Computer Security

Part 2

Contents

User authentication techniques

The usual scheme for access control is a two layer structure. When users need to access resources (like data), they first authenticate. This proves their identity within the system: who they really are, with confirmation it's really them. After authentication, they go through a *reference monitor*. This component is expected to mediate and control the access by expressing a decision about it. The reference monitor is tied to access rules defined by a *security administrator*. The rules can be seen as a static part of the system, while the reference monitor is the dynamic component that enacts them. The second part of access control is the *authorization level*. The use of the reference monitor varies depending on how the data is stored. Data is usually stored in an encrypted form; if that's the case, the reference control can just mediate the access to encryption keys. These keys can be made different for each user, to control the access directly through encryption. If data is not encrypted, the reference monitor needs to be tied to the data and control the access directly. In any case, all the activity in the system is recorded in *security logs*, audited by a professional figure (or a computer system) known as *auditor*. The auditor is tasked with finding security violations. Administration and auditing need to be separate to prevent attacks from within. If the roles were not distinct, administrators could manipulate the computer system to their advantage without anyone else being able to check. The combination of the three steps (Authorization, Authentication, Auditing) is known as AAA.

Let's dive into authentication. User-to-computer authentication allows the users to demonstrate they're the owners of a given identity. The process goes both ways. Users are also interested in demonstrating that the identity of the system they're connecting to is also correct. Attacks in which a malevolent computer system is disguised as a legitimate service are known as *phishing attacks*. In particular, an attacker might be interested in replicating the login form of some famous and critical service (e.g. a bank) to steal credentials from users (*spoofing*). To prevent phishing attacks, users need to check the URL carefully. Both the DNS entry and the path need to be correct. In the case of smartphone apps, the URL might be hidden, making this impossible. There is also computer-to-computer authentication, as in the case of APIs connecting to each other. Authentication can be seen as the primary security service, without which all the other steps are voided. Fortunately, the modern IT world provides many good options to secure this step.

All authentication relies on cryptography. The different types of user-to-computer authentication are distinguished by the type of secret they are based on. The secret might be some piece of knowledge known by the user, such as a password. Alternatively, it might be a code stored in, or generated by, something owned by the user. This is the case of tokens, access cards or smartphone apps. Finally, authentication might rely on the physical identity of the user. This is known as biometric access. A combination of more authentication methods based on different types of secrets is called *multi-factor authentication*. As it is the case for many other computer security issues, there is a significant trade-off between cost and security. The cost is not only related to the development and integration of new security component, but also with false negatives. Proper, legitimate users, being wrongfully locked out of a computer system when they need it, result in a loss of functionality and of revenue.

Secret-based authentication

Password-based authentication is the oldest and most common authentication method. The user trying to log into a system first provides a username, which is a unique identifier. Then, the insertion of a secret password proves that the user is the legitimate owner of that identity. A password is the simplest authentication secret for the user. It's easy to understand conceptually. It's also cheap and easy to implement for the creators of a computer system. However, as it usually happens with many things in this domain, this ease of use and implementation comes at the expense of security. Passwords are easily guessable in many cases, with an additional trade-off between security and rememberability. A long, high-entropy password is more secure but also harder to remember. The user might find the need to write it down, either on paper, on a digital note or in a password manager, making it more exposed to a potential attack. In addition, since passwords are inserted via a keyboard or keypad, a physically present attacker might take a look at the user during password insertion, or film it through a spy camera. This is known as *snooping* or *shoulder surfing*. The password might also be intercepted by spying network communications (sniffing) or acquired by impersonating the login interface of a legitimate system (*spoofing*). Finally, the password might be found by carrying out an exhaustive search on the character set domain (*cracking*).

Basically, whoever gets their hands on a username (which is usually public) and its associated password could impersonate the user within the target system. This operation is known as *masquerading*. However, this lower level of security does not mean that password-based identification should not be used. It only means that it has to be used with an additional level of care compared to other techniques. Password-based authentication is still the standard practice for logging into most computer services. Additional authentication steps may be required, in a multi-factor fashion, for more advanced functionality. Weak passwords are a vulnerability to the system, even if they are not the only access method. However, modern systems are moving in the direction of multiple authentication layers. A weak password may be less of a problem if critical operations require additional identification (e.g. identification apps). This kind of additional verification may also be activated by the use of a different location or a different device during access.

Many sources of password vulnerability are extrinsic. For instance, the chances of a password being exposed increase over time. However, forcing frequent password changes has a significant cost for the users. This price is paid in terms of time and cognitive overload, in the form of delayed access and a higher chance of forgetting. Another risk comes from account sharing: the more users share the same password, the more the risk of it getting spread around even more widely. Weak passwords that are easy to guess, along with the use of the same password on different systems, are other common vulnearability sources. Finally, the storage of passwords on insecure media can also be a source of trouble. Modern *password managers*, as client applications or browser components, provide a level of security that is higher than digital notes but lower than paper notes.

Storing the passwords in plaintext at server side increases the vulnerability of passwords used on multiple systems. This is why the hash digest of a password is usually saved instead of its plaintext. To further increase security, it is common practice to employ *salted hashing*, which consists in hashing the concatenation of the password with a random nonce (the *salt*) which is stored along the hash in the server database. When the password is received at login, it is hashed along with the salt and matched against the hashed salt in storage. This is required because non-salted hashing is vulnerable to *rainbow table* attacks. As explained in the first part of the course, a rainbow table is a data structure containing all possible input / output tuples for a given hash function. At the relative expense of space (usually a few terabytes), a hacker could perform a high-speed reversal of a non-salted password hash to find the original password.

Other possible vulnerabilities might come from overall system design rather than the password itself. For instance, many web services offer a *Single Sign On* (SSO), which is a single login system for all the possible services made by a company, or for accessing any of the terminals in a distributed computer system. Such a system makes access easier to users at the cost of vastly increasing the attack surface. Another major source of threats are password recovery systems. Since any system is only as secure as its weakest link, if the recovery questions or secondary passwords are not as strong as the primary password, the whole system will be made weaker. Old-school recovery questions were a favorite target for attackers because they were usually imposed to be easy to remember (fiscal code, mother's surname, school, first dog's name, …). Such a process can only be robust if a variety of different questions are asked, because the entropy of each distinct answer is very low. Using false answers to increase security is only an option if it does not violate correctness-of-declaration constraints. Another threat related to account recovery is that most modern password recovery systems are email-based. Attackers could exploit the "I forgot my password" option: stealing the credentials to a mailbox will allow taking control of all, or at least most, of the web accounts tied to that email address.

Another security versus ease-of-use trade-off comes from the policy used to handle failed login attempts. For example, it might seem a good idea to block login access, at least temporarily, after a certain number of wrong password insertions, to protect the system from brute force attacks. However, this can be exploited to block a legitimate user from accessing their own account. An attacker might keep on inputting the wrong password on purpose with the intent of blocking the system. Another related mechanic could come by the exploitation of stolen security logs. Legitimate login attempts with slightly wrong spelling in the password, if logged, could be used to infer the correct password. Other password-related issues could come from a change of context breaking the initial assumptions. For instance, Bancomat cards were originally meant to only be used for cash withdrawal at ATMs, which are a protected environment. Their use case extension to direct in-shop payments increased the exposure of their short PIN codes, lowering the overall security of the system. Finally, default passwords in

preconfigured IT systems can too be a source of vulnerability. For example, OracleDB used to include a demo account called `scott`, with password `tiger`, in all new installs. If the system administrators did not delete it manually, that account could be exploited by attackers to access the database.

After having extensively discussed the potential weaknesses of passwords and how they could be exploited, let's now analyze what could make a password strong. For starters, some enforcement on password structure could be a good practice. Passwords are stored as hashes on the backend, so it's hard for a system to run preventive cracking on user passwords to check their strengths. An easier option is to impose constraints on minimum size, force the use of unusual characters and discourage the use of common natural language words, via frontend on the web form where the user is asked to choose a password. Of these, checking that the password is not a dictionary word is probably the most important. Dictionary words have a much lower entropy than non-dictionary strings of the same length. The gamification of password choice can make the process less annoying to the user. On the other hand mandatory password expiration, while being a commonly used historical solution, has been proven to be counterproductive. If users are forced to change passwords frequently, they will choose on average weaker passwords than if they were asked to choose a non-expiring password only once. When used, forced password rotation needs to check that the passwords are not used more than once by storing old hashes. Some regulations in Italy force password rotation. Given the variability of hashes on even small changes to the plaintext, if a system can detect that a new password is too similar to the old password, it means that the system is not keeping the hashes but the plaintext passwords.

As an alternative, systems may choose passwords for the users. This is usually not well accepted and also introduces secure password distribution issues. This problem has been solved by password managers within browsers: the browser can pick a password and store it with very low friction to the user. Another approach is One Time Password / One Time Pin (OTP). These are automatically-generated, one-time passwords. Historical approaches for initial password distribution may includ-e physical, on-site identification with manual password insertion, or giving a *pre-expired password* (for example the birth date) to be changed at the first access.

Ownership-based authentication

The second family of authentication solutions is based on ownership. Tokens are card-sized objects that contain a cryptographic key that can be read by the computer. They are much harder to copy than a password. By maintaining physical control over the token, users keep control over their identity. However, access to the token is sufficient to access the system, which can be a problem if the token is lost, stolen or cloned. This is why token-based solutions are often paired to knowledge-based solutions (token + password). In modern scenarios, tokens have been replaced by SMS-sent OTPs (less secure) or authentication apps (more secure).

There are two types of tokens. The first is a *memory card*, which can store but not process data. The PIN (Personal Identity Number) is stored and sent as plaintext, which make it prone to sniffing attacks. This is the approach of old gate remote controls and the code cards once used by banks. The second type are *smart tokens*, which also have some processing capacity. This allows for encryption. Modern car keys and electronic identity cards (like the Italian CIE) are of this type. Tokens can use a unidirectional channel (either with statical or dynamically generated codes) or a bidirectional channel (with a handshake or challenge-response protocol). Statical passcodes may be intercepted and used in a replay attack. Dynamic password generation can use random codes, progressive numbers (a counter) or SKEY codes. It is both more secure than static codes and simpler than bidirectional systems.

SKEY is based on recursive hashing. The client keeps the whole list of hashes as throwaway codes, the server only stores the last used. Random code systems need to keep a list of the previously used codes to check the freshness of accesses. In addition, overflow attacks may reset the counter on the server to make a replay attack possible. SKEY codes are limited in amount. Counters can lose their synchronization. If the token contains a reliable clock, the HMAC of the token ID merged with a timestamp (maybe in the order of minutes, to give some tolerance to clock drift) can be used as a key. In challenge-response systems, the server sends a challenge (e.g. a nonce) to the client, which has to respond with a reply based on the shared secret.

The military application of identification is known as IFF (Identify Friend or Foe). It may be subject to relay attacks: a foe may relay friendly communications back to the enemy when called. Similar attacks

are used to relay the signal of RFID keys to the attack location. To prevent this, IFF technology needs to relay the position along with the message, so that triangulation may prevent relay attacks.

Physical identity-based authentication (biometrics)

The last family of authentication techniques is based on biometrical data. It has to use invariant physical traits (fingerprints, blood vessels in the retina) or behavioral traits. Since these traits are still slightly variable, a *template* (average of different measurements of the same traits) needs to be built. The user is then checked for similarity to the template at each access. The similarity threshold needs to be adjusted appropriately to minimize both false positives and false negatives. Biometric technologies have become cheaper, but there are still some issues (privacy, lack of fingerprints, laser scanners for retinas). As always, security and convenience do not go hand in hand. For instance, fingerprints can be picked up from the environment and copied on slide film. Creativity and low resources can be easily used to break biometric access if physical access to the target is available. Iris control is easy to perform and store (around 500 bit of information) but, unlike retina control, it can be stolen by photography even from a distance. Two-factor authentication via SMS OTPs is susceptible to attacks where hackers may get a new SIM in the real owner's name and intercept the OTPs.

Access control

Computer security tasks used to be mostly concerned with the setup and configuration of LAN networks and firewalls. Nowadays, *access control* has become the main and most fundamental component of computer security. To clarify the concept, let's now list some fundamental requirements of access control. First of all, users should be differentiated by the privileges they enjoy over the system and its resources. To maintain this differentiation, a user should not be able to access the account of some other user with higher privileges. It should also not be possible to improperly alter the access rules. Authentication does not only differentiate the users: it also helps in tracking the responsibility for wrongful actions within the system. This is why the old concept of having all administrators share a single root account has been mostly phased out. Instead, it's better to give a temporarily accessible administrator privileges (such as a *superuser mode*) to the personal accounts of the administrators. Even commercial web services have started to incorporate similar dynamics into their accounts. Some sensitive actions, such as changing the billing information or deleting the account, may require an additional level of authentication.

An important distinction to be made in the access control domain is that between *policies* and *mechanisms*. A policy can be defined as a set of high level rules. If these rules forbid the access to all users by default, except for those from an allowed list, the policy is called *closed*. On the other hand, an *open policy* allows all users by default except for those on an exclusion list. Open policies used to be rare in the computer world. However, in recent years, they have become the main driver for many web service companies. Corporations with an advertisement-based revenue model, such as Google and Meta, have a high interest in maximising the number of users (and therefore of impressions on the ads). All users can join their service, with the only exceptions being bots and repeated offenders of policy violations. A mechanism is the low level implementation (either hardware or software) of a policy.

The component that enacts access control in a computer system is known as a *reference monitor*. It is tasked with verifying the compliance of accesses from users to resources with the relative policies. To be secure, a reference monitor needs to have three fundamental properties. First, it needs to be tamper-proof, meaning that it can't be altered or disabled. Then, it should be non-bypassable, therefore controlling all possible transactions. This is especially important when using encryption. Finally, a good reference monitor must provide a security kernel. This can be defined as a compact implementation, stored in a limited part of the system. This is because the possibility of security risks increases with the size of the code base. Another reason is that, in practice, rigorous formal methods can only be applied to small portions of code. Reference monitor are also required be able to provide an answer to access requests within 10 ms.

The implementation of policies in the form of mechanisms is non-trivial. The complications arise from the necessity to avoid *storage channels* and *covert channels*. A storage channel, also known as a *residue problem*, is the set of traces left by the execution of operations in a computer. These include memory pages and disk sectors abandoned by a terminated process. To avoid leaving potentially sensitive information around, these memory portions need to be cleaned before reassignment. Historically speaking, encryption used to be more computationally expensive than storage channel clean up. However, the introduction of hardware encryption has reversed the trend. Covert channels include the analysis of system responses and system load (known as a *timing channel*). These can be used to infer information about the processed data without accessing it directly.

The development of access control is usually carried out in multiple phases. These steps move from policies to mechanisms. Such an approach requires the definition of a formal *model*. The model needs to be *complete*, as to specify all the security requirements. It also needs to be *consistent*, that is, without contraditctions. There are two main advantages to a multi-phase approach. Firstly, the evaluation of policies, models and mechanisms can be separated. Secondly, the properties of the sytem can be proved formally through the model.

Access control systems can be divided in three main classes. The first is based on discretionary policies and is therefore known as *Discretionary Access Control* (DAC). The second is based on mandatory policies and is therefore known as *Mandatory Access Control* (MAC). The third is built on role-based policies and is therefore known as *Role-Based Access Control* (RBAC). Less used (and known) approaches include *Attribute-Based Access Control* (ABAC) and *View-Based Access Control* (VBAC). The first is based on attributes given to both users and the target objects of their requests. The second is used in relational DBMS systems and tied to the construct of views. Users do not have direct access

to the tables. They are instead given user-specific views that contain only the data they are individually allowed to access. Finally, *Relation-Based Access Control* acts on a similar principle, but employing relations instead of views as its building blocks.

Discretionary Access Control

Discretionary policies are based on user identity. They are known as *discretionary* because a user having privileges over a resource may pass them over to other users. For an analysis of a security model in the discretionary domain, three components are required. These are the *entities* in the system, the *authorization/protection status* and the *axioms* that need to be satistfied. In particular, entities include the subjects who can access objects, the objects that need to be protected and the actions that can be performed on them. These components can be represented in a model known as the *access matrix*. This model is not commonly known, but is a good approach nevertheless. The structure was proposed by Lampson in 1971, then refined by Graham and Denning the following year. The most notorious version is the one formalized by Harrison, Ruzzo and Ullmann in 1976. This variant usually referred to as HRU.

As the name suggests, in the access matrix model the authorization state is represented by a matrix. The overall system state is described by a (S, O, A) tuple. A is the access matrix. The row set of A, known as S, represents the subjects. The column set O describes the objects. Each matrix cell $A[s, o]$ contains the privileges of subject $s \in S$ over the object $o \in O$. All elements, including privileges, rows and columns, can be created, deleted or edited. Access matrices tend to be very large but also highly sparse. While being good theoretical models, their high level of dimensionality and sparseness limits their practical usability. The main alternatives are three. The first alternative model is known as an *authorization table*. It stores non-null triples (s, o, a) in a three-column table. This model is frequently used to represent authorizations within DBMSs. The second alternative model is known as an *Access Control List* (ACL). This is a list of entries, one per object. It's like an access matrix memorized by split columns. This approach is highly used in operating systems over their file systems. Individual entries are usually stored within the single files they represent. The third and last alternative mdoel is known as a *capability list*. Each entry, known as a *ticket*, represents a list of all permissions of a single user over all the resources within the system. Basically, each ticket maps to a row of the access matrix. This last model is particularly common in SSO distributed systems.

ACLs, capability lists and authorization tables are different approaches for the implementation of the same underlying concept. ACLs always require subject authentication. The capability list model may require this too, but not necessarily. It requires, however, non-falsifiability and control over capability propagation. The ACL model is more efficient on object-centric control. On the other hand, capability lists are more efficient in handling user-centric modifications to the policy. Finally, authorization tables are well-suited to highly centralized environment. For all of the three approaches, changes to the system can be modeled with a command that contains arguments, an optional conditional component and finally the changes to be applied. The execution of a command will result in a change of system state from $Q = (S, O, A)$ to a new state Q', but only if the conditional part of the command is satisfied. Otherwise, the whole command won't be executed at all. This model for modification handling is an application to ACLs of the Access Matrix modification rules. The following are some examples of commands. The first is used to create a new file for a certain owner. The second is used by a file owner to grant read rights to another user. The last is the opposite, with the owner removing read rights to another user.

```
command CREATE(subj,file)
  create object file
  enter Own into A[subj,file] end.

command CONFER_read(owner,friend,file)
  if Own in A[owner,file]
    then enter Read into A[friend,file] end.

command REVOKE_read(subj,exfriend,file)
  if Own in A[subj,file]
    then delete Read from A[exfriend,file] end.
```

Administrative authorization, that is, authorization that permit the transfer of privileges, can be modeled by associating a flag to access modes. One possible flag is called *copy flag* and is usually associated with the symbol *. Subjects with a copy flag are able to copy their authorizations to other

users while keeping their own. The other flag, known as *transfer-only flag*, is associated with the symbol +. Subjects with a transfer-only flag can only give their privileges to other subjects by losing their own.

Privilege propagation is the cause of an issue known as *safety problem*. By a chain of privilege propagation, non-allowed users might get unearned privileges over resources they were originally barred from. The formulation of the safety problem for security policies is particularly relevant. Its statement is: *is it possible for the security policies (and system states) to evolve in a way that leads to a non-allowed, insecure state?* This is is an undecidable computational problem. It is so because it can be reduced to the Turing halting problem. However, the problem can become decidable by simplifying some of the assumptions. For example, the Turing halting problem is only undecidable because of the infinite amount of tape in the Turing machine. By assuming a finite amount of tape, the problem would converge. Similarly, the safety problem for policies is convergent for mono-operational commands, which are based on a single primitive step. The policy safety problem also converges in a finite amount of steps if the number of resources and users is also finite.

Take-Grant model

The Take-Grant model was introduced by Jones, Lipton and Snyder in 1976. It only considers a restricted class of systems and models the protection state as a graph. Since graphs can be represented by their adjacence matrix, the model can be reduced to an access matrix. This makes the safety problem decidable in polynomial time $O(n^3)$ in the number of nodes n over the whole system, and linear $O(n)$ when considering a specific authorization. The state is represented by a triple (S, O, G). S are the subjects, O are the objects, $G = (S \cup O, E)$ is an *authorization graph*. The authorization graph is bipartite. Its nodes represent subjects and objects while its arcs map to the authorizations. The allowed access modes are: *read*, *write*, *take* (take any privilege from another user you have "take" rights from) and *grant* (grant any privilege to another user you have "grant" rights on). Other operations are *create* and *remove*, to give or take privileges from oneself over a certain subject or object. The *new* modifier in a *create* statement can be used to create a subject or object that did not exist before.

The context of the Take-Grant model is slightly different compared to that of the traditional Access Matrix. Its main disadvantage lies in its semantical poorness. For instance, there is no way fo restricting the type of authorization that can be taken or granted. There is also no control on granted rights. A subject can grant a privilege to other users immediately after receiving it, with no restrictions. Oppositely, a subject may have a recently-acquired right be immediately stripped away. Another issue arises from the lack of an explicit restriction over loops in the Take-Grant graph. A loop may cause a reversal in the privilege flow. For example, the last recipient of a chain of privilege transfers may get privileges over its superiors, and thus, indirectly, over themselves.

DAC weaknesses

A *Trojan horse* is a piece of useful software that contains hidden malicious code. The user will grant privileges to it, deceived by the useful features, but the Trojan horse will exploit those permissions for malicious intents. Other types of malware include computer *viruses* and *worms*. A computer virus, just like a biological virus, has to infect another program because it cannot run directly. Oppositely a worm, like a biological parasite, can run directly on the OS as independent programs. Let's now demonstrate the behavior of a Trojan horse through an example. Jane, a user, has exclusive access to a file called "Market". Unknowingly, Jane invokes a Trojan horse created by John, mistaking it for a real program. The malicious application, having been invoked by Jane, will be able to use her rights to read "Market". Its contents will be transcribed into a file called "Stolen", owned by John, because Jane has write privileges (but not read privileges) over John's files. This way, John can now read a copy of "Market". Jane won't be able to tell, because she doesn't have the rights to look inside John's "Stolen" file.

Mandatory Access Control

Mandatory Access Control (MAC) is an access control model that splits access control between users and the programs that users run. The basic intuition behind it is simple: users can generally be trusted more than the programs they run; those programs are known as *subjects* in this domain. In the example of the previous paragraph, if the system had employed MAC, the attack could have been prevented. The Trojan horse, being run from Jane's account, couldn't possibly have created a file on John's behalf. Even if Jane had been an administrator with write privileges over John's account, her programs wouldn't automatically get those rights too.

Bell-LaPadula model

Mandatory access policies can be roughly divived into two families: *secrecy-based* and *integrity-based*. The reference for secrecy-based policies is the Bell-LaPadula (BLP) model. For integrity-based policies, the reference is the Biba model. Both are based on the classification of subjects and objects. Classification is defined as an element of a partially ordered set of classes. The aforementioned partial ordering is defined by a dominance relation $\succeq$. The security classes are made up of two components. The first component is a *security level*, which is an element taken from a hierarchical set (e.g. Top Secret > Secret > Confidential > Unclassified or Crucial > Very Important > Important). The second component is a *category* label, which is a subset of a non-hierarchical set of elements. This set can include "need-to-know" restrictions. Its intented goal is to partition the compentence domains within the system.

The dominance relation is defined as:

$$(L_1, C_1) \succeq (L_2, C_2) \longrightarrow L_1 \geq L_2 \wedge C_1 \subseteq C_2$$

The security classes along with $\succeq$ produce a lattice $(SC, \succeq)$ which is enjoys the following properties: reflectiveness, transitivity, antisymmetry, single least upper bound, single greatest lower bound. Each user is assigned a *clearance*, which is the combination of a security level and of a category. When connecting to the system, users get access to all, and only to, the classes that are dominated by their clearance. Security classes represent the level of trust upon users who access them, but also the criticality of the objects within them. In a way, categories provide domains of competence of users and data. Two policies are implemented to prevent the transfer of information from a higher level of confidentiality to a lower, or between different classes. The first is known as *no write down* and states that a subject can only write objects whose classification dominates that of the subject. The second rule, *no read up*, states that a subject can only read objects whose classification is dominated by that of the subject. In such a scenario, the flow of information necessary for a Trojan horse would be limited. However, MAC systems only control legitimate channels within a system, leaving open the possibility of side channels.

The concepts of no read up and no write down were originally introduced in the Bell-LaPadula (BLP) model. The model is an interesting concept but the modelization itself is not-so-creative and quite academic. A system is modeled with states and state transitions. A state $v \in V$ is an ordered triple (b, M, λ), where: $b \in \gamma(S \times O \times A)$ represents the set of current accesses to state v; M is an access matrix with S rows and O columns, and $\lambda : S \cup O \to L$ is a function that returns the classification of subjects and objects. The BLP model defines a system as secure if and only if enjoys two properties known as *simple security* and $*$*-security*. A state is simply secure if and only if $\forall(s, o, a) \in b, a = \text{read} \rightarrow \lambda(s) \succeq \lambda(o)$. In words, a state is simply secure if and only if all possible reads follow the no read up principle. A state is $*$-secure if and only if $\forall(s, o, a) \in b, a = \text{write} \rightarrow \lambda(o) \succeq \lambda(s)$. In words, a state is $*$ -secure if and only if all possible writes follow the no write down principle. A state transition function $T : V \times R \to V$ moves from one secure state to another secure state. The Bell-LaPadula *Basic Security Theorem* states that a system (v_0, R, T) is secure if and only if v_0 is a secure state and all possible states reachable, directly or indirectly from v_0 through T by requests from R, are also safe.

The Bell-LaPadula model suffers from a few weaknesses. First of all, the Basic Security Theorem does not put any restrictions on T, which might be exploited for information transfer. Another unsolved issue within the model is that state transitions are not restricted from containing steps that lower the security requirements of the system. In other words, in the BLP model a system where the security is progressively lowered into non-existence would still be marked as safe. A property called *tranquility* puts in place some additional restrictions that prevent the degradation of the security policy during transitions.

MAC Policy Limitations

The effectiveness of the MAC policy is limited by the many exception to it that would be required for real-life scenarios. One example is the process of *declassification*, which consists in the reduction of classification of data after a certain time, when it falls out of relevance. Another example is *sanitization*, the production of lower-class data as the result of a process that consumes highly classified data. Another difficulty lies in the not-so-clear determination of the class of some resources. For example, associations and aggregations might be of higher classification than their individual components. For example, the position of individual military ships is unclassified, but the full positioning of a fleet is a secret.

DAC and MAC policies are not mutually exclusive. For example, BLP also applies a discretionary policy ($b \subseteq \{(s, o, a) \mid a \in M[s, o]\}$). If both MAC and DAC are applied, the only allowed inputs are those who satisfy both. DAC provides discretionality within MAC conflicts.

Another MAC limitation, even when tranquility is applied, comes from its lack of control over side channels. A side channel could be defined as any shared or observable resource that is potentially employable as an indirect source of information. For example, a low-level users may send requests to write on higher level files. If this is not handled properly, the user may deduce whether the file exists or not from the error messages. Additionally, a lower-level processes trying to access a shared resource occupied by a higher-level process may have their requests denied. Oppositely, a higher-level process might get a hold over shared resources to alter the time response of lower-level processes and guess their behavior over that resource by analyzing their delay. This is an example of a *timing channel*. These are the reasons why locking and access control mechanisms need to be redesigned in multi-level systems to prevent both covert channels and denial-of-service issues. In general, covert channels are not a modelling problem, but a physical level problem. Still, covert channel analysis is usually performed during modelling, with the intent of strengthening the implementation. The solution is usually the *non-interference principle*, which states that higher-level processes should not have any effect on lower-level or non-comparable processes. It's important to remember that correct modelling will not guarantee perfect security.

Biba Model

By default, mandatory policies protect secrecy through access restrictions but don't protect integrity. For instance, Bell-La Padula does not stop lower-level users from writing (and overwriting) secret documents way above them. A dual policy, protecting integrity, could be implemented aside of the secrecy policy. Subjects may be classified by a *level of integrity*. For user, this level represents the degree of correctness of the information they have, and the trust in the fact that they won't alter information inappropriately. For objects, this level represents the value of the information they store, and also the degree of damage that might come from the corruption of the information they store. The *Biba model* defines a mandatory policy about integrity. It defines two properties that are dual to those of the BLP model. Its *strict integrity policy* is made of two sub-policies. The $*$*-property* states that a subject may write to an object only if its integrity class dominates that of the object. The *simple property* states that subjects can only read from objects which have a higher integrity class than the subject itself. This results in a *no write up, no read down* overall policy. This prevents the corruption of high-level data from the insertion of lower-quality data coming from the lower levels. An example of this is the policy employed by the initial versions of Microsoft's Internet Explorer. This browser would restrict its own permissions while opening non-trusted websites to limit the potential of system damage. However, the Biba policy has a serious limitation. It only checks integrity compromises caused by improper flows. Real-life integrity violations may come from a wider range of sources.

Water marks

A different, more dynamic implementation of both Bell-LaPadula and Biba might come from the concepts of *water marks*. For BLP, this translates is a *high-water mark*. The policy can be based either on subjects or on objects. In the subject-centric variant, subjects are initialized with a static clearance level and a dynamic clearance level that is initialized as empty. Subjects are still limited to only read the sources allowed by the static class. The dynamic level increases progressively each time they access a higher-level object, matching the class of what they read. This prevents them from writing down the higher-class information they have accessed, because they are limited to writing to objects whose class dominates the dynamic clearance. However, this might become a disadvantage: subjects become increasingly limited in their writing abilities as time passes. In the object-centric variant, there are no static and dynamic clearances. Subjects can only read objects that are dominated by their clearance, and write to any object. After a write operation, the object clearance is set as the lowest upper bound between its original clearance and the clearance of the writing subject. The disadvantage of this object-centric approach is that the use of the system causes a frequent increase in the confidentiality level of resources.

The Biba model, on the other hand, can use a *low-water mark* policy. In the subject-centric policy, subjects are initialized with a static clearance level, and a dynamic clearance level which is initialized at the level of the static clearance. Subjects can only write to objects for which the subject's dynamic

clearance dominates the clearance level of the object. Subjects can read any object, but after reading their dynamic clearance is set at the greatest lower bound between the previous dynamic clearance and the object clearance. This means that the dynamic pivileges of the subject are progressively lowered over time. In the object-centric variant, subjects can only read from objects if the object clearance dominates the subject's static clearance. Subjects can write to any object, but after being written on, their clearance level is set to the greatest lower bound between its previous clearance and the static clearance of the writer subject. This variant does not really protect integrity, but only marks the objects that have been compromised.

Let's now state some other facts about mandatory access that do not fit within any of the previous paragraphs. Both BLP and Biba are known as *multi-level security*. The most famous MAC implementation, used in Android and Linux distros (and adapted by iOS) is SELinux. Covert channels need to be analyzed during the implementation phase. Interface models try to eliminate the chance of covert channels in modelization. The *non-interference principle* states that high-level activity should not interfere at all with lower level (or non-comparable) processes. Many mandatory access control systems, including SELinux, are set up at boot time, to make them harder to alter at runtime.

Multi-level databases

The variant of MAC designed for databases is known as *multi-level database*. The information in the database is given different levels of visibility in the following way. Each data column is accompanied by an auxiliary column that represents the clearance level required to read it. Depending on their own clearance level, users are only shown the columns they are allowed to see. Alternatively, one single clearance column may represent the level of the whole tuple, which might be shown or hidden in its entirety. In the "one clearance per column" variant, the column(s) that provide the primary key need to all have the same clearance level, otherwise this wouldn't be compatible with the underlying relational model. In other words, partial views are not allowed on the primary keys. Additionally, the clearance of non-key attributes must dominate that of the key attributes.

Polyinstantiation must also be supported to provide a finer level of granularity. Polyinstantiation can be defined as the symultaneous presence of multiple objects with the same name but different contents. In the context of relational databases, this translates to multiple tuples with the same key, but with a different clearance over the key (*tuple polyinstantiation*) and/or different values with different clearances for one or more attributes (*element polyinstantiation*). In tuple-level polyinstantiation, lower-level users simply don't see the columns they are not allowed to see. In element-level polyinstantiation, lower-level users are shown a lower-level version of the tuple, containing either a less detailed version of the information or decoy data that serves as a *cover story* for the real information.

Polyinstatiation is usually seen as the possible solution of some problems that may arise in the handling of data in mandatory policy systems. To undestand why, it's important to note that polyinstatiation can be either *visible* or *invisible*. In the case of invisible polyinstatiation, a low-level subject inserts data in a tuple that already contains higher-level or non-comparable data. In visible polyinstantiation, a high-level subject inserts data in a tuple that already contains lower-level elements. Invisible polyinstantiation is tied to three possible scenarios. The low-level subject tries to write or update a tuple, but a tuple with same primary key and a higher clearance already exists; informing the subject that the tuple already exists would be a form of information leakage. Silently subsituting the data with the new values would cause integrity issues. The third possible option is to create a new tuple that will exist alongside the previous. This would be a polyinstantiated tuple. Visible polyinstantiation is also tied to three possible scenarios. The high-level subject tries to write or update a tuple, but a tuple with the same primary key and a lower clearance already exists; communicating the existence of the tuple and refusing the insertion would be a form of denial-of-service. Substituting the tuple would allow information leakage. The third possible option is inserting a new tuple alongside the previous, which is a form of tuple polyinstantiation.

A possible interpretation of what polyinstantiation means is the following. Polyinstantiated tuples could represent different real-world entities, while polyinstatiated elements could represent the same real-world entity. However, polyinstantiation is hard to control. For this reason, multi-level databases like Trusted Oracle have enjoyed very limited commercial success. Other reasons include the higher difficulty of performing data integrity checks and the possibility of inference channels. Alternative, more successful approaches have arisen to substitute polyinstantiation with less problematic procedures.

Tuple polyinstantiation can be avoided by making all primary keys visible, partitioning their domain and only allowing insertion by trusted subjects. Element polyinstantiation can be avoided through the use of restricted values instead of nulls for non-accessible elements and a single clearance level for all key attributes.

A different, architecture-based approach could be used instead of a multi-level database. In a *trusted subject* system, data from different levels is all memorized in a single database. The DBMS and the underlying OS need to be trusted to ensure compliance to the mandatory access policy. In the alternative *trusted computing base* system, the data is partitioned into different databases depending on its level. Only the OS needs to be trusted. Each database is handled by a different DBMS, which is restricted to only accessing the data that its subjects can access. This approach needs auxiliary data decomposition and data reconstruction algorithms.

Administrative policies

Administrative policies define who can grant and revoke authorizations to users. In a *centralized administration* approach, one administrator controls the entire process. In a *hierarchical* scenario, a top-level administrator grants lower-level administration permission to other administrators. Finally, in a *cooperative administration* policy, more administrators need to agree to be able to grant, modify or revoke user authorizations. Another possibility lies in the concept of *ownership*. The owner of a resource can administrate the access to it. This can also take a decentralized form, in which multiple users share the administration of an object, or a hierarchical form in which the resource owner can delegate its administration to others. The possible choices with regards to decentralized administration are the following. First comes the *level of granularity*. Then, potential *restrictions on right concession* could be applied. These are potentially useful, albeit not very common. Another choice is about who can *revoke authorizations*. Revocations could be handled either by all administrators, only the original granter, or others. Finally, there's the issue of *dangling grants*. These can be defined as grants made by someone whose granting right is subsequently revoked. Possible approaches include canceling all grants down the chain, leaving them active but marking them as potentially unsafe, or ignoring the issue all along.

In a DAC policy, authorizations may embed restrictions that limit their validity. These may be *location-dependent*, *history-dependent* or *content-dependent*. System-dependent restrictions evaluate the matching of system predicates, including time and location. An example might be the conditional application of GDPR-compliant policies for users that access a service from within the EU. History-dependent restrictions are based on previous requests. An example may be rate-limited access to prevent DDoS attacks. Content-dependent restrictions are based on the content of data. This may translate to the restriction of access on certain objects, or to certain parts of those objects. In the context of databases, data-dependent restrictions may be achieved through *query modification*. For example, a public healthcare doctor may access only the tuples in the national database for patients that have him recorded as their primary healthcare provider.

In hierarchy-based administration, permissions may be applied to *groups* instead of individuals. The hierarchy itself can be represented by a direct acyclic graph (DAG), because a tree-like structure is usually not apt to record the complex dynamics of the hierarchy. For instance, users may be part of multiple groups. For this reason, Linux separates between a EGID (Effective Group ID) and a EUID (Effective User ID) when using discretionary access. Different policies may be adopted for *non-minimal elements* of the hierarchy, which include non-leaf nodes, like folders. For example, in Unix-based systems, privileges on a directory do not automatically propagate to its contents, but may be required in order to obtain privileges on the contained elements.

Groups are a useful abstraction because they simplify the management of multiple users who need to have the same privilege. Exceptions to the group policy can be supported by the use of *negative authorizations*, which specify restrictions from the baseline group policy for single group members. These are especially useful in large groups, where trying to exclude one single user without resorting to exceptions would mean to give an authorization to all the other individuals within the group except for the excluded one. Negative authorizations were initially introduced separately from regular authorization. The base policy can be either *open* or *closed*. In an open policy, all users are authorized except when explicitly denied access. In closed policies, the opposite happens: all are resricted by default, except for those on an authorization list. Modern policies support both. One example is the Apache web server, which allows both negative and positive authorizations. Its two models are known as *deny, allow* abd *allow, deny*. In general, this dual-policy model can cause inconsistencies. For instance, a user may get both a positive and a negative authorization over the same resource. It may also create incompleteness, in the form of an unspecified authorization over an object, neither positive nor negative. There are two possible solutions to these issues. One is *completeness by default*: each access needs to have specified either a positive or negative authorization. This approach is very heavyweight. The other is to assume either an open or a closed policy by default, to be used wherever explicit authorizations, either positive or negative, are not explicitly marked.

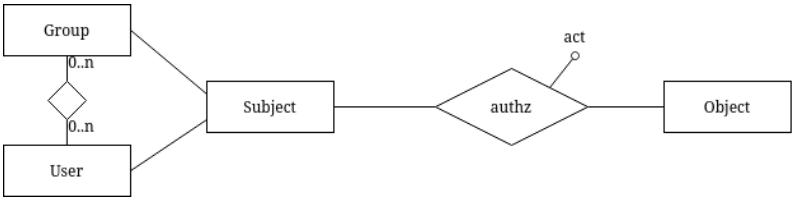
Some conflicts may arise while evaluating policies. There are multiple available strategies for conflict resolution. The first is known as *denial-takes-precedence*. On conflicting policies, request accesses are denied by default. This is fail-safe, but potentially overrestrictive. The second strategy is known as *most-specific-takes-precedence*: the most specific authorization wins the conflict. A variant of this, known as *most-specific-along-a-path-takes-precedence*, is used in file systems that support soft (symbolic) links. A

file may be accessible when seen through the soft link within a directory the user has access to, but not in the original directory if it is not visible to the user. Conflict resolution techniques are not mutually exclusive. They may therefore be applied in a certain order: if one does not bring a resolution to the conflict, the next is attempted.

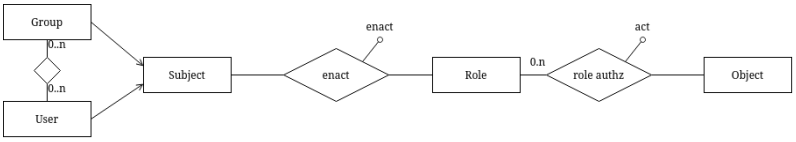
Recent DAC models allow at least the following: both positive and negative authorizations, hierarchy-based propagation, conflict resolution and decision policies, additional implication rules. The main goal is to be flexible and allow for different policies. This helps with dealing with the different requirements for users and administrators, but also with the same user or administrator needing different policies over different objects.

Role-based Access Control (RBAC)

This is an entity-relationship diagram of regular, group-based system:



The following is an entity-relationship diagram of a role-based system:



The main idea is to decouple the identity from its *role*. For example, university professors can assign grades not because of their personal identity, but because they are assigned the role of professor within the IT system that manages grades. This model provides a higher grade of flexibility. If the teacher of a course changes, the role of course owner can be reassigned to the new professor, because all of the associated data and privileges are tied directly to the role, instead of belonging personally to the previous teacher. In other words, the connection between subjects and objects is dynamically mediated by the roles. Many modern operating systems have some form of the "run as Administrator" (Windows) or "sudo" (Unix-like) command, which is an example of role-based access. The personal account of the user is still the same, but the command temporarily increases their role from regular user to system administrator. In regular access systems, groups can be seen as static collections of users. Similarly, in a role-based system, each role can be seen as a dynamic collection of privileges. Like users and groups, roles may themselves be organized hierarchically through *subroles* with privilege inheritance. Modern systems usually also provide support for *least privilege* policies. The higher level of flexibility provided by a role-based system comes at the obvious cost of putting an additional layer of complexity within the system.