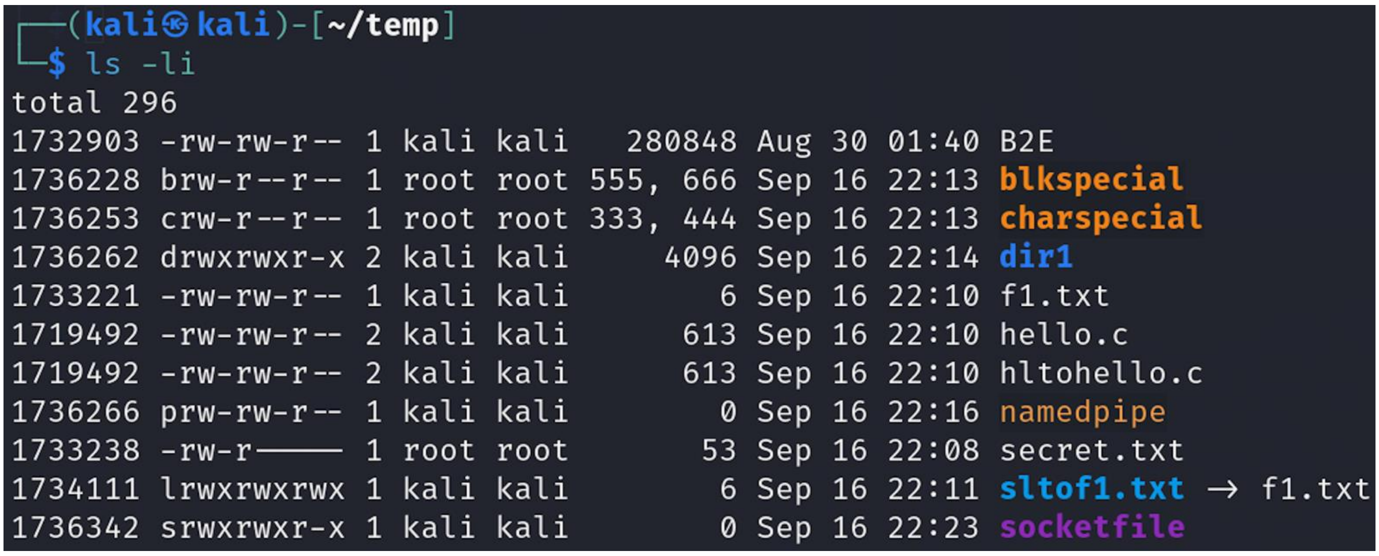
Operating System Security

File Permissions Related Commands

Every file stored on the file system of a computer must be secured. No unauthorize person should be able to access it. UNIX traditional privilege scheme defines what actions a user or process can perform on files, directories, and other system resources. The following screenshot shows the long listing of a directory contents having seven different file types in Linux:

`inode:perm:linkcount:owner:group:size:time:name`



```
┌──(kali㉿kali)-[~/temp]
└─$ ls -li
total 296
1732903 -rw-rw-r-- 1 kali kali   280848 Aug 30 01:40 B2E
1736228 brw-r--r-- 1 root root 555, 666 Sep 16 22:13 blkspecial
1736253 crw-r--r-- 1 root root 333, 444 Sep 16 22:13 charspecial
1736262 drwxrwxr-x 2 kali kali     4096 Sep 16 22:14 dir1
1733221 -rw-rw-r-- 1 kali kali        6 Sep 16 22:10 f1.txt
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hello.c
1719492 -rw-rw-r-- 2 kali kali      613 Sep 16 22:10 hltohello.c
1736266 prw-rw-r-- 1 kali kali        0 Sep 16 22:16 namedpipe
1733238 -rw-r----- 1 root root       53 Sep 16 22:08 secret.txt
1734111 lrwxrwxrwx 1 kali kali        6 Sep 16 22:11 sltof1.txt -> f1.txt
1736342 srwxrwxr-x 1 kali kali        0 Sep 16 22:23 socketfile
```

- **Protection in Linux**
    - **Users**: Each user in a Linux system is assigned a unique User ID (UID). Usernames and UIDs are stored in the `/etc/passwd` file. Users cannot read, write, or execute files owned by others without proper permission.
    - **Groups**: Users are also assigned to groups with unique Group IDs (GIDs), stored in the `/etc/group` file. Each user has a private group by default but can belong to other groups for additional access. Members of a group can share files that belong to that group.
- **Three Classes of Users**
    - **User/Owner**: The owner is the user who created the file. Any file you create is owned by you.
    - **Group**: The owner can grant access to the file to members of a designated group.
    - **Others**: The owner can also provide access to all other users on the system.
- **File and Directory permissions in Linux:** File permissions in UNIX/Linux have different implications for files and directories:
    - **For Files**
        - **Read (r)**:
            - Allows users to open and read the file's contents.
            - Commands used: `less, more, head, tail, cat, grep, sort, view`

- **Write (w)**:
  - Allows users to open and modify the file’s contents.
  - Editors used: `vi, vim, peco, nano`
- **Execute (x)**:
  - Allows users to run the file as a program or script.
  - Example: `./script.sh`

- **For Directories**
  - **Read (r)**:
    - Allows users to list the contents of the directory.
    - Command used: `ls.`
  - **Write (w)**:
    - Allows users to create new files and directories, as well as delete files they own within the directory.
    - Commands used: `mkdir, touch, cp, rm.`
  - **Execute (x)**:
    - Allows users to access the directory and perform actions within it, such as searching or changing into it. Without Execute. Permissions, read/write permissions on directory are ineffective.
    - Command used: `cd`.

- **Changing File Ownership:** The `chown` command is used to change ownership of files and directories.
  - To change the owner of a file `myfile.txt` to arif:
    ```
    # chown arif myfile.txt
    ```
  - To change the owner to arif and the group to developers for a directory `mydir` and its contents:
    ```
    # chown -R arif:developers mydir
    ```
- **Changing File Permissions:** The `chmod` command is used to change existing permissions of files and directories.
  - Set read (r), write (w), execute (x) permissions for the owner, and read and execute permissions for the group and others on a file `script.sh`
    ```
    # chmod 755 script.sh
    ```
    or
    ```
    # chmod u=rwx,g=rx,o=rx script.sh
    ```
  - Add execute permission for the owner of a file `myfile.txt`
    ```
    # chmod u+x myfile.txt
    ```
  - To remove write permission for the group from a directory mydir:
    ```
    # chmod g-w mydir
    ```
  - To set the SUID bit on a program `myprog`:
    ```
    # chmod u+s myprog
    ```
- **Special permissions in Linux**
  - **SUID bit**
    1. When a program has this bit set, it runs with the privileges of the program's owner.
    2. The SUID bit is typically set for executable programs.
    3. It is denoted by an 's' in the owner's execute permission or a capital 'S' if the owner's execute permission is off.
    4. For instance, the `passwd` program has its SUID bit set, allowing users to modify the /etc/shadow file owned by root.
    5. To identify executable files with the SUID bit set:
       ```
       $ find / -type f -perm -u=s -ls 2> /dev/null
       ```

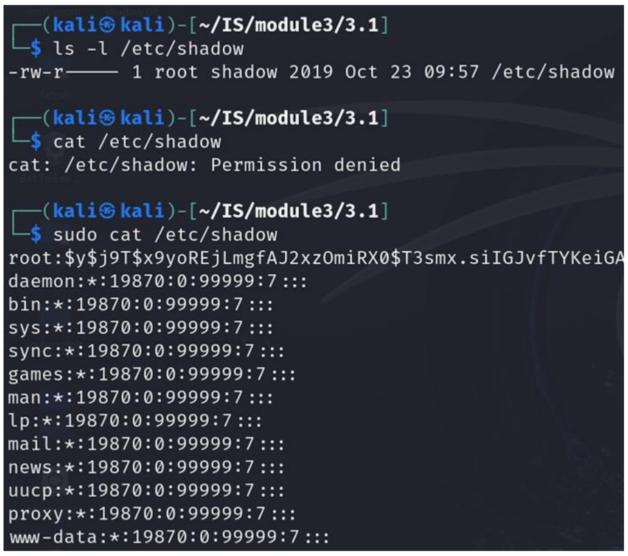
- **SGID bit**
    1. When a program has this bit set, it runs with the privileges of the program's group.
    2. The SGID bit is set for both executable programs and directories.
    3. It is indicated by an 's' in the group's execute permission or a capital 'S' if the group's execute permission is off.
    4. For example, the `chage` program has its SGID bit set, allowing users to modify the /etc/shadow file.
    5. To find executable files with the SGID bit set:

    ```
    $ find / -type f -perm -g=s -ls 2> /dev/null
    ```

    6. SGID bit on directories is useful in shared group environments. Any file created within a directory with the SGID bit set inherits the group membership of that directory.
- **Sticky bit (On Directories):**
    1. If a directory with full permissions has this bit set, users cannot delete each other's files.
    2. It is indicated by a 't' in the others execute permission or a capital 'T' if the others' execute permission is off.
    3. For example, the /tmp directory has its sticky bit set, preventing users from deleting each other's files.
    4. To identify directories with the sticky bit set:

    ```
    $ find / -type d -perm -o=t -ls 2> /dev/null
    ```

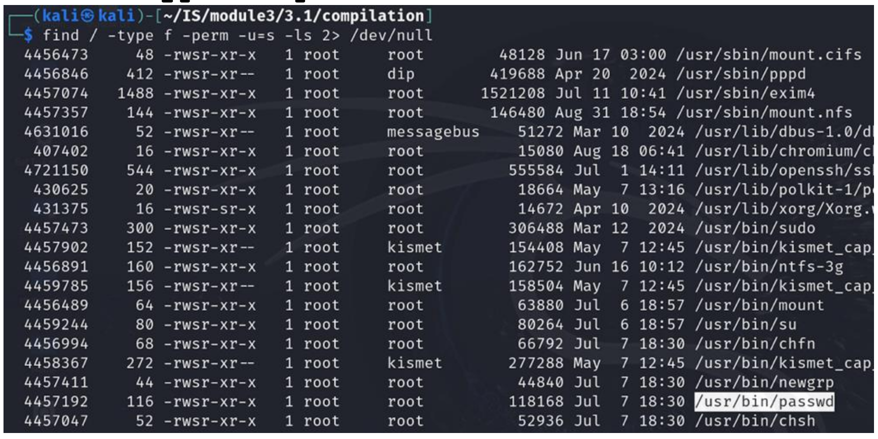
- **Understanding SUID Privileged Programs:**
We all know that the `/etc/shadow` file is a critical file in Unix-like operating systems that stores user account information, specifically the hashed passwords and related security attributes for user accounts. The `/etc/shadow` file is usually only accessible by the `root` user and certain privileged processes. This is to protect sensitive information from unauthorized access. From the screenshot, one can see that the permissions on this file are set to `640`, meaning that the owner (`root`) has read and write permissions, the group (`shadow`) has only read permissions, and others have no access. So, a regular user cannot even view the content of this file. However, this can be done by a user who is in the `sudo` group, and since the user `kali` is a member of this group, so he/she can view its contents using the `sudo` command as shown in the screenshot.



```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -l /etc/shadow
-rw-r----- 1 root shadow 2019 Oct 23 09:57 /etc/shadow

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ cat /etc/shadow
cat: /etc/shadow: Permission denied

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ sudo cat /etc/shadow
root:$y$j9T$x9yoREjLmgfAJ2xzOmiRX0$T3smx.siIGJvfTYKeiGA
daemon:*:19870:0:99999:7:::
bin:*:19870:0:99999:7:::
sys:*:19870:0:99999:7:::
sync:*:19870:0:99999:7:::
games:*:19870:0:99999:7:::
man:*:19870:0:99999:7:::
lp:*:19870:0:99999:7:::
mail:*:19870:0:99999:7:::
news:*:19870:0:99999:7:::
uucp:*:19870:0:99999:7:::
proxy:*:19870:0:99999:7:::
www-data:*:19870:0:99999:7:::
```

Now a 100$ question is how come every user can use the `passwd` command to change his/her password, that resides in this `/etc/shadow` file. One way is to run a privileged daemon running at all times that can do this job for regular users, but this is of course expensive. So, the solution used by all UNIX systems for such tasks is the SUID bit. The SUID (Set User ID) bit is a special file permission in Unix-like operating systems that allows users to execute a file with the permissions of the file owner rather than the permissions of the user executing the file. The advantage of SUID bit is that it allows users to perform tasks that require higher privileges without giving them full access to the system. It is denoted by an 's' in the owner's execute permission or a capital 'S' if the owner's execute permission is off. An example of a program having its SUID bit set is the `passwd` program, that is used by regular users to change their own password by modifying the contents of `/etc/shadow` file owned by root. To identify executable files with the SUID bit set:

```
$ find / -type f -perm -u=s -ls 2> /dev/null
```



```
┌──(kali㉿kali)-[~/IS/module3/3.1/compilation]
└─$ find / -type f -perm -u=s -ls 2> /dev/null
  4456473     48 -rwsr-xr-x   1 root     root        48128 Jun 17 03:00 /usr/sbin/mount.cifs
  4456846    412 -rwsr-xr--   1 root     dip        419688 Apr 20  2024 /usr/sbin/pppd
  4457074   1488 -rwsr-xr-x   1 root     root      1521208 Jul 11 10:41 /usr/sbin/exim4
  4457357    144 -rwsr-xr-x   1 root     root       146480 Aug 31 18:54 /usr/sbin/mount.nfs
  4631016     52 -rwsr-xr--   1 root     messagebus    51272 Mar 10  2024 /usr/lib/dbus-1.0/d
   407402     16 -rwsr-xr-x   1 root     root          15080 Aug 18 06:41 /usr/lib/chromium/c
  4721150    544 -rwsr-xr-x   1 root     root         555584 Jul  1 14:11 /usr/lib/openssh/ss
   430625     20 -rwsr-xr-x   1 root     root          18664 May  7 13:16 /usr/lib/polkit-1/p
   431375     16 -rwsr-sr-x   1 root     root          14672 Apr 10  2024 /usr/lib/xorg/Xorg.
  4457473    300 -rwsr-xr-x   1 root     root         306488 Mar 12  2024 /usr/bin/sudo
  4457902    152 -rwsr-xr--   1 root     kismet       154408 May  7 12:45 /usr/bin/kismet_cap
  4456891    160 -rwsr-xr-x   1 root     root         162752 Jun 16 10:12 /usr/bin/ntfs-3g
  4459785    156 -rwsr-xr--   1 root     kismet       158504 May  7 12:45 /usr/bin/kismet_cap
  4456489     64 -rwsr-xr-x   1 root     root          63880 Jul  6 18:57 /usr/bin/mount
  4459244     80 -rwsr-xr-x   1 root     root          80264 Jul  6 18:57 /usr/bin/su
  4456994     68 -rwsr-xr-x   1 root     root          66792 Jul  7 18:30 /usr/bin/chfn
  4458367    272 -rwsr-xr--   1 root     kismet       277288 May  7 12:45 /usr/bin/kismet_cap
  4457411     44 -rwsr-xr-x   1 root     root          44840 Jul  7 18:30 /usr/bin/newgrp
  4457192    116 -rwsr-xr-x   1 root     root         118168 Jul  7 18:30 /usr/bin/passwd
  4457047     52 -rwsr-xr-x   1 root     root          52936 Jul  7 18:30 /usr/bin/chsh
```

How Operating Systems Implement Security

Operating systems implement security through a variety of mechanisms designed to protect data, prevent unauthorized access, and ensure system integrity. OSs continuously evolve to address emerging threats and vulnerabilities, integrating new technologies and approaches to maintain robust security. Some of the key methods are mentioned below:

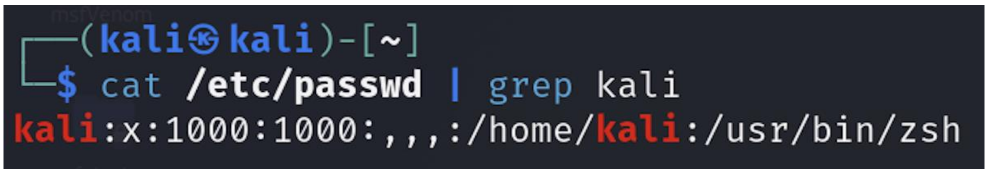
- **User Authentication:**
    - Passwords: A common method requiring users to provide a secret passphrase.
    - Multi-Factor Authentication (MFA): Combines something you know (password), something you have (security token), and something you are (biometric) to enhance security.
- **Access Control Models:**
    - Discretionary Access Control (DAC): In DAC, owner of the object decides who all have what all kinds of access (rwx) on his object. This is the default for most Operating Systems.
    - Mandatory Access Control (MAC): In MAC, system decides who all have what all kinds of access (rwx) on the objects. Every object has a security classification associated with it (e.g, top secret, secret, confidential, restricted, unclassified). Every user has a security clearance to access a specific class of object.
    - Role-Based Access Control (RBAC): Users are assigned roles, and roles have specific permissions, simplifying management and enhancing security.
- **Access Control:**
    - UNIX Traditional privilege Scheme: Defines what actions a user or process can perform on files, directories, and other system resources. Permissions are usually categorized as read, write, and execute. The UID is compared with the object owner ID, group ID and the permissions are granted accordingly else other permissions are granted to the user.
    - Access Control Lists (ACLs): Provide a more granular control over access permissions by associating specific permissions with individual users or groups.
- **Encryption:**
    - Data Encryption: Protects data at rest (stored data) and in transit (data being transferred across networks) using algorithms like AES (Advanced Encryption Standard).
    - File System Encryption: Encrypts the entire file system to protect data from unauthorized access.
- **Kernel Security:**
    - Privilege Separation: Ensures that the kernel and user space are separated to prevent unauthorized access to kernel-level operations.
    - Secure Boot: Verifies the integrity of the operating system at startup to prevent boot-level malware.
    - Data Execution Prevention (DEP): Prevents code from being executed in certain regions of memory.
    - Address Space Layout Randomization (ASLR): Randomizes memory addresses used by system and application processes to make it harder for attackers to predict targets.
- **Auditing and Logging:**
    - Activity Logs: Record events and user actions to monitor and detect suspicious activities.
    - Audit Trails: Provide a detailed history of system access and changes to help in forensic investigations and compliance.
- **Firewalls and Network Security:**
    - Firewalls: Control incoming and outgoing network traffic based on security rules.
    - Intrusion Detection Systems (IDS): Monitor network and system activities for malicious activities or policy violations.

- **Isolation and Sandboxing:**
    - Process Isolation: Ensures that processes run in their own memory space and cannot interfere with each other.
        - Sandboxing: Restricts the execution of code within a controlled environment to prevent it from affecting the rest of the system.
- **Virtualization and Containerization:**
    - Virtual Machines (VMs): Isolate applications and services in separate VMs to reduce the risk of compromise affecting the entire system.
        - Containers: Offer a lightweight form of isolation for running applications in segregated environments.

How and Where Linux OS Store Passwords?

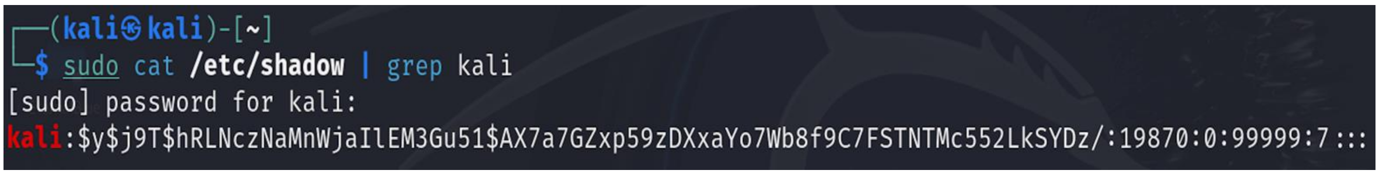
- Across different operating systems, password security is primarily handled through hashing algorithms and encryption techniques to protect against unauthorized access. While the specifics can vary—such as the use of NTLM in Windows, SHA-512 in Linux, or AES-256 in macOS—the common goal is to ensure that password data is securely managed and protected from compromise. On almost all modern Linux distributions, user passwords are hashed and stored in the **`/etc/shadow`** file. Access to this file is restricted to root users, and additional security measures like disk encryption and Pluggable Authentication Modules (PAM) modules help to protect and manage authentication securely.
- The early version of UNIX in 1971, used to store the hashed value of passwords in the second field of the world readable **`/etc/passwd`** file. This was because this file contains user related information other than passwords and many applications require that information to function properly. In the later versions of UNIX, and in today's Linux distros, the hashed password is saved in the **`/etc/shadow`** file, which is readable only by super users.
- The screenshot of the contents of the `/etc/passwd` file on my Kali Linux machine, with its seven self-explanatory fields is shown below:

```
loginname:en_passwd:UID:GID:GECOS:homedir:shell
```

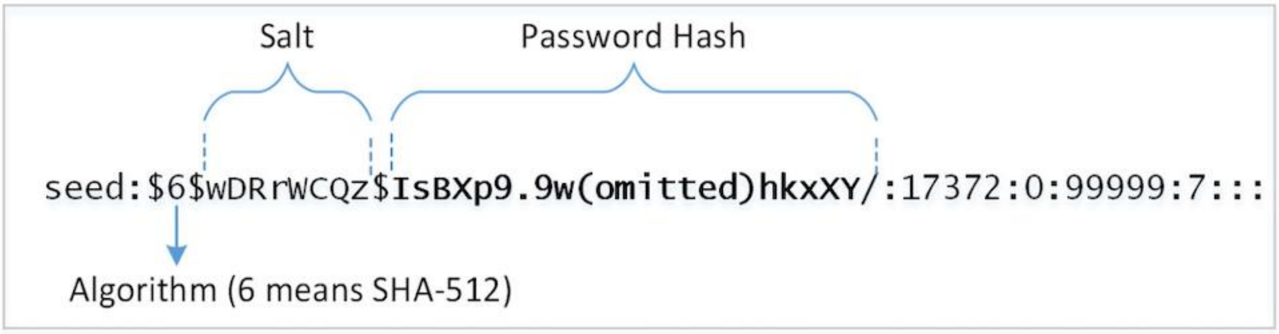


```
┌──(kali㉿kali)-[~]
└─$ cat /etc/passwd | grep kali
kali:x:1000:1000:,,,:/home/kali:/usr/bin/zsh
```

- Similarly, the screenshot of the contents of the **`/etc/shadow`** file on my Kali Linux machine, with its nine fields is also shown below. Every row contains one record having nine colon separated fields:



```
┌──(kali㉿kali)-[~]
└─$ sudo cat /etc/shadow | grep kali
[sudo] password for kali:
kali:$y$j9T$hRLNczNaMnWjaIlEM3Gu51$AX7a7GZxp59zDXxaYo7Wb8f9C7FSTNTMc552LkSYDz/:19870:0:99999:7:::
```


Salt
Password Hash
seed:$6$wDRrWCQz$IsBXp9.9w(omitted)hkxXY/:17372:0:99999:7:::
Algorithm (6 means SHA-512)

Every row contains one record having nine colon separated fields. Here is the breakdown of each field:

`user:$y$salt$hash:lastchanged:min:max:warn:inactive:expire:`

1. **`kali:`** Username
2. **`$ID$salt$hash:`** Indicates the hashing algorithm used, salt and hash
   - **`ID:`** This is a string that indicates the hashing algorithm used. Common values are:
     - `$1$` is MD5.
     - `$2a$` or `$2y$` is blowfish.
     - `$5$` for SHA-256.
     - `$6$` for SHA-512.
     - `$y$` for yescrypt.
   - **`Salt:`** It prevent two users with the same password from having duplicate entries in the /etc/shadow file. If user1 and user2 both has set their passwords as 'pucit', their encrypted passwords will be different because their salts will be different.
   - **`Hash:`** The salt and the un-encrypted password are combined and encrypted using the specific algorithm to generate the encrypted hash of the password that is saved here.
3. **`Lastchanged (19870):`** The date of the last password change, expressed as the number of days since Jan 01 1970 (UNIX epoch). If this field is empty, it might imply that the user hasn't changed their password since the account was created.
4. **`Minimum (0):`** Minimum password age specifies the minimum number of days required between password changes. An empty field or a zero means that password aging features are disabled.
5. **`Maximum (99999):`** Maximum password age specifies the maximum number of days that a password is valid. After this period, the user will be prompted to change their password.
6. **`Warn (7):`** Password warning period specifies the number of days before the password expires that the user will start receiving warnings about the upcoming expiration.
7. **`Inactive:`** Password inactivity period specifies the number of days after a password expires during which the account remains usable. After this period, the account will be disabled if the password isn't changed.
8. **`Expire:`** Account expiration date specifies the number of days since January 1, 1970, when the user account will expire. After this date, the user account will be disabled. If this field is empty, the account does not have an expiration date.
9. **Reserve:** This field is reserved for future use and is generally left empty in most implementations.

- The login process in the early UNIX versions was as follows:

```
if (md5($submitted_passowrd) == $stored_password_hash)
      login()
else
      display_msg("Wrong Password")
```

Some Basic Cryptographic Terms and Linux Tools to use them

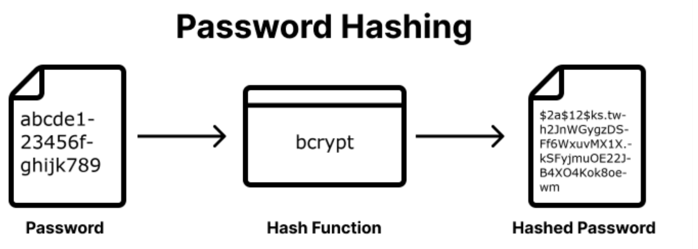
Encoding: It is a process of converting data to another format to be used on different device or system using a publicly known algorithm and without the need of a key to encode or decode. The main goal is not security but ensuring that data can be correctly interpreted or transferred. One of the main characteristics of encoding is its reversibility, i.e., the encoded data can be easily decoded back to its original form if you know the encoding scheme used. Some examples of different encoding schemes are octal, hex, base64, ASCII, UTF-8, and so on

- **Base 64 Encoding:** Encodes and decodes data in Base64 format, which is commonly used to encode binary data as ASCII text.
  - Encode a string to base64
    ```
    echo -n "Hello World" | base64
    ```
  - Decode a string from base64
    ```
    echo "SGVsbG8gV29ybGQ=" | base64 -d
    ```
  - Encode a file to base64
    ```
    base64 inputfile.txt > encodedfile.txt
    ```
  - Decode a Base64 file
    ```
    base64 -d encodedfile.txt > decodedfile.txt
    ```

- **xxd Encoding:** Creates a hex dump of a file or standard input, or converts a hex dump back into binary
  - Create hex dump of a string and display on screen
    ```
    echo -n "Hello World" | xxd
    ```
  - Create hex dump of a file and save the output in a file hello.dump
    ```
    xxd hello.c > hello.dump
    ```
  - Convert a hexdump back to its ASCII representation
    ```
    xxd -r hello.dump
    ```

Hashing: Hashing is used to verify data integrity. It transforms data into a fixed-size string of characters, which is typically a digest. Hashing is used to store passwords and also ensuring data integrity while downloading files from websites. The two main characteristics of good hashing algorithms are that they are irreversible (one way function) and deterministic (same input will always produce the same hash output)

- **Hashing using OpenSSL:** OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) network protocols and related cryptography standards required by them. The **openssl** is a command line program for using the various cryptography functions of OpenSSL's crypto library from the shell. It has many uses, but most common are:
    - Calculation of Hashes
    - Symmetric Encryption
    - Public/Private key generation
    - Asymmetric Encryption.

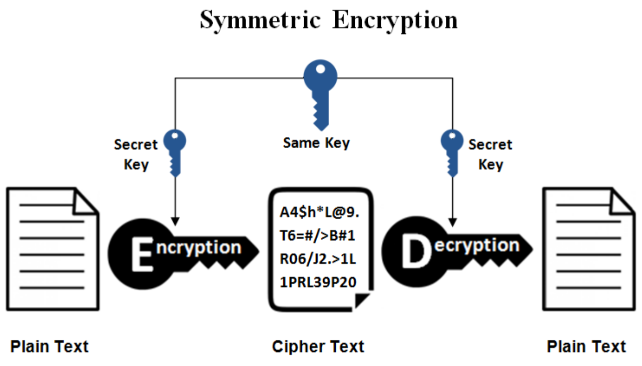


```
$ openssl version
OpenSSL 3.3.3 4 Jun 2024 (Library:OpenSSL 3.2.2 4 Jun 2024

$ openssl dgst -help
$ echo "password" > file.txt

$ openssl dgst -md5 ./file.txt
MD5(file.txt)= 286755fad04869ca523320acce0dc6a4       (128 bits hash)

$ echo "password" | openssl dgst -md5
MD5(stdin)= 286755fad04869ca523320acce0dc6a4          (128 bits hash)
```

The `openssl` command has many sub-commands like `version`, `base64`, `enc`, `dgst`, `ciphers`, `dsa`, `rsa`, and so on. The `openssl dgst` command (Message Digest Calculation), we can use many different hashing algorithms. In the above examples I have used the algorithm message digest 5 (`md5`) that will create a 128 bit hash. You can use the `sha1` (160), `sha256` (256 bit hash), `sha384` (384 bit hash), `sha512` (512 bit hash), and so on.

Encryption: It is used to protect data confidentiality. It is a process of scrambling data to make it decipherable only by the intended recipient. The goal is to ensure that only those with the correct decryption key can read or access the original data. One of the main characteristics of encoding is its reversibility, i.e., the encrypted data can be decrypted back to its original form if you have the decryption key. Some examples of different encoding schemes are octal, hex, base64, ASCII, UTF-8, and so on. There are two major categories of encryption that are taught in cryptography, namely symmetric encryption and asymmetric encryption.

- **Symmetric Encryption:** Symmetric encryption uses the same key for both encryption and decryption. This means that both parties must share the same secret key. The security of symmetric encryption depends on keeping this key secret. Some famous symmetric encryption algorithms are:
    - Data Encryption Standard (DES with 56 bits key)
    - Triple Data Encryption Standard (3DES with 168 bits key)
    - Blowfish (32 bit to 448 bit)
    - Advance Encryption Standard (AES with 128, 192, 256 bits key)

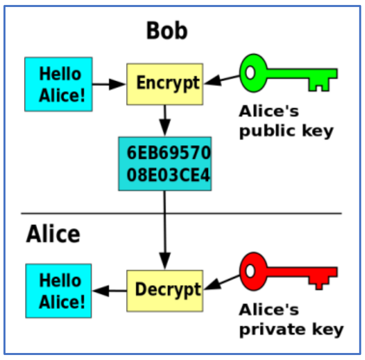


```
$ openssl enc -help

$ openssl enc -list

$ echo "hello world" > f1.txt

$ openssl enc -des -in f1.txt -out encrypted_des -K 0123456789abcdef -iv a0b0c1d4e5f27891

$ cat encrypted_des

$ openssl enc -des -d -in encrypted_des -out decrypted_des -K 0123456789abcdef -iv a0b0c1d4e5f27891

$ cat decrypted_des
```

Note:

- The **-K** argument specifies the encryption key as a hexadecimal string. DES requires a key that is exactly 8 bytes long (16 hexadecimal characters). If you provide a key that is longer or shorter, OpenSSL will either truncate it or pad it, potentially compromising security.
- The **-iv** argument specifies the initialization vector (IV) for certain modes of operation (like CBC - Cipher Block Chaining). It ensures that the same plaintext block will encrypt to different ciphertext blocks, providing additional security. For DES, the IV must also be 8 bytes long (16 hexadecimal characters).
- For the decryption process, one has to provide the same key and initialization vector that was used during the encoding process
- Limitation of symmetric encryption is, "How the sender and receiver share the encryption key?"

- **Asymmetric Encryption:** Asymmetric encryption uses a pair of keys: a public key for encryption and a private key for decryption. The keys are mathematically linked, but knowing the public key does not help in deriving the private key. Some famous asymmetric encryption algorithms are Rivest-Shamir-Adleman (RSA), Digital Signature Standard (DSS), and Pretty Good Privacy (PGP).



  - **Rivest-Shamir-Adleman (RSA):** is the first public key cryptosystem and widely used for secure data transmission. The key length is typically 2048 4096 bits for security. It can be used for both encryption and digital signature.
    - **Step 1: (Generate an RSA Key pair)**
      - Choose Two Secret Numbers: Start by selecting two large, secret prime numbers. These will be the foundation of the keys.
      - Create a Public Key: From these two numbers, generate a public key that anyone can use to encrypt messages meant for you. This public key consists of a large number (the product of the two primes) and another number that helps with encryption.
      - Create a Private Key: Also generate a private key, which is kept secret. This key is used to decrypt messages that were encrypted with your public key.

```
$openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out private_key.pem
$openssl rsa -pubout -in private_key.pem -out public_key.pem
```

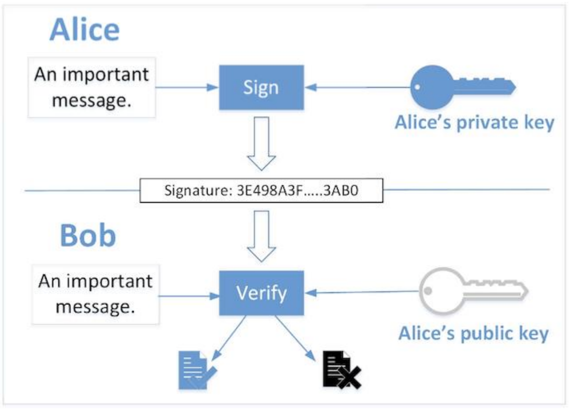
    - **Step 2: (Sender Encrypts with the Public Key of Receiver)**

```
$openssl pkeyutl -encrypt -inkey public_key.pem -pubin -in f1.txt -out encrypted.bin
```

    - **Step 3: (Receiver Decrypts with the his/her Private Key)**

```
$openssl pkeyutl -decrypt -inkey private_key.pem -in encrypt.bin -out decrypted.txt
```

Digital Signatures: Ensures that MiTM cannot do impersonation. It is an electronic mean of verifying some ones identity. It also use two keys. The sender signs the document using his/her private key and the recipient verifies it using the sender public key which answers in yes or No.



- **Step 1: (Generate an RSA Key pair)**
- **Step 2: (Sender Generate Signature with his/her Private Key)**
- **Step 3: (Receiver Verifies Signature. With sender's Public Key)**

Limitation is what if the sender or receiver denies that he/she has received the message. The sender may change his/her private:public key pair. Solution is Digital Certificates.

Digital Certificates: It is similar to Digital Signatures, but involves 3rd parties (Certificate Authorities) like GoDaddy, DigiCert, GlobalSign. Digital certificates are used to authenticate the identity of (individual, organization, webservers) and facilitate key exchange for encryption and verification of digital signature. You get your Digital Certificate from a CA and make it public. Rom this digital certificate a browser can extract the public key and use it to encrypt the data to be sent to server. Now you cannot say that I have not sent that message.

CyberChef:

CyberChef is a open source application also called the **"Cyber Swiss Army Knife"**. That can be run locally or accessed online at https://cyberchef.io. It is a simple, intuitive tool that can be used to carry out a variety of cyber operations. Most common of our interest are shown below:

- Encoding like Base64, Hexadecimal, Morse Code and so on.
- Symmetric Encryption like DES, 3DES, AES, Blowfish and so on.
- Asymmetric Encryption like RSA, DSS, PGP , and so on.
- Generating hashes like md5, sha1, sha256, sha384, sha512, and so on.
- Compression and decompression of data like zip, gzip, LZ4, and so on.