Secure Web Development Teaching Modules[1]

Introduction to Cryptography

Contents

1 Concepts

Secure communications on the Internet or web is the foundation of network security and web security. Cryptography is the practice and study of how to hide information from potential enemies, hackers or the public. The sender encrypts a message with a small piece of secret information (*key*), and then sends the encrypted message to the receiver. The receiver decrypts the encrypted message with a small piece of secret information (a key that is same or different from the key used by the sender) and recovers the original message. People who don't have the right keys would not be able to read the message even if they steal a copy of the encrypted version.

There are two categories of cryptographic systems: single key symmetric ciphers or dual-key public key ciphers.

1.1 Symmetric Secret Key Ciphers

With this approach (also called *conventional ciphers*), the sender and the receiver use the same *secret key* (secret information) to encrypt and decrypt messages. Many algorithms can do both encryption and decryption. The popular symmetric key algorithms (ciphers) include DES and AES. When the input plain data is long, they divide the data into equal-sized data blocks (except the last block) and encrypt/decrypt the successive data blocks with the same algorithm and key. In this lab you will learn how to use GPG (GNU Privacy Guard), the open-source version of PGP (Pretty Good Privacy), to experiment with symmetric key encryption/decryption.

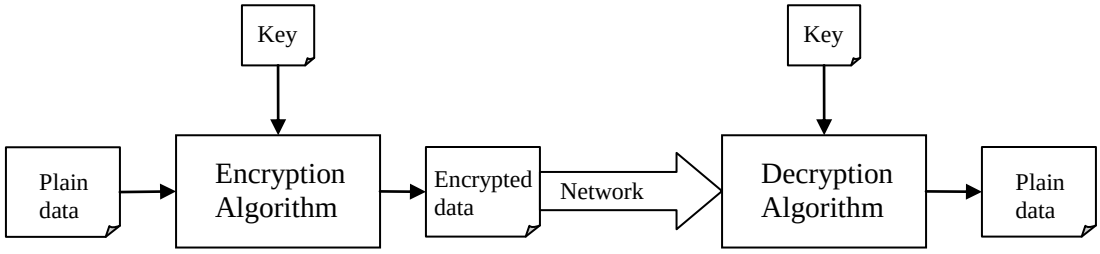


Figure 1 Data encryption/decryption

Symmetric key ciphers are also called secret key ciphers. They are much more efficient than public key ciphers described next.

1.2 Public Key Ciphers

With this approach a pair of public key and private key will be generated together. You can use either of the two keys to encrypt the plain data, and then use the other key to decrypt the encrypted data. For example you can encrypt data with the public key and use the private key to decrypt the data. RSA and Diffie-Hellman are the two most widely used public key algorithms

Typically, the key owner will keep the private key and distribute the corresponding public key to his/her potential communication partners. There are two typical application scenarios:

1. Author and contents validation. If the key owner needs to distribute a message to his friends and assure them the message is really originated from the owner without modification by any third parties, the owner would encrypt the message with the private key. If the receivers could decrypt the message with this owner's public key, they know that the message is really sent by that owner and the message has not been modified.
2. Many-to-one private messages. If a friend needs to send a private message to Bob, he could encrypt his message with Bob's public key and then send the resulting message to Bob, and only Bob, the owner of the right private key, could decrypt the private message.

Each computer maintains the public/private keys of the computer user in a file called key store, and the owner needs to set up passwords to limit the access to the key store.

Public key ciphers are less efficient than symmetric/secret key ciphers. They are mainly used for distributing the secret keys used by symmetric key ciphers, and authenticate and validate documents (here the document contents are usually not encrypted).

In this lab you will use GPG (GNU Privacy Guard), the open-source version of PGP (Pretty Good Privacy), to experiment with public/private key creation and public key encryption/decryption.

1.3 Hash Function and Digital Signature

While you could use public/private key pairs to authenticate the author of a message and validate the contents of the message, it would be slow if the message is long. Digital signatures are designed to make author and contents validation more efficient. When you digitally sign a document, you normally (not necessary) keep the document in plain form so everyone could read it, and you append a digital signature, which is a small piece of data, to the end of the plain document so the receiver could validate the author and validity of the public document if necessary.

You first need to compress the variable-length document into a short fixed-length string (popularly called *fingerprint*, *digest* or *hash code*). You use a hash function to do so. A hash function reads a long document, and produces a fixed-length short string, called *fingerprint* (*hash code* or *digest*), so that each bit of the fingerprint depends on as many bits of the input document as possible. Even though not possible in theory, in practice the hash function establishes a one-to-one mapping between the plain documents and the fingerprints with high probability: if someone modifies the plain document, its fingerprint would differ. The application of a hash function on the same document always generates the same fingerprint. SHA-1 and MD5 are both examples of hash functions. While MD5 uses 128 bits for fingerprints, SHA-1 uses 160 bits for fingerprints so it is less likely to produce the same fingerprint from two different files. In this lab you will learn how to use SHA1 and MD5 to generate fingerprints (sums) of files so you could be sure whether the downloaded large files have been compromised. There are also hash functions SHA224, SHA256, SHA384 and SHA512, which are all variants of SHA1 and use more bits for fingerprints to reduce the chance of fingerprint collision (different files have the same fingerprint).

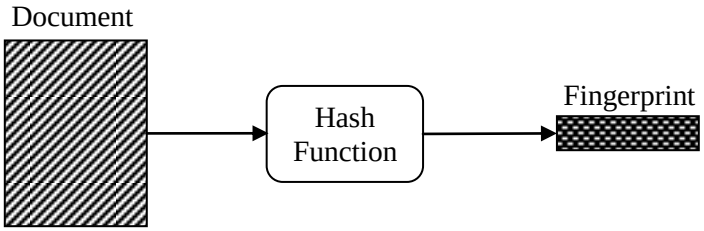


Figure 2 Hash function

Figure 3 shows how a document is digitally signed by its author. A hash function reduces the document into a fixed-size fingerprint, which is then encrypted by the author's private key into a digital signature. The digital signature is then appended to the end of the original document for distribution.

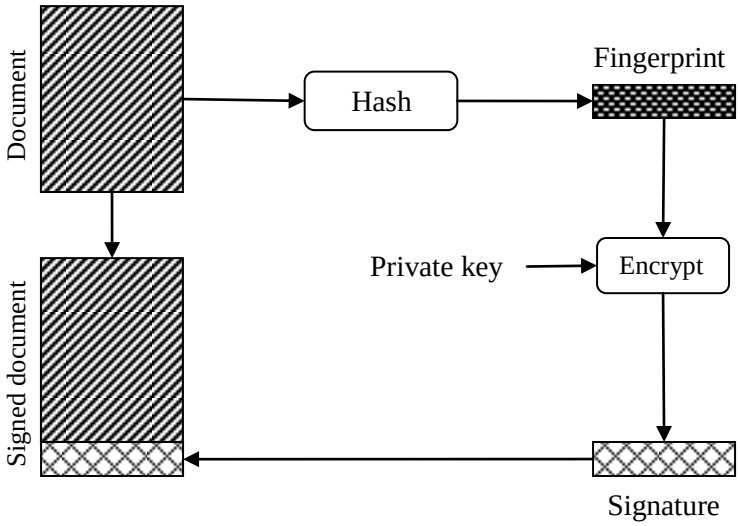


Figure 3 Digital signing of a document

Figure 4 shows how the signed document is authenticated for its author and validated against any compromises. The signed document is separated into the plain document and digital signature two parts. The same hash function maps the plain document to its fingerprint, which is then compared with the fingerprint decrypted from the received signature with the author's public key. If the two fingerprints are the same, then the document is from the author and its contents are intact, otherwise the document has been compromised.

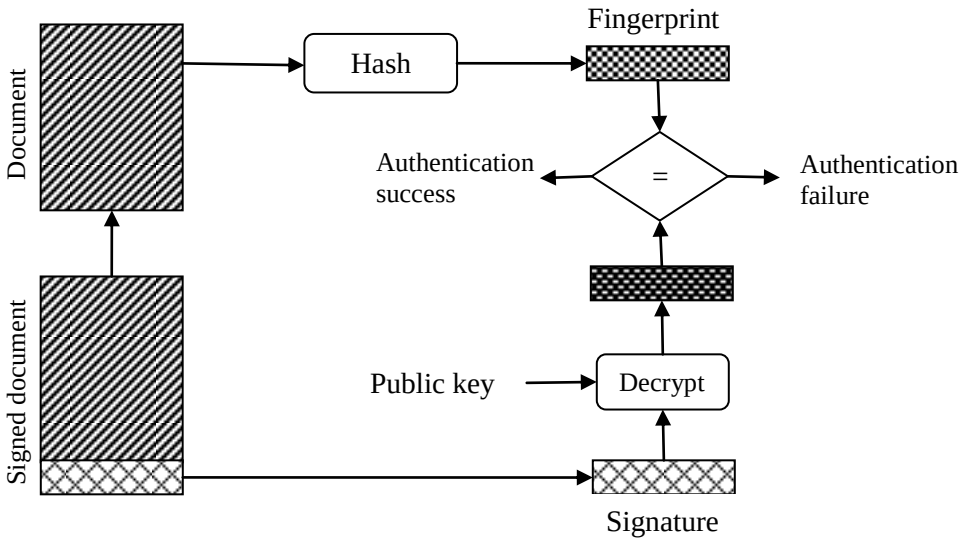


Figure 4 Authentication and validation of a signed document

1.4 Digital Certificates

One challenge of using public key ciphers is how to assure the public that the public keys they receive are actually those from their owners. Hackers could forge a public key and then distribute it under another person's name.

Digital certificates are designed to solve this problem. A few certificate authority (CA) companies are set up and supposed to be trusted by the public (can we really totally trust them or their employees?). VeriSign, GTE and U.S. Postal Service are a few example CAs. These CAs have generated their public/private key pairs, and distributed their public keys to the public computers in some "safe" way including hardcoding them in OS distributions (are they really safe?). When a person needs to distribute his public key to the public, he needs to apply to one of the CAs to create a digital certificate for him. A digital certificate includes the following plain text information: certificate format version number, certificate serial number with the CA, algorithm and parameters for signing the certificate, CA name, period of validity of the certification, the name of the person or company for which this certificate distribute its public key, the public key of the person of company that requested for this certificate, the algorithm and parameters to use the public key, and the digital signature of the above plain contents signed with the CA's private key. This certificate signing process is exactly the one described in Figure 3 where document is replaced with the above certificate plain information. When a computer receives such a digital certificate, if it finds that the certificate is correctly signed by the CA (with a process described in Figure 4), then the public key and its owner's name will be added to its key store and the certificate owner becomes a trusted entity of the computer.

But be aware that anyone can apply for a digital certificate with a CA. There are three classes of digital certificates. Class 1 certificates will be issued by a CA as long as the applicant has a valid email address. Class 2 certificate applications also need to go through an *automated* address check (a postal letter will be sent to the applicant to warn the creation of the digital certificate). Only for class 3 certificates that the applicants really need to make in-person appearances to produce ID documents as well as business records for organizations.

2 Lab Objectives

In this lab you will

1. Learn and practice how to use MD5 and SHA1 to generate hash codes of strings or large files, and verify whether a downloaded file is valid;
2. Learn and practice how to use GPG to encrypt/decrypt files with symmetric algorithms;
3. Learn and practice how to use GPG to generate public/private key pairs and certificates, distribute the certificate with public key to a friend, let the friend encrypt a document with the public key, and let the key owner decrypt the document with the private key.

3 Lab Setup

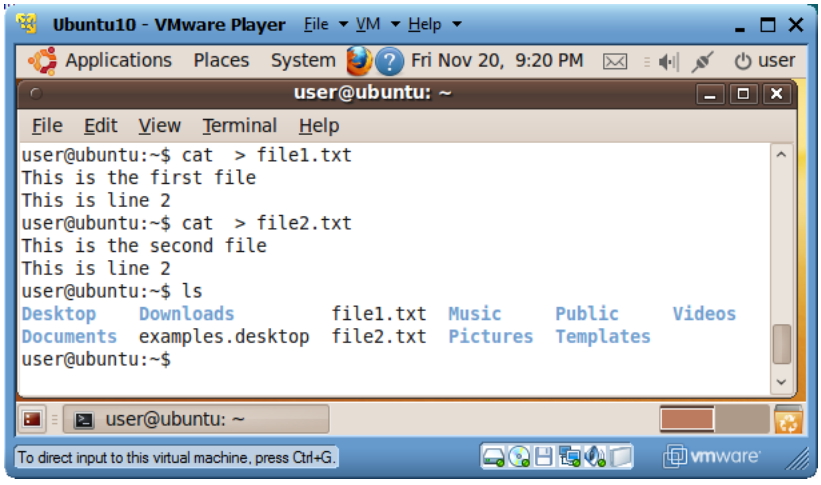
1. You will use the *ubuntu 10* virtual machine for SWEET teaching modules.
2. Extract the virtual machine from *ubuntu10tm.exe*.
3. Under the folder *ubuntu10tm*, double click on *ubuntu10tm.vmx* to start the virtual machine.

4. The username is “user” and the password is “123456”.

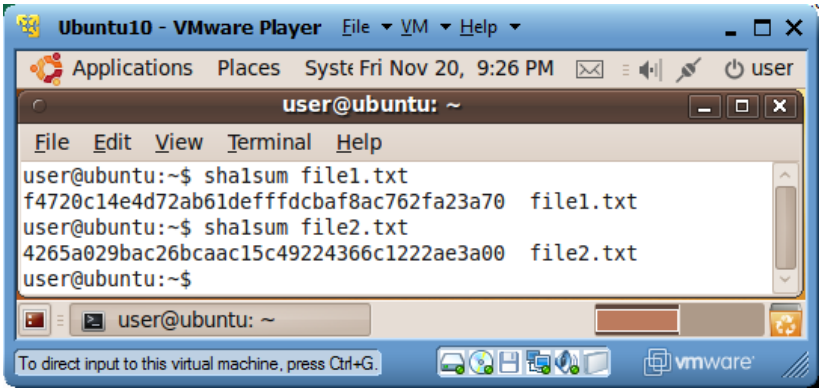
4 Lab Guide

4.1 Hashing Files with MD5 and SHA-1

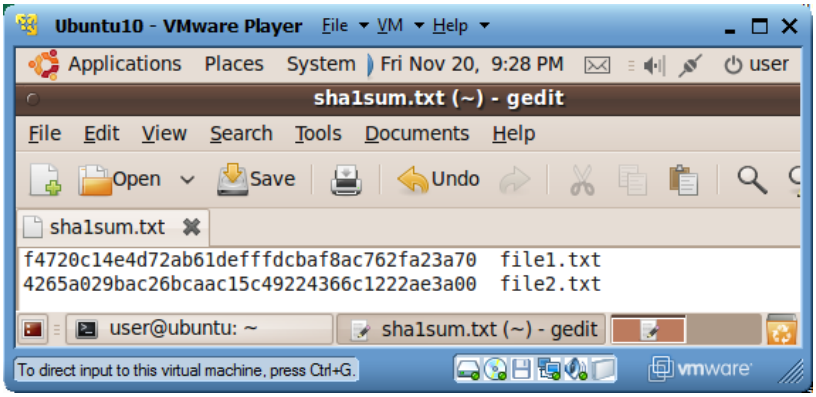
1. Start a terminal window in home folder ~ with menu item “Applications|accessories|Terminal”.
2. Create the first file “file1.txt” by typing
   $ cat > file1.txt [Enter]
   This is the first file[Enter]
   This is line 2[Enter]
   [Ctr-d]
3. Create the second file “file2.txt” by typing
   $ cat > file2.txt [Enter]
   This is the second file[Enter]
   This is line 2[Enter]
   [Ctr-d]



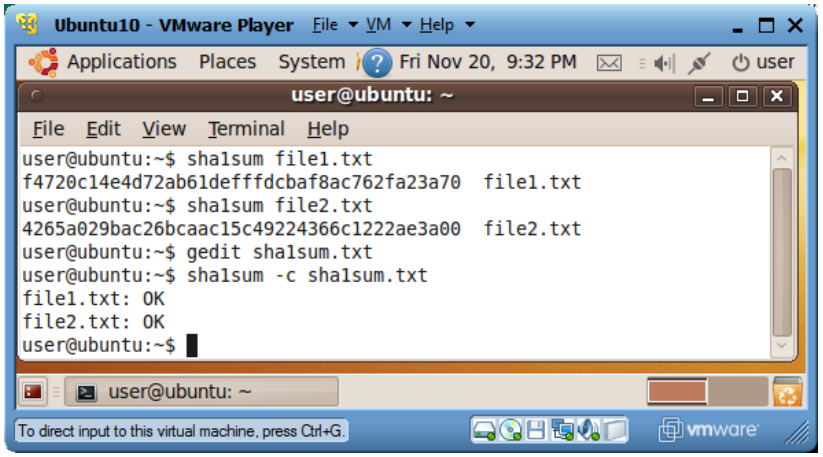
4. Run “sha1sum file1.txt” and “sha1sum file2.txt” to generate the hash codes (sums) for the two files. Each execution generates a line of two entries. The second entry is a file name, and the first entry is the hash code of the contents of the file whose name is the second entry. The hash codes are printed in hexadecimal.



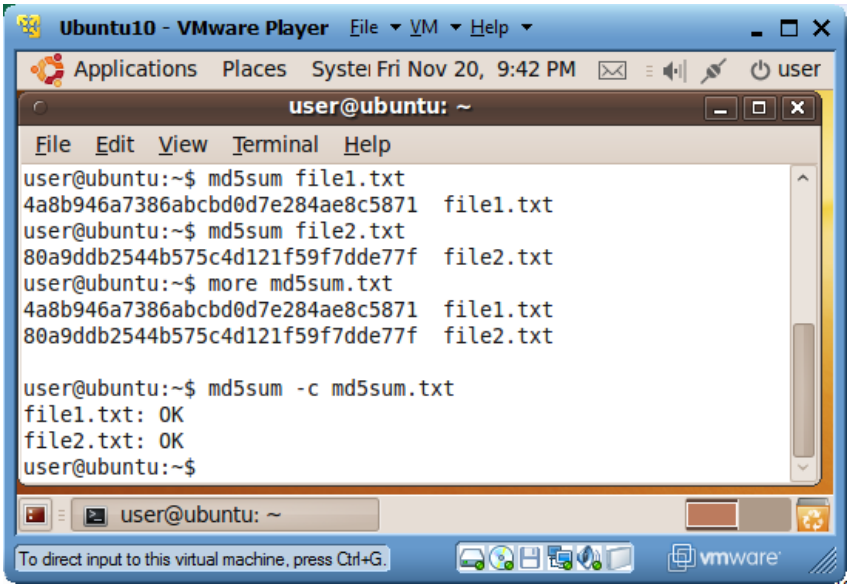
5. Run "gedit sha1sum.txt" to create a new text file "sha1sum.txt", and copy the two output lines of the last step into this file. Save the file.



6. Run "sha1sum -c sha1sum.txt". In this case program "sha1sum" will read file "sha1sum.txt". For each line in this file, it will check whether the SHA1 hash code generated for the contents of the second entry (file) is the same as the first entry (SHA1 hash code calculated beforehand). If they match, the program will print out OK for the file.



7. Now redo steps 4, 5 and 6 but replacing command "sha1sum" with "md5sum" and replacing file name "sha1sum.txt" with "md5sum.txt". You will notice that the MD5 hash codes are shorter and they serve the same purpose of file contents validation.



When you download large files, like ISO disk image files, you should download their corresponding MD5Sum or SHA1SUM files so you could check whether the downloaded files are valid or corrupted.

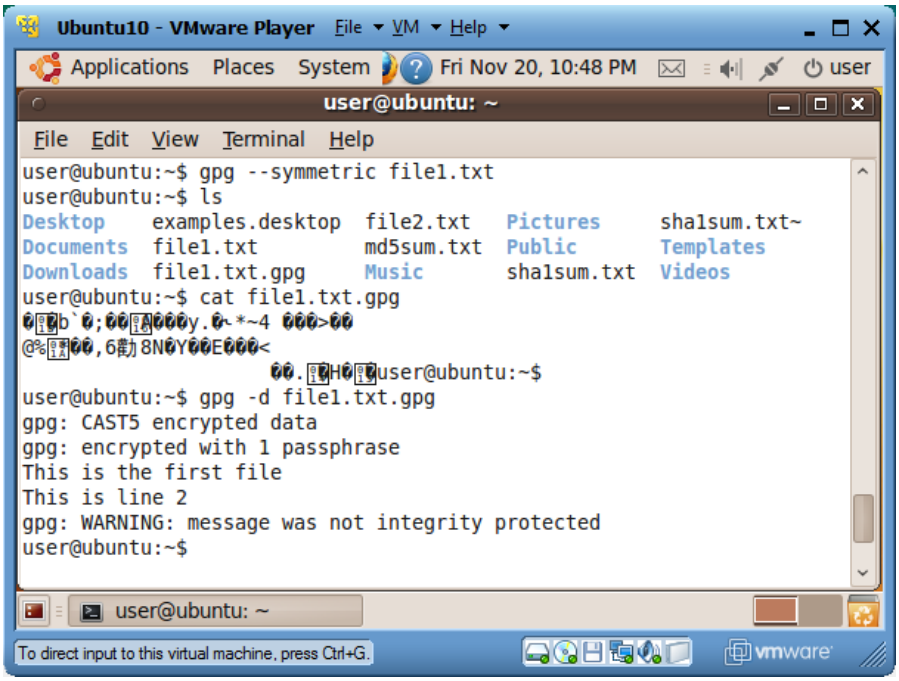
Question 1: Are all MD5 hash codes for different files of the same length?

Question 2: Do different files always lead to different MD5 or SHA1 hash codes?

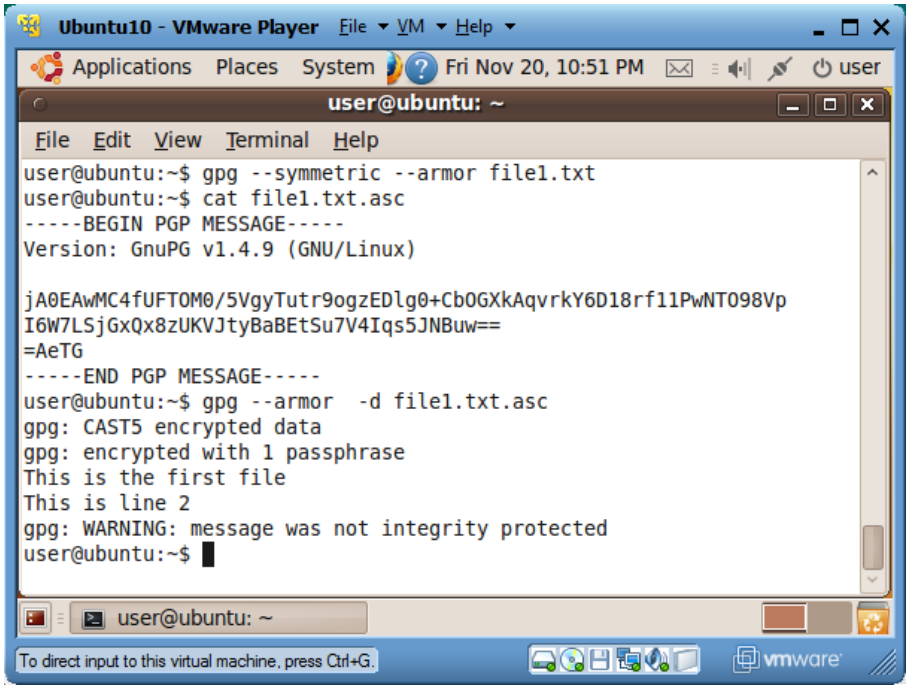
Question 3: In what sense SHA1 is better than MD5?

4.2 Symmetric Key Encryption/Decryption with GPG

1. Launch the Ubuntu-Master Install VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|accessories|Terminal".
3. If you have not created file "file1.txt" yet, do so by following step 3 of the last exercise.
4. Run "gpg --symmetric file1.txt" to encrypt file 'file1.txt". You will be prompted to enter the passphrase, it is 123456. The encrypted version is in file "file1.txt.gpg".
5. Run "cat file1.txt.gpg" to review the contents of file "file1.txt.gpg".
6. Run "gpg -d file1.txt.gpg" to decrypt the file. You will be prompted to enter the passphrase, it is 123456.



7. If you need to paste the encrypted data in email body instead of using email attachment, then you can run "gpg --symmetric --armor file1.txt" to generate the encrypted data in text form in file "file1.txt.asc". You will be prompted to enter the passphrase, it is 123456.
8. Run "cat file1.txt.asc" to review the contents of file "file1.txt.asc".
9. To decrypt file "file1.txt.asc", run "gpg --armor -d file1.txt.asc". You will be prompted to enter the passphrase, it is 123456.



GPG uses a strong cipher CAST5 to do symmetric encryption so it is much more resistant to attack than using WinZip. However the passphrase is the weak point: the longer and more complex the passphrase, the more secure the file. A single dictionary word can be brute forced in only a few hours, so use a complex passphrase of multiple words broken up with letters and symbols.

Question 4: If you receive a secret message in an email body and the message is encrypted by a command like "gpg --symmetric --armor message.txt", which steps should you take to recover the message?

4.3 Public/Private Key Creation and Encryption/Decryption

4.3.1 Basic Concepts of PGP (GPG) Digital Certificates and Public Key Ciphers

*PGP (Pretty Good Privacy) is a c*omputer program that provides cryptographic privacy and authentication. PGP supports public/private key pairs to implement secure data communications between communicating parties. GPG (GNU Privacy Guard) is the open-source version of PGP.

Suppose Mike needs to send a secure message in file, say *msg-to-Alice*, to Alice so that no other people can read the message. Both Mike and Alice need to have used the following command to generate their own public/private key pairs:

gpg --gen-key

Each of them will be prompted to enter a name, an email address, and a comment, which together make the person's *user-ID*, in form of "name (comment) email-address", capable of identifying the person. Suppose Mike uses email address mike@pace.edu, and Alice uses email address alice@pace.edu. Each person will also be prompted to enter a *passphrase* to protect his/her private key. Each time a person uses his/her private key, he/she needs to enter the passphrase to prove his/her ownership to the private key.

Alice needs to export her public key into a file, say "*alice-pk*", with a command like

gpg --armor --output alice-pk --export alice@pace.edu

Alice needs to send her public key (in file "*alice-pk*") to Mike in any secure way, like with a USB flash disk and making sure no substitution or modification of the public key file in the key distribution process. Mike now needs to import Alice's public key with the following command:

gpg --import alice-pk

Then Mike can use the following command to encrypt file *msg-to-alice* into a new file *secret-to-alice*:

gpg --recipient alice@pace.edu --output secret-to-alice --encrypt msg-to-alice

Now Mike can send file *msg-to-alice* to Alice in any way and only Alice could decrypt the file with command

gpg --output msg-from-mike --decrypt secret-to-alice

The decrypted message is now in file *msg-from-mike*. This command will only work if the computer has the private key of Alice.

You can also use command "gpg --list-keys" to list all public keys on your system, and use commands like "gpg --delete-key marge@pace.edu" to delete the selected public keys from your system.

In the above process, it is critical for Mike to be sure that the imported public key from Alice is from its real owner Alice. The *fingerprint* of a public key is an easier-to-compare short string uniquely identifying a public key. Alice and Mike can independently generate the fingerprints of Alice's public key and compare them in a trusting way like over the phone or in person. If the fingerprints are the same, Mike could *sign* Alice's public key to claim that he trusts the validity of the key, and this signing process will insert Mike's user ID in the key's signature list. To work on the tasks described in this paragraph, Mike could first run command "gpg --edit-key alice@pace.edu" to enter the key-editing user interface for Alice's public key, then use sub-command "fpr" to generate the signature for Alice's public key, use sub-command "sign" to sign Alice's public key with Mike's private key, and use sub-command "check" to review the key's list of endorsing signatures for the validity of Alice's public key.

From the above discussion we can see that for Mike to send a secure message to Alice, Alice needs to inform Mike of her public key as well as her email address used to generate the public key. Since the email address is part of the public key and listed when Mike imports Alice's public key, actually Alice only needs to pass her public key to Mike.

4.3.2 A Detailed Lab Guide for GPG

This lab exercise guides you to practice the above PGP (GPG) concepts with GPG in our Ubuntu-Master Install VM.

1. Create Linux Accounts for Alice and Mike

   - Launch your Ubuntu-Master Install VM, and start a terminal window.
   - Run command "sudo adduser alice" to create a Linux account for Alice. Use 123456 as password.
   - Run command "sudo adduser mike" to create a Linux account for Mike. Use 123456 as password.
   - Run command "sudo visudo" to launch file "/etc/sudoers.tmp" in a text editor, insert the following two lines at the end of the file, and then use Crtl+O to write out the revised contents, and use Ctrl+X to exit the editor. This step will enable Alice and Mike to use "sudo".

     ```
     alice   ALL=(ALL) NOPASSWD: ALL
     mike  ALL=(ALL) NOPASSWD: ALL
     ```

2. Run as Alice and Mike in two terminal windows

   - In the terminal window, run "sudo login", and then login as Alice.
   - Start a new terminal window, run "sudo login", and then login as Mike.

3. Generate keys for Alice

   - In Alice's terminal window, run "gpg --gen-key" to generate her public and private keys. Enter "DSA and Elgamal" for key kind, 2048 for key size, "key does not expire" for key expiration date, "Alice" for real name, alice@pace.edu for email address, "Alice's keys" as comment, and "Alice's passphrase" for passphrase. You may need to type over 284 random keys to generate enough entropy so the keys could be created.

4. Generate keys for Mike

   - In Mike's terminal window, run "gpg --gen-key" to generate his public and private keys. Enter "DSA and Elgamal" for key kind, 2048 for key size, "key does not expire" for key expiration date, "Michael" for real name, mike@pace.edu for email address, "Mike's keys" as comment, and "Mike's passphrase" for passphrase. You may need to type over 284 random keys to generate enough entropy so the keys could be created.

5. Export Alice's public key to Mike

   - In Alice's terminal window, run "gpg --armor --output alice-pk --export alice@pace.edu" to dump Ali's public key in file "alice-pk". You can run "more alice-pk" to review the public key.

   - Run "sudo cp alice-pk /home/mike" to copy Alice's public key file "alice-pk" to Mike's home folder.

   - In Mike's terminal window, verify the existence of file "/home/mike/alice-pk" by running "ls" in Mike's home folder ~ (/home/mike).

   - In the same Mike's terminal window, run "gpg --import alice-pk" to import Alice's public key into Mike's key store.

   - In the same Mike's terminal window, run "gpg --edit-key alice@pace.edu" to enter the editing session for Alice's public key. Type sub-command "fpr" to review the fingerprint of Alice's public key. Type sub-command "sign" to sign this key with Mike's key. You will be asked to enter Mike's passphrase, which is "Mike's passphrase". Type sub-command "check" to review who is on the signature list of Alice's public key, and we will see Alice (self-signature) and Mike on the list to confirm the validity of the key. You type sub-command "quit" to exit the editing session, and confirm to save the changes.

6. Create and encrypt a message

   - In Mike's terminal window, run "cat > msg-to-alice" followed by the ENTER key, type "Alice's secret message", and then type key combination Ctrl+D to close the file. You just created a new text file "msg-to-alice" with contents "Alice's secret message".

   - In Mike's terminal window, run "gpg --recipient alice@pace.edu --output secret-to-alice --encrypt msg-to-alice" to generate a new file "secret-to-alice" containing the encrypted version file "msg-to-alice".

- In Mike's terminal window, run "more secret-to-alice" to review the encrypted version of the message.

- In Mike's terminal window, run "sudo cp secret-to-alice /home/alice" to copy file "secret-to-alice" to Alice's home folder "/home/alice".

- In Alice's terminal window, run "ls" in Alice's home folder ~ (/home/alice) to verify the existence of file "secret-to-alice".

7. Decrypt the message

   - In Alice's terminal window, run command "gpg --output msg-from-mike --decrypt secret-to-alice" to decrypt the contents of file "secret-to-alice" and save the result in a new file "msg-from-mike".

   - In Alice's terminal window, run "more msg-from-mike" to review the decrypted message from Mike.

5 Review Questions

Question 5: Suggest some secure ways for distributing symmetric or public keys.

Question 6: Is email a secure way for distributing symmetric or public keys.

Question 7: Suppose Tom has the public key of Lisa. What is the best way for Tom to send his public key to Lisa?

Question 8: Suppose Tom has the public key of Lisa. What is the best way for Tom to send a secret message to Lisa?

Question 9: If a Word file is digitally signed, is the file also normally encrypted so it cannot be eavesdropped?

Question 10: Can you totally trust a company if that company has a digital certificate signed by VeriSign?

Question 11: Can technologies alone completely solve the network or web security problems?