Chapter 6

Of Users and Groups

We trust you have received the usual lecture from the local System Administrator. It usually boils down to these three things:

#1) Respect the privacy of others.
#2) Think before you type.
#3) With great power comes great responsibility.

– Output of `sudo(8)` upon first invocation.

6.1 Introduction

As was noted in Section 2.2.3, Unix was designed as a multi-tasking, multi-user system, allowing simultaneous access by different people. This concept, also found in all of today’s mainstream operating systems as well, demands important safeguards to prevent both accidental as well as intentional (i.e. malicious) access by unauthorized parties. Since this property affects virtually all aspects of system administration – ranging from account management to the way different software components interact with one another – it is important to fully understand the implications of the nature of a multi-user system.

In this chapter, we will discuss the different types of users and groups as well as talk about user authentication on our systems. Note that even though there are many parallels to authentication by remote clients against a service we may offer (such as users logging into our website), in this chapter we restrict ourselves to the “local” systems and how we access them internally.

Even if you are comfortable with the Unix multi-user model, you may still
find that explicitly identifying its inherent requirements and trust models will help in formalizing your access control requirements so as to express and apply them using your configuration management system, as discussed in the Chapter 7.

As one of our Pillars of Exceptional System Design, the topic of security weaves through this book like a red thread, and multi-user principles and the implications of different access- and trust levels necessitate a look ahead at topics and ideas discussed in more detail in Chapter 11.1. Hence, discussing the core concepts of authentication and authorization is a natural requirement and we will attempt to broadly cover them in this chapter as well. But let’s not get ahead of ourselves...

6.2 Types of Users

Allowing more than one person access to the resources of a system requires the computer to be able to distinguish between different users, which of course is done by providing distinct user accounts. When a user wishes to use the system, she identifies herself to the computer (for example by entering her login name), upon which the system authenticates her (for example by prompting her for a password). Upon successful authentication, she is then granted access to her files and/or is allowed to run certain programs. But this simple association of human beings to computer accounts – in the Unix world implemented using numeric user-IDs – does not paint a complete picture.

It is easy to assume that different “users” refer to distinct people accessing the resources, and ideally we’d want to keep the mapping between the two sets (“persons”, “accounts”) to be of a 1-to-1 (and onto) nature. But we quickly run into problems:

On a typical Unix system, you will find a perhaps surprising number of accounts that are not associated with actual human beings at all. To illustrate this, Listing 6.1 displays excerpts of the standard Unix password database, the /etc/passwd file, on a NetBSD system. Reviewing the first few rows, we find most of them reference so-called system accounts, present to allow specific services to run with only the permissions they need, a concept known as privilege separation or the Principle of Least Privilege. This ensures, for example, that a software error in one daemon cannot accidentally lead to access or deletion of files owned by another user.

In addition to these system accounts, we also frequently find what has
CHAPTER 6. OF USERS AND GROUPS

Listing 6.1: Excerpt of a typical /etc/passwd file on a NetBSD system

become to be known as role accounts: accounts that, while not specifically tied to a single person, are used to allow multiple people to initiate certain well-defined actions. A common example is a role account to allow for automated updates to a given service or application.

In order to manage the resources to be made available to different users, to control the system, to add or remove software etc., we require the use of an omnipotent account, known as the superuser. This account is commonly called the root account, and is identified by the numeric user-ID 0.\(^1\) In order to gain superuser privileges, a user needs to either log in as root, assume the superuser’s identity via the su\((1)\) command, or make use of, for example, the popular sudo\((8)\) utility (which also offers some more fine-grained control over who may gain what privileges).

\(^1\)As user privileges on a Unix system are only identified by the effective user-ID of a given process, the system does not care one bit whether or not the string mapped to the UID 0 happens to be “root” or “bob” or “susan”. That is, superuser accounts with other names are entirely possible; “root” just happens to be one of those old Unix conventions we’ve all grown so accustomed to. It is good practice – and part of most standard security scans – to periodically check for the existence of “other” accounts with UID 0.
CHAPTER 6. OF USERS AND GROUPS

Of Headless Users and Privilege

Role Accounts may be known by a variety of names, depending on the history of the organization: “service account”, “headless users” (because no human is associated with the account), or, frequently, by the name of the company or organization. Now a funny side-effect of using a service account intended for privilege separation to automate certain tasks is that this account becomes more and more powerful with time: every task it needs to accomplish (such as installing or updating software, starting or stopping services, monitoring services or alerting on conditions) requires the user to have the necessary privileges.

Take a look in your /etc/passwd – if you work at company “Gizmo App, Inc.”, chances are the most powerful user (besides root, of course) is “gizmo”, and gizmo’s access credentials are shared across multiple systems. We will discuss this dilemma and some possible ways to mitigate it in Chapter 11.

Perhaps somewhat paradoxically – usually, we wish to have a unique mapping from exactly one person to exactly one user-ID – access to this superuser account is shared: all system administrators within an organization need to have this kind of access by necessity.\(^2\)

Reviewing these access rules, we can identify mappings of zero, one or more people to a given account (Figure 6.1), as well as a single person with access to multiple accounts. From the system’s perspective the differences are irrelevant (all process privileges are granted solely based on the UID), but as system administrators, we have to treat each class of users differently. As a general rule of thumb, we want to ensure, for example, that only those accounts used by people have an interactive login session. All other accounts should be restricted such that they can only execute specific commands or access specific files\(^3\). This becomes difficult quickly especially when using role

\(^2\)A discussion of the limitations in the traditional Unix permissions model that make this shared superuser account a virtual necessity would be beyond the scope of this chapter. For completeness’s sake, however, we should mention that more modern access semantics, such as mandatory access controls (MAC), role-based access controls (RBAC), or POSIX “capabilities” can been added, albeit at the cost of increased complexity.

\(^3\)On some Unix systems, it is possible to restrict a given process to a specific subset
accounts, as often a certain set of interactive commands need to be allowed. Care must be taken to correctly identify and enforce the restriction to just these commands.

If you look closely at the example password file shown in Listing 6.1, you may notice that in addition to the root account there is a second account with UID 0, called toor. This account, often found on the BSD derived Unix variants, does in fact offer a second superuser account: it is usually given a different login shell to provide for a way to repair the system in case root is unable to log in (for example due to a broken login shell as a result of a software upgrade). This illustrates that it is possible for a Unix system to have multiple usernames mapping to the same UID, though it should be noted that generally speaking that is not a good idea and likely to be an error.

![Diagram of user and group mappings]

Figure 6.1: The set of users mapping to the set of account names is neither bijective (or “one-to-one”), nor surjective (or “onto”): some accounts are not used by an actual user, while others may be used by more than one user. Users are associated with local groups on a given host, and group membership may imply specific privileges across a set of hosts.

of the file system – see, for example, the chroot(8) command. In the past, assigning a user a so-called restricted shell was another method of limiting what actions a user may perform.
6.3 Groups of Users

In addition to different types of users, we can also identify different groups of users. Privilege separation is a fundamental best practice in the area of system administration: every user should be given exactly the level of access they require, with elevated or superuser privileges being particularly strictly controlled. But even outside this special group, there are a number of users who need to share resources with one another. The nature of the environment you are in plays an important role here, as (implicit) trust or lack thereof create vastly different use cases. For the purposes of this discussion, let us define a group of users as a number of people who are either collaborating in some capacity – meaning they will wish to share read and write access to certain resources – or require similar execute privileges, for example to start, stop, or otherwise control a specific service.

In a small commercial environment, such as a start-up, the good will and intent to collaborate for all users of your systems is implied. In such environments all users do frequently require the same or at least similar access to all systems, as responsibilities are shared and job duties overlap. As a result, you will find few distinct groups here, and getting access to one system tends to simultaneously get you (the same broad) access to all systems.

As the environment grows in size, however, you will find that you need to group accounts more carefully. For example, all developers on a given software project require access to systems on which to build the software, write access to the central source revision control repository and the like, and so are grouped together; they do not require – nor should they have – access to the systems processing credit card payments or which contain payroll information.

In even larger companies, you may find that groups sharing access to the same set of servers may have competing resource requirements, even if we assume that every user has been given an account for the ultimate benefit of the organization at large. In addition, the larger your organization grows and the more users you have, the higher the risk of any one account getting compromised becomes. Recall our earlier discussion of well-defined and documented policies in Section 3.3.2; it is no surprise to find a requirement for very specific access policies, that spell out in detail who should be granted what level of access on the one hand and for a complete audit trail of who was given said access on the other.
In contrast to these use cases, now consider the different groups of users present in an academic environment, such as a large university. Access to computing resources is made available to students, faculty, staff, and possibly visiting researchers. Even disregarding any outside compromise, you have at times almost directly conflicting privacy requirements: students should not be able to access faculty resources or files (and yet may have an explicit interest in doing so!), but students working as teaching or faculty assistants sometimes should; student homework assignments done on shared servers should not prevent research jobs from running; faculty may collaborate with each other on some projects, with outside researchers on others.

Finally, if you are maintaining public systems for an Internet Service Provider (ISP), you may find yourself trying to manage hundreds or thousands of completely unrelated user accounts with little to no collaboration amongst them.

Group definitions exist in every organization, and are often drawn outside the system administrator’s purview; it is not uncommon for computer systems to inherit a group structure by mapping the organization’s functional hierarchy into different user groups. However, sooner or later you will run into limitations and require more fine-grained control. A careful analysis of the different types of users, the different groups based on requirements and needs, and a directory service that allows you to easily create (possibly intersecting) sets of users are needed to allow your systems to grow with and adapt to the needs of whichever type of environment you are in. We will get back to this concept of grouping users and mapping them to similarly grouped sets of hosts in our next chapter.

All users are equal...

In almost every organization, there are a few users who have root who... shouldn’t: there’s the ex-system administrator, who now works in a different area but who occasionally is called upon when tribal knowledge from the olden days is required; there’s the competent software developer who always needed special resources and who finally managed to convince one of the system administrators to let him have superuser access to fix his own systems; there’s the CEO who simply “requires” superuser access to all machines, even though the last time he opened a terminal was 10 years ago when he updated the
single web server of his then-fledgling start-up.

And then there’s the start-up founder, who not only routinely tinkers with the OS kernel and system configurations, but oversees the hardware allocation requests in what has become an Internet giant with tens of thousands of hosts in data centers across the globe:

While working at Yahoo!, I continually tried to keep small the number of people whose accounts were included in the OS images. As an initially large number was slowly reduced (often only after many discussions, assurances of emergency failover procedures, and arguments), I finally had to tell David Filo (one of Yahoo!’s founders) that I intended to remove his account as well. After a few discussions, assurances of emergency failover procedures, and arguments, we eventually agreed to have it removed from the OS images, but to let our configuration management systems add it as necessary. To the best of my knowledge, Filo, as he is known to all employees, still logs into production servers to analyze and challenge current resource usage whenever engineers request new hardware, and he still traces and fixes kernel bugs that require him to have access to most, if not all, systems.

Every system administrator I know has a similar story to tell; they frequently end with “Well, he doesn’t need access, but he’s the boss, so...”.

Some users are more equal than others.

### 6.4 User Authentication

We mentioned earlier (albeit somewhat tersely) that the system authenticates the user at login time. Seeing how this is a rather important aspect of a multi-user system, let us spend a little bit of time to make sure we understand what exactly happens. As we will discuss in more detail in Chapter 11, when we are authenticating a user, we are ensuring that user is who they claim to be. More precisely, we are verifying that whoever is identifying themselves as user “alice” can in fact provide the credentials which we assume only Alice
to have access to.\textsuperscript{4}

\textit{Authentication} (frequently referred to as AuthN) is the act of providing a proof of identity, not a proof of \textit{authorization} (AuthZ). Although frequently entangled, the former only answers the question “Are you really who you say you are?”, while the latter concerns itself with the question “Are you allowed to perform this action?”. There are many different ways of providing a proof of authentication; we generally divide them into the following three classes\textsuperscript{5}:

- \textit{something you know}, for example a password
- \textit{something you have}, for example a physical key or a hardware token
- \textit{something you are}, a biometric factor, such as e.g. a fingerprint

Each authentication method has its own strengths and weaknesses, with usability and convenience being important factors upon the final security they may provide: passwords that are easy to remember are also easy to guess or crack, physical tokens can be lost or stolen, and biometric factors are near impossible to replace or change in case of a compromise.

In many cases, it is desirable to combine some of these authentication methods to yield the desired level of security and usability. So-called “two-factor authentication” or 2FA has recently become a popular protection mechanism used by many internet sites: in order to gain access to a service, the user needs to provide e.g. a password (\textit{something you know}) as well as a code generated by an app on their smartphone (the phone thus being \textit{something you have}). It is entirely possible – and at times desirable – to add more factors or vary combinations of factors, which is why the more general term “multi-factor authentication” (MFA) is more precise.

Lastly, let us note that even though we frequently only consider authentication of one party, \textit{mutual authentication} is an often desired or even required property of a secure system: when a user logs into a system, the system wishes to authenticate the user, but the user also needs to have assurance that the system they are logging in to is the system they think it is.

\textsuperscript{4}Unlike the system administrator in charge, the computer does not care if Alice gives her credentials to Bob and allows him to log in as her.

\textsuperscript{5}Cynical information security professionals, always happy to point out weaknesses in any system, like to refer to these categories as “something you forgot, something you lost, something you stopped being”.

Listing 6.2: Password authentication on the system console on a NetBSD server.

Listing 6.3: An example of mutual authentication by way of SSH keys.

6.4.1 Authentication Examples

To illustrate the many different ways we may use or encounter authentication in our day to day operations, let us look at a few examples:

Listing 6.2 illustrates the most simple example: password authentication on the system console of a NetBSD server. The user provides their username and password; if the password matches (see below), the user is logged in.

Listing 6.3 shows another method of authentication so frequently used that we hardly ever think about it. Here, we see asymmetric key cryptography as a means of authentication, effectively using a *something we have* (a private SSH key). But note that at the same time, the server also offers *us* a way to authenticate *it*: the SSH hostkey fingerprint presented on the first connection allows us to verify that the server is in fact the one we intended to connect to. If we know the server’s expected hostkey fingerprint (which itself is derived from the private hostkey that presumably nobody else would have access to), then we can compare and match them, thereby authenticating the server.\(^6\)

\(^6\)Unfortunately, the distribution of known hostkey fingerprints and changing identities in a large scale environment have lead most people to blindly accept any hostkey fingerprint
Listing 6.4 illustrates the use of a central authentication service by way of the Kerberos protocol. Here, we are authenticating ourselves to the central service using a password (something we know). We then use a time-based token (a “ticket” in Kerberos lingo, i.e. something we (now) have) to authenticate ourselves to the server we wish to access. At the same time, the Kerberos network authentication protocol also takes care of authenticating the server to us. This setup thus includes multiple authentication methods as well as an example of mutual authentication.

Listing 6.5 finally illustrates the use of several strong factors combined with time-based access credentials to form a particularly strong example of authentication. Here, we are making use of a Certificate Authority (CA) for use with SSH to issue short-lived access credentials. In order to receive such a certificate, the user must first authenticate to the sshca service, which requires both a password (again: something we know) as well as a cryptographic One-Time Password (OTP) generated by a hardware token (something we have). In addition to the client certificate we received from the CA, the server we are finally accessing also implements another form of two-factor authentication, offering to send a message to the user’s cell phone and requiring an interactive acknowledgement. If the cell phone in question requires a fingerprint to unlock, then we are even adding a biometric factor (something we are). Phew, that is one tough system to access!

Presented. Doing so is an example of Trust on First Use (TOFU). We will get back to this problem in later chapters.
**6.4.2 The Problem with Passwords**

Despite the variety of authentication methods available, password authentication remains the lowest common denominator and is worth looking at in a little bit more detail.

More often than not, the credentials used to authenticate at login time are simply a password of often pitiful complexity. Much like you may have not let your sister (or her friends) climb into your tree house unless they knew the secret passphrase, the Unix system will not let you log in without you entering the correct string of characters.

It is important to understand at this point that the Unix system does not actually compare the string you entered to the string it has stored in a database, but that instead it operates on password hashes. That is, the string you entered (together with a few bits of additional data known as the salt) is transformed via a one-way function that produces a fixed-length string of (different) characters, which are then stored and compared against at the time of authentication. Since this transformation is a one-way function, it is impossible for anyone to reproduce the original password from the hash – an important cryptographic property of the function used. If this password hash matches the one on record, access is granted, and you are logged in.
From that moment on, regular Unix semantics and permissions will apply to decide whether or not access to a given resources (e.g. the ability to write to a file) is granted.

Passwords are an easy and convenient way for users to authenticate to a system. However, there are quite a few drawbacks that it is important to be consciously aware of.

As noted above, we do not actually store clear text passwords in a database, as this would mean that anybody (and any process) able to access this database had access to all users’ passwords. Instead, the use of a hash ensures that even using a privileged account, we cannot see the users’ clear text passwords. Still, we wish to protect the password hashes from prying eyes, which is why our Unix systems no longer store them in the world-readable /etc/passwd database, but instead in a separate, protected file (such as /etc/master.passwd on BSD derived systems or /etc/shadow on many System V derived Unix versions).

But, and this is one of the problems when using passwords for local authentication, this data has to exist on all the hosts a user wishes to log in on. That, in turn, means that if a single host in the environment is compromised, the attacker gets their hands on all users’ password hashes. Once retrieved, they can then perform so-called offline dictionary attacks on the password hashes or look them up in widely available rainbow tables, large pre-computed mappings of common strings to their hashes.

The solutions, or rather, the efforts to at least partially address these issues include the use of a password salt, the previously mentioned small amount of data added to the user’s password prior to the application of the hash function, thereby yielding a different hash on unrelated systems despite the password being the same and thus defeating simple rainbow table look ups.

Another approach used frequently is to not make the password hash locally available on each host and instead rely on an authentication system where the actual password hashes and additional user information is stored in a central place and is accessed over the network. The Lightweight Directory Access Protocol (LDAP) is an example of this approach. However, it should be noted that the benefits of a central location of this information carries a certain prize, as well: if the host(s) providing this service becomes unavailable, thousands of hosts can become unusable or see debilitating errors. This problem can of course be defined in a more general statement: as an environment increases in size, relying on a central service poses an increasing risk of
becoming a Single Point of Failure (often abbreviated as “SPOF”), while distributing data across thousands of servers poses its own data replication and synchronization challenges while simultaneously increasing the probability of it being compromised.

What’s more, passwords are an inherently insecure means of authentication, largely due to human nature: most people are rather terrible at remembering complex passwords (which are harder for computers to crack) and hence tend to use and reuse across different sites a small set of a simple passwords. This means that accounts on your systems may get compromised by password leaks in another, completely different and independent environment!

Many solutions or improvements to this dilemma exist, ranging from multi-factor authentication protocols to randomly generated passwords stored and managed by specific password management tools. We will discuss some of these in our chapter on security later in the book.

6.4.3 Sharing root

As noted in Section 6.2, the root account is commonly shared, meaning multiple people know the password to authenticate as the superuser. This, however, poses a number of problems. When a system administrator leaves the organization, the root password needs to be changed, all systems need to be updated with the new password hash, and the new password has to be communicated to all members of the team. Despite many best practices of sharing or storing shared secrets like this, the more people who require said access, the larger the risk of accidental exposure of the password or its hash.

Secondly, since the root account is a system account, it may not be included in any central user directory (such as your LDAP service), and instead be managed either via your configuration management system, or your OS image. This may lead to the password hash being available in additional places, such as the configuration management’s repository or the OS image build system. Again: the more places such important access credentials are stored, the more likely it is that they can be accessed by unauthorized parties.

Finally, by mapping multiple people to a single account you lose an important audit-trail: no longer is it possible to identify who performed a given task; all we can do is identify a (possibly large) group of people who could have done so. Given the powers of the superuser account, this is particularly bothersome.
It is common best practice to disable logins from root over the network\textsuperscript{7}, requiring users to first log in with their usual account, and then to use the\texttt{su}(1) or\texttt{sudo}(8) utility to perform certain actions with elevated privileges. Since this tool logs all invocations, we nicely solve the problem of the missing audit trail; in addition, we gain much finer grained control of what access is given to which users.

Note, however, that this solution is no panacea: it is easy to fall into a false sense of security when restricting superuser privileges to a few explicit commands while forgetting that many of these commands allow a skilled user to invoke or otherwise trick the system into executing other, non-sanctioned commands.\textsuperscript{8} As a general rule of thumb, you should consider the use of\texttt{sudo}(8) primarily for its audit-trail capabilities and only grant its use to trusted users.

Secondly, this approach only works on actual server operating systems. Networking equipment, such as switches, routers, or load balancers, where the TACACS+ and RADIUS remote authentication protocols allow well-defined control of remote access do, for the most part, still follow an all-or-nothing access model locally. The access credentials for the privileged accounts on these devices (as example might be the “enable” password, used to enter privileged mode on a router or switch) need to be managed and shared with the same care that a root password would be.

6.5 Summary

Even though taken for granted nowadays, the nature of a multi-user system has had from the beginning a number of important implications for overall system security and operational procedures. The impact of these implications grows near exponentially as your environment scales up, which is why we make a point of identifying them explicitly in this chapter.

Users fall into a number of well-defined categories, or types of users. In particular, we distinguish for good reasons between user accounts used by ac-

\textsuperscript{7}It is also possible to completely disable the root account, though that may have implications on your ability to recover the system without physical access in the case of a severe failure.

\textsuperscript{8}Most Unix editors, for example, allow a user to invoke a shell; dynamically linked executables can be tricked into loading custom libraries; the possibilities to exploit access to a small number of tools running with superuser privileges are too numerous to account for.
tual humans and so-called system accounts. Different users (of either kind) have at times conflicting requirements and impose very specific trust models on the environment. Access privileges need to be defined and enforced, and authentication methods need to be considered. We briefly mentioned passwords as the most common form of authentication and noted a few of the problems associated with them; we also covered some of the implications of sharing root access with your peers.

More generally speaking, though, we looked at what it means for a system to support multiple users. System administrators with experience managing deployments in diverse environments are able to recognize and apply these general rules:

- **All users are equal.** We need to be able to accommodate different use cases and equally enable many different types of requirements. All groups of users should be treated with the same professionalism and their needs appropriately addressed.

- **Some users are more equal than others.** While all users’ needs should be addressed, there are, in any system, some users who have more specific requirements; who need certain elevated privileges; whose computing demands exceed those of others.

- **All users are to be given precisely the access rights they need, but no more.** The principle of least privilege needs to be rigorously applied, as any one account may become compromised, and the possible damage deriving from this scenario needs to be limited as much as possible.

- **Trust does not scale.** When building your infrastructure, remember that while you may trust all users in your organization today, you will eventually grow to a size where this no longer holds. It is near impossible to later on put in place restrictions on users’ privileges, just as it is to anticipate the possibly sudden departure of trusted employees.

- **You will always face tradeoffs.** No matter which authentication mechanism you choose, there are downsides. Eliminating single points of failure may increase your infrastructure’s complexity or increase the risk of exposure of confidential information. (This holds for many other aspects of system administration, too, but the impact may be most obvious when it comes to authentication of users.)
CHAPTER 6. OF USERS AND GROUPS

When you manage a sufficiently large or diverse group of systems, you will learn to abstract individual users (and their requirements) into more conceptual *groups of users*. These groups may then be translated to access privileges on a given host, or, as you scale up your environment, to mappings of privileges to *sets of hosts*.

Controlling only a few hundred machines, it is easy to think in terms of individual hosts. However, today’s Internet giants tend to operate by utilizing tens or hundreds of thousands of hosts. At this scale, services are defined not by which individual machines, but by which *datacenter* handles the requests. Individual hosts become irrelevant, even though access control, by and large, still follows the same model, even if operating on a different scale.

Due to this shift in scale, a distinct trend away from managing user access on an individual host basis and instead shifting towards a *Service Orchestration* model has formed in the last couple of years. In this world, interactive logins to any single host are unnecessary and imply a systemic failure, as unavailable hosts should automatically be taken out of a production-serving rotation and overall load be distributed to the remaining, working hosts.

Services are run as so-called system accounts unassociated with actual users. Nevertheless, regular multi-user semantics apply. The principles of how access to any given resource is granted, how services (or users) are authenticated, and how a clear separation of privileges provides the foundation for overall system security is not different when applied to user accounts that end up being mapped to humans versus those that are not.

A multi-user system implies the existence of a privileged account, which, by necessity, is shared amongst multiple people. The fact that this account is also – by definition – a system account embodies the paradoxical complexity of managing *all* user accounts (whether or not they may be mapped to people or service roles). The only way to retain a modicum of sanity when managing these many-to-many mappings of privilege in a large environment is a clear definition of host and user access groups. As we will see in the following chapter, creating and operating on these sets of resources in the abstract is best done using a configuration management system.

Looking towards the future (as we will in Chapter 17) – and catching up a bit with developments since I first started writing this chapter back in 2012 – we should also note that as the industry moves further towards more ephemeral systems, such as through the use of containers and virtualization
technologies, the overall headache of managing local user accounts may well be a thing of the past: systems are defined in a descriptive way and spun up or torn apart as needed. Nevertheless, as experienced system administrators, we are well aware that new headaches may well lie hidden here, and the concept of authenticated access by multiple users (for humans and services alike) continue to apply.

Thus, the burden of identifying the proper access model, however, remains with the system administrators. Let’s make sure that we understand the implications of multi-user access on all of our systems as we do!
Problems and Exercises

Problems

1. Review the passwd(5) manual page and make sure you understand what each field is used for. Is this password database used on the systems you have access to, or is authentication done by way of a central system, for example via LDAP? If so, what additional information can you find in this system?

2. Review the accounts present on your systems. How many of these are system accounts, and how many are user accounts? What different types of users can you identify? Are there any role accounts?

3. Review the different groups present on your systems. Identify the groups with the most users in it and what it is used for. What resources on the system are accessible only by belonging to a specific group?

4. Identify whether or not sudo(1) is used on the systems you have access to. Can you find out which users have which privileges? Which, if any, commands can you think of that might be dangerous to allow untrusted users to invoke? Try to think of non-obvious ways to circumvent the given restrictions.

5. Compare the default /etc/passwd file on a few different Unix versions. What kinds of differences do you notice? What kinds of accounts are present on one but not another?

6. Consider some of the online services you use. What types of authentication do they require? Which offer multi-factor authentication? What kinds of factors do they use?
7. Search the Internet for a list of the most popular passwords in use (such as, not surprisingly, “password”).

   (a) Generate hashes for each password using the following digest algorithms: DES (as used by the Unix crypt(3) family), MD5 and SHA1. Can you find the resulting strings in any rainbow tables on the Internet?

   (b) Repeat the previous exercise, but add a salt to the password. What do you notice about the results?

8. Write a tool to create a new account on a remote system. The tool should take as input the username of a local account; the account on the remote system should be identical with regards to UID, GID, supplementary groups, login shell etc. to that on the local system.
Bibliography

