**Module 10 – Authentication, Access control and System commands**

**Learning objectives:**

1) Describe common authentication techniques, attacks and prevention methods

2) Describe common access control systems and their applications

3) Explain common system commands for authentication and access control

Contents

[**1. Authentication** 2](#_Toc90166132)

[**2. Access control and policy** 8](#_Toc90166133)

[**Discretionary Access Control** 9](#_Toc90166134)

[**Mandatory Access Control** 10](#_Toc90166135)

[**Role-based Access Control** 11](#_Toc90166136)

[**3. System admin commands** 11](#_Toc90166137)

[**References** 16](#_Toc90166138)

# **1. Authentication**

When logging on to a computer you enter user name and password.

The first step is called identification: You announce who you are.

The second step is called authentication; You prove that you are who you claim to be.

Authentication is the process of identifying users that request access to a system, network, or device. Access control often determines user identity according to credentials like username and password. Other authentication technologies like biometrics and authentication apps are also used to authenticate user identity.

User authentication is a method that keeps unauthorized users from accessing sensitive information. For example, User A only has access to relevant information and cannot see the sensitive information of User B.

Cybercriminals can gain access to a system and steal information when user authentication is not secure. The data breaches companies like Adobe, Equifax, and Yahoo faced are examples of what happens when organizations fail to secure their user authentication.

Hackers gained access to Yahoo user accounts to steal contacts, calendars and private emails between 2012 and 2016. The [Equifax data breach](https://www.ftc.gov/enforcement/cases-proceedings/refunds/equifax-data-breach-settlement) in 2017 exposed credit card data of more than 147 million consumers. Without a secure authentication process, any organization could be at risk.

Cybercriminals always improve their attacks. As a result, security teams are facing plenty of authentication-related challenges. This is why companies are starting to implement more sophisticated [incident response](https://www.exabeam.com/incident-response/steps/) strategies, including authentication as part of the process. The list below reviews some common authentication methods used to secure modern systems.

**Password-based authentication**

Passwords are the most common methods of authentication. Passwords can be in the form of a string of letters, numbers, or special characters. To protect yourself you need to create strong passwords that include a combination of all possible options.

Passwords should be secrets shared between the user and the system authenticating the user. In an enterprise, users can collect their password personally. Otherwise, the password could be sent by mail, email, or phone, or entered by the user on a web page. You now have to consider who might intercept the message and who might actually pick it up.

* + E.g., a letter containing the password for an online bank account might be stolen or an impersonator may phone in asking for another user’s password.

If you have forgotten your password but are in the middle of an important task you need instant help. Procedures for resetting passwords are the same as listed previously, but now reaction should be instant.

* + Global organisations must staff a hot desk round the clock,
  + On a web site, auxiliary information may authenticate a user: mother’s maiden name, phone number, name of pet, etc.

Password support can become a major cost factor. Staff at hot desk needs proper security training.

**Attacks on passwords:**

* Exhaustive search (brute force): try all possible combinations of valid symbols up to a certain length.
* Intelligent search: search through a restricted name space, e.g. passwords that are somehow associated with a user like name, names of friends and relatives, car brand, car registration number, phone number,…, or try passwords that are generally popular.
* Typical example for the second approach: dictionary attack trying all passwords from an on-line dictionary.
* You cannot prevent an attacker from accidentally guessing a valid password, but you can try to reduce the probability of a password compromise.

**Defense against password hack:**

* Change default passwords: often passwords for system accounts have a default value like “manager”.
  + Default passwords help field engineers installing the system; if left unchanged, it is easy for an attacker to break in.
  + Would it then be better to do without default passwords?
* Avoid guessable passwords:
  + Prescribe a minimal password length.
  + Password format: mix upper and lower case, include numerical and other non-alphabetical symbols.
  + Today on-line dictionaries for almost every language exist.
* Password ageing: set an expiry dates for passwords to force users to change passwords regularly.
* Prevent users from reverting to old passwords, e.g. keep a list of the last ten passwords used.
* Limit login attempts: the system can monitor unsuccessful login attempts and react by locking the user account (completely or for a given time interval) to prevent or discourage further attempts.
* Inform user: after successful login, display time of last login and the number of failed login attempts since, to warn the user about recently attempted attacks.

**Password policy**

Users are forced to use long passwords, mixing upper and lower case characters and numerical symbols, generated for them by the system, and changed repeatedly

* + Users may have difficulty memorizing complex passwords.
  + Users may have difficulty dealing with frequent password changes.
  + Users may find ways of re-using their favourite password.

Passwords will be written on a piece of paper kept close to the computer.

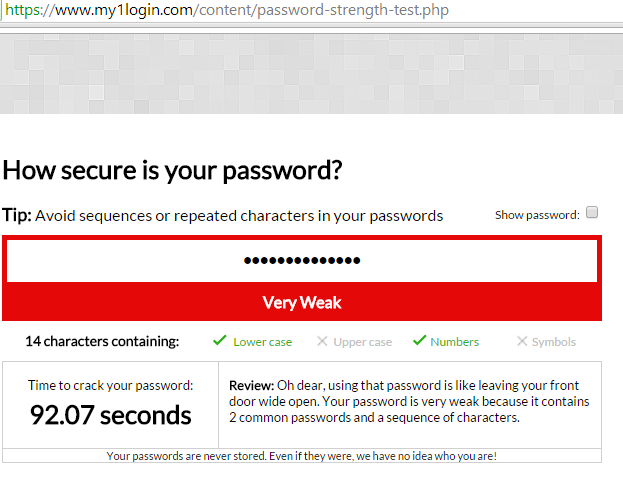
* + Security experts routinely look out for passwords on notes posted on computer terminals.
  + Is it a bad idea to write down your password?

People are best at memorizing passwords they use regularly. Passwords work reasonably well in situations where they are entered quite frequently, but not so with systems used only occasionally.

* Good advice:
  + When changing a password, type it immediately several times.
  + Do not change passwords before weekends or holidays.

**Password Strength Measure**:

* Entropy is an useful metric to measure how strong a password is (NIST 2006)
* Try out this website https://www.my1login.com/content/password-strength-test.php
  + It provides an indication how weak or strong a password is along with expected break time



**Spoofing attack - Phishing**

Attacker starts a program that presents a fake login screen and leaves the computer. If the next user coming to this machine enters username and password on the fake login screen, these values are captured by the program. Login is then typically aborted with a (fake) error message and the spoofing program terminates. Control returned to operating system, which now prompts the user with a genuine login request. To address this, two safeguards are used.

* Trusted path: guarantee that user communicates with the operating system and not with a spoofing program; e.g., Windows has a secure attention key CTRL+ALT+DEL for invoking the operating system logon screen.
* Mutual authentication: user authenticated to system, system authenticated to user.

**Protecting the password file**

Operating system maintains a file with user names and passwords located at /etc/password. Attacker could try to compromise the confidentiality or integrity of this password file. Options for protecting the password file includes below:

* + Store password in encrypted format (cryptography)
  + access control enforced by the operating system,
  + combination of cryptographic protection and access control, possibly with further measures to slow down dictionary attacks.

**One way function:** For cryptographic protection we can use one-way functions (cryptographic hash functions).

Definition: A one-way function *f* is a function that is relatively easy to compute but hard to reverse.

* + Given an input *x* it is easy to compute *f*(*x*), but given an output *y* it is hard to find *x* so that *y* = *f*(*x*)
* Instead of the password *x*, the value *f*(*x*) is stored in the password file; when a user logs in entering a password *x’*, the system applies the one-way function *f* and compares *f*(*x’*) with the expected value *f*(*x*).

**One way hash function example**

* hash( 'abc' ) = ascii('a')+ascii('b')+ascii('c')

= 97 + 98 + 99

= 294

* Given the hash value of 294, can you tell what the original string was?
* Obviously not, because 'abc' and 'cba' (and countless others) give the same hash value.
* Cryptographic hash functions work the same way, except that the algorithm is much more complex.
* If string s hashes to h, then it is very difficult to construct another string that also hashes to h

**Example steps of authentication in real-world applications**

* A hash-based account system is as follows:
  + The user creates an account.
  + Their password is hashed and stored in the database. At no point is the plain-text (unencrypted) password ever written to the hard drive.
  + When the user attempts to login, the hash of the password they entered is checked against the hash of their real password (retrieved from the database).
  + If the hashes match, the user is granted access. If not, the user is told they entered invalid login credentials.
  + Steps 3 and 4 repeat every time someone tries to login to their account.

**Hash function example:**

* Hash function converts any amount of data into a fixed-length "fingerprint" that cannot be reversed.
* If the input changes by even a little, the resulting hash is completely different
  + hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
  + hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
* The hash functions used to implement data structures are designed to be fast, not secure.
* Cryptographic hash functions are used to implement password hashing
  + SHA256, SHA512, RipeMD, and WHIRLPOOL are examples of cryptographic hash functions.

**John The Ripper (password cracker tool) - http://www.openwall.com/john/**

**Password salting:** Following common practice we refer to *f*(*x*) as the encrypted password; to be precise, it is the hash of *x*. To slow down dictionary attacks, a salt is appended to the password before encryption and stored with the encrypted password. If two users have the same password, they will now have different entries in the file of encrypted passwords.

* + $salt1 = "13$13aVc!kd";
  + $salt2 = "4kr$!vlmeoc";
  + $salted = $salt1 . $userinput . $salt2;
  + $encrypted = encrypt($salted);

**Single Sign-on:** Having to remember many passwords for different services is a nuisance; with a single sign-on service, you have to enter your password only once. E.g., ADP portal of KSU (subscribe to benefit, receive HR notification, update address/profile, get tax slips). A simplistic single-sign on service could store your password and do the job for you whenever you have to authenticate yourself. Such a service adds to your convenience but it also raises new security concerns. System designers have to balance convenience and security; ease-of-use is an important factor in making IT systems really useful, but many practices which are convenient also introduce new vulnerabilities.

**More on Authentication**

* If you are dissatisfied with the level of security provided by passwords, what else can you do?
* In general, the following options are open.
* You can be authenticated on the basis of
  + something you know,
  + something you hold,
  + who you are,
  + what you do,
  + where you are.

**Something you know:**

* The user has to know some secret to be authenticated.
* Examples: password, personal identification number (PIN), personal information like home address, date of birth, name of spouse (used e.g. by banks to authenticate customers on the phone).

**Something you hold:**

* User presents a physical token to be authenticated.
* Examples: keys, cards or identity tags (access to buildings), smart cards.
* Physical tokens can be lost or stolen.
* Anybody who is in possession of the token has the same rights as the legitimate owner.
* To increase security, physical tokens are often used in combination with something you know, e.g. bank cards come with a PIN or with a photo of the user.

**Who you are:**

* Biometric schemes use unique physical characteristics (traits, features) of a person such as face, finger prints, iris patterns, hand geometry (maybe even DNA at some time in the future).
* We will use the example of fingerprints to sketch how biometric authentication works.
* Biometric schemes are still quite new; it has to be seen whether results from experiments conducted in controlled environments are a good indicator for practical performance.

**Fingerprint:**

* Enrolment: reference sample of the user’s fingerprint is acquired at a fingerprint reader.
* Features are derived from the sample.
  + Fingerprint minutiae: end points of ridges, bifurcation points, core, delta, loops, whorls, …
* For higher accuracy, record features for more than one finger.
* Feature vectors are stored in a secure database.
* When the user logs on, a new reading of the fingerprint is taken; features are compared against the reference features.
* Biometrics are used for two purposes:
  + Identification: 1:*n* comparison tries to identify the user from a database of *n* persons.
  + Verification: 1:1 comparison checks whether there is a match for a given user.
* Authentication by password: clear reject or accept at each authentication attempt.
* Biometrics: stored reference features will hardly ever match precisely features derived from the current measurements.

# **2. Access control and policy**

Access control is one of the built-in features in operating systems. In access control, we refine the notion of a principal to be one of a:

* *user:*a human
* *subject:*a process executing on behalf of a user
* *object:*a piece of data or a resource.

Common examples of access control include discretional access control, mandatory access control and role-based access control.

## **Discretionary Access Control**

A *discretionary access control*(DAC) policy is a means of assigning access rights based on rules specified by users.  This class of policies includes the file permissions model implemented by nearly all operating systems.  In Unix, for example, a directory listing might yield "... rwxr-xr-x ... file.txt", meaning that the owner of file.txt may read, write, or execute it, and that other users may read or execute the file but not write it.  The set of access rights in this example is {read, write, execute}, and the operating system mediates all requests to perform any of these actions.  Users may change the permissions on files they own, making this a discretionary policy.

A mechanism implementing a DAC policy must be able to answer the question:  "Does subject S have right R for object O?"  Abstractly, the  information needed to answer this question can be represented as a mathematical relation D on subjects, objects, and rights:   if (S,O,R) is in D, then S does have right R for object O; otherwise, S does not.  More practically, the same information could also be represented as an *access control matrix*.  Each row of the matrix corresponds to a subject and each column to an object.  Each cell of the matrix contains a set of rights.  For example:

|  |  |  |
| --- | --- | --- |
|  | file1 | file2 |
| Alice | rwx | r-x |
| Bob | r-- | rw- |

Real systems typically store the information from this matrix either by columns or by rows.  An implementation that stores by columns is commonly known as an *access control list* (ACL).  File systems in Windows and Unix typically use such an implementation:  each file is accompanied by a list containing subjects and their rights to that file.  An implementation that stores by rows is commonly known as a *capability list*, by analogy with the use of capabilities in authentication.  Each subject maintains an unforgeable list of the rights it has to objects.  Both implementations make certain questions easier to answer than others.  For example, it is easy in an ACL implementation to find the set of all subjects who may read a file, but it is difficult to find the set of all files that a subject may read.

The underlying philosophy in DAC is that subjects can determine who has access to their objects. There is a difference, though, between trusting a person and trusting a program. E.g., A gives B a program that A trusts, and since B trusts A, B trusts the program, while neither of them is aware that the program is buggy.  Suppose a subject S has access to some highly secret object O. Moreover, suppose that another subject S' does not have access to O, but would like to. What can S' do to gain access? S' can write a program that does two things, the first of which is the following sequence of commands:

* Create a new object O'.
* Grant S write access to O'.
* Grant S' read access to O'.
* Copy O to O'.

The second thing the program does is to act like a video game (or some other application that S might be interested in running). If the program is run by S, then S' will get access to the contents of O (now in O'). This type of program is referred to as a Trojan horse.  DAC mechanisms are typically insufficient to protect against Trojan horse attacks.

## **Mandatory Access Control**

A *mandatory access control*(MAC)policy is a means of assigning access rights based on regulations by a central authority.  This class of policies includes examples from both industry and government.  The philosophy underlying these policies is that information belongs to an organization (rather than individual members of it), and it is that organization which should control the security policy.  MAC policies strive to defend against Trojan horse attacks.

**MAC Example.  Domain Type Enforcement:** One way to address the problem of trusted subjects in MLS is to introduce a new kind of matrix. It looks like the access control matrix, but this is only a superficial resemblance. The rows in this matrix correspond to *domains* and the columns to *types*. Each entry contains a set of access rights. An entry [s,t] is the maximum permissions that domain s has with respect to an object of type t. In contrast to an access control matrix, this type enforcement matrix does not have commands associated with it. It cannot be manipulated by owners of objects; instead, it is controlled by system administrators. This makes it MAC, as opposed to DAC.

Note that a type enforcement matrix allows us to encode more than a lattice. For example, information flow is not necessarily transitive, and the matrix lets us express this, whereas the lattice does not. Consider the following example: a program sends data to an encryption routine that then sends encrypted data to the network. We would like an application program to be able to write to the encryption routine. The encryption routine should be able to read from the application program and write encrypted data to the network, and the network should be able to read from the encryption routine. The matrix is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Application Program | Encryption | Network |
| Application Program |  | write |  |
| Encryption | read |  | write |
| Network |  | read |  |

This matrix provides stronger constraints than simply making the encryption routine a trusted subject. A trusted subject can do what it wants, but here we make the encryption program's access rights more restrictive. Thus, if we still wish to do an analysis of the encryption program (e.g. to make sure any data that it writes is encrypted), we don't know need to worry about it writing anywhere other than to the network, so the scope of the analysis is narrowed (and therefore the analysis is easier.)

## **Role-based Access Control**

In the real world, security policies are dynamic.  Access rights, whether discretionary or mandatory, need to change as the responsibilities of users change.  This can make management of rights difficult.  When a new user is authorized for a system, the appropriate rights for that user must be established.  When a user changes job functions, some rights should be deleted, some maintained, and some added.

Role-based access control (RBAC) addresses this problem by changing the underlying subject--object model.  A *role* is a job function or title---i.e., a set of actions and responsibilities associated with a particular working activity.  Now, instead of an access control policy being a relation on subjects, objects, and rights, a policy is a relation on roles, objects, and rights; this is called a *right assignment*.  For example, the role "513 TA" might be assigned the right to grade 513 homeworks.  Further, subjects are now assigned to roles; this is called a *role assignment*.  Each subject may be assigned to many roles, and each role may be assigned to many subjects.  Finally, roles are hierarchical.  For example, the role "513 Professor" should have all the rights that a "513 TA" does, and more.

Roles are similar to groups in Unix file system DAC, with two important distinctions.  First, a group is a set of users, whereas a role is a set of rights.  Second, a user is always a member of a group, whereas a subject may activate or deactivate the rights associated with any of the subject's roles.  This enables finer-grained implementation of the Principle of Least Privilege.  Subjects may login with most of their roles deactivated, and activate a role only when the rights associated with the role are necessary.

Nearly all real-world systems (including most operating systems and database systems) implement some form of RBAC.  Either discretionary or mandatory policies can be built using RBAC as the underlying model.

# **3. System commands**

While users can be either people (meaning accounts tied to physical users) or accounts that exist for specific applications to use, groups are logical expressions of organization, tying users together for a common purpose. Users within a group share the same permissions to read, write, or execute files owned by that group.

Each user is associated with a unique numerical identification number called a user ID (UID). Likewise, each group is associated with a group ID (GID). A user who creates a file is also the owner and group owner of that file. The file is assigned separate read, write, and execute permissions for the owner, the group, and everyone else. The file owner can be changed only by root, and access permissions can be changed by both the root user and file owner.

**Reserved User and Group IDs**

Red Hat Enterprise Linux reserves user and group IDs below 1000 for system users and groups. By default, the **User Manager** does not display the system users. Reserved user and group IDs are documented in the setup package. To view the documentation, use this command:

cat /usr/share/doc/setup\*/uidgid

The recommended practice is to assign IDs starting at 5,000 that were not already reserved, as the reserved range can increase in the future. To make the IDs assigned to new users by default start at 5,000, change the UID\_MIN and GID\_MIN directives in the /etc/login.defsfile:

*[file contents truncated]*

UID\_MIN 5000

*[file contents truncated]*

GID\_MIN 5000

*[file contents truncated]*

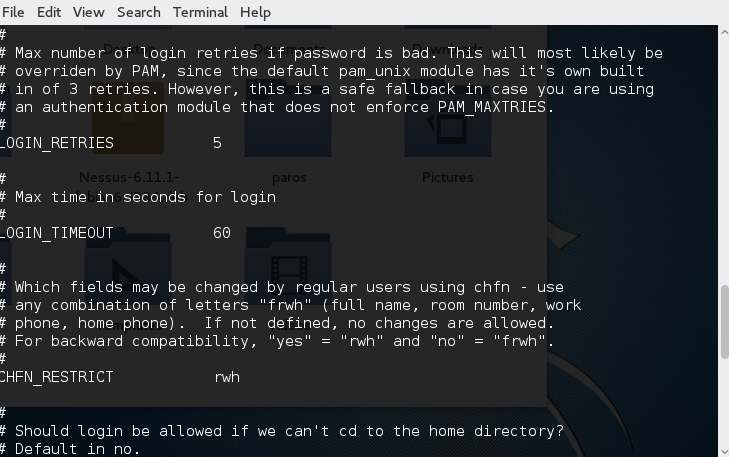
For users created before you changed UID\_MIN and GID\_MIN directives, UIDs will still start at the default 1000.

**Shadow passwords**

In environments with multiple users, it is very important to use shadow passwords provided by the shadow-utils package to enhance the security of system authentication files. For this reason, the installation program enables shadow passwords by default.

The following is a list of the advantages shadow passwords have over the traditional way of storing passwords on UNIX-based systems:

* Shadow passwords improve system security by moving encrypted password hashes from the world-readable /etc/passwd file to /etc/shadow, which is readable only by the root user.
* Shadow passwords store information about password aging.
* Shadow passwords allow to enforce some of the security policies set in the /etc/login.defs file. Below is a screenshot of a defs file.



Most utilities provided by the shadow-utils package work properly whether or not shadow passwords are enabled. However, since password aging information is stored exclusively in the /etc/shadow file, some utilities and commands do not work without first enabling shadow passwords:

* The chage utility for setting password aging parameters. For details, see the [Password Security](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Security_Guide/chap-Hardening_Your_System_with_Tools_and_Services.html#sec-Password_Security) section in the Red Hat Enterprise Linux 7 Security Guide.
* The gpasswd utility for administrating the /etc/group file.
* The usermod command with the -e, --expiredate or -f, --inactive option.
* The useradd command with the -e, --expiredate or -f, --inactive option.

**Adding a new user**

To add a new user to the system, type the following at a shell prompt as root:

useradd [options] username

By default, the useradd command creates a locked user account. To unlock the account, run the following command as root to assign a password:

passwd username

**Common useradd command-line options**

| **Option** |  |
| --- | --- |
| -c 'comment' | comment can be replaced with any string. This option is generally used to specify the full name of a user. |
| -d home\_directory | Home directory to be used instead of default /home/username/. |
| -e date | Date for the account to be disabled in the format YYYY-MM-DD. |
| -f days | Number of days after the password expires until the account is disabled. If 0 is specified, the account is disabled immediately after the password expires. If -1 is specified, the account is not disabled after the password expires. |
| -g group\_name | Group name or group number for the user's default (primary) group. The group must exist prior to being specified here. |
| -G group\_list | List of additional (supplementary, other than default) group names or group numbers, separated by commas, of which the user is a member. The groups must exist prior to being specified here. |
| -m | Create the home directory if it does not exist. |
| -M | Do not create the home directory. |
| -N | Do not create a user private group for the user. |
| -p password | The password encrypted with crypt. |
| -r | Create a system account with a UID less than 1000 and without a home directory. |
| -s | User's login shell, which defaults to /bin/bash. |
| -u uid | User ID for the user, which must be unique and greater than 999. |

The following steps illustrate what happens if the command useradd juan is issued on a system that has shadow passwords enabled:

1. A new line for juan is created in /etc/passwd:

juan:x:1001:1001::/home/juan:/bin/bash

The line has the following characteristics:

* + It begins with the user name juan.
  + There is an x for the password field indicating that the system is using shadow passwords.
  + A UID greater than 999 is created. Under Red Hat Enterprise Linux 7, UIDs below 1000 are reserved for system use and should not be assigned to users.
  + A GID greater than 999 is created. Under Red Hat Enterprise Linux 7, GIDs below 1000 are reserved for system use and should not be assigned to users.
  + The optional *GECOS* information is left blank. The GECOS field can be used to provide additional information about the user, such as their full name or phone number.
  + The home directory for juan is set to /home/juan/.
  + The default shell is set to /bin/bash.

1. A new line for juan is created in /etc/shadow:

juan:!!:14798:0:99999:7:::

The line has the following characteristics:

* + It begins with the user name juan.
  + Two exclamation marks (!!) appear in the password field of the /etc/shadow file, which locks the account.

**Note**

If an encrypted password is passed using the -p flag, it is placed in the /etc/shadow file on the new line for the user.

* + The password is set to never expire.

1. A new line for a group named juan is created in /etc/group:

juan:x:1001:

To delete users using the userdel command, you need to be logged in as root or a user with **sudo** access

The syntax for the userdel command is as follows:

userdel [OPTIONS] USERNAME

Type following passwd command to change your own password:  
*$ passwd*  
Sample Outputs:

*Changing password for juan*

*(current) UNIX password:*

*Enter new UNIX password:*

*Retype new UNIX password:*

*passwd: password updated successfully*

The user is first prompted for his/her old password if one is present. This password is then encrypted and compared against the stored password. The user has only one chance to enter the correct password. The super user is permitted to bypass this step so that forgotten passwords may be changed.

# **References**

[1] <https://www.techtarget.com/searchsecurity/definition/authentication>

[2] <https://www.idrnd.ai/5-authentication-methods-that-can-prevent-the-next-breach/>

[3] <https://linuxize.com/post/how-to-delete-users-in-linux-using-the-userdel-command/>

[4] <https://www.cyberciti.biz/faq/linux-set-change-password-how-to/>