Assignment 1

secure programming 2016

Dimitri Diomaiuta - 2553339

Question 1:

Which requirement is violated when a third-party passively monitors unencrypted network packets exchanged between two hosts?

(a) **Confidentiality**

Which requirement is violated when a third-party modifies network packets exchanged between two hosts?

(b) **Integrity**

Alice downloaded a Twitter client and once she installed it in her smartphone, the application asked her Twitter credentials to access her Twitter feed. Which requirements are affected?

(c) **Confidentiality/Authentication**

Describe a methodology which allows two parties to exchange messages through a protocol that ensures the integrity of the communication:

A common protocol that ensures the integrity of the communication is the MAC (=Message Authentication Code) protocol. This protocol consist in generating a small piece of information, the MAC, before sending the encrypted message giving to the tag algorithm both the original message and the secret key. Once the message has arrived the receiver, having both the secret key and the MAC can use once again the protocol algorithm used to generate the MAC in order to verify the integrity of the message. In the case the file has been changed the protocol algorithm will detect that. The drawback of this algorithm is that not only the secure key must be sent through a secure channel but also the MAC piece of information must be sent in a secure way.

Question 2:

This is the first encrypted message:

dtucluclmlcuy xqlyqccmsqlqzobi dqplgudtlmlcuy xqlobi d!sbm tuolmxs!budty

In order to decipher it I tried to see if there were some pattern occurrence. Since the one I found were not enough to let me think about being encrypted witht the Vigenere algorithm I thought that every letter must have been encoding by shifting it using the Caesar cipher algorithm. I then wrote a small program that could bruteforce through all the possibilities in order to decrypt it. The first time I tried to bruteforce it I failed. I then added the '!' ' ' characters to the alphabet used to decrypt. This time the program find a solution among all the possible output:

Shifting letters by 16 index position:

this is a simple message encrypted with a simple cryptographic algorithm

The source of the program can be found in “./Problem2/caesarCipherBruteForce/”.

I compiled with: *javac Problem2/caesarCipherBruteForce/*.java*

and run with (after changing the current working directory to Problem2 one): *java caesarCipherBruteForce.CaesarCipherBruteForce 'dtucluclmlcuy xqlyqccmsqlqzobi dqplgudtlmlcuy xqlobi d!sbm tuolmxs!budty'*

This is the second encrypted message:

xqm drwieiwrqypndvi wjkndnrlvftaimddmaliewdrqyvxznhig!ayxxk!eylrgieukxvrxqq

First I tried to decipher it by using the ceasarCipherBruteForce small program I already wrote. The program did not success in finding a possible solution. I then analyzed the text in order to find possible patterns. I found 'dr' 'ie' 'nd' as different two-letters patterns, with ie recurring three times and the other two times. I also found 'rqy' as a three-letters-pattern, recurring two times. Since a three-letters-pattern is also a two letter pattern I had four different two-letters-pattern. I hence decided to write a small bruteforce program in order to find all the possible solutions with a key size of length two. I used the same alphabet as the one used for decrypting the previous string. Since the program would have produce 784 different output strings (28^2: the size of the alphabet to the power of the size of the key) the output strings were parsed in order to keep just the ones written in English (at least with 90 % of English words the string should contain). The words of each resulting string were checked in order to see if they were contained in the English unix dictionary (“/usr/share/dict/american-english”). I provided the dictionary file used by the program under the ./Problem2 directory in the case the code was not tested under unix-like system. This was the resulting output:

Decrypted text with the following key: yt

this is a simple message encrypted with an improved cryptographic algorithm

All the words were contained in the dictionary except from 'cryptographic' but the string was still accepted since 90% of its words were contained in it.

The source code of the program can be found at: “./Problem2/vigenerCipherBruteForce/”

I compiled with: *javac Problem2/vigenerCipherBruteForce/*.java*

And run with (after changing the current working directory to the Problem2 one): *java vigenerCipherBruteForce.VigenerCipherBruteForce 'xqm drwieiwrqypndvi wjkndnrlvftaimddmaliewdrqyvxznhig!ayxxk!eylrgieukxvrxqq'*

Question 3:

The source code of the playfair algorithm implementation can be found at: “./playfair/”

I compiled the program with: *javac playfair/*.java*

If the program it is runned without any arguments (*java playfair.Playfair*) then it will print on the standard output how it should be run:

Usage: <--encrypt or --decrypt option> <input file> <output file>

--encrypt

the program will run in encryption mode and encrypt the file given as input

--decrypt

the program will run in decryption mode and decrypt the file given as input

<input file>

This is the path to the input file, it should be a plain text file. Encrypted or decrypted but it should be readable.

<output file>

This is the path to the output file.

If we create an input file we can run the program and test it: *echo "This is a small part of plain text to be encrypted with the playfair algorithm" >> playfairInput*

We can the run the program as follows:

java playfair.Playfair --encrypt playfairInput playfairOutput

the program will ask us for a key that will be used to encrypt the text by shifting the alphabet matrix (5*5 character matrix containing all alphabet letters except from 'j') by the key used. In this case the if we use “random” as a key the program will print the alphabet used for this key and encrypt the file text accordingly. This is the alphabet (I left it printed by the program in order to show how it works):

R A N D O

M B C E F

G H I K L

P Q S T U

V W X Y Z

Using a key as “1234random” or “random random” or “random%^$” it is equivalent since only the singular alphabet characters are used by the cipher (since the alphabet can contain just alphabetical characters that are unique).

The *playfairOutput* file will contain:

“QK SX SX NQ BR IZ GU NA UD MU HO SC YK YS UD CF CD MN VT YK AY KS KQ KB UG DW BO GN OH LR NG QK FV”

I decided to keep the letters in tuples since it was easier to track down how the algorithm works and check if the letters were encoded correctly.

If we now decrypt the file with:

java playfair.Playfair --decrypt playfairOutput playfairOutputDecrypted

The *playfairOutputDecrypted* output file will contain the following:

“TH IS IS AS MA LX LP AR TO FP LA IN TE XT TO BE EN CR YP TE DW IT HT HE PL AY FA IR AL GO RI TH MZ”

Since the spaces are not kept during encryption the decryption will result in a string of tuples. Further more the 'x' character for padding equal letters and 'z' character for padding a string with an odd length are kept since the algorithm cannot know whether they are produced by the encryption or they are part of the plain text (see 'SMALXL' that is actually 'SMALL' and 'ALGORITHMZ' that is actually 'ALGORITHM').

Question 4:

The source code of the program implementing the file vault based on AES encryption can be found at: “.filevault”

I compiled the program with: *javac filevault/*.java*

The program it is runned as follows: *java filevault.Filevault*

Once running the program will prompt the user to different menus in order to give him the possibility to encrypt/decrypt file to/from the vault and manage the vaults by creating one or selecting one. The user can type 'exit' followed by 'Enter' to exit the program at any time.

The first menu that appears says as follows:

Type 'exit' to exit the program or 'help' to display this menu

select or create a file vault:

Usage: <option> <directoryPath>

Where <option> is 'cd' or 'mkdir'

The cd option will select an already existing vault, the mkdir one (I chose to use the same names as the unix commands names for convenience) will create one.

Examples:

- *mkdir /home/user/myVault*

  This will create a vault called '*myVault*' under the /home/user directory. If the vault already exist than it will be just selected.

- *cd /home/user/Documents/alreadyExistingVault*

  This will select as vault the one called '*alreadyExistingVault*' under /home/user/Documents

After selecting/creating the vault the user will be prompted to a new menu:

Type 'exit' to exit the program and 'help' to display this menu

List the files in the vault, decrypt or encrypt a file

Usage: <option> <filePath>

Where <option> is 'encrypt' or 'decrypt' or 'ls'. 'encrypt' and 'decrypt' options have to be followed by a filePath

The path must be absolute for the files to be encrypted and relative to the vault for the ones to be decrypted

The decrypt option can also have a destination path of the decrypted files, if it doesn't the decrypted file destination will be the vault

Here the user can do three actions:

1. add a file to the vault by encrypting it
2. retrieve a file from the vault by decrypting it
3. list the contents of the vault

Examples:

- *encrypt /home/dimitri/Documents/importantFile*

  This encrypts the file *importantFile* under the */home/dimitri/Documents* with the AES algorithm. The user is prompted to digit the password. The digits cannot be seen by the user while typing (as in a linux/unix terminal, in order to be more secure) but the password will be saved once pressed 'Return'. The user will be asked to type it twice in order not to encrypt a file with the wrong password by mistake. The file will then be encrypted and added to the vault.

- *ls*

  This lists the content of the vault that was selected before. In this case, after running *mkdir /home/user/myVault* and *encrypt /home/dimitri/Documents/importantFile* the command will produce the following output:

  importantFile.encrypted

  I decided to add a .encrypted extension to the encrypted file in order to make them recognizable and do not overwrite unencrypted files.

- *decrypt importantFile.encrypted*

  This decrypts the *importantFile.encrypted* we previously encrypted and writes the output file to the vault. Decrypt command works with relative path since it let the user decrypt just the content of the vault. Now if we do *ls* we will have as output:

  importantFile.decrypted

  importantFile.encrypted

- *decrypt importantFile.encrypted /home/user/Documents*

  This decrypts the *importantFile.encrypted* in the vault and writes the decrypted file under the */home/user/Documents* instead of writing the decrypted file to the vault.

The password is hashed with the SHA-1 algorithm in order to let the user use a password bigger or smaller than 128 bit (16 characters). I decided to create a ParsingFileVaultException in order to handle user input errors and to change the flow of control of the program. By typing 'help' the user will display the last prompted menu.