaconda Python, you can install `pycrypto` easily with the following command in command prompt/terminal:

```
conda install pycrypto
```

[1] Lab handouts are available online on the Blackboard. Please report typos and send comments to francis.bui@usask.ca.

Example 2.1 (*Symmetric Encryption with pycrypto*)
The following program implements DES encryption algorithm. It uses a key `'01234567'` to encrypt the message `'abcdefgh'`. The DES works here in ECB mode, which means each of the plaintext blocks is directly encrypted into a cipher block independent of any other block. For detailed information of working modes, please refer to `https://www.dlitz.net/software/pycrypto/api/current/`. Also note that the key is 8-byte long, therefore, the text's length needs to be a multiple of 8 bytes.

```
>>> from Crypto.Cipher import DES
>>> des = DES.new('01234567', DES.MODE_ECB)
>>> text = 'abcdefgh'
>>> cipher = des.encrypt(text)
>>> cipher
b'\xec\xc2\x9e\xd9] a\xd0'
>>> decrypt_text = des.decrypt(cipher)
>>> decrypt_text
b'abcdefgh'
```

Asymmetric encryption, also called public-key encryption, is another technique to encrypt the transmitted data. Different from symmetric encryption, a pair of keys (public key (PU) and private key (PR)) are used in asymmetric encryption. PU is made for other network participants while the PR is only known to its owner. In the sender side, the sender uses the PU of the receiver to encrypt the message. In the receiver side, the receiver will use its PR to decrypt the ciphertext. Other participants cannot decrypt the ciphertext because they do not have the receiver's PR. One of the widely used asymmetric encryption algorithm is RSA algorithm.

Example 2.2 (*Asymmetric Encryption with pycrypto*)
The following program implements RSA encryption algorithm. The key pairs is generated with a random generator. Also the size of the key is specified to be 1024-bit. The message is encrypted with the public key. And the ciphertext is decrypted with the private key. When doing encryption, the second parameter, 32, is just a random parameter. According to the official documentation, it is used for compatibility only and is ignored when performing the encryption.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto import Random
>>> random_generator = Random.new().read
>>> key = RSA.generate(1024, random_generator)
>>> key
<_RSAobj @0x526ff28 n(1024),e,d,p,q,u,private>
>>> public_key = key.publickey()
>>> enc_data = public_key.encrypt('abcdefgh', 32)
>>> enc_data
(b'\x95\x91Q\xb7h\x00\xac\x82\x92#\x8em
=\x8e\xe7\xdc\x19O\xdb\xbcq\xf9\x1b\xdf\xdb\xc5J1\xce|U\xd3\x8d\x15gM
\xb1\xd1OV\x9eD\x87\xd3\x9c\xc9\x9d\xf6{\xc8f\x10\x01\xc2\xa1\xe9\xf9
\xb8_\xde\x8a\xd0\xa6&r\xa8\xe0\xbb\xae\x1f\xc7\x8c\xc7\xe3\xdf\x00\xae
\x03j4\x91 \xdb\x99I\xe6\xec\xd0\x89\xde\x1c\xa0[\x96w:\x18\xc0\x17\x8b
\xe3\xe73@$K|+\xcb\xa0\xc7\xf3\x80"$\xec\xca\xa5b\x04w\x11\xda\xf3\x89!\xa5',)
>>> decrypt_text = key.decrypt(enc_data)
b'abcdefgh'
```

3 Authentication and Digital Signature

Active attacks are the falsification of data and transactions. To protect data transmission from active attacks, the authentication techniques are applied. Authentication is to verify the received message is not altered and the source is authentic.

In order to verify the integrity of data, various hash functions can be used. Hash function can produce a fixed-length string based on various size of input. The hash function has three basic requirements:

- It is infeasible to guess the original message based on the hash string.
- It is infeasible to find 2 different input messages that have the same hash string.
- It is infeasible to modify the input string without modifying the hash string.

MD5, SHA-1, and SHA-256 are commonly used hash function. These hash function can be easily implemented using `pycrypto`.

Example 3.1 (*Hash Functions*)
The following program shows how to use hash functions with `pycrypto`. MD5 generates a 128-bit hash value. SHA-1 produces a 160-bit hash value. And SHA-256 outputs a 256-bit hash value.

```
>>> from Crypto.Hash import MD5
>>> from Crypto.Hash import SHA
>>> from Crypto.Hash import SHA256
>>> # Generate MD5
>>> hash_md5 = MD5.new(b'CME451 Course').digest()
>>> # digest() return binary format hash value
>>> # use hex() to make it readable
>>> print(hash_md5.hex())
41297691a2a72437702f9b1217227773
>>> # Generate SHA1
>>> hash_sha1 = SHA.new(b'CME451 Course').digest()
>>> print(hash_sha1.hex())
7e2d8f626f5cd2c55da0d3334859b79013f2d923
>>> # Generate SHA256
>>> hash_sha256 = SHA256.new(b'CME451 Course').hexdigest()
>>> # hexdigest() directly return the printable hash value
>>> print(hash_sha256)
dd779f6741eae2026ea05343d5ae006b12363682cd918fa55f4809533c1484dd
```

The asymmetric encryption can also be used in authentication, which is called the digital signature algorithm. In the normal encryption, the public key is used to encrypt the message and the private key is used to decrypt the message. In authentication, the key pair are used in a different manner. The sender uses his private key to sign the message. The receiver can only use the sender's public key to verify the message. No one else has the sender's private key and therefore the receiver can confirm that the message comes from the authentic sender. In order to be more efficient, the sender will sign the hash value of the original message instead of signing the original message.

Exercise 3.1 (*Digital Signature*) *
Bob and Alice want to communicate with each other using asymmetric encryption. In order to verify the source is authentic, they also need to sign their message so that the other one can verify the source of the message. In this exercise, you do not need to consider how the message is transmitted over the network. Write a python script to accomplish this process. Specifically, you need to accomplish these tasks:

1. Generate key pairs for both Bob and Alice.
2. Bob and Alice encrypt and sign their own message.
3. Bob and Alice decrypt and verify their received message.

(*Hint*: When performing signature and verification, you can use the `private_key.sign(hash_of_message, '')` and the `public_key.verify(hash_of_decrypt, signature)` method of `_RSAobj`. You can choose any hash function for the signature. For more detail of these methods, please refer to `https://www.dlitz.net/software/pycrypto/api/current/`, click `Crypto.PublicKey.RSA` in Submodules and then click `_RSAobj` in Classes.)

Exercise 3.2 (*Symmetric and Asymmetric Encryption*) †
In this exercise, you will send an encrypted message from a socket client to a socket server. The message will be encrypted using AES symmetric encryption algorithm in the client. The server receives this ciphertext and decrypt it using the same symmetric key. Write a python program to create a socket server and socket client to accomplish this process.
(*Hint*: For communication over symmetric encryption, you need to first securely distribute the symmetric key between sender and receiver. You can use asymmetric encryption to help distribute the symmetric key. The client can generate the symmetric key and encrypts it with server's public key. The server can receive the encrypted key and decrypts it using its own private key. After the key distribution, the message transmission can start.)

CME 451 Lab 4 - Instructor Verification Sheet

Turn this page to your grading TA.

Name:

Student No.:

Instructor Verification: Exercise 3.1	
Verified:	Date & Time: