Secure the Border and Build the Wall

A Compendium of Access Control on Unix-Like OSes

Patrick Louis

2023-02-28

Published online on venam.nixers.net

©Patrick Louis 2023

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of the rightful author.

First published eBook format 2023

The author has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

Introduction **5**

Access Control In Operating Systems **9**

Models **11**

Proving Who We Are **17**
- The Password And Group Files . . . 17
- Dynamic/Pluggable Authentication . . . 21
  - BSD Auth . . . 21
  - PAM . . . 22
- Super-User and Switching Subject/Domain/User . . . 27
  - Super-User Concept . . . 27
  - setuid and setgid . . . 28
  - `su` and `newgrp` . . . 31
  - `doas` and `sudo` . . . 33
  - SunOS Derivatives Profiles . . . 38
- Identity and Access Management Solutions . . . 40

System-Wide Access Control **43**
- Background Knowledge and Theories . . . 43
  - Access Control Lists, Access Profiles, and Flow Policies . . . 43
  - Discretionary Access Control and Mandatory Access Control . . . 44
  - Role-Based Access Control . . . 45
  - Attribute-Based Access Control and Context-Based Access Control . . . 46
  - Capability-Based Access Control . . . 47
- Basic File Permission . . . 47
- POSIX(IEEE 1003) 1e and 2c . . . 50
- POSIX.1e/2c Access Control Lists . . . 52
- POSIX.1e/2c Capabilities . . . 57
  - POSIX Capabilities on TrustedBSD . . . 60
  - POSIX Capabilities on Linux . . . 62
  - POSIX Capabilities on SunOS Derivatives . . . 69
- POSIX.1e/2c Mandatory Access Control . . . 71
  - POSIX.1e/2c Information Labeling & Extended Attributes . . . 73
  - Mandatory Access Control on BSD . . . 75
  - Mandatory Access Control on Linux . . . 81
    - Linux Security Module Interface . . . 81
    - SELinux . . . 82
    - AppArmor . . . 98
    - Other LSMs . . . 105
    - RSBAC Another Linux Modular Security Framework . . . 110
  - Mandatory Access Control on Other Unix-Like Systems . . . 114
- OpenBSD relationship with POSIX.1e/2c . . . 116
- Particular Role-Based Access Control . . . 117

RBAC on SunOS Derivatives . . . 117
RBAC on Linux using RSBAC Framework . . . 119
RBAC on Linux using GrSecurity . . . 120
Capability-Based Security . . . 126
Capability on FreeBSD using Capsicum . . . 128

Putting in Boxes: Isolation and Constraints as Access Control **132**
Classic Constraints . . . 133
Resource Limits . . . 133
Niceness . . . 133
ulimit, rlimit, and sysctl Tunables . . . 134
File System Quotas . . . 137
`chroot` . . . 139
Isolation on OpenBSD . . . 141
`systrace` . . . 141
`unveil` & `pledge` . . . 144
Isolation on FreeBSD . . . 145
Capsicum as a Sandbox . . . 145
FreeBSD Jail . . . 146
Isolation on Linux . . . 149
Linux Control Groups . . . 149
Linux Namespaces . . . 155
`landlock` & `seccomp` . . . 159
Linux Software Relying on Isolation . . . 167
Isolation on SunOS Derivatives . . . 171
Solaris Projects & Pools . . . 171
Solaris Zones . . . 173
macOS and Android Sandboxes . . . 175
Virtual Machines as Sandboxes . . . 177
Image-Based OS & Immutable Distro . . . 177

Action-Based Access Control **179**
SunOS derivatives auths . . . 179
Polkit/D-Bus . . . 181
macOS Extension Points & Android Intents . . . 187

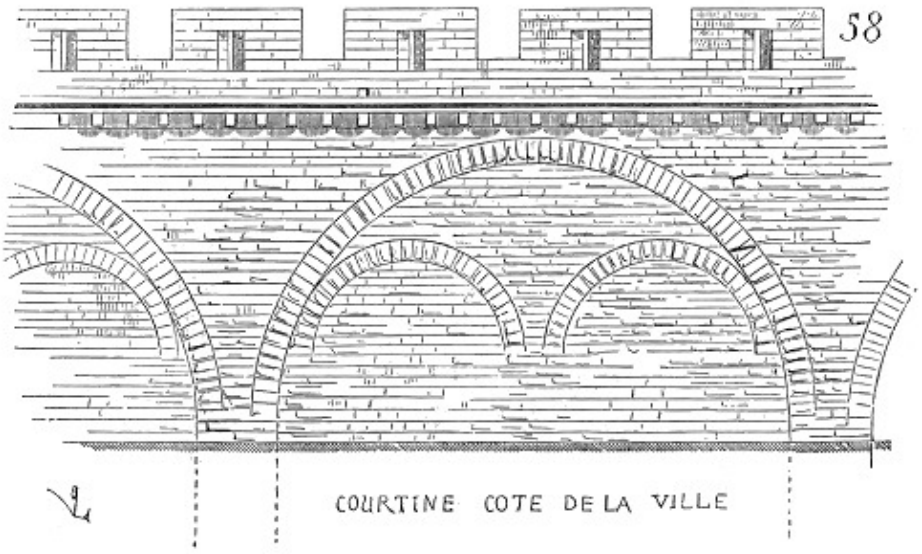
After the Facts: Logging & Auditing **189**
Classic System Logger . . . 189
POSIX.1e/2c Auditing . . . 190
Basic Security Module Auditing . . . 191
Linux Auditing System . . . 193

General Security & Trusted Computing Base **197**

Conclusion **199**

Bibliography **200**

Introduction



Plenty of cheesy quotes often say that total security stands on the opposite of total freedom. Undeniably, in computers and operating systems this is a fact. However, universal privilege used to be the norm, and restricting actions was a concept that wasn't part of the vocabulary. Today, this idea is a must. Our machines are constantly interacting with the external world, exchanging information, and deliberately fetching and executing pieces of code and software from servers hosted in places we might never visit. Meanwhile, we trust and intertwine our lives with these machines.

A system that is trustworthy is not the same as a system we must trust. This distinction is important because systems that need to be trusted are not necessarily trustworthy.

This article will focus on the topic of access control on Unix-like systems. Sit back and relax as it transports you on a journey of discovery. We'll unfold the map, travel to different places, allowing to better understand this wide, often misunderstood, and messy territory. The goal of this article is to first and foremost describe what is present, allowing to move forward, especially with the countless possibilities already present. How can we better shape the future if we don't know the past.

To facilitate the reading and skimming, every section ends with a quick summary labeled "**What you need to remember**".

There are 8 parts to this article. In the first few ones we will go over theories such as what security and access control mean, along with different models of how to represent them. Then we have a section on the subject, proving someone is who they say they are. Afterward we move to practical access control with 3 sections: **system-wide**, **isolation/constraint**, and **action-based**, which will be used to categorize the mechanisms employed. Lastly, we'll finish with sections on auditing and logging, to see what happens after the facts, and finally give generic OS and hardware security tips, because otherwise all that was mentioned before would be useless.
You can think of the progression as a chronological one, from a user proving who they are by authenticating, to interacting with the system, and then leaving traces on it.

We'll start our introduction by pondering on what security means, and afterward continue to the main topic.

In everyday talk, security and computer security are ill-defined abstract concepts. Even experts don't all agree on what they mean. Additionally, as with anything scientific, what you can't measure, quantify, and experiment doesn't exist. Therefore, multiple standards, accreditations, definitions, jargon, guides, evaluation schemes, principles, best practices, and models have been created, not all coherent with one another.

For example, as a teaser, the following series of words could refer to different concepts depending on the context, which can lead to confusion.

- Protection
- Permissions
- Privileges
- Policies
- Capabilities
- Policies
- Trust
- Ownership
- Access
- Authentication
- Authorization
- Limits
- etc..

The standards and accreditations have separate ways to evaluate the levels of security of a system, each focusing on different aspects. Some popular ones include: NIST FIPS 140-2 security requirements for cryptographic modules, multiple ISO certifications such as ISO 27k for information security, the famous PCI DSS that is targeted at the payment card industry, the Common Criteria framework (aka ISO/IEC 15408), etc..

As far as access control standards go, the Common Criteria framework, which replaced the older Orange Book (aka TSEC, Trusted Computer System Evaluation Criteria), is the international de facto. The testing laboratories, which evaluate the claims companies make about their products, are scattered around the world and they mutually agree through a treaty to recognize each others' security assessment results. Common Criteria is mandatory for software used within some government systems and types of industries.

Let's consider it a good base and extract the generic definition of a secure system from the Orange Book (TSEC):

> A secure system will control, through the use of specific security features, access to information such that only properly authorized individuals, or processes operating on their behalf, will have access to read, write, create, or delete information.

The evaluation schemes grade the level of security of systems based on how they apply policies, which are rules and practices on the system. These policies are used as the definition of security. For instance, the classic principle of least privilege or the CIA triad (Confidentiality, Integrity, Availability).
The principle of least privilege dictates that subjects should be given just enough privileges to perform their tasks, it ensures failure will do the least amount of harm. Privilege, sometimes also called permission or rights, is loosely defined as the abstract ability to perform a task, whichever form it takes; a key concept in access control. This is also linked to the idea of compartmentalization, separating entities from one another.
In CIA: *Confidentiality*, is a property of objects/information only getting to where it's supposed to, conserving privacy. *Integrity* is the property of the object not being tampered by unauthorized parties and how the data should reflect and maintain its consistency. Finally, *Availability* is another property of objects that makes it accessible and usable upon demand.
How these vague terms apply and are interpreted depends on the developed policy description, one that would allow it to be measured and controlled.

The policies are then implemented, proved, and formalized using a security model, The model could then be mandated, or not, within the scope of the evaluation. A security model is used to determine and visualize how security is applied, the relations between subjects and objects access. This is also called access control theory by some.
We'll take more time to dive into models later, but for now, an example would be a state machine of who controls which resources along with transitions, or markings/labels on objects, or a matrix of users and resources on the system. More on this later..

Apart from the policy and models, TSEC and Common Criteria have additional requirements that should be included in a secure system:
- Accountability, how individual subjects (users and processes) are identified, along with related Auditing of their actions on the system. - Assurance, in the form of hardware/software mechanism to enforce other requirements, along with continuous protection of the life-cycle of the system.

The systems are then graded in one of four divisions that represent the strength of the certification. These can even go up to formally verified systems; in the Orange Book they are: D, C, B, A, with A being the highest security level, while in Common Criteria EAL1 is the lowest and EAL7 is the highest.
When it comes to Unix-like systems, multiple versions of RHEL meet the Common Criteria, some certified under BSI with Protection Profile at EAL4+ others by the NIAP (National Information Assurance Partnership).

As you can see, calling a system "secure" isn't straight forward, nor is measuring it. Certifications such as Common Criteria don't really measure the security of the system but simply state at what level the system was tested and against which requirements.
Security is a moving target on an axis that is constantly extending. Furthermore, even certifications like the ones mentioned could be criticized to be *security theaters*, focusing essentially on documen-

tation and evidence rather than day to day operation.

Let's move on from the abstract concept of security to the particular topic of operating system access control.

What you need to remember: *Defining "security" is not straight forward, there are a lot of standards and specs, each with their own criteria and levels. However, one thing is in common: it's about applying a policy of who has access to what and respecting it.*

Access Control In Operating Systems

In this section we'll get a better idea of what access control in operating systems means. The best way to understand it, in my opinion, is to think of the following three ideas: the goal, the entities in the equation, and the domain where they're at play.

The part of security related to access control, as we've seen, emphasizes on making sure the right subject is able to perform the actions they need on the right object; indirectly it also means blocking access to those who shouldn't.
This is the Goal: to prevent malicious misuses of the system by applying the right policy.

> Access control is defined by the National Institute of Standards and Technology (NIST) as the set of procedures and/or processes that only allow access to information in accordance with pre-established policies and rules (among multiple other similar definitions).

There exist multiple entities that join together to achieve this goal. There are what we call subjects, or sometimes called "principal", something doing an action, which is usually a user, group, role, or a process/program. On the other side there are objects, usually passive entities on which the action is applied, these can be system resources, files, devices, programs, and even other users or the set of actions themselves.

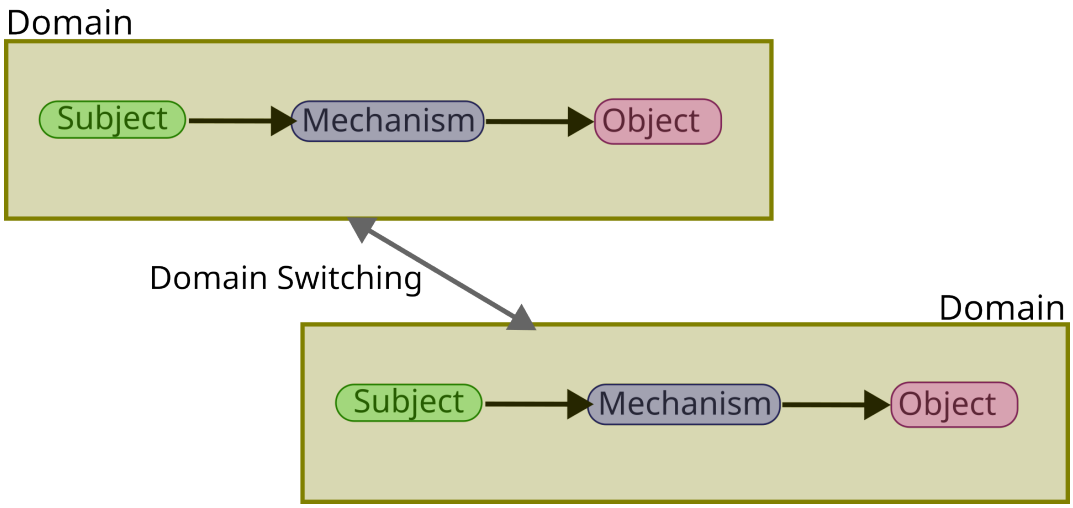
These two interact through a mechanism or facility, the entity that allows the action and gives it form. This mechanism is surrounded or includes ways to enforce rules, the guiding policies. Often these are described using models to explain the flow and prove that the mechanism functions properly.

> Mechanisms determine how something is done; policies dictate what is done. Flexibility requires the separation of policy and method.

The mechanism and its inner workings should follow what's called the principles of protection, it's the mindset used to perceive the overall system access control and good practices, basically how it applies the policy. It can contain the concept of privilege or permission: being able to do some task or not. The privileges a subject has, or is able to get, are its access rights.
Additionally, the mechanism could also include rules on the modes of operations which would be defined according to the types of objects/resources on which the action is done.

All of these: subjects, objects, and mechanisms, are part of a domain of protection; The "Who", "What", "Where", "When", etc.. Policies are applied within each domain, which could possibly be different. In that case, we could have higher-level mechanisms that would allow switching between domains.
Sometimes, in literature, subjects are called domains. They could live and pick their own rules, under their control, in a dynamic way. Switching user could be said to be switching domain, think of setuid/setgid, which we'll come back to later. This is in contrast with a more static application of policies such as the standard Unix permission model, or the more rigid Mandatory Access Control, which we'll also dive deeply into later.

These three ideas are abstract: the goal, the entities, mechanisms along with how to imagine them in a domain, yet they make it much simpler to understand all the access control content that will follow in this article so keep them in mind.



What you need to remember: *Access control in OS has the goal of preventing malicious usage through a policy, the actors are the subject, objects, and they interface together through a mechanism/facility. These all exist within a domain, changing domain would mean changing some of these params*.

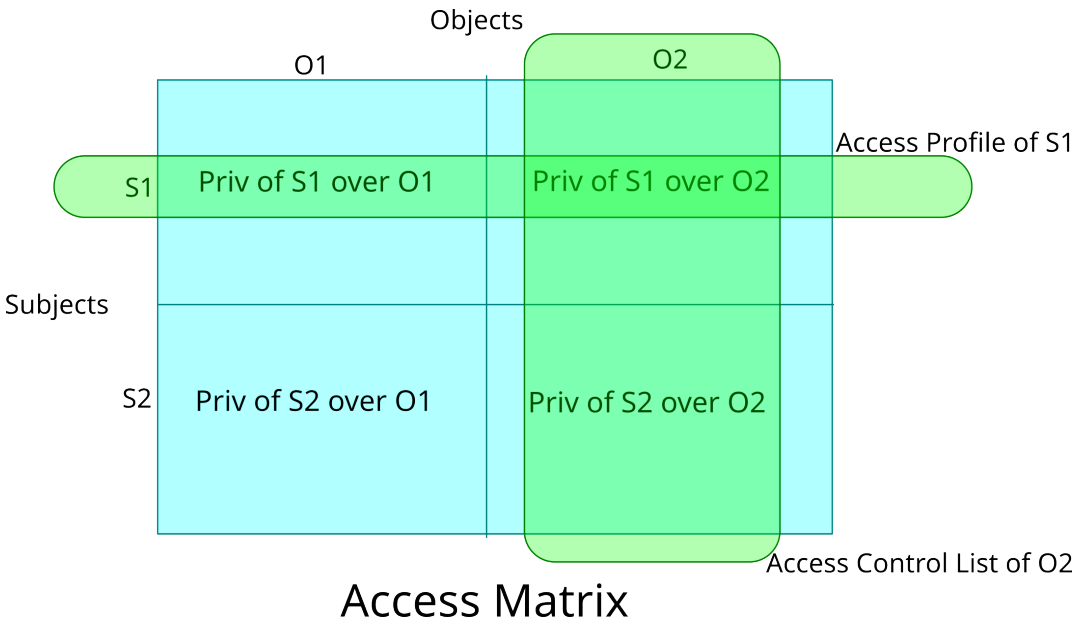
Models

In this section we'll discuss, with big brush strokes, a couple of security models.
Models are used to make sense of the policies and their implementations. They describe how the mechanism/facility protects the system by juggling the interaction between subjects, objects, and other possible entities. It's a more formal description of what's going on in the system. However, keep in mind that the map is not the territory, none of these representations are perfect, nor address all security issues.

For now, don't worry about the actual programming aspect, we'll come back to it in the next sections. You can still imagine how each model would take concrete form, it's a painful read but a rewarding one. Often multiple models can be used to describe the same system. Sometimes the implementation exists previous to the model, and sometimes it's the opposite, the model impacts how the mechanism of protection is conceived.
We'll walk through a couple of examples, covering as much ground as possible.

Let's start with one of the most intuitive model, the access control matrix model.
As the name implies, it's a matrix, column and rows. The columns represent the objects, the rows the subjects, and the entries in the matrix indicate the privilege/permission/rights that the subject can exercise on the object. So far that's simple enough.

The model gives two views, if you look at it column-wise we can see what's called an access control list for the object: who can do what on the object. If we look at it row-wise we can see what's called an access profile for the subject (or capability list): what a subject has the ability to do on which object.



Access Matrix

Concretely, it could get translated in a programming implementation that stores the access matrix as a bitmask, or any other kind of structure, on every file in the system and it would contain the possible

conceptions of subjects (user, group, roles, etc..) along with their privileges, the access control list (ex: standard POSIX permission, or POSIX.1e ACL, if you think about them in this novel way it can be mind-bending).
Or it could be a system-wide storage matrix, or one existing in the subjects themselves, that would contain all objects that the subjects currently have access to, the access profile (ex: capability-based security). We'll come back to this topic.

Extending on those, subjects in the matrix could be allowed to dynamically switch role with another subject, sometimes called domain switching. There's also the possibility to have a rights-transferring mechanism, copying them from one subject to another, which can include restriction on the propagation of the rights (how far and for how long they can propagate). In these cases, the matrix becomes dynamic.

In general an access matrix is considered an incomplete description of security policy as the model doesn't really enforce rules but simply describes the current state.

A model that is more formal and expands on the access matrix is the Graham Denning (G-D) model. Similar to the typical access matrix, rows are subjects and columns are objects, and the entries/elements are the rights of the subject on the object.
However, the model adds a set of 8 protection "rules" to the mix, which in this model actually means "actions", so that the description of the policy becomes more complete. Along, it also defines normal subjects as "users" and a new special subject role called "controller", which is a user that has ownership over other users.

Rules, which are about how to perform actions, are associated with preconditions to make sure the rights are respected. When a rule is executed, for example creating an object, the matrix is changed accordingly. The rules are as follows:

- Create an object.
- Create a subject.
- Delete an object.
- Delete a subject.
- Provide the read access right.
- Provide the grant access right.
- Provide the delete access right.
- Provide the transfer access right.

After Graham Denning model, another model went further on these ideas, the Harrison-Ruzzo-Ullman (HRU) model. It dissected the rules into more primitive operations (called commands, of which there are 6) and conditions, making it more like the ACID of database transactions.
It is described formally using mathematical procedures.

- Subjects: `S`
- Objects: `O` (`S` can also be considered an `O`)
- Rights: `R`
- Commands: `C`
- Access Matrix: `P`

The current system then exists in a "configuration" defined by a tuple `(S, O, P)` and can change through commands with preconditions.

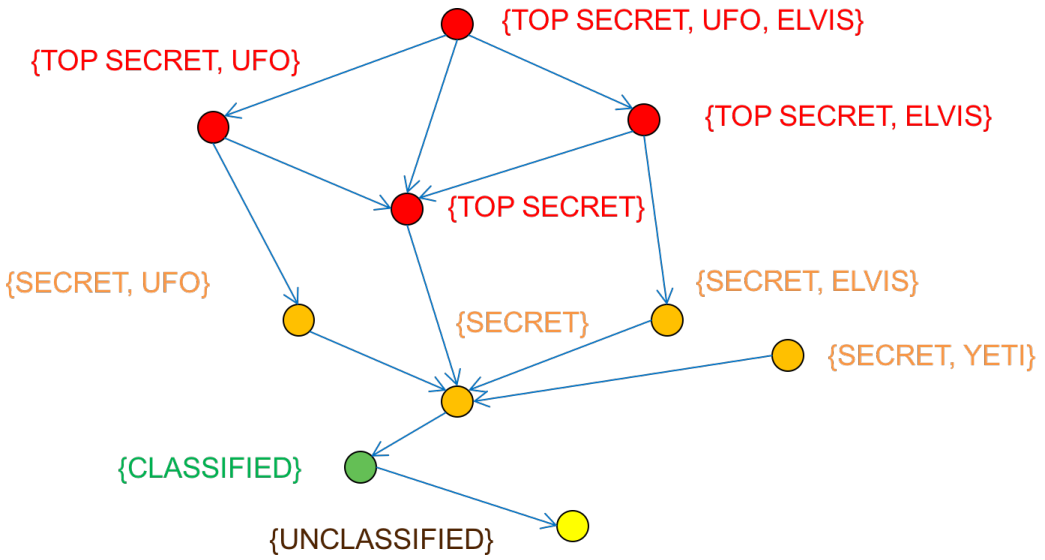
These two models, G-D and HRU, are better than a plain access-matrix at visualizing whether or not the system stays in a secure state and can later on help thinking of the algorithms to use when doing a software implementation.

Another category of models are state machine models, based on finite-state machines. These models emphasize the transition between states based on action and secrecy. It's a more dynamic view where permission/privilege is always on the move depending on the state of the system. In these models we think in terms of levels/layers of access, gained or revoked rights, and classification of subjects and objects (aka non-discretionary or mandatory control, explained later). When every state transition, from booting to power off, is proven to be secure then, by induction, the whole system is secure. That's the notion of "secure state".

There are many ways to visualize a finite-state machine, usually it's better that it's shown as deterministic (DFA). This can be done in a table showing all transitions, or as a graphical representation. You could go back to your CS course on automata for a refresher.

Yet, as with the access matrix, a state machine on itself isn't a very complete description of a security policy. A more formal category that extends from the state machine model are information flow models, also called latticed-based information flow models. This includes, among others, the Biba model and the Bell-LaPadula model.

The information flow models consist of objects, state transitions, and flow policy states. Basically, it's a state machine that makes sure the information can't flow in the wrong direction, avoiding unauthorized access. This is done by governing how subjects can get access to objects by verifying if the security level criteria matches.
Additionally, these flow models can live along other models and other systems, interoperating with them through the use of a guard in between, which is yet another fancy name for any sort of mechanism/facility that decides if something is allowed or not.



Source: Thinking about Security, Paul Krzyzanowski, January 31, 2022

One of these models is the Bell-LaPadula model. It's a model that focuses on information confidentiality. Information is perceived as an object that is tagged/labeled in categories. Users are also tagged/labeled in categories, and both of them are used to classify how the information can be accessed based on its sensitivity.
Example of classifications:

- Sensitive but unclassified
- Confidential
- Secret
- Top secret

The flow of information always validate the security clearance so that the subject level is allowed to access the object/information level. This is why the Bell-LaPadula model is often mentioned as a formalization of Multi-Level Security (MLS).

The Bell-LaPadula model is particular in that its authors, David Bell and Leonard LaPadula were the ones that started formalizing the idea of secure system state and models. Their work directly influenced the TCSEC/Orange book which we mentioned earlier, and it was initially made to evaluate criteria of MLS systems.

The term MLS can refer to two things, either the security environment/mode, the original definition, or as a capability. The environment is one in which the community has multiple levels of security and needs clearance to access information. The capability, on the other hand, is about the system itself supporting mechanism to implement and enforce an MLS model.

Back to the Bell-LaPadula model, the information flow rules property that we can apply to subjects and objects with labels are the three following:

- The Simple Security Property (`ss` property): A subject at one level of confidentiality can't read information at a higher level of confidentiality. Aka "no read up".
- The Star Security Property ($*$ property): A subject at one level of confidentiality is not allowed to write information to a lower level of confidentiality (thus the level of confidentiality never goes down). Aka "no write down".
- The Strong Star Security Property (Strong $*$ property): A subject at one level of confidentiality is not allowed to neither read or write information of higher or lower confidentiality.

Furthermore, Bell-LaPadula model enforces that subjects and objects can't change their levels of classification while they are being referred to. This is called the tranquility principle, which can be weak or strong.

The model has its limit, it didn't make a difference between the idea of general security and protection of data integrity, it emphasizes confidentiality and controlled access in MLS environment only.

Another model that also relies on the information flow paradigm is the Biba model. It emphasizes the policy of integrity so that data remains internally and externally consistent, there's no unauthorized changes, and the information/process given the same input produces the expected output.

In the Biba model (also called low water-mark policy), similar to Bell-LaPadula, data and subjects are categorized, this time into levels of integrity. The design is such that if something is at a higher level it cannot get corrupted by a lower level subject.

The properties that could be applied for the flow of information to ensure the data integrity are as follows:

- The Simple Integrity Property: A subject at one level of integrity can't read information at a lower level of integrity. Aka "no read down".
- The Star Integrity Property ($*$ property): A subject at one level of integrity can't write information to a higher level of integrity. Aka "no write up" (What is higher keeps its integrity at that high level).
- The Invocation Property (Strong $*$ property): a subject at one level of integrity can't invoke/request a subject at a higher level of integrity. Aka a subject can't be promoted to a higher level, only to equal or demoted to lower level.

As you can notice, the first two properties are the reverse of the Bell-LaPadula model. For integrity we had: "no read down, no write up" and for confidentiality: "no read up, no write down".

Someone came up with a good metaphor for Biba:

> After a long journey on your search for Shangri-La and true security awareness, you arrive at a Tibetan monastery. You discover the monks are huge fans of the Biba model and as such, have defined certain rules that you, the commoner, must abide by.
>
> - A Tibetan monk may write a prayer book that can be read by commoners, but not one to be read by a high priest.
> - A Tibetan monk may read a book written by the high priest, but may not read down to a pamphlet written by a commoner.

The Clark-Wilson model is another model also focused on addressing the goal of integrity. However, it's using a more holistic abstract approach than the information flow. It tries to formalize what information integrity is, how the data items in the system should be kept valid.
The model uses security labels to grant access to objects via transformation procedures and a restricted interface model. It adds to Biba an enforcement of separation of duties: subjects must access data through applications, and auditing of their actions is required. More elements are defined as part of the model: users, applications, duties, etc..

Clark-Wilson achieves this through access control triplet. The access control triplet is composed of the user, transformational procedure, and the constrained data item.
Authorized users/subjects cannot change data in an inappropriate way (we'll dive later into what authorization is in the next section, for now keep in mind it means making sure we know who we claimed we are). Subjects are restricted to their own domain, a subject at one level of access can read one set of data, whereas a subject at another level of access has access to a different set of data.

Modification in that model only happens through a small set of programs. These programs perform well-formed transactions, which are the transitions keeping the system consistent.
Multiple roles in the models are assigned to achieve this. To keep the internal consistency there is a concept of Integrity Verification Procedure (IVPs). The data is change through a Transformation Procedure (TP, sort of like relational db ACID that we mentioned in HRU), on which the data integrity is checked for Constrained Data Items (CID), and there could be items/objects outside the model seen as Unconstrained Data Items (UDI).
This model is interesting as a lot of implementations employ this mindset of only allowing change by passing through a set of specific applications (see action-base access control section).

There are a lot of other models to discover such as the Take-Grant model, the Brewer Nash model (Chinese Wall model), and the NIST RBAC model, but we'll keep it at that for now.
As you can see there exists a lot of different ways to visualizing access control. Let's move on to one of the pieces of the equation: subjects.

What you need to remember: *There exists a lot of models which are used to describe how the security policy (a definition of security) is implemented and prove it's working as expected. Two main categories exist: the matrix and the information flow. The information flow focus on levels of access while the matrix focuses on rights. Understanding models can help visualize the implementation.*

Proving Who We Are

We've mentioned subjects before, however we haven't dived into who they are and how to make sure of who they say they are. This is what we'll tackle in this section.
To know who a subject is and let them pass the gates leading to a system, we need to discuss *identification*, *authentication*, and *authorization*.

Identification is the process of being able to indicate the identity of a person or a thing: what makes it unique. This is a generic term, that is more human than computer-related.

Authentication (`authN`) is the act of proving to a certain degree of confidence an assertion, verifying that something is what it claims to be: the identity of a subject or any other assertion.
There exists plenty of ways to achieve this, we talk of authentication factors. They are the following: the knowledge factors (ex: password, pin) the ownership factors (ex: security token, ID card), the inherence factors (ex: fingerprint, voice, DNA, retinal pattern), and the geotemporal factors (ex: place and time). In common parlance it's the "something you know, something you have, something you are, and when/where you are" (the last one often omitted from text books).
The more of these you can mix, the more multi-factor the authentication, the more it is considered strong; A single-factor authentication is weak. Additionally, the authentication process can either happen once or continuously, asking again from time-to-time.

While *authentication* is the process of verifying that "you are who you say you are", *authorization* (`authZ`) is the process of verifying that "you are permitted to do what you are trying to do". Often, *authorization* happens immediately after *authentication* (ex: upon login), but this does not mean *authorization* presupposes *authentication*: an anonymous subject could be authorized to some limited privileges.

To sum it up, we got the list of subjects (*identity*), asserting that they are who they claim to be (*authentication*), and finally checking if they have access, granting them privileges (*authorization*). Access control is about having a system only used by those authorized and detect and exclude unauthorized usage, as we said before.
On Unix-like systems there's a couple of ways to achieve the above, from `passwd` file, `login.conf` and `login.defs`, to BSD Auth and PAM, passing by `su`/`sudo`/`doas` along with special identity management solutions.

What you need to remember: *Identification is a generic word relating to proving the identity of something, its uniqueness. Authentication is the assertion of a claim, usually an identity claim. Authorization is about checking if the subject has enough privileges*

The Password And Group Files

We can start with the classic username-/group-password combination provided by POSIX. This is the classic way to describe the set of user accounts and groups on a system through the `/etc/passwd` and `/etc/group` files. The passwd file contains a list of users with info and the group file similarly

contains the groups with their info. These are the basic subjects that can interact in a system: users and groups. A user can act either as itself or as the group it is part of.

Most Unix-like systems have them (`/etc/passwd` and `/etc/group`) and they have a typical layout representing a standard structure from POSIX that should be returned when using the functions to read them: `getgrent` for `struct group` and `getpwent` for `struct passwd`.

The files are textual and contain rows (records) that have fields separated by colon `:`. The `/etc/passwd` file has the following fields:

- Username aka login. The login name is usually a small string that starts with a letter and consists of letters, numbers, dashes and underscores. In general it's a bad idea to have a dash ('-') at the beginning, and it's better to avoid uppercase characters and dots within the username so that it doesn't mess with the behavior of certain programs and the shell.
- Encrypted password (if present)
- User ID (UID, a number in decimal)
- Principal group ID (primary GID, also a number)
- GECOS field (General Electric Comprehensive Operating System field). A deprecated field that is used these days for comment and random user information. It can be used by utilities such as `finger(1)`.
- Home directory
- Login shell to use

And the `/etc/group` file has the following fields:

- Group name. Should follow a similar convention as the user name.
- Encrypted password (if present)
- Group ID (GID, a number in decimal)
- A comma-separated list of user names (users who are in this group)

The username, UID, group name, and GID are theoretically unique values and can be used as reference to the identity of the subjects. However, on some systems it is still possible to have multiple entries with the same values, but it is considered a logical mistake to do so and can lead to several security issues.

When a user executes processes, they inherit the same UID as the one of the user. That is true unless privilege is dropped (as we'll see in the super-user section), or if some special mechanism allows changing it (as we'll see in the setuid section).

By convention, certain UIDs have special meaning. For example, the Linux Standard Base Core Specification says that the values between 0 and 99 should be allocated by the system and not created by applications, while UIDs ranging from 100 to 499 should be reserved for dynamic allocation. Other systems and daemons have different conventions, Debian uses the 100 to 999 as dynamically allocated system users and group range. As for FreeBSD, the range that should be used by package porters is 50 to 999, and macOS start allocating new UID starting from 500. There's really nothing standardized across systems.
Apart from this, specific UIDs could mean particular things, such as negative ones which are often used to specify unallowed or blackhole users, such as `-1` that is unallowed, and `-2` often used for the `nobody` user.

While these files and their formats are simple in themselves, they are also world-readable and this

causes a lot of vulnerabilities. In the past it was hard to crack passwords, but these days having access to a hash will eventually lead to it being cracked. That is why today, most Unix-like systems don't store the passwords as-is in the files but have them in a separate place that can only be read with elevated privileges (super-user/root), this additional file also has new configurations that are useful for password policies and features.
What these files are is system-dependent and isn't mandated by POSIX.

The password in the files `/etc/passwd` and `/etc/group` have been replaced with any character that isn't a valid hash, such as `x`, `!`, and `*`. It is then interpreted, or not, as having special meaning. If the character isn't valid and can't be interpreted then the user is locked.
Note that an empty password field means that the user or group can be used without entering a password (if no other mechanism on top is in place to disallow empty passwords, such as PAM as we'll see).

Practically, Solaris uses the `/etc/shadow` file to store the passwords of users along with configurations related to password.

The same is true for Linux, which copied Solaris, it uses `/etc/shadow` for securely storing user passwords, and `/etc/gshadow` for group passwords. It also contains information such as the age of the password, the last login, inactivity, expiration, etc.. These can be manipulated with the configuration file in `/etc/login.defs`, the shadow password suite configuration, or through command line utilities such as `chage` to change the password expiry information for example.
There exists a couple of scripts to convert to-and-from shadow passwords and groups such as `pwconv`, `pwunconv`, `grpconv`, and `grpunconv`. However, these days, everything is directly stored in secure shadow files, so there's no need to convert back and forth.

The `login.defs` configuration file of the shadow password suite is important as it contains control knobs for the behavior of most of the utilities related to passwords and accounting. It is a text file with key/value entries.
For example, `ENCRYPT_METHOD` specifies which algorithm to use to encrypt the password. `FAIL_DELAY` create a delay before allowing another attempt between login failure, `LOGIN_RETRIES` is the maximum number of bad password retries, `PASS_MAX_DAYS` is the maximum age of password before being forced to be changed, etc..

On BSD systems, we have something similar to the separation done on Solaris and Linux that is achieved through the files `/etc/master.passwd`, `/etc/pwd.db` and `/etc/spwd.db`. The policy and behavior is controlled through the configurations in `/etc/login.conf`.
The `master.passwd` file is where all the passwords and user related information is stored and it is then used to generate two files using `pwd_mkdb(8)`: one in a secure and the other an insecure database format, `/etc/spwd.db` and `/etc/pwd.db`. The insecure database file in `/etc/passwd` is generated at the same time, removing fields such as the encrypted password, replacing them with asterisk `*`.

`master.passwd` is readable only using elevated privileged and it is a text files containing colon-separated records with the following fields, which are in extension of `/etc/passwd`:

- name: User's login name.
- password: User's encrypted password.
- uid: User's login user ID.
- gid: User's login group ID.
- class: User's general classification (we'll dive into it later).

- change: Password change time.
- expire: Account expiration time.
- gecos: General information about the user.
- home dir: User's home directory.
- shell: User's login shell.

Also, like Linux, the behavior of the tools used to manipulate user accounts and their password policies are controlled in a config file, this time it's `/etc/login.conf` (system-wide) and `~/.login_conf` (local), the login class capability database.
This file contains more than this, as we'll see in a bit with BSD Auth. As far as password control goes, it can also set the password cipher to use `localcipher`, the `passwordtime` used for expiry date, `idletime` as maximum idle time before automatic logout, and much more. It even contains a way to only allow specific hosts to login using specific users, which pertains to our previous discussion of the "when/where" of authentication (similar to postgresql `pg_hba.conf` for those familiar).
In general, there are many more password related configs in BSD `login.conf` than there are in `login.defs`. Yet, that doesn't make much a difference, because `login.conf` is used for BSD Auth and not only as a shadow password file, its real counterpart is PAM and not `login.defs` as we'll see in the next section. Keep in mind that these files are not the only way to list users and groups in a system.

In general the `passwd` and other specific files used to defer encryption should never be edited directly but should only be accessed through command line utilities. For `/etc/passwd` and `/etc/group` there are `vipw` and `vigr` respectively. To verify their integrity after editing them there are the `pwck` and `grpck` commands. These script do the appropriate locking, processing, and consistency checks on the entries so that they aren't mangled or corrupted.
Other utilities are used to change the passwords, create, and edit users or groups, such as `chpasswd(8)`, `passwd(1)`, `useradd(8)`, `usermod(8)`, `userdel(8)`, and `gpasswd(1)`. On OpenBSD we also find another way to change the user database information through `chpass(1)` with its many other aliases and functional equivalents such as `login_lchpass(8)`.
NB: `passwd` sometimes offer the `--lock` option to add a '`!`' at the start of the password of the user, indirectly locking it.

Let's move to the more advanced and modern management of authentication, instead of relying on a single authentication scheme we could rely on a range of varied ones that are pluggable to many external methods ranging from LDAP/AD, Oauth2, HSM and hardware keys, kerberos, certificates, and more.

What you need to remember: *The password and group files are classic and simple files to store user and group info along with password. However, today this isn't secure to store as publicly accessible, thus different solutions exists to store the password encrypted in a separate place that is only accessible by the super-user. These include the shadow password suite on Solaris and Linux and the `master.passwd` on BSD. The new mechanisms also offer a couple of options to configure password policies.*

Dynamic/Pluggable Authentication

BSD Auth

BSD Authentication is a mechanism initially created by the now-defunct BSD/OS to support dynamic authentication "styles", it is predominantly used in OpenBSD. It consists of stand-alone processes that communicate over a narrowly defined IPC API to dictate how the authentication will happen. This separation of programs and scripts follows the principle of least privilege, not getting the same power as the parent process but only what it needs, it's a way to do privilege separation.
The modules/scripts are configured through `login.conf` as methods of authentication.

We mentioned `/etc/login.conf` and `~/.login_conf`, the login class capability databases, earlier but didn't explain its format (`getcap`). As the name implies it consists of a list of "classes" along with specific features and configuration related to them. A class is simply an annotation to categorize users, independent of their groups. If you go back to the last section you'll notice that the class a user is in is mentioned in the `master.passwd` file. If it isn't in there then the class used will be the one named `default` (root account will always use the `root` entry regardless if present or not).

In `login.conf`, you'll find all sorts of things referred to as capabilities, such as resources constraints and quotas (we'll go into that in the isolation section seeing also `ulimit` and `cgroups`), the password format and expiry, session accounting, user environment settings, and much much more.
It's a textual file with the name of the class followed by a colon `:` and then a list of capability entries that are also separated by colons.
Example:

```
default:\
    :localcipher=bcrypt:\
    :copyright=/etc/COPYRIGHT:\
    :welcome=/etc/motd:\
    :path=/sbin /bin /usr/sbin /usr/bin /usr/games /usr/local/sbin ↙
        ↳ /usr/local/bin ~/bin:\
    :nologin=/var/run/nologin:\
    :filesize=unlimited:\
    :maxproc=unlimited:\
    :umask=022:\
    :auth=skey,radius,passwd:
```

Any edit to the file will require it to be rebuilt into the system using `cap_mkdb`.

What we'll pay attention to in this section are the `auth` and `auth-<type>` list attributes which allow the user to be authenticated through the dynamic authentication "styles". By default `auth` only contains the `passwd` entry, however there are many more available.

OpenBSD lists the following modules, each having their own separate manpage documenting how to set them up (`login_<style>`).

- passwd
- reject (doesn't allow login)
- activ

- chpass
- crypto
- lchpass
- radius
- skey
- snk
- token
- yubikey

These are found in `/usr/libexec/auth/login_<style>`. Obviously, there's also a way to write your own local authentication as a separate script, but it's recommended to give them names starting with - to avoid collision with existing ones.
We won't dive into how to write such script but let's mention that the IPC protocol is based on file descriptors where strings are written (`authorize`, `reject`, `challenge`, etc..).
When a program needs to authenticate something it asks for a "service" which can either be `login`, `challenge`, `response`. In most cases, the `login` service is the one asked for.

You might also wonder how to specify the authentication method you want to use upon login when there's a whole list enabled for a user. That's done by adding after the username a colon `:` and the name of the auth method afterward. Example:

```
1 login: username:skey
2 otp-md5 95 psid06473
3 S/Key Password:
```

What you need to remember: *BSD Auth is a way to dynamically associate classes with different types/styles of authentication methods. Users are assigned to classes and classes are defined in `login.conf`, the `auth` entry contains the list of enabled authentication for that class of users.*

PAM

PAM, the Pluggable Authentication Module, not to be confused with Privileged Access Management - a generic term for corporate infrastructure sensitive data security (something used with IdM) - plays a similar role as the BSD Auth styles, that is to delegate the authentication mechanism to a wide-array of different technologies.

It achieves this through a suite of shared libraries allowing the admin to pick how applications authenticate users (in contrast with separate programs communicating via IPC of BSD Auth). If an application is PAM-aware, if it is using the PAM library to perform authentication, then the mechanism can be switched on the fly without touching the application itself.
Instead of relying on `/etc/passwd`, `/etc/group`, or the shadow password suite to map identifiers, PAM will handle and often override this mapping and multiple of other features provided by them. PAM is now the default authentication mechanism on most systems, including Linux and FreeBSD, that means that the default login interface to the system will use it (`login`).

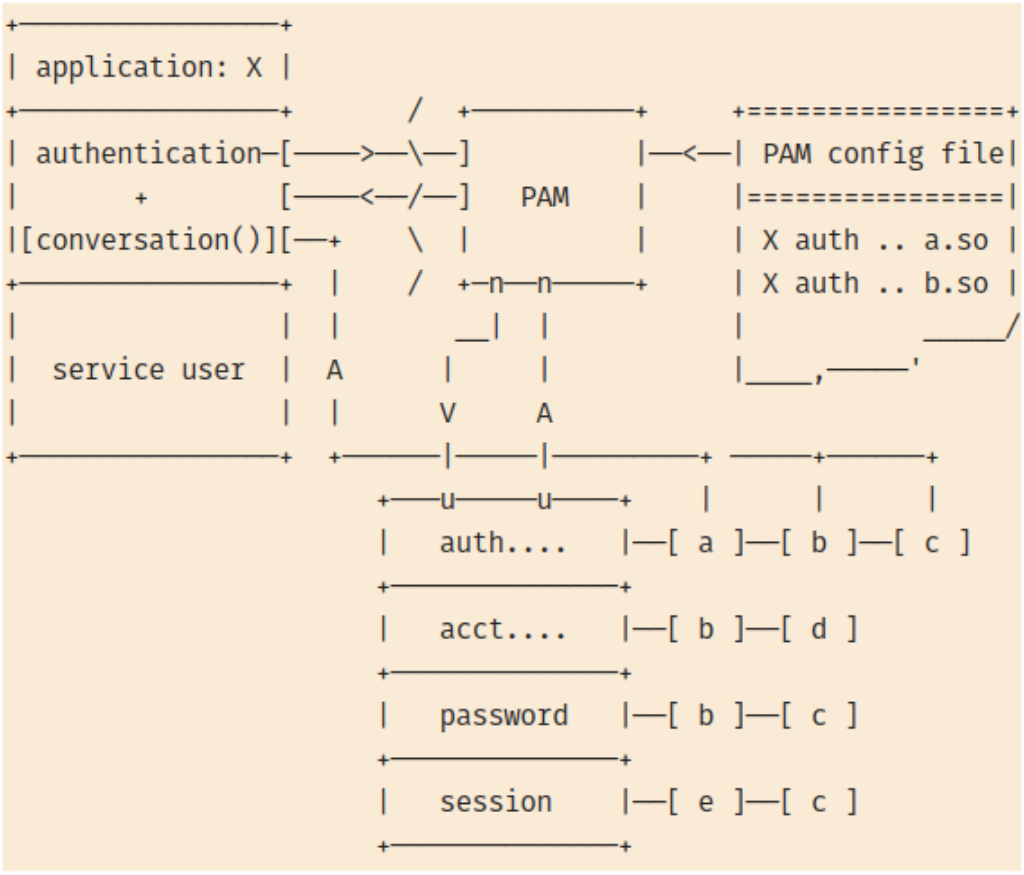
The PAM modules are not only limited to authentication but also include standard programming interfaces for session management, accounting, and password management.

One big caveat is that there isn't a single PAM implementation but multiples: OpenPAM, Solaris PAM, and Linux-PAM.
OpenPAM is a continuity of Solaris PAM pushed forward by FreeBSD as part of a USA DARPA-CHATS research contract program, the lib is used by FreeBSD, PC-BSD, Dragonfly BSD, NetBSD, macOS, IBM AIX, and some Linux distributions. On Linux, Linux-PAM is almost the default authentication everywhere, now being a base meta package dependency.

Both of them are very similar in their workings and only differ on a few points. PAM was somehow indirectly standardized in POSIX, from what I understood, as part of a sub-specs within another spec in the X/Open Single Sign-on (XSSO) standard. That particular specification having a scope englobing more than PAM itself (single sign-on). But it's also standardized in OSF-RFC 86.0 (the Open Software Foundation later merge to become The Open Group): Unified Login With Pluggable Authentication Modules (PAM).
When it comes to differences, Linux-PAM has a wider range of modules available, linking these modules is more dynamic than OpenPAM which is more rigorous. The location of the modules and headers is different (macOS has them in `<pam/pam_appl.h>` while most systems have them in `<security/pam_appl.h>`) , even modules doing similar functionalities could take different parameters and be named differently between OpenPAM and Linux-PAM. Code-wise when implementing modules, the API syntax and how it's used is inconsistent, they each have specific structures and could have their own header files. While OpenPAM says that it follows the PAM specs to the letter, when taking a closer look it's not really true. As for Linux-PAM, it contains a lot of extensions to the specs. Another difference is that Linux-PAM has more community support and documentation than OpenPAM.

PAM, like so many other project, makes life harder by using its own vocabulary, which is ill-defined across implementations, but somewhat makes sense if someone takes the time to explain it.
An *account* is the set of credentials/identifiers an *applicant* wants to get access to from the *arbitrator*. The *applicant* is the entity requesting it, and the *arbitrator* the entity that has the ability to grant/deny and verify that request. The *applicant* performs the request through a *client*, an application that is PAM aware, toward a *server*, the module or piece of code acting on behalf of the *arbitrator*. The *server* can be grouped into *services* providing the same functionality, usually a service has the same name as the program (ex: `ssh` using `ssh` *service*). This request asks for a *facility* which is one of the predefined function in the API categories provided by the PAM library: authentication, accounting, session management, and password management. The request will launch a *chain*, which is a sequence of ordered *modules* that will handle the request. Multiple requests over the usage of the application create a *transaction* or *conversation*. The client usually needs to use its *token*, which could be a password or any other piece of information to prove its identity. The *session* to use the *account* is what is returned to the client after its successful request. Finally, the set of all rules and configurations statements that handle a particular request are called its *policy*.

This figure from the Linux-PAM documentation makes things much clearer than the above mumbo-jumbo.



In practice a good way to understand how PAM works is to take a look at the files provided by its package, how to configure them (syntax), and then take a look at a few example flows.

There are three categories of locations that the package installs files at: the configuration files for the *services* and modules, the module libraries, and the documentation.

The PAM configurations are found in either a single file `/etc/pam.conf` or in a series of configuration files named after every *service* in `/etc/pam.d`. The main difference is that if all the *services* are in a single file, then every lines related to that *service* will need to be prepended with its name, meanwhile in `/etc/pam.d`, the name of the file is the name of the *service* (ex: `passwd`). For extra services that are outside the base system, the package itself will need to install its policy file in these locations. Specific modules configurations are found in `/etc/security`. Every module can have its own syntax and settings, either when used in the PAM configuration or from a separate module-specific config file (ex: `faillock.conf`).

The modules, the `.so` dynamic libraries, are installed in either one of these locations: `/lib/security` or `/lib64/security`, or in `/usr/lib/pam` for OpenPAM. The default installation should come with a good set of useful modules but you can always create your own, which we won't dive into.

The PAM documentation comes in a set of man pages (`pam(8)`, `pam.d(5)`) and for Linux-PAM an extensive built-in administration guide as HTML pages in `/usr/share/doc`. Additionally, every module comes with its own man page that starts with `pam_`, for example `pam_faillock(8)`.

Now that this is out of the way, we can dive into how to configure the *policy* (set of all rules) of a *service*. The first rule is that if no service file or entry matches, the `other` *service* will be used as a black hole. The `other` *service* usually contains a denial policy along with warnings.

Within every *service policy* file we find sequential rule lines calling modules to perform a feature of one of the API *facility* (a task category in the API management: authentication, accounting, session, and password), furthermore we could also find `include` or `substack` entries that allow stacking other configuration files as dependencies.

The syntax goes as follows:

```
service type control module-path module-arguments
```

The `service` part should be omitted if using the `/etc/pam.d` style of configuration. The `types` is the *facility* we've been talking about, asking for a certain API category in a *module*, one of these:

- `account`: For non-auth related account management
- `auth`: Two aspects of user auth, find who the user is by prompting password, and grant group membership and privileges
- `password`: For updating auth *token* associated with user
- `session`: For doing things that need to be done before/after giving access to a *service*, including logging, opening data exchange, etc..

NB; The type can be prepended with a `-` to silence logging on missing library errors.

The `module-path` can either be the full filename or relative path of the `.so` file.

The `module-arguments` are a space-separated list of arguments to modify the behavior of the module. Every module manages this themselves, and you can usually find what the argument means in their man page (`pam_<module>`). Moreover, as we said before, this can also be configured in `/etc/security` if the module offers a configuration file there (ex: `/etc/security/limits.conf`).

Lastly, the most flexible part of the rule is the one before the module, the `control` part which allows control flow based on the success or failure of the module: stopping processing, continuing, jumping a few lines, etc..
There are two syntax for that, either a simple word, or square-bracket of `value=action`. Here's the list of words:

- `required`: failure leads to PAM-API returning failure, but only after the other modules have been invoked
- `requisite`: like required but returns directly or superior stack
- `sufficient`: if module succeed and no prior module failed, it will return successfully
- `optional`: success or failure is not important
- `include`: include all the lines from the config file specified
- `substack`: include line of the config file as arg. It differs from include in that the action in the substack doesn't skip the rest of the complete module stack.

Otherwise, it will use the advanced syntax: `[value1=action1 value2=action2 ...]`, the values are what is returned by the module and the actions can either be a predefined ones such as `ok`, `ignore`, `die`, or a digit which will indicate how many lines/rules to skip in the *policy*. Example from Linux-PAM calling `pam_unix.so` module:

```
1 auth  [success=1 default=bad] pam_unix.so  try_first_pass nullok
```

This means that on success of `pam_unix` the next line will be skipped.

That's about all there is to PAM, let's have a look at a few modules as examples.

The `pam_unix` module is one that will use the standard Unix authentication we've seen before, relying on the shadow password suite. On Linux-PAM, it can take some additional arguments that are interesting such as `remember` which will remember the last N password on change, the specific encryption algorithm to use for the password, the minimum password length, and more. Most Unix-like systems have a similar module, either with the same name or split into different ones (Solaris has `pam_unix_auth`, `pam_unix_account`, and `pam_unix_session`, as separate modules).

On Linux-PAM, the `pam_nologin` module prevents non-root users from login to the system when `/var/run/nologin` or `/etc/nologin` file exists.

Let's mention other modules important for access control:

- `pam_group`: grant group membership dynamically according to a specific syntax.
- `pam_limits`: Limit the system resources that can be obtained in the user-session.
- `pam_setquota`: Limit the disk quotas on session start.
- `pam_access`: Control who can access the system based on login name, host or domain names, IP, and more
- etc..

Furthermore, there are many modules related to using devices such as PKCS#11 enabled, HSMs, LDAP, etc. to authenticate the user to a system. And there are also multiple modules that have logging and audit options (We'll have a full section on auditing too, as this is a must).
Most distros package managers offer hundreds of different modules for countless types of integrations, ranging from captchas to time-based one-time password (ex: Google Authenticator).

Overall, PAM alleviate the weight from applications by taking it upon itself. The admin then has a centralized place to configure each service authentication, accounting, and password management. It's especially useful since PAM offers a lot of modules, much more than BSD Auth, and a high-level of flexibility.

What you need to remember: *PAM is the most popular way to dynamically perform authentication through modules, it's used on systems such as FreeBSD, macOS, and Linux. There exists multiple PAM implementations that differ in a couple of ways but not in the PAM configurations: there's OpenPAM and Linux-PAM. There are countless PAM modules, each have their own man page and can be configured independently in `/etc/security`. The PAM configuration file (`/etc/pam.d`) is per-module and consist of a series of entries read sequentially, each calling a module for a purpose. The important part of the config is the control flow which allows to take actions based on the response of the module.*

Super-User and Switching Subject/Domain/User

We've seen how we can prove who we are, but sometimes there's a need to swap subject, to become someone else. This was mentioned briefly as switching domain dynamically in a previous section. That switch can either be temporary, delegating authority to run a command, or permanent by continuing running the session as another subject.
Before initiating the discussion, let's have a meaningful detour to talk about the concept of super-user, it will come in handy when uncovering the rest of the topic.

Super-User Concept

A super-user is a generic computing concept of a subject that has administrator privileges on the system, that is, they bypass all the rest of the security features and can carry all the possible actions, absolute power over the system. This applies in all modes, single- and multi-user, and ranges from the ability to change permissions of other users, to using low-numbered ports (or whatever is configured as such), to manipulating raw devices.
This could be a unique user or a role/group or any other mean to associate an identity with this feature.

On Unix-like systems, it is the user with UID zero (`uid=0`) that gets this privilege. Historically, this account's name is root in relation to the `/` root of the file system and the user who owns it, but the actual name is irrelevant. Some systems such as FreeBSD even provide additional alternative super-user called toor (with a non-standard shell).
Because of this, on Unix we often refer to super-user as root, but that is not very precise, so we can talk of root privileges instead.

As we mentioned previously, it's a security risk to have multiple users with the same UID, however on most systems it's still technically feasible to have these entries in the `passwd` file.

You can check your own user id by doing:

```
1 > echo $UID
```

Only the root user has the ability to change its UID to that of another user but once it does, there's no way back. This drop in privileges, as a security measure, keeps the integrity. No real widely-used command-line utility exists to do this root drop apart from Bernstein's setuidgid and derivatives script (Part of daemontools) and Linux's `runuser` from `util-linux` (also used for daemons privilege drop).

As we'll see in the capability and access-control list sections later, there's many more ways to define a super user, and many more identifiers (real and effective user and group ids). We can note on Linux that the super-user is also a Linux's capability role defined as `CAP_SYS_ADMIN` or TrustedBSD's `CAP_ALL_ON`.

After reading this we can clearly say, based on the principle of least privilege, that the super-user should not be used to perform daily tasks and should be restricted to specific scenarios only as otherwise it could lead to disastrous damages with no safety net. Yet, by default Unix-like systems are

made in such a way that ordinary users don't have access to most part of the systems, so it can be tempting to unnecessarily rely on the root account and its derivatives.
For that reason, it is preferable to rely on a middle-man, a mechanism or facility as we've come to call them, that would intermediate between user switching. That's what we're going to see.
Yet, we'll need another preamble to introduce these tools, because as we said: Only root privilege allows to change between UIDs. The trick to allow normal users to do this are found in the following two terms: `setuid` and `setgid`.

What you need to remember: *A super-user is one with full privileges. On Unix-like system that's a user with `UID=0`, the user name root doesn't matter, it could be anything else. This is a possible issue with double entries with `UID=0` in `/etc/passwd`. Only the super-user can change its UID.*

setuid and setgid

The setuid and setgid bits, short for set user identity and set group identity, are special access rights flags that can be attached to files. As we'll see later, files have a series of other access rights attached to them (read, write, execute), and are owned by a group and a user.

The special flags, if set on an executable, allow the user running it to gain the privilege of the owner of the file, user or group depending on the flag set. This means we can bypass the rule saying that none other than root can switch user, at least we can bypass it temporarily for the time the executable is running. In many instances of Unix-like Os, for security reasons, the process gaining privilege is stopped from self-modifying its own process memory, otherwise that would lead to privilege escalation.

When set on a directory, the files and directories created underneath will inherit the permission set in these special bits. However, not all Unix-like systems will do this for setuid, as far as I know only FreeBSD allows to configure setuid to work similar to setgid when set on directories.

To add setuid on a file we can do:

```
1 > chmod +s ./executable
```

We'll see how to set all kinds of flags on files and more later, but now we're armed with the knowledge required to understand the rest of this section. For now, just keep in mind the trick that allows us to switch user through an executable owned by them.

One thing we need to add here is that `setuid` and `setgid` are not only bits used on files, but also exist as functions specified by POSIX (often implemented as system calls). Along with them we have another set of functions called `seteuid` and `setegid`, for set *effective* UID and set *effective* GID. Additionally, there's a combination of both previous ones found in `setreuid` and `setregid`, for set *real* and *effective* user and group IDs. Furthermore, there's even a third and fourth set of functions `setresuid` and `setresgid` for setting the previous ones along with the *saved* user ID and group ID, and `setfsuid` and `getfsuid` to set the *filesystem* IDs.

We've seen how it made sense to have bits such as `setuid` and `setgid` on a file, which would allow the process to be run as the UID of the owner of that file, but what about a calling a function programmatically in a process: who can perform them, how, and what's the difference between the *real*, *effective*, *saved*, and *filesystem* IDs.

Remember when we said that after root drops privileges there's no way to gain it back, well euid, the effective user id, file system user id, and saved set-user ID are tricks to bypass that.
When a process is executed it gets "credentials", an identity allowing it to perform tasks. These include the process identifier, the parent process identifier, the session id, but more importantly for us: the real and effective user and group ID.

As you can imagine, upon login, a user gets associated with the IDs it has in the password file we've seen, these are its real user and group identifiers. Yet at the same time, in the background all the other identifiers, effective, saved are also set to these values. You can fetch your user ID using `getuid(2)` function call for instance.
The processes spawned by a user inherit these IDs, and thus in most cases a process real and effective user and group id are the same, making them redundant.

However, these fall into place when executing one of the setuid bit program we've mentioned, such as `ping` for example. At this moment the process will change its effective user or group id to the one of the file owner. The kernel will use the effective IDs to make most privilege decisions (with some exceptions), thus allowing the behavior we've seen.
So far, it means that the real user ID is who is actually owning the process, and the effective user ID is the one the OS looks at to make decisions.

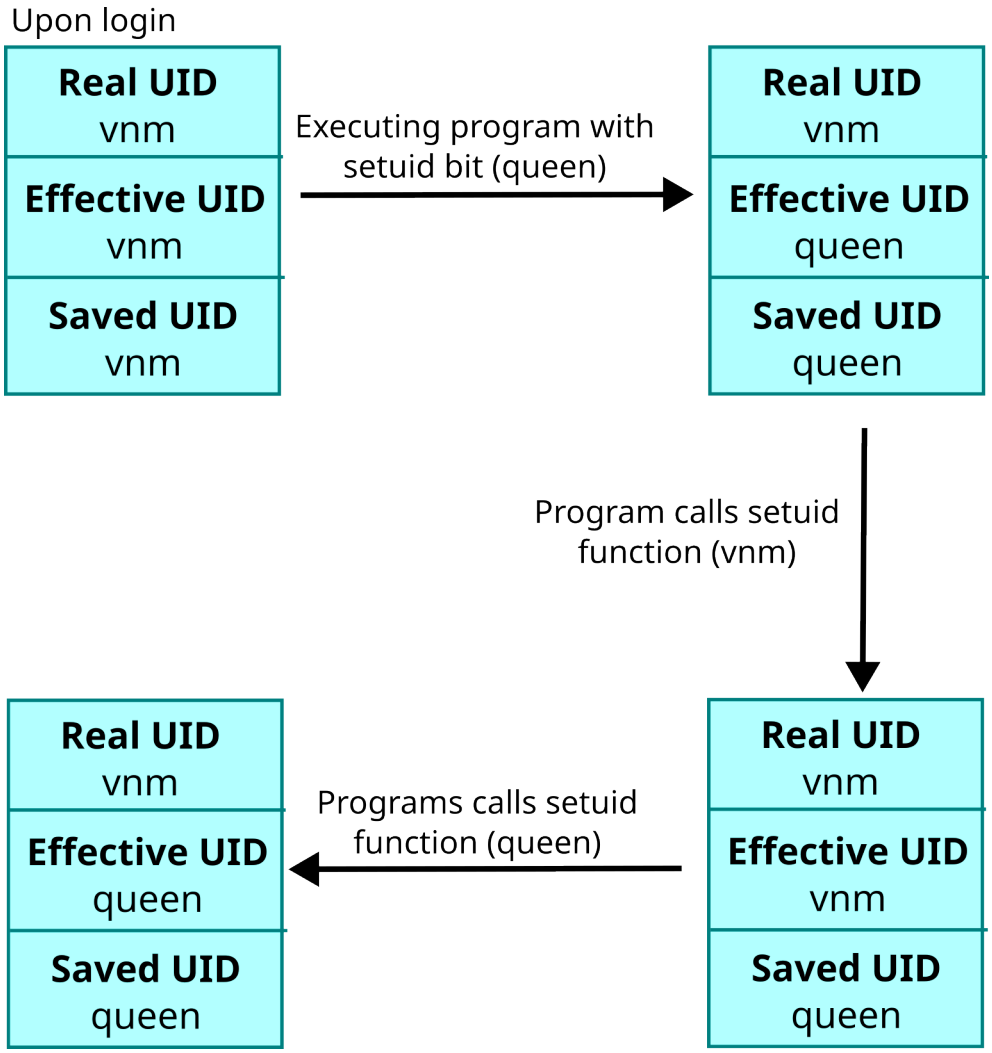
The reason the real user ID is stored is to allow to switch back to it. To make this happen, the effective user ID is backed up in a temporary place called the saved-set user ID, and even when the real-user ID is swapped with the effective user ID, we we can get it back. This mechanism allows a super-user to drop its privilege to a normal user, and then switch back, thus keeping intact the least-privilege principle.

The exceptions to the privileges allowed by the effective IDs depend on the Unix-like system's implementation. On most systems, it allows accessing the file system as the effective UID, however on Linux this is done through the file system ID (`fsuid`) instead, which is usually equal to the effective user ID unless explicitly set otherwise (`setfsuid`).
Additionally, depending on the semantic, the creation of files might or might not inherit the effective ID. For instance, on BSD Unix the group ownership of files created under a directory is inherited from the parent directory, while on AT&T UNIX and Linux the files created inherit the effective group ID. The effective user ID can be propagated, for example when spawning a new shell, but that depends on the shell used and the parameters passed.

Let's also note that a process can be killed if either the real or effective UID match, this allows stopping a process that a user starts as setuid.

The above functions mentioned should make more sense, but why have so many of them.
On most systems the behavior of `setuid` depends on the user, for super-user everything is allowed and all IDs are set to the one passed, while for normal user it will only set the real user ID and it will be allowed depending on the effective and saved-set user ID (On Linux there's a special capability called `CAP_SETUID` which we'll see in the POSIX Capabilities section).
The `seteuid` will only set the effective user ID, and let you perform this call if the ID passed is either

the saved-set ID or the real user ID.
The `setreuid` sets both of them, it will also change the saved-set user ID to the new effective user ID.




```c
// for Linux example getresuid
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>

// 1000 is vnm
// 996  is queen
int main() {
    // all UID are 1000 on login
    // this is setuid bit to user queen 996

```

```
12      int ruid, euid, suid;
13
14      // this is not POSIX
15      getresuid(&ruid, &euid, &suid);
16      printf("real uid: %d\n", getuid());
17      printf("effective uid: %d\n", geteuid());
18      printf("saved uid: %d\n\n", suid);
19      // effective and saved uid are now 996
20      // real uid is 1000
21
22      setuid(1000);
23      printf("real uid: %d\n", getuid());
24      printf("effective uid: %d\n", geteuid());
25      getresuid(&ruid, &euid, &suid);
26      printf("saved uid: %d\n\n", suid);
27      // effective and real UID are now 1000
28      // saved UID is still 996
29
30      // allowed because saved
31      setuid(996);
32      printf("real uid: %d\n", getuid());
33      printf("effective uid: %d\n", geteuid());
34      getresuid(&ruid, &euid, &suid);
35      printf("saved uid: %d\n\n", suid);
36      // effective and save UID are now 996
37      // real UID is still 1000
38  }
```

This surge of information is dizzying with its plethora of IDs but will come handy in the next few sections.

What you need to remember: *setuid and setgid allow someone to gain the privilege of the user or group owning an executable when calling it. There's a dance of real, effective, and other types of IDs allowing this to happen. That's a way to bypass the restriction saying that only root can change its UID and that there's no way to switch back and forth.*

`su` and `newgrp`

Setuid and setgid are what makes administrations tools that we've seen before work such as the ones manipulating the shadow password suite configuration files, ex: changing your password as a normal user.
They also allow the creation of generic tools to switch between users and groups: `su`, substitute user, and `newgrp` substitute/change group.

Both of these are straight forward, doing what is intended, switching to the user or group but asking the user or group password beforehand. If user `vnm` wants to become user `queen` after issuing `su - queen`

they'll have to enter `queen`'s password. Meanwhile to switch to another group with `newgrp secret` they'll have to enter the `secret` group's password
`su` is more advanced than `newgrp`, allowing to run interactively or non-interactive shell, to pick the shell that is going to be used (`-s`), set the environment variables and whether to start as a login shell, to change the directory to the home of the user or not (`-`), to run a command and exits afterward (`-c`), etc.. `su` also has the possibility to switch group (`-g` and `-G`) but this is only allowed for the root user, which isn't as practical as `newgrp`.
By default `su` without arguments switches to root and without changing the directory.

On systems using PAM, `su` can have its own *policy* file, allowing special authentication behavior, and further auditing and logging (apart from the default logging upon login behavior) of who has used the command.
For example, on FreeBSD installations it is common to have a rule that only allows users part of the `wheel` group to use `su`, this is done through the `pam_group` module restrictions in the `su` PAM *policy* file.

Moreover, on BSD systems the `su` utility can be used to switch between login classes by precising it as an additional `-c` option. Weirdly, this is the same option used to run commands so you have to precise it twice to make it work.
For instance, to switch to the `staff` login class of `login.conf`:

```
1 su -c staff bin -c 'makewhatis /usr/local/man'
```

There are countless reasons why you'd want to switch from one login class to another, one of them is to raise your resource limits, but as we'll see later it can also be used in an Mandatory Access Control system to raise or lower privileges.

While the setuid and setgid flags solve a lot of system management issues they also open the door to an astonishing number of security risks. It is the source of the so called "confused deputy" problem: One in which one program of higher privilege is tricked by another lower privilege program into doing something it wasn't supposed to do, misusing its authority on the system (capability-based security is one of the solutions to this problem, that we'll dive into later).

For this reason, a common recommendation is to only use the `-c` option with `su` to exits immediately after the execution of the command.
Another issue is that `su` doesn't create a new pseudo-tty for the session, which can lead to escalation, thus the `-P` option should be passed to achieve this.

Yet, this isn't enough, FreeBSD even disables the `newgrp` command by default by not assigning the setuid flag to it, considering it too insecure and discouraging it.
The alternative to all these is to use either a completely different mindset or to use a relatively more configurable versions of the previous tools: `sudo` and `doas`.

What you need to remember: *`su` and `newgrp` are tools that rely on setuid to substitute the user and group. To perform the command the user needs to be aware of the password of the subject they'll switch to.*

`doas` and `sudo`

The tools `sudo`, substitute user and do, and `doas`, literally "do as" someone else, offer the same functionality as `su` and `newgrp`, allowing it through the same setuid trick (changing all the IDs real and effective), but differ in some minor but important theoretical and practical ways.

First and foremost, while `su` and `newgrp` require the password of the subject we'll switch to, both of `sudo` and `doas` don't, instead they require the current user's password, the one invoking the command, or having a rule in the configuration file allowing such action without authentication.
Second of all, `sudo` and `doas` have more granular configurations, allowing a range of things that aren't possible with `su` such as only allowing particular commands or certain hosts, logging and auditing, and more.
Thirdly, both of these tools follow the mindset of executing a single command and exiting afterward, which we talked about earlier as a good practice.
Lastly, these tool have a "persist" feature which allows configuring a timeout to not re-ask for the password after authenticating.

For the actual authentication part, similar to `su`, `sudo` usually relies on PAM library and `doas`, which is usually on OpenBSD, relies on BSD Auth. However, there exists a portable version of `doas`, OpenDoas which works with PAM.
With `doas`, because it relies on BSD Auth, you can specify the authentication style to use when authenticating with the `-a` argument.

By default, if you call `sudo` or `doas` with only the command you want to run, it'll execute it as the root user (`UID=0`).

```
1 > sudo cmd
2 > doas cmd
```

They also allow running it as another user through the `-u` argument, and `sudo` also allows running the command as another group with the `-g` argument.

One particularity with `sudo` is that when invoking the command it will set two environment variable with the invoking user's values: `$SUDO_USER`, `$SUDO_UID`, `$SUDO_COMMAND`, and others. This can be useful to keep track of who has initially called `sudo`.

Let's take a look at the rules we can configure for both tools. `sudo` has its configuration files in `/etc/sudoers` or `/etc/sudoers.d` along with `sudo.conf`, while `doas` has a single simple configuration in `/etc/doas.conf`.

We can start with `doas` since it's simpler. There's no specific tool to edit `doas.conf` however you can double check everything is OK in the configuration by issuing `doas -C /etc/doas.conf`.

The file contains a series of line with rules used to match what is allowed. They have this format:

```
1 permit|deny [options] identity [as target] [cmd command [args ...]]
```

This is pretty straight forward, it either permits a command issued by someone to be executed as someone else, or not.
The *identity* can be either a username or a group, to specify it as group you have to prepend it with a colon (`:`). The *target* is the user we'll be substituted with, or if not present all of them are allowed.

Additionally, the rule can be restricted to only allow a particular *command* with or without certain arguments (*args*). In the *options* part, there can be a couple of things such as setting or keeping environment variables, making the rule not require a password, use the "persist" option, and more.

Here are two examples:
To allow users in the group `wheel` to run any command as any other user:

```
permit :wheel
```

To allow members of the `test` group to run `helloworld` without password as root:

```
permit nopass :test as root cmd /usr/bin/helloworld
```

Let's move on to `sudo`'s configurations, which consist of the same idea as `doas`: lines with rules. To edit the `sudoers` file (or `/etc/sudoers.d` set of configurations), which is the equivalent of `doas.conf` rules, it is recommended to rely on the `visudo` tool. Furthermore, `sudo` offers a nifty option to debug the rules applying to the current user: `sudo -ll` which will display whatever applies at the moment. `sudo` also has an additional configuration file for its "frontend", unrelated to the rules but to display, plugins, logging, and debugging (`sudo.conf`). Some plugins can even allow storing the rules remotely, such as in LDAP (See `sudoers.ldap` man page).

```
> sudo -ll
Matching Defaults entries for vnm on identity:
    passwd_tries=100

User vnm may run the following commands on identity:

Sudoers entry:
    RunAsUsers: ALL
    Commands:
    ALL

Sudoers entry:
    RunAsUsers: root
    Options: !authenticate
    Commands:
    /usr/bin/pacman ^[a-zA-Z0-9 -_'"]+$

Sudoers entry:
    RunAsUsers: root
    Options: !authenticate
    Commands:
    /bin/systemctl restart adsuck
```

The syntax of the rules and patterns found in the `sudoers` file is much more advanced than `doas`. Yet for the typical average user of `sudo`, all they ever know is that there's a group called `wheel` or `sudo` and that their user is in there by default, making it possible to switch to any other user after entering their password. The truth is that this is possible only because the configuration on these systems is like that at package installation, there are way more options than what is commonly believed. There's so many options that the syntax can be a bit convoluted, it's even explain in EBNF in the `sudoers(5)` manpage.

The `sudoers` file is composed of three types of entries (even though the man page says there's only two): aliases (which are like variables), user specifications (the rules on who can run what), and defaults which are expensive configuration options changing `sudo`'s behavior. The rules are applied in linear order, and the last one matching will be the one that is applied.

Aliases have types which need to be specified when defining them, there are `User_Alias`, `Runas_Alias`, `Host_Alias`, and `Cmnd_Alias`. The type defines where they can be used. For example, this is an alias `ADMINS` for a list of users.

```
User_Alias     ADMINS = millert, dowdy, mikef
```

The definition of user includes both username, user id, group name, and group id. They are specified as follows:

- `user-name`
- `#user-ID`
- `%group`
- `%#group-ID`

The groups having a `%` prepended and the IDs having a `#`.

The Defaults entries can be used for so many options that we can't cover them all, they can be applied to a specific host, user, command, or run-as. The syntax is as follows:

```
Default_Type ::= 'Defaults' |
                 'Defaults' '@' Host_List |
                 'Defaults' ':' User_List |
                 'Defaults' '!' Cmnd_List |
                 'Defaults' '>' Runas_List

Default_Entry ::= Default_Type Parameter_List

Parameter_List ::= Parameter |
                   Parameter ',' Parameter_List

Parameter ::= Parameter '=' Value |
              Parameter '+=' Value |
              Parameter '-=' Value |
              '!'* Parameter
```

In sum, this means you write it as `Defaults` followed by where you want to apply it, or nothing to apply it globally, then a list of key = value, the keys are the configs you can set. Here's a couple of them, you can find more in the `SUDOERS OPTIONS` of the man page `sudoers(5)`:

- `insults`, `lecture`: when a wrong password is entered, a message will be displayed
- `rlimit_<what>`: a series of configurations related to limiting resources. While these can be also set at PAM level, here it can be more specific to a rule, command, or user.
- `syslog`: use syslog to log events happening in `sudo`
- `passwd_timeout`: Number of minutes the prompt asking for the password times out.
- `timestamp_timeout`: Number of minutes that can elapse before sudo will ask for a password again.

- `passwd_tries`: Maximum number of password attempts before being locked
- `secure_path`: `$PATH` to use instead of the environment variable
- etc..

There is even option to run the command in a chroot environment, but we haven't reached the section on isolation yet to discuss this.

Example, change the maximum number of password attempts for user `vnm`:

```
Defaults:vnm passwd_tries 100
```

The user specification, which determines which commands a user may run (and as what user), is the most complex part of the sudoers syntax and is the one confusing people. Get ready for the relevant EBNF:

```
User_Spec ::= User_List Host_List '=' Cmnd_Spec_List \
              (':' Host_List '=' Cmnd_Spec_List)*

Cmnd_Spec_List ::= Cmnd_Spec |
                   Cmnd_Spec ',' Cmnd_Spec_List

Cmnd_Spec ::= Runas_Spec? Option_Spec* (Tag_Spec ':')* Cmnd

Cmnd ::= Digest_List? '!'* command |
         '!'* directory |
         '!'* Edit_Spec |
         '!'* Cmnd_Alias

Runas_Spec ::= '(' Runas_List? (':' Runas_List)? ')'

Runas_List ::= Runas_Member |
               Runas_Member ',' Runas_List

Runas_Member ::= '!'* user name |
                 '!'* #user-ID |
                 '!'* %group |
                 '!'* %#group-ID |
                 '!'* %:nonunix_group |
                 '!'* %:#nonunix_gid |
                 '!'* +netgroup |
                 '!'* Runas_Alias |
                 '!'* ALL

Option_Spec ::= (Date_Spec | Timeout_Spec | Chdir_Spec | Chroot_Spec)

Date_Spec ::= ('NOTBEFORE=timestamp' | 'NOTAFTER=timestamp')

Timeout_Spec ::= 'TIMEOUT=timeout'

Chdir_Spec ::= 'CWD=directory'

Chroot_Spec ::= 'CHROOT=directory'
```

```
38
39 Tag_Spec ::= ('EXEC' | 'NOEXEC' | 'FOLLOW' | 'NOFOLLOW' |
40               'LOG_INPUT' | 'NOLOG_INPUT' | 'LOG_OUTPUT' |
41               'NOLOG_OUTPUT' | 'MAIL' | 'NOMAIL' | 'INTERCEPT' |
42               'NOINTERCEPT' | 'PASSWD' | 'NOPASSWD' | 'SETENV' |
43               'NOSETENV')
```

As you can imagine, this is probably impossible for the average user to comprehend, or have the patience to visually parse EBNF (:<). Let's explain the format in a way that's understandable by humans (:D).
The format is composed of a user (same definition as before: user, user id, group, and group id), followed by a host then the = character. On the right side of this we find optionally within parenthesis the users that we can run as (all separated by : and ,), then a set of possible options such as timeout, after that we also have optionally some options in the form of tags followed by the : character, these tags are keywords options such as `NOPASSWD` to skip password auth, finally we have at the complete right side a list of commands, directories or aliases. In another format, while not precise, it looks like this:

```
1 USER_NAME HOST_NAME = (RUN_AS_USERS) OPTIONS TAGS: COMMANDS
```

The special keyword `ALL` can be used to replace certain values, allowing all of them (host, commands, etc..)

The command should be a fully qualified file name, the full path, but note that this can include shell-style wildcards (glob), or a regex that starts with ^ and end with $ (start and end). If a directory is specified in the command part it means that any executable in that directory can be accessed (but not sub-directories). When no command line arguments are specified then all of them are allowed. The commands can also be prepended with ! to disallow them.

That's about it for the `sudoers` syntax, you should now know more than 99% of people using `sudo`. Let's put it into practice with a few examples.

Allow user `ray` on the host `rushmore` to issue the commands `kill`, `ls`, and `lprm` as root without entering a password (no authentication).

```
1 ray      rushmore = NOPASSWD: /bin/kill, /bin/ls, /usr/bin/lprm
```

Allow user `alan` on all hosts to run as root, or user bin with optional operator group, or system, to run all commands. (The user should be specified with `-u` and group with `-g`).

```
1 alan     ALL = (root, bin : %operator, system) ALL
```

Allow user `john` on all host to run the `passwd(1)` command as root but don't allow it to be called with the "root" argument, stopping the user from changing root's password (not so secure though).

```
1 john     ALL = /usr/bin/passwd ^[a-zA-Z0-9_]+$,\
2                !/usr/bin/passwd root
```

There's a couple of extra examples that can be found upon package installation in `/usr/share/doc/sudo`.

As you've certainly noticed, `sudo` is an advanced method of allowing a user to substitute itself with another one, a lot more complex than `doas`. However, one could still ask why we'd prefer these two

over the simpler `su` of before.
We've said in the introduction that what is secure highly depends on the definition of security, which is often bound to the policy put in place on a system for access control. In that case, `sudo` and `doas` allow to define the policy in extensive configuration files, while `su` would only allow that indirectly through PAM or `login.conf`. Additionally, it might contradict the policy to share the authentication tokens/passwords of other users with one another, which is an obligation for `su` to work. While it's true that a lot of the simpler setup can achieve a simple policy with PAM and `su`, most advanced policies will consider that as a breach. For instance you could rely on `pam_wheel` plugin to only allow users in the wheel group to use `su` but that might not be enough. Furthermore, there is also the question of fine-grained rules and audit trails which are harder to perform without `sudo` or `doas`, they both are important as we might only want to allow a single program for a user and not full access to another account.
Additionally, with `sudo` and `doas` the root account can be locked completely (`passwd --lock root`) without losing the usability to manage the system.
Yet, one could argue that `sudo` is still a dirty way to split privilege and that other methods would be more favorable, the kinds we'll talk about later (Mandatory Access Control, Capability-Based security, Isolation, or Action-Based Access Control).

What you need to remember: *`sudo` and `doas` are more advanced versions of `su` that offer a granular way to define policies through configurations. No more need to know someone else's password, only the one of the current user is needed. In the configurations of these tools we can set who has access to what command and can run it as which user.*

SunOS Derivatives Profiles

SunOS and its derivatives, ranging from Solaris, OpenIndiana, to illumos, don't use neither `sudo` nor `doas`. While these can still be installed separately, they instead rely on something called "profiles".

The profiles allow a user to switch to another user to perform a command, and also to gain other functionalities which we'll plunge into in other sections.
The profile is a combination of two things: execute attributes (`exec_attr`), and profile attributes (`prof_attr`). Profiles can be combined together to construct appropriate access control.

The profile attributes in `/etc/security/prof_attr` is a file that contains the execution profile names, their descriptions, along with a set of attributes assigned to each of them: "auths" and "privileges". We'll skip both for now as we'll see them in future sections, the action-based access control and POSIX capabilities section. The first is used to give access to specific custom features in programs that choose to check them, and the later is used to split super-user access control into granular pieces.

On the other side, the execute attributes in the `/etc/security/exec_attr` file enumerates commands along with process attributes, such as the effective user and group IDs that the profile is allowed to run as. If the same command appears in multiple profiles' execute attributes, then the first occurrence, as determined by the ordering of the profiles, is used for process-attribute settings.

The profiles are then assigned to users in a file similar to `login.conf` and `login.defs`, found in `/etc/user_attr`, the extended user attribute database file. This file is similar to BSD's `login.conf`

and Linux's `login.defs` but we haven't mentioned it before because it has very few options related to passwords and is more related to other types of access control (profile-based, action-based, role-based, and capability-based).
The entries in the file are composed of colon separated key-values within the `attr` field. It practically looks like `user:qualifier:res1:res1:attr`, however the fields `qualifier`,`res1`, and `res2` are reserved for future use, so useless.

```
username::::key=val;key=val
```

We can also access them using NSS:

```
> getent user_attr username
```

The file can either be edited manually or through the `usermod(8)` and `rolemod(8)` system utilities with the `-K` flag. Roles are something we'll dive into later in the RBAC section, but for now just think of them like any other user.

The field we're interested in here is the `profiles` which contains a comma separated list of profiles (found in `prof_attr`) that the user can switch to.

The profiles found in `user_attr` are merged with the default profiles is defined in `/etc/security/policy.conf` field `PROFS_GRANTED`. This is a key-value file with system-wide default policies for different access control.

The `prof_attr` format is similar to the `user_attr` file, a colon-separated list of profiles.

```
profname:res1:res2:desc:attr
```

The `res1` and `res2` are unused, the `desc` is a generic description of what the profile is, and the `attr` contains a semicolon `;` separated list of `auths`, other profiles `profs`, and `privs` (POSIX capabilities called privileges on SunOS derivatives).

Similarly, `exec_attr` has a colon-separated list of entries in the form:

```
profname:policy:type:res1:res2:id:attr
```

The `profname` should reference the profile found in `prof_attr`, `res1` and `res2` are unused, `type` can only be set to `cmd`. There are two types of `policy`, the `suser` for standard users and `solaris`, the difference is that `solaris` can use privileges, which we'll see in the POSIX capabilities section.
The interesting part of the line are the `id` and `attr`, the `id` is a string representing the command and the `attr` under which effective uid or gid. A `*` can be used in the `id` field to specify all commands.

The `attr` field is formatted as as semicolon-separated `;` key-value pair, setting the following possible values `euid`, `uid`, `egid`, `gid`, `privs`, and `limitprivs`. Again the `privs` and `limitprivs` will be seen in the POSIX capabilities section. The IDs can be either strings or numerals.

Example, to allow the profile "Audit Control" to run the command `/usr/sbin/audit` as effective UID 0.

```
Audit Control:suser:cmd:::/usr/sbin/audit:euid=0
```

In summary:
- Profiles are defined in `prof_attr` - Profiles are associated with executable attribute in `exec_attr` - Profiles are assigned to users in `user_attr`

To list and manage profiles assigned to users, the command `profiles(1)` can be used (`getent prof_attr` NSS utility can also be used).

```
> profiles tester01 tester02
tester01 : Audit Management, All Commands
tester02 : Device Management, All Commands

> profiles -l tester01 tester02
    tester01 :
        Audit Management:
        /usr/sbin/audit          euid=root
        /usr/sbin/auditconfig    euid=root    egid=sys
        All Commands:
        *
    tester02 :
        Device Management:
        /usr/bin/allocate:       euid=root
        /usr/bin/deallocate:     euid=root
        All Commands
        *
```

Practically the user access the profiles through a command interpreter called the profile shells `pfexec(1)`, it internally relies on subshells such as `pfcsh`, `pfksh`, and `pfsh`. The profiles are searched in order to see if it matches the command passed and then the shell is launched with it. Additionally, a special `-P` flag can be passed to set privileges (See POSIX capabilities section).
For this to work, the `pfexecd` daemon needs to be running.

Example:

```
> pfexec /usr/sbin/audit
```

What you need to remember: *SunOS derivatives don't use `sudo` and `doas`, they instead use profiles. The profiles are assigned certain access control such as "auth", "privileges" (`prof_attr`), and importantly an execution environment (`exec_attr`) allowing to run commands as another effective UID or GID. The users are assigned these profiles in an ordered list in their extended attributes (`user_attr`), there is also a set of default profiles in `/etc/security/policy.conf` key `PROFS_GRANTED`. The `pfexec` command allows running commands in "profile" mode.*

Identity and Access Management Solutions

There exists a more high-level view of authentication, one that has appeared in the corporate scene and relies on separate services that have as role to manage identities and access in a decoupled manner.

They defer the decisions, storage, identification, authentication, and authorization we've talked about. These services can be centralized or even decentralized.

Most of them are generic and not Unix-specific, and apart from identity management and access management they could include features such as a vault system: storing and using secret tokens securely across multiple systems for a wide-variety of usages. These tokens can range from digital certificates, to hardware security modules encryption features.

We refer to these services as Identity Management (IdM) or Identity and Access Management (IAM or IdAM). They emphasize on the abstract and pure concept of digital identity with all that it entail: the axioms making up the relationship of an entity/subject with the real world. In a way, it's more human, keeping up information that aren't usually necessary in other systems, such as real name, date of birth, etc..
In practice the system should allow anything that has to do with the life-cycle of an identity, such as the creation, management, and deletion, along with what authentication credentials are used to prove the identity (regardless of the system it'll be applied on). It then has to pick, in a centralized way, what the subject has access to and keep track through auditability and monitoring functions of all the actions taken (something we'll dive into in the last section of this article).

There exists a lot of specifications and standards that make sense of all this. One of them that we've mentioned before uses the single-sign on concept: X/Open Single Sign-On Service (XSSO). However, there are many more, for example:

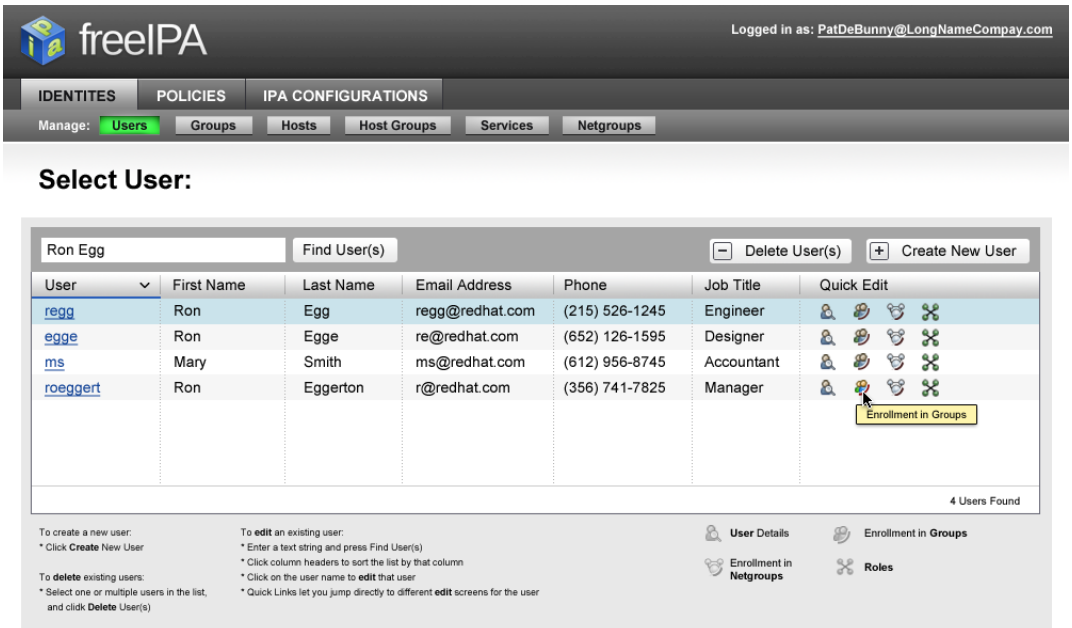
- ISO/IEC 24760-1 A framework for identity management-Part 1: Terminology and concepts
- ISO/IEC 24760-2 A Framework for Identity Management-Part 2: Reference architecture and requirements
- ISO/IEC DIS 24760-3 A Framework for Identity Management-Part 3: Practice
- ISO/IEC 29115 Entity Authentication Assurance
- ISO/IEC 29146 A framework for access management
- ISO/IEC CD 29003 Identity Proofing and Verification
- ISO/IEC 29100 Privacy framework
- ISO/IEC 29101 Privacy Architecture
- ISO/IEC 29134 Privacy Impact Assessment Methodology
- Role-Based Access Control (RBAC) ANSI INCITS 359
- Administrative Role-Based Access Control (ARBAC02)

Some of them often rely on a base protocol and extend it, for example: OpenID, OAuth, Kerberos, LDAP.

Practically, on Unix-like systems, the implementations will rely on tech we've mentioned before, mostly PAM and its modules. Otherwise, the solution will have to hijack the system calls through custom libraries, which is what the `sssd`, System Security Services Daemon does on the client side. `SSSD`, can integrate with Microsoft Active Directory, FreeIPA, or LDAP domain (such as Apache Fortress) to use remote definitions of identities, policies, and other authorization mechanisms.
Microsoft Active Directory, Apache Fortress, AWS Cedar within Amazon Verified Permissions service, Polar's OSO policy language, the Open Policy Agent, Google's Zanzibar, auth0 IAM, and FreeIPA (the free version of RHEL Identity Management), are examples of IdM server solutions. They add a nice GUI, a nice skin on top of access control, abstracting details across multiple systems.

Many of the above rely on what's called RBAC, role-based access control, which is a concept we'll dis-

cuss later. “Roles” are annotations separate from groups, used to assign certain privileges. Sometimes these systems also allow managing other advanced features we didn’t mention yet (ex: FreeIPA), such as the standard Unix file permission, Mandatory Access Control and extended attributes, role-based access control, and Capability-security.



One non-Unix-like system we haven’t mentioned that offers an identity server solution is Plan9 with its factotum. It is a user-level file system that lives on every host, with one owner of all resource on that host that acts as the authentication agent for users wanting to access that host. Similar to PAM and BSD Auth, it offers plugins for different methods of authentication that the user can pick from, and has associated keys which represent a collection of information used to authenticate a particular action.
Similar to BSD Auth, upon login, the user can pick the mechanism they want to get access to the resources on that host.
Plan9 is also what inspired Linux namespaces, which we’ll see in the isolation section.

This is it for this section, you should now have a generic idea of all the ways used to identify and authenticate users. From the password files, the shadow password suite, PAM, to setuid trick, `su`, `sudo`/`doas`, and much more.
We’ve seen what there are to see about subjects and the time has come to move on to focus on the object side of the equation along with the mechanism of control in between.

What you need to remember: *Identity Management platforms (IdM or IdAM) exist to take centralized authentication decisions across systems. FreeIPA is a good example, it relies on a daemon on the client-side called* `sssd` *that hijacks system calls related to authentication. These platforms can also manage other access control features.*

System-Wide Access Control

While the vocabulary around access control is colorful, making sense of it by splitting the practical concepts into categories is an epic task. The approach taken in this article is to split the access control mechanisms in one of three categories: **system-wide**, **isolation/constraint**, and **action-based**.
We'll start with the common approach of having a mechanism applying a security policy over all objects on the system; this is what is meant in this article by "System-Wide".

Background Knowledge and Theories

Before diving into the actual implementations, yet again, there is some background knowledge we'll need to digest. This time we'll quickly review the categories of security models we've seen before, but then switch to a new approach to access control: How the user interfaces, controls, and sets the policies.

Access Control Lists, Access Profiles, and Flow Policies

Let's recap the main categories of models we've looked at. On one side we have the matrix-like models that include the access control list, where within each object is stored the privileges each user has, and the access profile, where each user carries with them the list of privileges they have over objects. On the other side, we have the flow policies models, where we associate different security levels with certain privileges and sets of rules on how to move from one level to another.
These ideas will come handy in this section as they can be used to categorize the implementations we'll see.

A novel idea that we can add is to consider an action as an object. In that case, the mechanism would allow to perform the generic action, anywhere or within a specific software. This can be applied to all the models we've seen. Example: The mechanism allows user john read access on all files between 1am and 2am.
By itself, that doesn't seem very "secure", nor relevant, but that is going to be applied in the Action-Based Access Control section.

The previous example also includes something we've been missing in the models, the notion of time, and more broadly, the environment/context. While some will argue that the context is an object, others will say that it's omnipresent and ambient. Yet others will say that it inherently changes the subject, becoming part of it as a "geotemporal" factor during authentication.
Regardless, if the context is missing from the models it will render it less flexible, further away from the security policy, or even completely irrelevant.

Lastly, another part we've missed when modelizing is to emphasize the revocation of privileges. Depending on how we choose to implement our model, revocation can be a risky point. Will the revocation be immediate, atomic, or delayed, and what will happen in between. How granular is it, can it be applied to whole groups, to a subset of access rights, and for how long (temporal).
In the case of the access control list, the access revocation looks straight forward: modify the object and you'll get the result. However, in the case of access profiles, the subject carries with it these privileges and thus a mechanism needs to be in place to be able to modify them. One way to solve this is to have a validating point for the user's rights, sort of like a PKI with a revocation list (CSR), signing the validity of the privileges that they currently have, as long as they match they keep the access. As for the flow security model, revocation is fairly simple too as the levels are described globally.

We can now move to a related topic: In what ways users have control over their policies.

What you need to remember: *A recap of the categories of models: matrices with access control lists and access profiles, and flow policies. Three new ideas are added: an "action" could be an object, "context/environment" could either be an object or part of the subject, and how to think about privilege revocation.*

Discretionary Access Control and Mandatory Access Control

The two historical and classic means of access control are discretionary access control (DAC) and mandatory access control (MAC).

Discretionary access control is any security policy where users are involved in the definition and assignment of security attributes and privileges. In other words, subjects can assign, based on their current privileges, access control rules upon objects to other subjects (at their own discretion). This is the case of most access control list, such as the usual POSIX permission.

In contrast with discretionary access control where users themselves set the policies, mandatory access control is a security policy that is tightly controlled by a system security policy administrator. It is a system-wide policy that cannot be overridden by normal users, either accidentally or intentionally. That means the policy is dictated in a centralized way, guaranteeing that it's enforced on all users, and usually checked at the kernel level.

MAC is closely associated with the rigorous multi-level security (MLS) that we've seen before in the models section with flow-based security that has security modes and clearance levels. For a long time MAC and MLS were mostly synonyms, however, these days MAC doesn't have to be a multi-level security.

Practically, a mandatory access control policy is either implemented as pathname-based or as label-based.
The pathname-based approach to MAC, is one in which the privilege of a subject is associated with the path of files. This means, there's a configuration somewhere associating users to files and what they can do on them. It's a simple approach that works across multiple systems, however the permission is not carried with the files themselves (if they're moved). Some implementations we'll see are AppArmor and TOMOYO Linux.

Labels, on the other hand, require a special file system construct that allows adding arbitrary extended attributes on files. These attributes consist of key-value pairs that are used to make privilege decisions. They are set by the system administrator, like all MAC, and used to decide if the level of security of the users can access the security level of the files (flow-based security model). Some implementations we'll see are SELinux and TrustedBSD's MAC modules.

What you need to remember: *Discretionary Access Control (DAC) and Mandatory Access Control (MAC) are the historical classification of how users interface with their security policies. DAC: normal users can control their own policies to their own discretions. MAC: a policy administrator enforces a policy on the whole system.*

Role-Based Access Control

Role-based access control (RBAC) is a newer approach to policies, in-between DAC and MAC and which can be used to implement either of them. In the past, if a policy wasn't categorized a MAC it was automatically considered a DAC, however, research in the late 90s has proven that it's not always the case.
Some standards have been emphasizing this category of access control, such as the *NIST/ANSI/INCITS RBAC standard (2004)* which recognizes three levels of RBAC.

Role-based access control consist of "roles" which is a grouping mechanism used to assign a set of privileges to subjects. Roles carry with them permissions to certain functions, and users acquire these permissions through the roles they are assigned. Hence, if a user has no role, they have no privileges. In other words, they don't have permissions but acquire them through their roles.
The roles are often given according to the job, responsibilities, or functions of the users. Indeed, this makes a lot of sense in corporate and government organizations. This creates three main relationship: role-permissions, user-role, and role-role (hierarchies) relationships.

A role, unlike in an ACL, can assign permissions to operations and actions to several entities in one go. These don't have to necessarily be files but could also include action-based privileges (See Action-Based Access Control). A role can thus be a set of operations within a larger activity. Additionally, roles can be hierarchical, one role containing another subset.
A minimal role-based access control can be equivalent to ACL when there's a 1-to-1 match between roles and groups of subjects.

In practice, RBAC can be implemented through SELinux and SunOS roles for example.

One advantage of RBAC is that it reduces the abuses we've seen with setuid bits and instead provides a coarse-grained approach to access control that relies on the principle of least privilege, but this can also be said about all other access control mechanisms we'll see. Yet, RBAC has been criticized to lead to "role explosion", where an enterprise creates so many roles that nobody is able to manage them properly (considering it could either be managed centrally or discretionary depending on the implementation).

A very-similar category, which I believe is linked, is Organisation-based access control (OrBAC). It relies on hierarchies of organizations, roles, activities, views, and context to apply constraints. The

roles are a set of users, an action is a group of activities, and a view is a set of objects to which the same security rules apply.
OrBAC is more of an abstract concept implemented using RBAC than something that exists on its own.

What you need to remember: *Role-based access control (RBAC), is in-between DAC and MAC, it can be used to implement both. It consists of roles which have privileges associated with them, users only get privileges through the roles associated with them. Minimal RBAC is equivalent to ACL.*

Attribute-Based Access Control and Context-Based Access Control

Attribute-Based Access Control (ABAC, also called Context-Based Access Control (CBAC), or policy-based access control (PBAC), or even claims-based access control (CBAC)) is a category that evolved from role-based access control by considering additional attributes apart from roles.
The attributes can be associated to one of the following, which basically consists of mostly everything possible, even the environment (context):

- Subject/User attributes e.g. citizenship, clearance
- Object/Resource attributes: e.g. classification, department, owner
- Action attributes: e.g. view, edit, delete
- Context attributes: e.g. time, location, IP

Attribute-based access control is policy-based, it has a system-wide policy that is applied by the system administrator, or locally by users, and evaluates according to the matching attributes whether access is allowed or not. This is widely considered one of the best practice by NIST and other institutes.

As you might have noticed, this allows complex rules based on different attributes that can be context-aware. For example, these rules are possible:

- A user can view a document if the document is in the same department as the user
- A user can edit a document if they are the owner and if the document is in draft mode
- Deny access before 9 AM

A similar category of access control is relationship-based access control (ReBAC), which is a type of attribute-based access control, in which the main attribute checked is the relationship between subjects. This term is mostly used in Google's Zanzibar authorization system, "Google's Consistent, Global Authorization System".

Currently, there exists no pure Unix implementations, however it is available on Windows OS and Web APIs frameworks. There is a Linux framework that can be used to implement it and other access control mechanism called RSBAC, rule-set based access control, we'll see it in the particular role-based access control section. Another example is the XACML, the eXtensible Access Control Markup Language, used to implement APIs and that can be used in IdAM tools (ex: FreeIPA). Yet other web solutions are AWS Cedar policy language, Polar's OSO, and the OPA (Open Policy Agent) that contain ABAC support.

What you need to remember: *Attribute-based access control (ABAC), extends RBAC by having all attributes possible on the system: subject, object, action, and context. It employs a policy matching any attributes in this set against the rules created.*

Capability-Based Access Control

The last category we'll look at is the capability-based access control. This is basically another name for Access Profiles, which is about associating the privilege to the subject instead of the object. It needs a mention because of its particularity. For example, the user could get a transferable privilege but that comes with a limit of forwarding to a maximum of 3 other users. Essentially, it's a type of discretionary access control, with the capabilities being inherently part of the user.

Indirectly, as we said a while back, in capability-based access control what is important is not who sets the policy but the idea of the integrity of the "capability". It is considered an abstract atomic protected resource that exists without being directly accessible by the user.
This can be achieved by having processes contain extra information that represents the capabilities. It could be a non-modifiable file-descriptor, or a tag, or a part of memory segment inaccessible by the program itself. This is then continually checked for consistency and integrity.

Now that we have a good idea of what the landscape looks like we can move to how these system-wide policies are implemented on Unix-like systems.

What you need to remember: *Capability-based access control isn't a way to set policies, it's inherently DAC because the user contains its privileges. A small mention here is about what the "capability" actually consists of: an atomic unchangeable value by users, which can be a file descriptor, tag, or memory segment.*

Basic File Permission

The basic POSIX.1 permission, ubiquitous to all Unix-like systems, compares bits set on files and directories, representing actions, to the process (effective) user and group ID, and decides whether these actions are allowed or not.

There are 3 types of permission bits that can be assigned to 3 classes of users (`3x3`). The 3 types of permissions are: *read*, *write*, and *execute*. Since there are only 3, they are easily represented by a set of 3 bits that are either on or off for the specified permission. And that's how it's implemented, the first bit is for *read*, second for *write*, and third for *execute*.
The classes are the *owner*, *group*, and *other* classes. Accordingly, the 3 bit sets of the 3 classes are represented in the mentioned orders as a big set of 9 bits (`3x3`).

Those bit sets are also represented in octal or decimal notation for every class (here it doesn't matter because the numbers are always less than 10), so for example the `777` permission means that all the

classes get all the privileges on the file. The file permission can obviously be printed in a more human-readable form a symbolic notation, most commands these days allow that.

Thus, the permission to do one of the three actions (read, write, execute) for a class on the file is checked against either the user ID of the process and the group ID or supplementary groups the user is part of. The "other" class is used for process that don't match neither the user ID nor any of the groups, everyone else.

It's obvious what the permissions do on normal files, however, on directories the behavior is a bit different. The read permission gives the ability to list the file names within that directory, the write permission gives the ability to create new files and directory under that directory, and the execute permission gives the ability to enter the given directory (`cd` in it). Think of it as if the directory was a normal file containing a list of names of the files within it (as it used to be the case in early Unix versions).

	Read	Write	Execute
Files	Read file	Append, Modify, Rename, Delete	Execute file
Directories	List directory	Append, Modify, Rename, Delete	Traverse directory

The main commands used to manipulate these permission bits are `chown` change owner, `chmod` change file mode bits, and `chgrp` change group ownership. All of them have a way to set the permission in a user-readable way, sometimes differing between a system and another.
When normal users create files they will create them by default under their user ID and primary group ID, and will only be able to change ownership and group of the files they own. Meanwhile, the super-user has unrestricted access to change ownership. As can be seen, this is a good use-case of discretionary access control.

Here's an example of changing the group of a file when the user is part of the "newgroup" group and owns the file "helloworldfile":

```
chgrp newgroup helloworldfile
```

Additional bits that can be set using the `chmod` command are the `setuid` and `setgid` bits which we've talked about a lot in a previous section. As with the other bits, the user needs to be the owner of the file or the super-user. This leads to a possible scenario in which a user might be the owner of a file that is owned by a group that they are not part of, and yet be able to set the `setgid` bit on it, thus gaining the group membership.

For instance, the file "helloworldfile" is currently owned by user "vnm" and group "git", but user "vnm" isn't part of the group "git", yet the following is valid:

```
chmod g+s helloworldfile
```

The last special bit that can be set on files is the *restricted deletion flag*, also called sticky bit. It affects directories and normal files differently.
For directories, it prevents unprivileged users from removing or renaming a file in the directory unless they own the file or the directory. In practice, it's a way to restrict what the write access does: the user will be able to modify file content, but not change the file names (avoiding modifying the directory itself, as if it was a textual file with a list of the file names found underneath it).
For regular files on some older systems, the bit saves the program's text image on the swap device so it will load more quickly when run. On other systems, it is useless.
To add the restricted deletion flag to the current directory:

```
chmod +t .
```

All and all, this is how the permission bits appear when issuing `ls -lah`; going from left to right, showing whether it's a directory or not, the *read-write-execute* permissions for *owner*, *group*, and *others*:

```
total 12K
drwxr-xr-t  2 vnm users 4.0K Jan  2 18:41 .
drwxr-xr-x 22 vnm users 4.0K Dec  9 18:18 ..
-rwxrwxr--  1 vnm git     35 Jan  2 18:25 helloworldfile
```

Yet, one question remains to understand these bits: how are they initially set, what's the default *read-write-execute* permissions for each class when creating new files and directories?
This is where "**umask**", the creation mask, comes into play.

Masking bits is the process of taking a group of bits and apply another group of bits to it as a mask, which consists of a single bitwise operation to set some bits of the first group either on or off based on the second group. Basically, it's performing a bitwise operation between two groups of bits. For instance, you can do an *OR* operation and use a mask to set some bits to 1 or use an *AND* operation to turn some bits to 0.

On Unix there's a conventional default permission of "*666*" (*read* and *write* for all classes) for files and "*777*" for directories (usually hard-coded values). But, every user on the system has a "**umask**", which is then applied to this default permission, computing the permissions of newly created files. It's applied by doing an *AND* operation on the default permission and the *NOT* of the "**umask**". In practice, it disables the bits of permissions set in the "**umask**", the reverse of what is actually allowed. Nevertheless, "**umask**" cannot add permissions that aren't present in the default ones, hence a file will never get execute permission by default.
You can issue the command `umask(1)` to check the current value.

The "**umask**" can either be set in the `/etc/login.defs` and `/etc/login.conf` files, upon user creation on the home directory, in PAM through plugins such as `pam_umask`, or upon mounting a file system. When mounting a file system, we can set different masks such as "**dmask**" to set the "**umask**" for directories only, and "**fmask**" for files only.

What you need to remember: *The basic POSIX.1 file permission consists of read-write-execute bits set on files for user, group, and others. The meaning depends on whether it's set on a normal file or a directory. On directories execute means searching, and writing means creating files underneath. Additionally, there are the `setuid`/`setgid` bits that can be set on files, and the "restricted deletion mask" for directories to only allow editing files and not renaming them or creating them. The default permission is a mix of hard-coded values (`666` for files and `777` for directories), along with a mask to disable the unwanted bits.*

POSIX(IEEE 1003) 1e and 2c

For a long time the basic POSIX permission seemed to have done the trick to keep systems secure, but soon a need arose to have advanced security policy mechanisms, especially implementations based on theories such as the ones we've seen in the security policy models section.

The POSIX.1e and 2c draft standards were a response to this need. The Portable Operating System Interface (IEEE 1003) draft extensions 1e (C interfaces) and commands 2c (shell and utilities) defines security extensions (protection and control interfaces) allowing a range of flexibility in how to implement different policies. Yet, the standard doesn't deal with security evaluation criteria, but only with the standardization of common interfaces to implement them.
The scope includes the following five optional sets of interfaces, each with new functions and security constraints on previously existent ones (ex: `open(2)`).

1. Access Control Lists (ACL)
2. Capability (Separation of privilege)
3. Information Labeling (IL)
4. Mandatory Access Control (MAC)
5. Security Auditing

Sponsorship for the standard was withdrawn in January 1998 when multiple parts of the documents were already of high quality. One big issue leading to this was the lack of support from companies, and the complexity in having the scope of the standard too wide for a single document. The draft was later released and published to the public. Yet, even as a draft, multiple Unix-like OS got inspired and implemented parts of it.

The document is a pillar in the standardization of OS security features and terminologies. Many of the terms listed have similar definitions to the ones we've seen such as "security", "policy", "policy model", "access", "availability", "confidentiality", "access control", "access control list", "security domain", etc..
Specifically, it has terminology regarding each of the five sections: access control list, auditing (event, log, record), its own definition of capabilities, and terms related to mandatory access control flow and labeling.

The standard also introduced the concept that is often called "type enforcement" (TE), an access clearance logic: When a process requests permission, it firsts checks if one of the alternate access control is in place, if there are none then the usual POSIX.1 permission is used, otherwise it will use the alternate one. For example, this can allow to layer MAC with DAC, giving priority to MAC.

Lastly, the standard was important because of its indirect impact on so many Unix-like OS vision of security features. These were often called "trusted" extensions, resulting in branches of the OS implementing them being named after it, such as TrustedBSD, Trusted Solaris, Trusted AIX, etc.. Then later, some merged the features into the main branch of the OS.
Let's have a look at what POSIX.1e/2c has in store by first surveying the POSIX Access Control List, a name we're familiar with but not in this context.

What you need to remember: *POSIX.1e and 2c is a dropped/draft standard defining extensions for new security interfaces and commands. It covers the topics of ACL, it's version of "capabilities", information labeling, MAC, and security auditing. The draft was important because of the way it defined security terms and inspired multiple Unix-like OSes to still implement the extensions regardless of its official acceptance status.*

POSIX.1e/2c Access Control Lists

The POSIX.1e/2c access control list (ACL), like the name implies, are a form of access control list, as we've seen in the matrix model section, and thus is also a discretionary access control mechanism. The idea is to extend the basic POSIX.1 permissions as a super-set allowing more fine grained subject tags while re-using the `rwx` permission bits as the action tags.

When listing files, it displays a `+` after the basic bits, indicating that extra access rules are present:

```
1 -rwxrwxr--+  1 vnm git     35 Jan  2 18:25 helloworldfile
```

To allow backward compatibility the draft defines a mapping between the previous file *owner*, *group*, and *other* bits, to the new tags defined by the POSIX ACL. It is achieved through a masking mechanism and a redefinition of what the *group* classification means.
As a consequence, this implies the implementers of this feature will have to add support at multiple levels: file system, functions, and utilities.

In POSIX.1 basic permission, every file was associated with only 3 permissions, 3 classes of subjects, and extra setuid, setgid, and sticky bits that we've seen. The POSIX.1e/2c ACL redefines the subject classes into the following:

- `ACL_USER_OBJ`: The basic *user* class, same as before.
- `ACL_GROUP_OBJ`: The basic *group* class, same as before.
- `ACL_OTHER`: The basic *other* class, same as before, no addition in ACL.
- `ACL_USER`: A new class called "*named users*", a list of specific users.
- `ACL_GROUP`: A new class called "*named group*", a list of specific groups.
- `ACL_MASK`: A new class used as a mechanism, called "*mask*", or maximum access rights, or "*upper-bound*". It applies over `ACL_USER`, `ACL_GROUP_OBJ` and `ACL_GROUP`.

Along with these, there are new concepts introduced such as the *minimum* ACL, the usual user/-group/others equivalence with basic POSIX permission. These are part of the *required* ACL entries, the minimum that should be present on files. Any new access permissions assigned to the files are called *extended* ACL.
Besides, another category of ACL can be associated with directories called *default* ACLs. They are used to determine the initial permissions to set on files created underneath, dismissing the *umask* when present. Keep in mind that the hard-coded default permissions in the kernel still applies (`666` for files and `777` for directories).

The way the draft standard resolves the conflict between *named users/groups* and the basic user/group is through the `ACL_MASK` mechanism. To achieve this, the *named users and groups* (`ACL_GROUP` and `ACL_USER`) are assigned under the standard group class, and the basic group class functionality is replaced by the *mask* (`ACL_MASK`), which acts as a maximum access right, the *upper-bound*, applied over *named users and groups*. These new semantics allow the backward compatibility.
For instance, if an access control list is created for a particular user, let's say "queen", with `rwx` permission, but the *upper-bound* only contains `r` permission, then "queen" will have as *effective* permission only `r` permission.

Let's note that POSIX.1e doesn't actually say that ACL tag types are limited to *named users and groups*, but only say that it defines a minimum set. Yet, no implementation actually adds more than these, and if ever, they rely on another feature called "extended attributes" (EA) instead.

With all these new terms, how does permission checks takes place:

- If the effective UID matches the UID of the file object owner and the permission matches, then permission is granted.
- If the effective UID matches one of the *named user* and the permission is both present in the *mask/upper-bound* and the *named user*, then permission is granted.
- If the effective GID matches any of the *named group* and the permission is both present in the *mask/upper-bound* and the *named group*, then permission is granted. If the *named group* and *mask/upper-bound* are not present, the same is applied to basic group permission.
- if the other entry contains the permission then permission is granted.
- If UID or GID matches in one of the above but the permission doesn't, then access is denied.

In sum, it's the same mechanism as before, but *named users and groups* are included in the mix along with their *mask/upper-bound*.

On the whole, there's a wide number of Unix-like systems implementing POSIX.1e/2c ACL, however it also depends on file system support for it. For example, AIX, SunOS derivatives, FreeBSD, macOS have implemented it. It is supported by the file system UFS (through shadow vnodes), NFSv4 and v3, ZFS, Ext2, Ext3, Ext4, IBM JFS, ReiserFS, SGI FS, and many more. Note that OpenBSD doesn't implement this feature, and none of the POSIX.1e/2c for that matter.
As you can imagine, these additional attributes of variable length set on files can potentially affect the access check time. Statistics have indeed shown that this can have a minor difference on certain file systems. Moreover, most Unix-like systems and file systems will limit the number of ACL entries on a file to keep it efficient. The same applies to many of the features in the following sections when they add attributes on files.

The draft defines a couple of ACL manipulation functions in an ACL library (`libacl`, `-lacl`) that comes bundled with the OS choosing to implement it. Some systems have additional extensions such as Linux found in separate header files (`acl/libacl.h` vs `sys/acl.h`).

On FreeBSD to enable ACL the following kernel option needs to be set:

```
1 options UFS_ACL
```

Furthermore, it also needs to be added to the mount-time options in `/etc/fstab` with `acls` flag.
Meanwhile, on Linux it is also enabled as a mount-time option `acl` if not enabled by default in the kernel compile (ex: `CONFIG_FS_POSIX_ACL`, `CONFIG_EXT4_FS_POSIX_ACL`, or `CONFIG_BTRFS_FS_POSIX_ACL`, depending on the file system).

On Linux, the actual implementation of the ACL relies on "extended attributes" (EA) which we'll see in a later section. The rational is to provide all metadata through the same interface, at least kernel-wise. It stores them as extended attributes on files named `system.posix_acl_access` and `system.posix_acl_default` for the access and default ACL respectively. However, if you dump them with the command `getfattr` (which, again, we'll see later) it'll output something hard to visually parse.

```
1 getfattr -n system.posix_acl_access .
2  # file: .
3 system.posix_acl_access=0sAgAAAAEABwD/////AgAGAFkAAAACAA\
4   YA6AMAAAQABQD/////EAAHAP////8gAAUA/////w==
```

The actual utilities used to manipulate the ACL are defined in POSIX.2c. Besides having `chmod`, `chown`, `cp` and others being backward compatible and respecting ACLs, they add the commands `getfacl(1)` and `setfacl(1)` to get and set ACLs.

For `getfacl(1)`, the draft defines 3 representations, an exportable/external form, an internal form that is dependent on the storage (file system), and multiple textual representations. There are countless functions to manipulate these structures defined in POSIX.1e such as `acl_to_text` and others that can be used to display them, which is what `getfacl(1)` relies on.
There are two types of text forms, the long form and the short/abbreviated one. In the long one, every line is an `acl_entry` that is colon separate with 3 entries: tag type (user, group, other, mask), entry qualifier (uid or gid or empty), and discretionary access permissions (`rwx-`), and implementation specific additional fields. Comments start with `#` and can be used to display effective permissions when applying the mask/upper-bound.

`setfacl` is used to set the ACL entries on files and directories, including the mask/upper-bound, and default ACL (only allowed to be set on directories).
Here's an example:

```
> setfacl -m u:vnm:rwx kk

> getfacl kk
    # file: kk
    # owner: vnm
    # group: users
    user::rw-
    user:vnm:rwx
    group::r--
    mask::rwx
    other::r--

> ls -l kk
-rw-rwxr--+ 1 vnm users 8 Jan  6  2021 kk
```

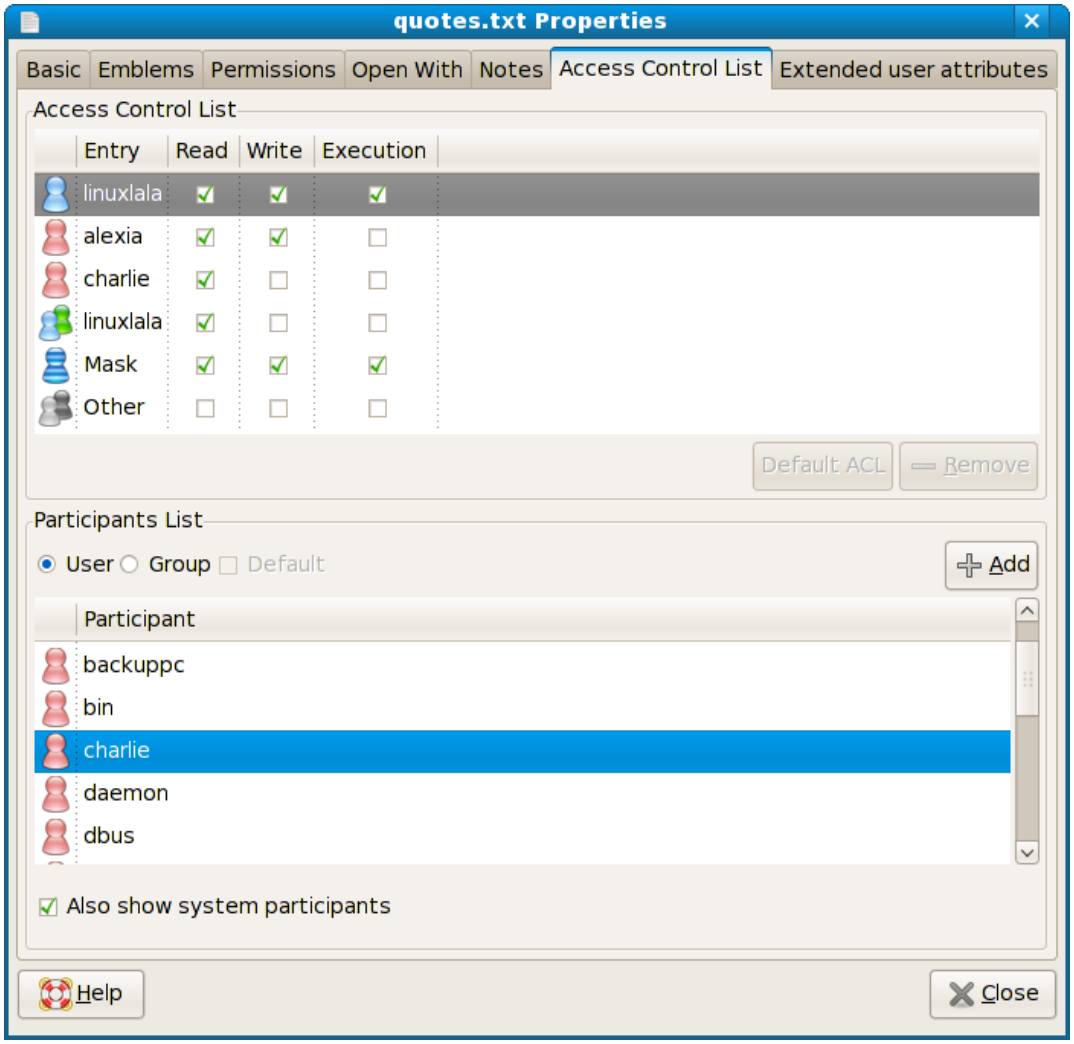
To set the default ACL on a directory:

```
> setfacl -d -m group:toolies:r-x dir

> getfacl --omit-header dir
    user::rwx
    user:joe:rwx
    group::r-x
    mask::rwx
    other::---
    default:user::rwx
    default:group::r-x
    default:group:toolies:r-x
    default:mask::r-x
    default:other::---
```

Now if you perform, `chmod g+w` it's the ACL mask/upper-bound that will be updated and not the group.

The last thing we have to think about is the support across other tools.
For instance, the copy utility `cp` can possibly either support it by default, copying ACL, or will require the `-p` flag to preserve them. The move utility `mv` usually always preserves them.
When it comes to backup and restore, GNU tar and GNU cpio used to not support them, however, recently most tar versions have added flags for storing ACL and EA ("extended attributes" which we'll see later), but that depends on the archive format chosen. The same applies to most other features in this part of the article.

Sadly, most front-ends, graphical user interfaces, barely have support for POSIX.1e/2c ACLs and usually only allow manipulation of standard permissions. This greatly reduces its adoption as the only way to edit them is either through the command line or through IdAM solutions. The only file manager that includes an add-on is nautilus, through the eiciel extension (It also has support for "extended attribute", EA).
Unfortunately, this issue will also be true for most of the features of POSIX.1e/2c found in the following sections.



What you need to remember: *POSIX ACL builds on top of the basic file permissions by adding new subjects: named users and named groups. It achieves backward compatibility by relying on a mask/upper-bound that replaces the definition of groups and is applied over the previous group and the named users and groups. Support for this feature is required at the kernel and file system level, however the GUI tooling lacks support, not only for POSIX ACL but for most POSIX.1e security extensions.*

POSIX.1e/2c Capabilities

POSIX.1e/2c capabilities defines interfaces that allow splitting root privileges and associating them with files and processes. Similar to the `setuid`/`setgid` that can be set on processes and files, capabilities also have an inheritance mechanism between parent process and child processes, a strict logic of what is passed down generations. The split privileges is a way to avoid relying on super-user and `setuid` when only needing to perform a specific task. Instead, the privileges are granular, they are fine-grained categories that encompass a group of allowed actions/functions. This is nice in theory, however, depending on what these capabilities are, they could still open the way to privilege escalation (let's say a capability allows to write directly to raw memory, or to manipulate `setuid` on files). While the name contains the word "capabilities", it is not to be confused with the capability-based models we've seen. POSIX capabilities can be associated with objects/files which is entirely different than the subject-only approach of capability-based security.

Thus, with POSIX capabilities, if a process needs networking privilege to bind on low ports they won't require full root privileges but only the related capability instead. It is another step in the direction of least-privilege.

The POSIX.1e specifications defines the inner workings of these capabilities, how to manipulate them, and lists a couple of example categories that are possible to implement. Additionally, there's a focus on how to keep the system secure while passing these privileges from parent to child process. However, the specifications in POSIX.2c, utilities, is vague and doesn't enforce how these could be manipulated, and thus every system that chose to implement them does it in their own way.
Indeed, only a few Unix-like OSes have this feature, namely TrustedBSD, Linux, and all SunOS derivates such as Solaris, OpenSolaris/OpenIndiana, Nexenta OS, illumos, Tribblix, OmniOS, SmartOS, etc.. All of them approach POSIX capabilities by inspiring themselves from the draft standard and sprinkling their own style unto it.

The draft, just like with POSIX ACL, defines functions to manipulate the capabilities, which are never edited directly but only through functions. Some structures they rely on are defined while others aren't, such as `cap_flag_t` which is defined as the capability flag, and `cap_t` which is opaque and internally defined (ex: on Linux it's relying on `_cap_struct` distributed in `libcap`/`lcap`). The actual storage is also implementation dependent, and as with ACL, relies on file system features. POSIX.1e also describes a textual grammar format to manipulate capabilities and convert them from one format to another, as we'll see.
Systems can create capabilities for any functionality, including other security features, such as the POSIX.1e ACL we've seen, and even capabilities themselves (On Linux there exists deprecated capabilities such as `CAP_SETPCAP` and `CAP_SETFCAP`).

Let's get some definitions down.

A **capability** is an attribute associated with a process or file, it is used to determine whether the process has the privilege to perform a privileged action (something usually only given with root privileges).

A **capability flag** is a per-capability attribute indicating how the capability can be used during execution, think of them like the setuid real/effective/saved UIDs. The capability can have one or many of the following three flags: *permitted*, *effective*, and *inheritable*. Some systems have additional flags, but the previous three are the minimal ones for POSIX capabilities implementations.

The capabilities with the *permitted* flag are the ones that are available to the current process, they can be "activated" through functions such as `cap_set_proc`.
The capabilities with the *effective* flag are the currently usable ones in the process, the ones that the kernel will check, this is a subset of the *permitted* capabilities.
The capabilities with the *inheritable* flag are the ones that may be passed to child processes.
While the meaning of these flags should be the same on both processes and files, in some implementations they differ, such as in Linux.

The flags don't apply to users with root privileges, they instantly have the full set of permitted and effective capabilities. Meanwhile, their *inheritable* capabilities set is usually empty for security purpose.

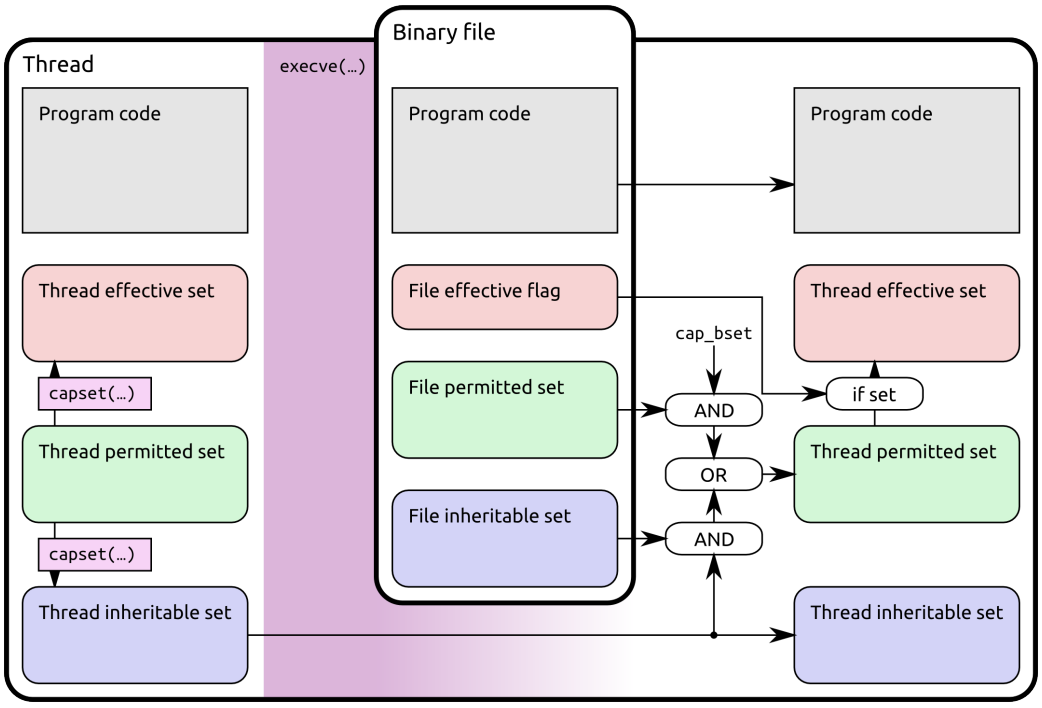
It follows, with the same mindset as with `setuid`/`setgid`, that these flags should be used to limit the propagation of privileges. This is why a basic algorithm is defined to calculate the capabilities a process will have when invoking an executable file. The process capabilities will be re-evaluated as follows (binary operations: `&` for `AND`, `|` for `OR`).

```
pI' = pI

pP' = (fP & X) | (fI & pI)

pE' = (fE & pP`)
```

Where `I` stands for *inheritable*, `E` for *effective*, `P` for *permitted*, `p[IPE]` represents the starting process capability sets, `p'[IPE]` represents the new process sets, and `f[IPE]` represents the file executable being invoked. The special `X` represents a possible global bounding set which can be used to limit what a process is capable of doing.

In human terms, this means that the new process will keep the *inheritable* capabilities from its parent while the *permitted* sets of capabilities will be all that is *permitted* in the file along what is in common between the *inheritable* from the file and parent, meanwhile the *effective* capabilities will be what is in common between the new *permitted* capabilities and the file's *effective* capabilities.
This implies that a process having capabilities is useless without the file being marked with the matching capabilities. People don't have privileges but executables do, in contrast with privileges associate with users. In other words, if a user session starts a shell with capabilities then they'll only have these capabilities through the shell, unless they have access to readable executables that matches the capabilities they want. Another way to get capabilities unto a process would be if another process was able to assign them dynamically, which is only available on certain systems (Linux).

The special bounding set `X` is also particular because it does only limit the new *permitted* set. This means that if a process has in its *inheritable* set some permissions that were supposed to be limited by the `X` bounding set, it'll still be able to exercise them when executing another file that has them in its *inheritable* set.



Source: Understanding Capabilities in Linux from ploetzli.ch blog from December 01, 2014. The "if set" is Linux specific instead of the `AND` *operator.* `cap_bset` *is the global bounding set shown as* `X` *above.*

For example, if a process starts with no capabilities but the file it executes has `CAP1` as *permitted* and *effective* then the process will follow this algorithm:

```
Consider the X bounding set allows all capabilities
pI' = {} # empty set
pP' = ({CAP1} & X) | ({} & {}) = {CAP1} & X = {CAP1}
pE' = ({CAP1} & {CAP1}) = {CAP1}
```

Thus, the process will be able to perform `CAP1` but any new executable it will invoke will lose this capability.

Let's note that this algorithm varies between implementations, other implementations might include new flags that can affect it (Linux has an *ambient* flag).

Comparably to the above `EIP` syntax, POSIX.1e defines a grammar for manipulating capabilities in a textual format. Unsurprisingly, `e`, `i`, and `p`, stand for *effective*, *inheritable*, and *permitted*. The name of the capabilities list is forced to be case-insensitive.
The available operations that can be performed on capabilities are either to add a capability `+` with a flag, to remove a flag `-` from it, or to fix it to a set of flags after resetting them `=`.
The special keyword "all" or "" (empty) represents all available capabilities in the bounding set.
This textual representation looks like this:

```
Set cap_chown to permitted and then add the flag effective to
it this is equivalent to "cap_chown=ep"
cap_chown=p cap_chown+e

Give "all" capabilities as permitted and effective then
remove effective flag of cap_chown and permitted and effective
flag of cap_kill
This is equivalent to "=ep cap_chown-e cap_kill-ep"
all=pe cap_chown-e cap_kill-pe"
```

All of the above gives rise to two methods of avoiding super-user privileges by using POSIX capabilities.
One of them is about creating "capability-dumb" binaries by swapping the `setuid`/`setgid` bits with the appropriate capabilities that are actually needed by the executable. These capabilities will often have the *effective* and *permitted* flags.
The other approach is "capability-smart", which consist of having the application capability-aware. The executable file will start with a set of *permitted* and *inheritable* capabilities which it will know how to programmatically set as *effective* or not depending on the situation (through `cap_set_proc` for example, see Linux example from k3a).

We can also wonder what happens when invoking executables that have both `setuid` bits and capabilities set on them. The behavior is unsurprising: the UID will be changed and the capabilities will apply to the new effective UID. Essentially, this means that even when invoking a root-`setuid` program, we can limit the available capabilities, and thus reduce the root privileges. Keep in mind that this behavior has more edge-cases depending on the system.
Another scenario is when a process has capabilities but execute a root-`setuid` program, the executable not having capabilities set on it this time. This usually results in capabilities being ignored, and the process acting with root privileges, keeping Unix semantic untouched.

What you need to remember: *POSIX capabilities, unrelated to capability-based security, is a way to split root privileges into granular ones and assign them to files and processes. Each assigned capability, to either the executable file or process, has one of 3 flags: effective, inheritable, and permitted, along with an optional bounding set. These flags decide, through a defined algorithm, which capabilities will apply after invoking an executable. The capabilities can be set programmatically (capability-smart) or statically on files (capability-dumb).*

POSIX Capabilities on TrustedBSD

Let's move to the implementations and start with TrustedBSD which calls this feature "fine-grained privileges" to avoid a conflict with capability-based security.

While many of the TrustedBSD patches were merged into FreeBSD, the fine-grained privileges never were as they represented a "substantial risk" to how the super-user privilege model worked and could introduce unexpected vulnerabilities. Yet, the real reason was that it wasn't reviewed properly and the authors probably weren't convinced. Hence, the code lives as an unmaintained patch for FreeBSD

5.0 (as of this article the latest stable version of FreeBSD is 14.0).
The patches are still available online for download and contain, along with the code changes, scripts, and man pages.

For instance, the possible capabilities are listed in the `cap(3)` man page (`/lib/libc/posix1e/cap.3`), along with the implementation as a 64bit bit set which are stored, similar to FreeBSD's POSIX ACL, as shadow vnodes. Each bit in the set represents a capability being turned on, however, the user would barely have to handle this internal representation and would instead rely on the POSIX.1e textual layer implemented in `cap_text.c`.

TrustedBSD defines many capabilities, such as `CAP_CHOWN` to allow changing the owner of any file regardless of the current owner, `CAP_KILL` to allow killing processes regardless of effective and real UID, `CAP_NET_RAW` to allow creating raw sockets, `CAP_SYS_MODULE` to allow loading/unloading kernel modules, etc..
Most notably, the `CAP_ALL_ON` will turn on all capabilities, which is the logical equivalent of becoming super-user.

When it comes to the algorithm used to propagate capabilities, the only difference with the one we mentioned previously is that TrustedBSD doesn't implement the concept of bounding-set, yet implements something different through a per-user maximum set.

One particularity of TrustedBSD is that it maintains a capability database to associate a "default" and "maximum" capability set to users upon login. The "maximum" set is the equivalent of a bounding set that exists on a per-user basis, and not a global one. This also means, that a user not listed in the capability database will not have any capabilities in its set.
The file is found in `/etc/capability` and is compiled into `/etc/capability.db` by issuing `pcap_mkdb(8)` to regenerate the database file. It consists of a colon-separated list of users, their default capability sets and the maximum sets. The format of the sets follows the POSIX.1e/2c grammar we've seen above.

```
username:default_set:maximum_set
```

For example, give root all capabilities:

```
root:all=ep:all=eip
```

Remove all capabilities of the root user:

```
root:all=:all=
```

Give the backup user the possibility to read any file on the system:

```
backup:CAP_DAC_READ_SEARCH=eip,CAP_MAC_READ=eip:\
CAP_DAC_READ_SEARCH=eip,CAP_MAC_READ=EIP
```

For the management of POSIX capabilities, TrustedBSD offers the commands `getpcap`, `setfcap`, and `getfcap`. These print the process capabilities in text format, alter the capability set of a file, and print the capability set of a file in text format, respectively.

While TrustedBSD's implementation was promising, it was halted short and many of the features are half-working such as the "maximum" per-user set, which seems to be ignored in the code.

What you need to remember: *TrustedBSD's "fine-grained privileges" is an implementation of POSIX capabilities that includes a per-user bounding set and a capability database to associate "default" and "maximum" sets to users. However, the implementation is lacking and unpolished, it lives as a patch for FreeBSD 5.0.*

POSIX Capabilities on Linux

One system that embraces POSIX.1e/2c capabilities open-heartedly is Linux. It applies and extends it to suit its needs with new modes. On one side, it has the typical POSIX.1e headers in `<sys/capability.h>` and has its customized interfaces in `<linux/capability.h>` (`cap_iab(3)` for example). Furthermore, the manpage `capability(7)` goes to great length at explaining the fine details, so much that it becomes dizzying and confusing.

Similar to TrustedBSD, Linux implements the capabilities as a 64-bit set, each bit holding a capability. It can also be inspected through the procfs virtual file system in `/proc/<pid>/status`, the maximum bit that could possibly be set is found in `/proc/sys/kernel/cap_last_cap`.
On the file system side, it is implemented the same way that POSIX ACL are and for the same reason: stored as an extended attribute in the `security.capability` attribute, so that all metadata are accessed through the same interface from the kernel side. Let's note here that the format of the capabilities has changed over the years, and the current is `VFS_CAP_REVISION_3`.

```
1 > grep -i cap /proc/self/status
2 CapInh: 0000000000000000
3 CapPrm: 0000000000000000
4 CapEff: 0000000000000000
5 CapBnd: 000001ffffffffff
6 CapAmb: 0000000000000000
```

To no one's surprise, Linux also offers a wide array of different capabilities such as `CAP_SETUID` which allows arbitrary setting the setuid bit, `CAP_NET_RAW` allowing using raw sockets, `CAP_MKNOD` allowing creating special files using `mknod(2)`, and much more.
In particular, `CAP_SETFCAP` and `CAP_SETPCAP` are capabilities related to setting capabilities on files and processes/threads respectively.

The catch-all capability, which is somewhat equivalent to super-user, similar to `CAP_ALL_ON` on TrustedBSD, is called `CAP_SYS_ADMIN` capabilities. While it doesn't include all capabilities, it is still overloaded and allows so much that it's close to being a new root privilege.

One particularity on Linux, is that shared object files can have capabilities and, upon linking, the invoked executable will get associated with them.

When it comes to the algorithm used to propagate capabilities there are a couple of differences with the POSIX draft. The first, is that the bounding set is implemented on a per-thread/task basis (in the latest version). Linux also fixed the issue with *inheritable* capabilities that could be outside the bounding set by disallowing them. Not to mention, there are also two other particularities: a new tag called the *ambient* capability set, and another meaning for what the *effective* set flag does when put

on a file. Furthermore, Linux goes into great details and flexibilities of configuration when it comes to special scenarios, such as when capabilities are present on `setuid` files, or special treatments when changing from super-user to normal users and vice-versa.

The *ambient* capability set is one that is omnipresent across the lifetime of a process, across `execve(2)`. That is all until a file is executed that either has a `setuid`/`setgid` bit or capabilities set on it. In that case, the *ambient* is cleared. We call these files "privileged". The *ambient* set makes it easy to give non-super-user capabilities without relying on file capabilities but through the specific Linux functions such as `prctl(2)` (process control) and `capset(2)`, and some command line tools such as `capsh(1)` as we'll see. This changes the concept of capabilities, giving priority to the parent process and its environment, instead of files.
The *ambient* set is filled through these specific Linux functions for processes/threads, and, upon setting them, the *ambient* capabilities are limited to the ones already present in the *permitted* and *inheritable* set.

The difference in the *effective* capability sets on files, is that on Linux this isn't an actual set but a bit that is either turned on or off. When this boolean is on, all the new *permitted* capabilities are copied in the new process *effective* set, otherwise only the *ambient* set is taken into consideration. What's more, Linux also used to call the *permitted* set on files the *forced* set, and the *inheritable* one the *allowed* set, which makes sense considering how they'll be used in the algorithm.

With these in mind, this is how the Linux capability transformation algorithm looks like:

```
1  P'(inheritable) = P(inheritable)    [i.e., unchanged]
2
3  P'(bounding)    = P(bounding)       [i.e., unchanged]
4                    # new: it's now per-process/thread
5
6  P'(ambient)     = (file is privileged) ? 0 : P(ambient)
7
8  P'(permitted)   = (P(inheritable) & F(inheritable)) |
9                    (F(permitted) & P(bounding)) | P'(ambient)
10                   # new: the P'(ambient)
11
12 P'(effective)   = F(effective) ? P'(permitted) : P'(ambient)
13                   # new: the P'(ambient)
14                   # new: F(effective) is a boolean
```

It would be easy if Linux only used the above as an overlay over the POSIX.1e algorithm, however, it goes further and introduces configurations to control the behavior of the algorithm, especially when changing between users that are privileged (`UID=0`) and those who aren't.
As we mentioned before, the behavior of the algorithm changes when the `setuid`/`setgid` bit is involved. Depending on whether the process has capabilities, or the file has capabilities, things might happen differently.

For one, we mentioned that when invoking a `setuid`/`setgid` binary or one that has capabilities on it, "privileged", the *ambient* set it cleared.
Additionally, the mechanism of `setuid` programs that have capabilities we mentioned in the POSIX.1e still applies: that is if the real UID of the process isn't 0 and the effective UID of the process becomes

0, then the capabilities on the file will be set on the process via the algorithm, without giving full root privileges.

Now, if any of the UID are changed, and one of the previous UID was 0, and the change results in this UID 0 disappearing, then all capability sets are cleared.
If the same scenario happens but the UID 0 is kept in one of them, and the effective UID has changed from 0 to non-zero, then the *effective* set is cleared.
If the same scenario happens but the UID 0 is kept in one of them, and the the file system UID has changed from 0 to non-zero, then only the capabilities related to the file system are cleared from the *effective* set (`CAP_CHOWN`, `CAP_FOWNER`, etc..).
If the *effective* UID is changed from non-zero to 0, then the *permitted* set is copied to the *effective* set.

To add more to all this, the above behavior depends on further configurations at different levels.

At the kernel level, if it is booted with the option `no_file_caps` then the kernel will not honor file capabilities, only process ones.

At the process level, there are a couple of different options affecting the capabilities decisions.
A simple one is the `no_new_privs` bit, which is a generic mechanism that stops a process from having more privileges than before invoking an executable with `execve()`. This applies to file capabilities, `setuid`, and others. This is set with `prctl(2)` on a thread/process.
Likewise, `prctl(2)` has another series of configurations to change the inheritance of capabilities for privileged users that it calls "secure bits". These can also be set with functions such as `cap_set_secbits(3)`. For example, setting the `SECBIT_KEEP_CAPS`, will disable the clearing mechanism when changing all UID from 0 to non-zero. To completely disable all the clearing in all scenarios we mentioned above, then the `SECBIT_NO_SETUID_FIXUP` will do that. Another interesting secure bit is the `SECBIT_NOROOT`, which will avoid granting users with root privileges all capabilities by default.
All of the secure bits have a companion "locked" flag which will prevent further change, making them irreversible.

All and all, this makes POSIX capabilities on Linux very advanced but also very hard to grasp. This is why the authors of the Linux' libcap came up with another approach to simplify the complexity which is called "libcap modes". Yet another layer on top!
The libcap modes encapsulate a set of configuration to assign to processes using specific functions (`cap_set_mode(3)`), they have names such as `CAP_MODE_NOPRIV` (the equivalent of `no_new_privs`), `CAP_MODE_PURE1E_INIT`, `CAP_MODE_PURE1E`, and `CAP_MODE_HYBRID`. In the `PURE1E` modes, the *ambient* set is completely disabled, keeping somewhat more true to the POSIX.1e draft, and being root doesn't come with super user privileges, basically enabling the `SECBIT_NOROOT` securebit (yet it still owns a lot of files). This means everything only runs on capabilities.

The libcap maintainers also came up with a novel approach to handling capabilities that only includes the *inheritable*, *ambient*, and bounding sets. They call it the capabilities "IAB" format.
If you only care about these sets then the inheritance will be more simple to think about. The *ambient* set can't contain more capabilities than the *inheritable* and *permitted* set, so the *inheritable* set will act as a sort of bounding set for it, and the bounding set will limit all of them. Confusingly, the IAB style inheritance is summarized as: `I'=I; A'=A&I; P'=A&I&P`.

Fortunately, there are enough tools and commands to handle POSIX capabilities on Linux to make this less painful.

First and foremost, we can glance at what `security.capability` extended file attribute actually looks like. As expected, it's a binary format, and as we've said there are multiple versions, the current is `VFS_CAP_REVISION_3`.

```
> getfattr -n security.capability newone
    # file: newone
security.capability=0sAQAAAgAEAAAAAAAAAAAAAAAAAAA=
```

Like TrustedBSD, Linux has the `getcap(8)` and `setcap(8)` commands, both only valid for getting and setting capabilities on files.

```
> getcap newone
newone cap_net_bind_service=ep
```

To get process capabilities the `getpcaps(8)` command exists, it takes the process ID as a param, and also allows showing the capabilities in IAB style (`--iab`).

```
> getpcaps 458
458: cap_net_bind_service,cap_net_admin,cap_net_raw=ep

> getpcaps --iab 458
458: "cap_net_bind_service, cap_net_admin, cap_net_raw=ep" ↙
    ↳ [!cap_chown,
!cap_dac_override, !cap_dac_read_search, !cap_fowner,
!cap_fsetid, !cap_kill, !cap_setgid, !cap_setuid,
!cap_setpcap, !cap_linux_immutable, !cap_net_broadcast, !cap_ipc_lock,
!cap_ipc_owner, !cap_sys_module, !cap_sys_rawio, !cap_sys_chroot, ↙
    ↳ !cap_sys_ptrace,
!cap_sys_pacct, !cap_sys_admin, !cap_sys_boot, !cap_sys_nice,
!cap_sys_resource, !cap_sys_time, !cap_sys_tty_config, !cap_mknod, ↙
    ↳ !cap_lease,
!cap_audit_write, !cap_audit_control, !cap_setfcap, ↙
    ↳ !cap_mac_override, !cap_mac_admin,
!cap_syslog, !cap_wake_alarm, !cap_block_suspend, !cap_audit_read,
!cap_perfmon, !cap_bpf, !cap_checkpoint_restore]
```

Three other commands are great to list capabilities that exists on all processes and files on the system. `netcap`, specifically for network-capable processes, `pscap(8)` for any process, and `filecap(8)` to get and set capabilities and search for all the files which have them set in `$PATH`.

```
> filecap $PWD/newone
set       file                  capabilities  rootid
effective /home/vnm/junk/newone    net_bind_service
> filecap        # list all files in $PATH
set       file                  capabilities  rootid
effective /usr/bin/dumpcap    dac_override, net_admin, net_raw
effective /usr/bin/rcp    net_bind_service
effective /usr/bin/rlogin    net_bind_service
effective /usr/bin/pcsx2-qt    net_admin, net_raw
effective /usr/bin/newuidmap    setuid
effective /usr/bin/newgidmap    setgid
```

```
effective /usr/bin/gnome-keyring-daemon    ipc_lock
effective /usr/bin/rsh    net_bind_service
```

```
> pscap
ppid  pid   name        command           capabilities
 ------------------------------------------------------
1     439   root        haveged           sys_admin
1     1490  root        agetty            full
1     1511  ntp         ntpd              net_bind_service, sys_time +
```

`captest` is another interesting command which you can use to test setting capabilities, it will then attempt to access `/etc/shadow` and print the current capabilities.

The most advanced capability tool is `capsh(1)`, a capability shell wrapper. It lets you set specific capabilities, debug them, and launch a new shell with them.
To achieve this it uses a neat assortment of functions part of Linux's special capabilities that is called `cap_launch(3)`.

The tool even allows listing and setting modes, debugging current capabilities, setting securebits, starting a `no_new_privs` envs (through related mode), relying on IAB style capabilities, and much more.

```
> capsh --modes # list them
Supported modes: NOPRIV PURE1E_INIT PURE1E HYBRID
> sudo capsh --mode=PURE1E -+
>
> capsh --print
Current: =
Bounding set =cap_chown, cap_dac_override, cap_dac_read_search, ↙
    ↳ cap_fowner,
cap_fsetid, cap_kill, cap_setgid, cap_setuid, cap_setpcap, ↙
    ↳ cap_linux_immutable,
cap_net_bind_service, cap_net_broadcast, cap_net_admin, cap_net_raw,
cap_ipc_lock, cap_ipc_owner, cap_sys_module, cap_sys_rawio, ↙
    ↳ cap_sys_chroot,
cap_sys_ptrace, cap_sys_pacct, cap_sys_admin, cap_sys_boot, ↙
    ↳ cap_sys_nice,
cap_sys_resource, cap_sys_time, cap_sys_tty_config, cap_mknod, ↙
    ↳ cap_lease,
cap_audit_write, cap_audit_control, cap_setfcap, cap_mac_override, ↙
    ↳ cap_mac_admin,
cap_syslog, cap_wake_alarm, cap_block_suspend, cap_audit_read, ↙
    ↳ cap_perfmon,
cap_bpf, cap_checkpoint_restore
Ambient set =
Current IAB:
Securebits: 00/0x0/1'b0 (no-new-privs=0)
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
```

```
uid=1000(vnm) euid=1000(vnm)
gid=100(users)
groups=10(wheel), 14(uucp), 50(games), 54(lock), 81(dbus),
90(network), 91(video), 92(audio), 93(optical), 95(storage),
98(power), 100(users), 963(realtime),994(docker)
Guessed mode: HYBRID (4)
```

Here's an example of debugging:

```
> capsh --suggest="net_bind"
cap_net_bind_service (10) [/proc/self/status:CapXXX: ↙
    ↳ 0x0000000000000400]

    Allows a process to bind to privileged ports:
      - TCP/UDP sockets below 1024
      - ATM VCIs below 32

> capsh --explain="cap_setuid"
cap_setuid (7) [/proc/self/status:CapXXX: 0x0000000000000080]

    Allows a process to freely manipulate its own UIDs:
      - arbitrarily set the UID, EUID, REUID and RESUID
        values
      - allows the forging of UID credentials passed over a
        socket

> capsh --decode=0x0000000000000410
0x0000000000000410=cap_fsetid,cap_net_bind_service
```

To start a capability shell using IAB style:

```
> sudo capsh --user=$(whoami) --iab='^cap_setuid' --
```

A similar tool is `setpriv(1)` which will launch a program with the specific capabilities/privileges set. It allows `no_new_privs`, securebits, and IAB style capabilities.

```
> setpriv --no-new-privs tree -L 1
.
|-- docu
|-- dot
|-- downloads
|-- Dropbox
|-- junk
[-- media

7 directories, 0 files

> setpriv --dump
uid: 1000
euid: 1000
gid: 100
```

```
egid: 100
Supplementary groups: 10,14,50,54,81,90,91,92,93,95,98,100,963,994
no_new_privs: 0
Inheritable capabilities: [none]
Ambient capabilities: [none]
Capability bounding set: chown,dac_override,dac_read_search,fowner,\
fsetid,kill,setgid,setuid,setpcap,linux_immutable,\
net_bind_service,net_broadcast,net_admin,net_raw,\
ipc_lock,ipc_owner,sys_module,sys_rawio,sys_chroot,\
sys_ptrace,sys_pacct,sys_admin,sys_boot,sys_nice,\
sys_resource,sys_time,sys_tty_config,mknod,lease,\
audit_write,audit_control,setfcap,mac_override,mac_admin,\
syslog,wake_alarm,block_suspend,audit_read,perfmon,\
bpf,checkpoint_restore
Securebits: [none]
Parent death signal: [none]
```

There are multiple other small utilities such as `captree`. A noticeable project is `sucap` a version of the `su` command that only relies on capabilities and not on setuid bit (see).

Unlike TrustedBSD, Linux doesn't offer a global configuration to set capabilities to users as they login, instead there exists a pam module that does a similar thing: `pam_cap`. Curiously, it's this PAM module that was the reason for the creation of the IAB style capabilities. It stores the users along with their capabilities in a default configuration `/etc/security/capability.conf`. It has its own syntax, which we won't dive into to apply the IAB capabilities. This makes more sense to put an *ambient* set on users as it'll stay present across the user's session.

Another approach with POSIX capabilities on Linux is to use them when launching services. `systemd` has a couple of configuration that can be applied in its unit/service files to configure what needs to be removed or set in the bounding set, ambient set and others. This is an excellent approach to not give full root privileges to users, and avoid escalation.

Altogether, as this section has portrayed, Linux' take on POSIX capabilities is genuinely advanced. Linux has succeeded in adopting them in its own way. However, even though there are countless tools to manipulate capabilities, the average user's knowledge of them is far from being deep, considering the complexity of man pages such as `capabilities(7)`. Hopefully, this article will spark curiosity while explaining it in a more approachable tone.

What you need to remember: *Linux's take on POSIX capabilities expands and modifies it by adding an ambient set that is omnipresent until we execute privileged files, it also changes the meaning of the effective flag on files making it a boolean instead, and changes the transformation algorithm to include edge cases when switching UIDs. The last point, the change in the capabilities when switching UIDs, can be controlled through "securebits" and others. An IAB (Inheritable, ambient, bounding) format was included to make it easier to manage capabilities, along with a lot of small tools such as `capsh`. There's no system-wide configuration to pre-configure users with capabilities, but there exists a PAM module `pam_cap` to do something similar, and a way to fix systemd unit files with bounding sets and others.*

SunOS and all its derivatives, ranging from Solaris, OpenIndiana, to illumos, have their own quirky approach to POSIX.1e/2c capabilities and access control in general, (see also the SunOS profiles and RBAC sections). Like TrustedBSD it also calls them "fine-grained privileges" but the resemblance stops there. Unlike both the previous systems, it has none of the POSIX.1e headers and instead has `<priv.h>` with its own privilege-related functions, all starting with `priv_*`.

Similar to the previous two systems, it stores the capabilities as an integer, however, its naming convention is different. The SunOS derivatives define their privileges starting with `PRIV_` instead of `CAP_`. Furthermore, they don't only include operations that would be allowed by root, but also include operations that could be performed by normal users, grouping them in a set of "basic" privileges that is given by default to all processes. These "basic" privileges include: `PRIV_FILE_LINK_ANY`, `PRIV_PROC_INFO`, `PRIV_PROC_SESSION`, `PRIV_NET_ACCESS`, `PRIV_PROC_FORK`, and `PRIV_PROC_EXEC`.
Moreover, `setuid` binaries lose their ability to work unless the user has the capabilities: `PRIV_PROC_SETID` and `PRIV_PROC_AUDIT`.
The list of all privileges can be found in `/etc/security/priv_names`, the privilege definition file.

The biggest difference with POSIX.1e capabilities is that SunOS derivatives privileges are only assigned to processes and users, not to files. This is done either through user management configurations (`user_attr`, `prof_attr`, …), or dynamically on the command line with `ppriv(1)`.
They have the same sets of capabilities as POSIX.1e but the bounding set has been renamed the "limit" set, and, as with Linux, is per-process.

Having the privileges assigned only to subjects makes SunOS derivatives implementation of POSIX capabilities closer to real capability-based security systems. Yet, it still picked another naming convention than "capability".

Since it's a process-only privilege system, they have added the possibility to either turn privilege awareness on or off in what's referred to as the *Privilege Awareness State* (PAS). This can also be done either through the command line or through functions such as `setpflags` with the `PRIV_AWARE` flag. Privilege awareness is a mechanism akin to the security bits on Linux and capability-smart executables, avoiding or not a change in *effective* and *permitted* capabilities when executing `setuid` executables. When a process is privilege-aware, their "observed" capabilities don't change, otherwise, when it switches to privilege-unaware mode then the *effective* and *permitted* sets will change based on whether the effective UID is 0 or any of the UID is 0.
This means a non-privilege-aware process that has an effective UID of 0 can exercise all privileges within their limit/bounding set, basically returning the functionality of `setuid` bits on executables.

Note that, whenever a user executes a program, the kernel directly tries to relinquish privilege awareness and sets the "implementation" set to the *inheritable* set restricted by the limit set.

This all can seem a bit messy, but it makes more sense when we take a look at command line utilities and system management.

The `ppriv(1)` command is used to inspect and modify process privileges and attributes on-the-fly. It can also be used to start commands, in a sense, it's similar to Linux' `setpriv` command.

```
1 > ppriv $$
2     387: -sh
```

```
    flags = <none>
    E: basic
    I: basic
    P: basic
    L: all

Remove PRIV_PROC_SESSION (Allow a process to send signals or trace
  processes outside its session)
This means we can't send signals to the parent process
> ppriv -s EI-proc_session $$

> ppriv $$
    387: -sh
    flags = <none>
    E: basic,!proc_session
    I: basic,!proc_session
    P: basic
    L: all

Inspecting a process
> ppriv -S `pgrep rpcbind`
    928: /usr/sbin/rpcbind
    flags = PRIV_AWARE
    E: net_privaddr,proc_fork,sys_nfs
    I: none
    P: net_privaddr,proc_fork,sys_nfs
    L: none
```

Yet, it can be hard to know which capabilities/privileges a program will need, that's why there's a useful utility called `truss` (similar to OpenBSD's `systrace` which we'll see in the isolation/constraint section). It's a tracing utility that can be used to list which privileges were needed to accomplish what the program was doing.

We said that users have "basic" privileges set on them by default, but haven't discussed how to set more than those upon login (and not with a utility). That's where the `/etc/user_attr` extended user attribute database file that we've seen before comes in!
We're interested in two keys in the `attr`, the `defaultpriv`, for the *inheritable* sets upon login, and the `limitpriv` for the limit/bounding set upon login.

```
jdoe::::defaultpriv=basic,proc_clock_highres;type=normal
```

We can also modify it using `usermod(8)` and `rolemod(8)` system utilities.

```
> usermod -K defaultpriv=basic,proc_clock_highres jdoe
```

The default system-wide privileges can also be set in `/etc/security/policy.conf` in the `PRIV_DEFAULT` and `PRIV_LIMIT` keys.

Privileges can be assigned to the profiles we've seen earlier, in the file for profile attributes `/etc/security/prof_attr` within the `attr` field as a `privs`, which is a list of comma separated privileges that the profiles will get access to. It can also be set in the execute attributes in the `privs`

and `limitprivs`. This means that with commands such as `pfexec` when switching profiles we can select to turn on certain privileges with the `-P` argument.

Example:

```
1 > pfexec -P all chown user file
```

Lastly, SunOS derivatives allow privilege/capabilities debugging through a system-wide configuration files `system(5)` by setting the variable `set priv_debug = 1` in it. It also offers the option within its debugger called `mdb(1)`.

All things considered, SunOS derivatives have it weird when it comes to POSIX.1e capabilities, if it can still be called that. They're only process-based, include a "basic" set that isn't privileged operations, and are handled in quite a peculiar way. However, it's still a good mechanism to implement granular-privileges.
That's it for capabilities, now we should move to another POSIX.1e/2c draft functionality: Mandatory Access Control, something we've learned about but haven't actually seen in practice.

What you need to remember: *SunOS derivatives have their own quirky way to implement POSIX capabilities, which it calls fine-grained privileges. They have their own privilege-related functions and only allow applying them to processes/users. This means processes need to be capability-aware to activate the functionality. Additionally, we can configure, through the extended attribute file `/etc/user_attr`, the privileges the users will have. In SunOS derivatives there's a group of "basic" capabilities, not root-privilege-related, that are given to all users by default and can be customized.*

POSIX.1e/2c Mandatory Access Control

POSIX.1e/2c mandatory access control defines abstract interfaces to implement all kinds of MAC.

The research and papers on such non-bypassable, centralized way to perform access control aren't novel. There are countless methods of implementing it such as the MLS (multi-level security), which is a form of MAC, that is explained in the Orange Book. Likewise, all the flow-based access control models with their clearance levels can be implemented with MAC, either as path-based or label-based. None of this was new but there was a need for standardization, which is where the POSIX.1e/2c draft comes in: to make it mainstream.

The goal of the draft is to pick a middle-ground between the performance overhead in security checks that comes with MAC, as with all previous POSIX.1e features we've seen, and to still allow the flexibility to implement any MAC policy without intrusively being tied to the kernel. Moreover, there should possibly be a way to support multiple simultaneous policies at the same time, layering them.

Thus, the MAC framework should lie as a thin layer in-between the kernel, policies, and the security-aware applications. System calls should be changed to be intercepted by this layer, adding the needed checks.

The POSIX.1e/2c implementation chose to rely on a label-based approach to achieve this. The labels are additional metadata assigned to files (objects) and processes (subjects), that are policy agnostic, and where the persistent storage is specific to every implementer. The association with files is similar to POSIX capabilities and POSIX ACL that we've dived into earlier, however with MAC even the textual representation isn't defined.

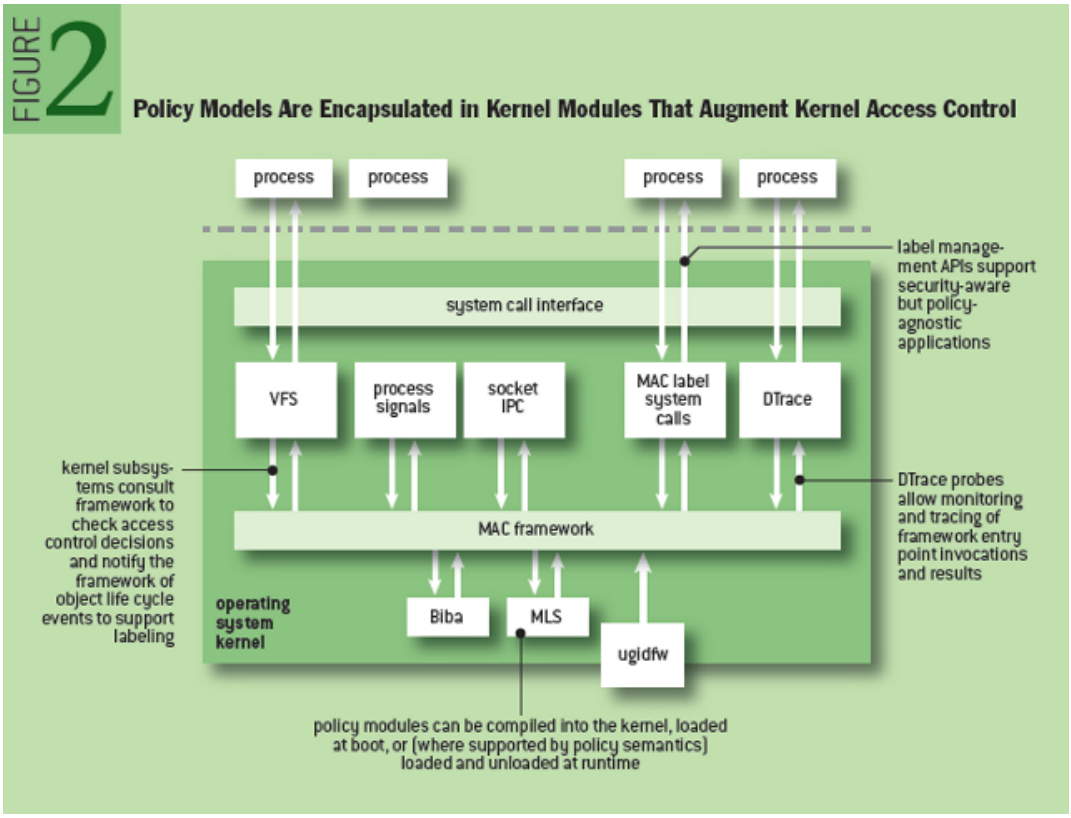
To create a policy means defining the functions that are used to compare these labels, giving them their inherent meaning, and using functions to read and write them as opaque data objects, only accessible through the implementation's API.

An implementation of a policy should follow certain concepts related to labels.
First, all subjects and objects on the system must have MAC labels on them at all time.
Second, the meaning of the relationship between labels should be defined, the terms "dominance" and "equivalence" are results of implementation-specific functions. A dominance means that there is a partial order between labels, one is above the other, and an equivalence means that there's no dominance. Strict dominance is a dominance in which there is also no equivalence.
Third, there can be operations to change the value of MAC labels, upgrading or downgrading them, to one that dominates or not the current label.
Fourth, the practical meaning in access control of the dominance takes form as follows, considering it's the MAC subject that is the active entity causing information to flow between controlled objects.:

- `P`: The fundamental statement mandatory of access control policy.
  Subjects cannot cause information labeled at some MAC label `L1` to become accessible to subjects at `L2` unless `L2` dominates `L1`.
- `FP.*`: The refinement of `P` that applies to file objects.
  - `FP.1`: The MAC label of a file should be dominated by the label of the subject to be able to read data and attributes of that file.
  - `FP.2`: The MAC label of a file should dominate the label of the subject for that subject to write to this file.
  - `FP.3`: For FIFO and pipes, the `FP.1` and `FP.2` apply.
  - `FP.4`: A new object shall be assigned a MAC label which dominates the MAC label of the creating subject, it can also be equivalent.
- `PP.*`: The refinement of `P` that applies to processes/system-calls/signals.
  - `PP.1`: An `L1` process can write to an `L2` process if it is dominated by `L2`.
  - `PP.2`: A new process has a MAC label which dominates the MAC label of the creating process, it can also be equivalent.

In general, this could be summarized as read access being allowed when the process dominates the file or other process, write access being allowed when the process is dominated by the file or other processes, and new files or processes having a higher or equal dominance than the current one. Theoretically, this is somewhat equivalent to the Bell-LaPadula model, however, since the relationship between labels is abstract, Biba, or any other security policy models can be implemented with this.

Practically, POSIX.1e defines a set of headers in `<sys/mac.h>` with label-management functions and definitions of objects, leaving implementation-defined functions as abstract, such as `mac_dominate(mac_t labela, mac_t labelb)`.



Source: A Decade of OS Access-control Extensibility - Open source security foundations for mobile and embedded devices, Robert N. M. Watson from January 18, 2013

Before moving further into the mandatory access control specificities on all kinds of Unix-like OSes, let's jump into information labeling aka extended attributes, which we kept mentioning and which is intimately tied to MAC.

What you need to remember: *POSIX MAC uses the concept of labels on subjects and objects, along with abstract ordering functions to allow implementers to define their own policies. The implementers have to define how a label dominates another, which affects the subject's access to the object.*

POSIX.1e/2c Information Labeling & Extended Attributes

POSIX.1e/2c information label is metadata, as simple as that. It can represent a security attribute of subject/object but it isn't mandated, nor used for neither MAC nor DAC.

As with POSIX.1e MAC label relationships, information labels are omnipresent on subjects and objects and can have equivalence and dominance functions defined, together with other functions to operate on labels such as `inf_float(inf_p1, inf_p2)`, `inf_default()`, etc..

The "information label policy" is the name given to these optional functions that operate on label flow, deciding how labels change when reading/writing to files/processes. Files that are outside this scope are called "non-floating".
The functions are found in the header `<sys/inf.h>` along with the related structures such as `inf_t`.

Despite being a good theoretical idea, information labels actual implementation didn't take the same form as the draft, nor their relationship or their omnipresence on subjects/objects. None of the POSIX.2c commands (`getfinf`, `getpinf`, `setfinf`) to get and set file or process information label, including the functions, exist on any popular Unix-like system.
Instead, "extended file attributes" became the de-facto method to associate metadata to files.

While the standard file attributes are represented in a `stat` structure, the extended attributes, often called `xattr`, need support at the `libc` level and is returned as a list of strings. The functions manipulating them are usually named like `listxattr`, `getxattr`, `removexattr`, `setxattr`, etc..
The strings don't have to be formatted in any particular manner, however, they are frequently colon separated key-value pairs. These key-value pairs are then interpreted, sometimes for access control features, as we've seen with POSIX ACL and POSIX capabilities.

On Linux, these attributes are also prepended with a namespace identifier followed by a period: `namespace.attribute_name:value`. The namespaces are limited to either `user`, `trusted`, `security`, or `system`. For instance, the `system` namespace is reserved for kernel and access control, the `security` namespace is used by SELinux, the `user` namespace is used for arbitrary information, and lastly the `trusted` namespace is like the `user` namespace but can only be read by super-user or users with the `CAP_SYS_ADMIN` capability.
On MacOS, downloaded files are tagged with the `com.apple.quarantine` extended attribute.

As we mentioned before, the support for these extended attributes depends on the OS and file system used. They are also limited in size, both the list size and the string size, for performance reasons.
Multiple Unix-like OSes have them, ranging from MacOS, Linux, FreeBSD, AIX, Solaris derivatives, and more. In file systems, the support is found in UFS1, UFS2, ZFS, ext2/3/4, HFS+, JFS, Squashfs, ReiserFS, XFS, Btrfs, and many more. The feature sometimes has to be enabled at the kernel too, for example on Linux it is configured a compilation time with options such as `CONFIG_REISERFS_FS_XATTR` and `CONFIG_TMPFS_XATTR`.

Similarly, since it's not a standard attribute, the tools need extra functionality to support them. That's more important with file manipulation tools and backup tools. Support is found in GNU tar and others with dedicated flags.

The commands to manipulate, get and set, the extended attributes are usually called `getfattr(1)` and `setfattr(1)`.

Here's an example on Linux:

The command `getfattr` can be used to dump all extended attributes in the `user` namespace by default, all other namespaces need to be fetched explicitly:

```
1 > sudo getfattr -d yes
2   # file: yes
3   user.testing="hello world"
4   user.yes="aslkdfj"
```

Setting a value for a `user` namespace attribute, we can set any namespace apart from the `system` which is only kernel-accessible:

```
> setfattr -n user.checksum -v "3baf9ebce4c664ca8d9e5f6314fb47fb" ↙
    ↳ foo.txt

> getfattr -d foo.txt

  # file: foo.txt
  user.checksum="3baf9ebce4c664ca8d9e5f6314fb47fb"

> setfattr -x user.checksum foo.txt
```

Additional special attributes can be also found on a per-file-system basis, file system attributes. These are distinct from extended attributes and are set as flags with meanings related to file change, readability, and mutability.

For instance, on Linux there is the `lsattr(1)` and `chattr(1)` commands to list and change file system attributes part of ext2/3/4 file systems. It offers features such as making a file append-only, immutable, securely deleted (zeroed out), undeletable, etc.. Most of these features are only allowed by the superuser or through POSIX capabilities (ex: `CAP_LINUX_IMMUTABLE`, inode modification).
Similarly on macOS and most BSDs, the commands `chflags` is used to change the attributes and `ls` to list them (with new flags). This can also be used to make the system append-only and immutable.

Lastly, the idea of extra attributes can also be found on the network such as with the CIPSO labels on packets of Trusted Solaris computers. It modifies the IP option of outgoing packets, which is used upon the receiving end for extra checks.

What you need to remember: *POSIX information labeling is about adding metadata to files, the draft also focused on operations applied to labels. However, no useful implementations have gone this way and instead went with extended file attributes, which are extra list of strings attached to files. The support depends on the file system and OS. On Linux these are separated by namespaces and used for some access control features like ACL and POSIX capabilities. Additionally, there are other sorts of metadata such as file system attributes (file immutability, undeletable, etc..) and packet attributes.*

Mandatory Access Control on BSD

The work on TrustedBSD's MAC framework was proposed in 1999 and started being merged into FreeBSD 5.0 in 2003 as an experimental feature. Later, it got included by default as a production-ready feature in FreeBSD 8.0 in 2009. FreeBSD, macOS, and the systems based on them, are the only BSDs having such feature.

MAC policies are loadable kernel modules that implement well-defined kernel programming interfaces (KPI). Simply said, the modules define functions that will augment the access control decisions by relying on the concept of subject and object labels we've seen.

These policies don't override the POSIX basic permissions and super-user checks, but consist of checks done afterward.

To enable MAC, the kernel should be compiled with the `options MAC` to allow dynamically loading appropriate modules either using `kldload mac_<name>` or to set them in a configuration file that will load them during boot such as `/boot/loader.conf`. Furthermore, the system can possibly allow stacking multiple label policies at the same time but it needs to be enabled either dynamically in single-user mode using `tunefs -l enable filesystem` or by adding the `multilabel` flag during mount in `/etc/fstab` (or when creating a new file system).
Nonetheless, a single label system is easier to manage than a multi label one. Yet, not all MAC modules rely on labels, and thus these non-label modules can co-exist in a single label system.

MAC labels are arbitrary formatted data that is interpreted and given meaning by the policy, it lives on system subject and system objects. The policy can be enforced on different parts of the system such as sockets, file system, pipes, processes, virtual-memory, etc.. Each system object has its own method of setting the label on it, either through new commands or through additions to pre-existing commands used to manipulate such objects.

- File system object: `setfmac(8)` and `setfsmac(8)`, `getfmac(8)`
- Network interface: `ifconfig(8)` through the `maclabel` parameter
- TTY (by login class): `login.conf(5)`
- User (by login class): `login.conf(5)`

Notice the login class capability database file we've mentioned in a previous section, it can also be used to set labels.

Additionally, the `su(1)` (by changing class name) and `setpmac(8)`/`getpmac(8)`, set and get process MAC labels, utilities let users run commands with a different process label than the current one, in the same mindset that `setpriv` and `ppriv` did for POSIX capabilities on Linux and SunOS derivatives.

While labels are theoretically arbitrary, they are mostly used to create levels/grades and compartments of subject-objects, the dominance relationship. Some policies have predefined labels such as `low`, `equal`, and `high`, but more generically numeric labels are used to precisely say which level dominates another. With these label policies, users are usually assigned a default/effective starting level and a range (minimum, maximum) or a set of levels/grades that they can access by switching to them using `setpmac`. The previous is called hierarchical labels, but they can be accompanied with compartments or non-hierarchical labels, which is the equivalent of groups for MAC labels, they are used to give access to generic features in a system and not a particular level of access.
The actual syntax to set the above might differ from module to module.

To set a label in the capability database, `/etc/login.conf`, we use the syntax `module_name/<labels>`. Don't forget that after any change to this file `cap_mkdb` needs to be run. For example:

```
1 default:\
2         :label=partition/13,mls/5,biba/10(5-15),lomac/10[2]:
3
4 example_user:\
5         :label=biba/10:2+3+6(5:2+3-20:2+3+4+5+6):
```

The Biba module syntax is interpreted as follows:

```
[labeltype]/[effectivegrade]:[effectivecompartments](
    [lowgrade]:[lowcompartments]-[highgrade]:[highcompartments]
)
```

Thus, the first default policy in the above example tells the Biba policy that a process's minimum integrity is 5, its maximum is 15, and the default effective label is 10. The process will run at 10 until it chooses to change label, perhaps due to the user using `setpmac`, which will be constrained by Biba to the configured range.

Let's see a couple of interesting modules available on TrustedBSD/FreeBSD. Keep in mind that all of this requires a lot of planning from administrators, a deep understanding of what the policy of the modules imply, and how to set the labels on the system. This requires multiple trials as this can also possibly lock the super-user account.

The `mac_none` and `mac_stub` modules have no effect, one completely empty and the other filled with no-op.

The `mac_seeotheruids` is a module controlling whether a user can see other users' processes and sockets. It doesn't require any label and is instead configured through `sysctl` tunables. It is similar to the tunables `security.bsd.see_other_uids` and `security.bsd.see_other_gids` but is more extensible. For instance, the module is enabled/disabled through `security.mac.seeotheruids.enabled` (automatically set to 1 on module load) and has options to see processes in the same primary group `security.mac.seeotheruids.primarygroup_enabled` and have whitelist groups `security.mac.seeotheruids.specificgid`.

Analogously, the `mac_partition` allows splitting processes into partitions which can only see other processes within the same partition. This policy is based on MAC labels and has the form: `patition/value`, in which the value can be either a number or `none`. The policy can be enabled or disabled using the tunable `security.mac.partition.enabled`.

Example in `/etc/login.conf`:

```
vnm:\
       :label=partition/10:
```

Then reducing what a process can see:

```
As user vnm
> getpmac
partition/10
> ps ZU root
LABEL PID TT STAT TIME COMMAND
partition/10 3452 p0 S+ 0:00.08 systat

As root
> sysctl -w security.mac.seeotheruids.enabled=1

As vnm
> ps ZU root
LABEL PID TT STAT TIME COMMAND
```

Notice the `Z` option to display the label, which can also be used with `ls(1)`.

Two other simple modules not relying on labels are the `mac_ifoff` and `mac_portacl`. The first is used to create a silence policy for network interfaces and the second to create access control lists for port range usage.

`mac_ifoff` works by relying on `sysctl` tunables to enable and disable network interfaces through `security.mac.ifoff.<interface_name>_enabled`. For instance: `security.mac.ifoff.lo_enabled=0` will disable the loopback interface.

Meanwhile, `mac_portacl` also uses `sysctl` tunables to set the high port `security.mac.portacl.port_high` and rules for who can bind to local TCP and UDP ports. The rules are set in `security.mac.portacl.rules` and is a series of comma separated `idtype:id:protocol:port`, where `idtype` is either `uid` or `gid` and it defines the following `id` parameter. The protocol is either `tcp` or `udp`. The module also has special tunables to allow super user or enable automatic port allocation.

The `mac_biba` module is the implementation of the Biba model which we've see in the models section. A similar module is named `mac_lomac` with the exception that it permits access by a higher integrity subject to a lower integrity object by temporarily downgrading the integrity level of the subject. The "no read down, no write up" being respected.

The `mac_mls` module is the implementation of the Bell-LaPadula model, the "no write down, no read up", reversing the dominance rules.

Here's a practical example:

```
We start with a user called vnm with biba policy
The effective level, along with min-max are all "low"
> whoami
vnm
> getpmac
biba/low(low-low)

The root user, effectively "high" level but able to access low-high
Creates two files, both effectively "high" level
Keep in mind that MAC is done after usual POSIX permissions
> whoami
root
> getpmac
biba/high(low-high)
> touch /home/vnm/test
> touch /home/vnm/test2
> chown vnm:users /home/vnm/test
> ls -lZ
-rw-r--r-- 1 vnm users biba/high 0 Jan 31 09:59 test
-rw-r--r-- 1 root users biba/high 0 Jan 31 09:59 test2

Now as user "vnm" we can read-up but not write-up.
Even when we own the file.
> cat test
> echo test > test
test: Permission denied.
```

```
> rm test
override rw-r--r-- vnm/users for test? y
rm: test: Permission denied

Meanwhile, as root user when lowering the level of a file
we can then not read-down but only write down.
> setfmac biba/low /home/vnm/test
> cat /home/vnm/test
cat: /home/vnm/test: Permission denied

Back as user "vnm" the file now being "low".
We can then write to it, yet not erase it.
> cat test
> echo testing > test
> rm test
rm: test: Permission denied

As root we set it to "equal" policy.
We're allowed to read it.
> setfmac biba/equal /home/vnm/test*
> cat /home/vnm/test1/test
testing

As normal user we can now manipulate everything.
But remember that MAC doesn't override POSIX DAC permissions.
> cat test
testing
> echo testingagain >> test
> echo testing > test2
test2: Permission denied.
> rm test
rm: test: Permission denied

Yet, we still can't remove the file, and that is because
removing a file is about modifying the parent directory!
The home dir was set as "high" level.
Let's change this as root:
> setfmac -R /home/vnm biba/low
> rm test

However, now that it's "low", root cannot read under its level:
> cd /home/vnm
/home/vnm: Permission denied

A way to solve this would be to use numerical levels or set the dir
as "equal", or let the root user call `setpmac` to lower its ↙
    ↳ grade/level.
```

This advanced Biba scenario displays how complex the flow-policies can actually become, the idea of information only going one way can quickly become a headache and requires a lot of pre-planning.

A more pleasant module, which isn't based on a flow-policy but on firewall-like rules, or path-based as we called it, is the `mac_bsdextended` module. Rules are entered through the `ugidfw(8)` utility, which has a syntax similar to firewall rules in `ipfw(8)`, but instead sets access for subjects to different objects on the system.

The rules allow subjects to access "modes" on "types" of objects. The "types are the following:

- `a` any file type
- `r` a regular file
- `d` a directory
- `b` a block special device
- `c` a character special device
- `l` a symbolic link
- `s` a UNIX domain socket
- `p` a named pipe (FIFO)

And the "modes" are the following:

- `a` administrative operations
- `r` read access
- `s` access to file attributes
- `w` write access
- `x` execute access
- `n` none

We can list the current rules using `ugidfw list`.
The rule syntax is extensive and can be found in the `ugidfw(8)` man page. Rules are checked in order, so there's a possibility to specify this with the `set` sub-command. The generic syntax goes like this:

```
add subject ... object ... mode arswxn
```

The subject and object defined as:

```
subject [not] [[!] uid uid | minuid:maxuid] [[!] gid gid |
mingid:maxgid] [[!] jailid jailid]

object [not] [[!] uid uid | minuid:maxuid] [[!] gid gid |
mingid:maxgid] [[!] filesys path] [[!] suid] [[!]
sgid] [[!] uid_of_subject] [[!] gid_of_subject]
[[!] type ardbclsp]
```

Here's an example to make more sense of this:

```
> ugidfw add subject uid 1002 object ! filesys /home type rd mode n
> ugidfw add subject uid 1002 object filesys /usr type rd mode rxs
```

The above only allows full access to the home directory, disallowing access to anything outside `/home`. Afterwards, read-execute and attribute access is allowed to files and directories within `/usr/` so that the user can issue basic commands.

To facilitate management, especially when installing services, `mac_bsdextended` also comes with default rules stored in `/etc/rc.bsdextended`. Indeed, if a user/subject isn't mentioned in any rule, it will have access to nothing.

One last module, which we won't describe here, is the `SEBSD` module, which can't be dynamically loaded and only set in `/boot/loader.conf` as `sebsd_load="YES"`. It is an experimental module that implements FLASK/SELinux, which we'll dive into in the next section, thus will skip the inner workings here.
Let's just say that it relies on SELinux reference policy and stores them in `/etc/security/sebsd/targeted/src/policy` and installs the compiled version in `/etc/security/sebsd/targeted/policy/policy.20` by default. The `/usr/sbin/load_policy` command needs to be used when modifying the policy.

Finally, let's think about some scenarios in which we can combine different MAC modules. The most intuitive way to do this is to mix specific non-label-based policies with ones that use labels.
The easiest approach for example is to mix `mac_seeotheruids`, `mac_portacl`, and `mac_bsdextended`, creating a system in which the admin has full control over which user does what and what they see of the rest of the system. In general, it is hard to pick the right choice, and a case-by-case study of the system is needed.

What you need to remember: *TrustedBSD/FreeBSD includes a MAC framework allowing to dynamically load modules that will be used to check access rights after the POSIX basic permissions. Some rely on labels, which can be set in the capability database `/etc/login.conf` in the `label=` parameter, while others don't. There are multiple modules ranging from network interfaces and port restrictions (`mac_portacl`, `mac_ifoff`), to system visibility (`mac_seeotheruids`, `mac_partition`), flow-policy models (`mac_biba`, `mac_mls`), and path-based access rules (`mac_bsdextended`), and more. Some of the policies can be combined.*

Mandatory Access Control on Linux

Linux Security Module Interface

In the 1990s researchers at the USA's National Security Agency released multiple papers on operating system security architecture. One in particular was about FLASK, part of the Fluke OS project, appearing in USENIX Security Symposium in August 1999. It was inspired and extended some earlier attempts such as the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula. Later, this research project morphed into a practical application: a patch to the Linux kernel implementing a mandatory access control architecture under the codename SELinux.

The team proposed a merge into the kernel mainline but it was refused because it would have tied the kernel to a specific security model. Instead, Crispin Cowan proposed a better solution: a generic interface that would allow hooks to loadable module enforcing access control. This project idea was merged in August 2003 and got the name LSM, Linux Security Module, which is designed to answer all the requirements to implement all sorts of security modules with the fewest changes in the kernel.

Internally, it is akin to TrustedBSD/FreeBSD's approach, kernel modules that implement security checks. Like the above, they also need to be built into the kernel at compile time through configurations such as `CONFIG_DEFAULT_SECURITY_<MODULE>` (for example `CONFIG_DEFAULT_SECURITY_APPARMOR`).
These modules can be stacked, the checks being done one after the other, sequentially until one module allows the action. The order list of stacked modules is hard-coded in the kernel parameter `CONFIG_LSM` and overridden via a boot-time parameter `lsm`, previously called `security`. Afterward, when the system is online, the modules can be registered and unregistered.
However, one hindrance of this stacking mechanism is that the chaining should be willingly given from the previous module. That means that modules are responsible to forward the decision request and can choose not to do so if it breaks their security model.

Each module, as we'll see, has a comprehensive security policy, however not all of them are concerned with MAC (for instance Landlock, which we'll see in the isolation section, bpf, and lockdown). Here's a couple of them: SELinux, Smack, Tomoyo, AppArmor.

Torvalds also suggested migrating the POSIX capabilities code into an LSM. Thus, what we've seen previous in the POSIX capabilities section is, under the hood, an LSM.
This particular module is always loaded by default, and will always be the first one checked in the ordered sequence of modules. Indeed, the list of LSMs can be found in the pseudo-fs under `/sys/kernel/security/lsm`.

Example:

```
> cat /sys/kernel/security/lsm
capability,landlock,lockdown,yama,bpf
```

There exist alternatives to this approach that were created independently as patches to the Linux kernel, such as Grsecurity, Medusa, and RSBAC. This last one being a contending framework also allowing modular extension. We'll see an example of an RSBAC module in the RBAC on Linux using RSBAC Framework section.

What you need to remember: *Over the years, many papers were released about security frameworks, this eventually lead to the acceptance of the LSM, Linux Security Framework. It is a modular framework that allows modules to implement security features as hooks. POSIX capabilities are the first of such modules in the stack and can't be overridden. There are other less mainstream security frameworks, such as GrSecurity and RSBAC, that exist as patch outside the main kernel branch.*

SELinux As we mentioned, FLASK, the Flux Advanced Security Kernel, was an implementation by a collaboration of the NSA, SCC, and the University of Utah based on theoretical proofs of the properties and characteristics of the architecture of secure access control that was applied to a research operating system called Fluke.

SELinux, Security-Enhanced Linux, is a port of the concepts of FLASK unto the Linux kernel, bringing the idea to a mainstream OS. The architecture supports ways to enforce different mandatory access control policies, such as those based on type enforcement, multi-categories security (MCS), also allows implementing role-based access control (RBAC) and others.

Type enforcement (TE), is an access clearance mechanism based on rules attached to security context defined within a domain. In short, a security context is a bunch of categorized extended attributes, aka labels, that have meanings and are checked for access control. Labels are also what is is used to implement MLS, MCS, and RBAC, what differs is the way in which they are used. In essence, SELinux is thus a hybrid system, mixing different concepts.

Like all MAC, there needs to be a system-wide policy, however, keep in mind that the SELinux permission check happens after the usual POSIX basic permissions, like all LSM. If the regular permission system disallows an activity, then SELinux is not even consulted.

The SELinux labels, also called context, are grouped into a hierarchy of three to four levels, each a subset of the other. The context used to identify resources is kept at all time on both subjects and objects in the system.

The parts composing a context are the following:

- Username, usually the same as the real username found in the password file
- Role, a grouping mechanism
- Domain/Type, another mid-level grouping mechanism
- Sensitivity, another high-level grouping mechanism

Using these categories of context is how different access control theories are put in place. Practically, the context takes the form of a colon separated string, each part representing, in order, the username ending in `_u`, role ending in `_r`, domain/type ending in `_t`, and sensitivity starting with `s`.

All core utilities are augmented with a `Z` flag to display the context of processes and files. For instance `ls -Z`, `ps -Z`, `netstat -Z`, etc..

```
$ ls -lhZ
 dr-xr-xr-x.   6 root root system_u:object_r:boot_t:s0       5.0K ↙
     ↳ Jan 27 08:41 boot/
 drwxr-xr-x.  22 root root system_u:object_r:device_t:s0     4.1K ↙
     ↳ Feb  6 14:01 dev/
 drwxr-xr-x.   1 root root system_u:object_r:etc_t:s0        5.5K ↙
     ↳ Feb  6 14:01 etc/
 drwxr-xr-x.   1 root root system_u:object_r:home_root_t:s0    48 ↙
     ↳ Jul 14  2016 home/
 dr-xr-x---.   1 root root system_u:object_r:admin_home_t:s0  354 ↙
     ↳ Jan 30 19:37 root/
 drwxrwxrwt.  14 root root system_u:object_r:tmp_t:s0         300 ↙
     ↳ Feb  6 14:38 tmp/
 drwxr-xr-x.   1 root root system_u:object_r:usr_t:s0         174 ↙
     ↳ Nov 16 20:58 usr/

$ ps -eZ
 LABEL                             PID TTY        TIME CMD
 system_u:system_r:init_t:s0         1 ?     00:00:05 systemd
 system_u:system_r:kernel_t:s0       2 ?     00:00:00 kthreadd
 system_u:system_r:syslogd_t:s0    655 ?     00:00:05 systemd-journal
 system_u:system_r:policykit_t:s0 1155 ?     00:00:36 polkitd
```

The logical step here is to have of a mapping between the subject `user_u:user_r:user_t` context and a target file `object_u:object_r:object_t` context, stored as a system-wide access control policies so that SELinux can enforce them, which is exactly what is happening.
Depending on which policies are put in place, using which particular context, the access control methodology mindset differs.

One could choose to only rely on the domain/type context, and in that case we'd call such policy a type enforcement. It is one that uses an "access vector" containing on one side the source context as a type, such as `user_t`, and the target context, such as `lib_t`, along with the activity invoked on which class of object, such as "execute" on "file".

For instance, this is a policy allowing "execute" permission for users assigned the `user_t` type to objects assigned the type `lib_t` which are files.

```
allow user_t lib_t : file { execute };
```

The classes of objects, with the activities possible on each, are predefined by SELinux and can be found in the `/sys` pseudo-fs.

Here are the classes found on a system:

```
> ls /sys/fs/selinux/class


appletalk_socket  db_procedure
association       db_schema
blk_file          db_sequence
capability        db_table
capability2       db_tuple
chr_file          dbus
context           db_view
db_blob           dccp_socket
db_column         dir
db_database       fd
db_language       fifo_file

file              netlink_audit_socket
filesystem        netlink_dnrt_socket
ipc               netlink_firewall_socket
kernel_service    netlink_ip6fw_socket
key               netlink_kobject_uevent_socket
key_socket        netlink_nflog_socket
lnk_file          netlink_route_socket
memprotect        netlink_selinux_socket
msg               netlink_socket
msgq              netlink_tcpdiag_socket
netif             netlink_xfrm_socket

node              socket
nscd              sock_file
packet            system
```

```
packet_socket      tcp_socket
passwd             tun_socket
peer               udp_socket
process            unix_dgram_socket
rawip_socket       unix_stream_socket
security           x_application_data
sem                x_client
shm                x_colormap

x_cursor           x_screen
x_device           x_selection
x_drawable         x_server
x_event            x_synthetic_event
x_extension
x_font
x_gc
x_keyboard
x_pointer
x_property
x_resource
```

Each class has its set of privileges, for instance on the “file” we can do the following:

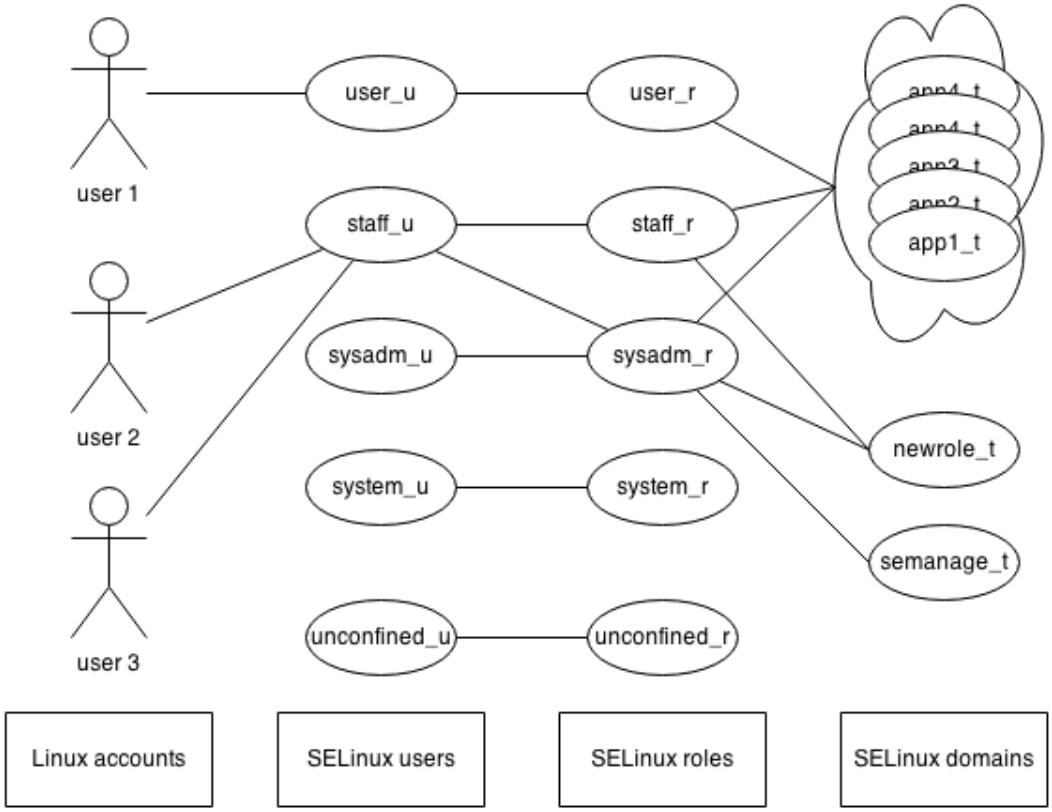
```
> ls /sys/fs/selinux/class/file/perms/
append       execmod            getattr
create       execute            ioctl
entrypoint   execute_no_trans   link
lock         quotaon            relabelto
mounton      read               rename
open         relabelfrom        setattr
swapon
unlink
write
```

Meanwhile, the supported permissions for a TCP socket are:

```
root #ls /sys/fs/selinux/class/tcp_socket/perms/
accept         bind        create        ioctl
acceptfrom     connect     getattr       listen
append         connectto   getopt        lock

name_bind      node_bind   recv_msg      send_msg
name_connect   read        relabelfrom   sendto
newconn        recvfrom    relabelto     setattr

setopt
shutdown
write
```

These create a lot of possibilities for access vectors just by relying on the “type” context.

Yet, one could use another approach and instead of creating a vector of type-to-type, could instead rely on the roles users are assigned, since all subjects have it in their context. Thus, SELinux policy would act as a role-based access control mechanism (RBAC), mapping roles to domains/types.



Still, this isn't a true RBAC since in a real RBAC users are only granted permissions through roles, and there aren't any restrictions in SELinux to limit the policy to only this. Additionally, users should be explicitly granted roles and otherwise will have no rights. Furthermore, unprivileged users only have access to a single role.

For instance, here's a utility listing the `user_r` role having access to which types. Do not worry about the commands and policy storage yet, we'll get back to them later, for now just keep in mind that we can get info, set the context to users, processes and files, and more.

```
> seinfo -ruser_r -x

  user_r
    Types:
      git_session_t
      httpd_user_script_t
      ...
```

Still further, there's another generic concept that SELinux calls User-Base Access Control (UBAC) which consists of creating the access vector by relying on the user part of the context. Since it overrides the basic POSIX DAC, this can be used for fine-grained permission, similar to ACL but as a MAC.

The SELinux user is immutable in the context, it is assigned at login through a fixed mapping between login name and SELinux user, deciding what the user has accessed to on the system.

For instance, the following `semanage` command lists the mapping, showing a default fallback user `user_u`:

```
> semanage login -l

Login Name          SELinux User

__default__         user_u
swift               staff_u
root                root
```

We can also list which roles are assigned to which users:

```
> semanage user -l

SELinux User        SELinux Roles

root                staff_r sysadm_r
staff_u             staff_r sysadm_r
sysadm_u            sysadm_r
system_u            system_r
user_u              user_r
```

Or through `seinfo` for a specific user:

```
root #seinfo -ustaff_u -x

  staff_u
    roles:
      staff_r
      sysadm_r
```

So far we've seen that subjects and objects are assigned a context which is divided in sub-parts, the user which is mapped from the login name, the domain/type assigned by what the "thing is", the roles which users are assigned to. When creating policies we can use any of these, creating a vector deciding what context criteria are needed to perform an action on a class of objects. Before seeing how to create the policies, the commands and management operations, let's see the last method of assigning permission: sensitivity.

The sensitivity is the fourth field in the SELinux context, it's a way to implement a flow-based policy with clearance levels. However, it isn't neither Bell-LaPadula nor Biba, once the clearance check is done everything is allowed.

The sensitivity field is split into security levels and categories. A subject gets clearance to an object if it belongs to the same categories and also has access to the security level the object is in.

Visually, the sensitivity is a string that separates the security level from the category by a colon. A dash `-` means a range in the security level, while a comma `,` means distinct values in the categories and a dot `.` means a range in the categories. The security levels start with `s` and the categories with a `c`.

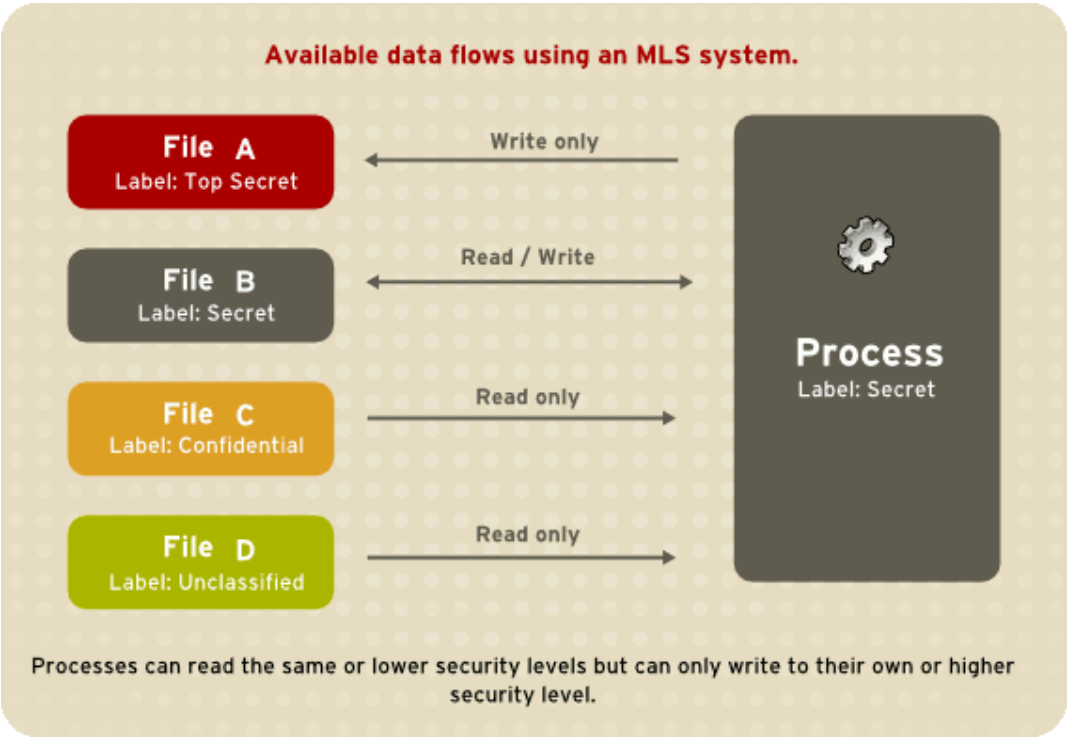
For instance:

```
s0-s5:c0,c4.c8
```

Means that this clearance runs in security level `s0` and is allowed to access resources with sensitivity up to `s5` and the categories needs to be `c0` and `c4` to `c8`. If the resource isn't part of the categories mentioned, then it is not part of the clearance and will not be accessible.

This is sort of equivalent to the compartments on TrustedBSD/FreeBSD MAC, however under SELinux, having this field is often called multi-categories security (MCS). Keep in mind that MCS isn't a subset of MLS, after a clearance dominates a file it gets the access that was explicitly defined.

SELinux acts as MLS when the category part is missing, but unlike Bell-LaPadula it allows users to read files at their own sensitivity level and lower, but can write only at exactly their own level (write-up isn't respected).



So far, SELinux sounds nice, the theoretical aspect should be a piece of cake by now. But we're missing a big part of the puzzle: how to create the policies, how to apply the context on files and subjects, and how to manage them.
Unfortunately, this is where the complexity of SELinux starts to appear. Writing a policy from scratch

is such a hassle that we have to heavily rely on tools to achieve this. Furthermore, since it is so advanced, a standard reference policy project exists upon which all distributions base policies are extended from.

An SELinux installation comes with multiple parts: A modified kernel with SELinux LSM, an SELinux library `libselinux` for API functions, command line tools, and configuration files.
The `libselinux` library is used by SELinux-aware applications that internally interpret security context. For instance, D-Bus packets can be labeled with the originator's context to decide whether they have access to a functionality.

The configuration files that exists are the following:

- Global SELinux configuration in `/etc/selinux/config`, `/etc/selinux/semanage.conf`, `/etc/selinux/restoredcond.conf`, `/etc/sestatus.conf`, etc..
  These configurations are not specific to any policy and are common. In `/etc/selinux/config` we can find what mode SELinux is currently in, the `SELINUX` parameter can be: `enforcing`, `permissive`, or `disabled`. The `permissive` mode allows everything but warning logs are written so that we can debug policy rules, this is useful when modifying the system. In that same file we have the `SELINUXTYPE` which contains the name of the current directory under `/etc/selinux` where the active binary policy and its configuration files will be located. The available ones are `targeted`, only for network daemons, `strict`, the full SELinux protection, `mls` and `mcs`. It defaults to `strict`.
  The `/etc/selinux/semanage.conf` controls the utilities `semanage(8)` and `semodule(8)`.
  The `/etc/sestatus.conf` is used by `sestatus(8)` when the verbose (`-v`) flag is passed to display the context of extra files and processes that are listed.

```
> sestatus

SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             strict
Current mode:                   enforcing
Mode from config file:          enforcing
Policy MLS status:              disabled
Policy deny_unknown status:     denied
Max kernel policy version:      28
```

- Policy store configuration files
  In `/etc/selinux/<SELINUXTYPE>/modules` or `/var/lib/selinux/<multiple_policies>`. These files are the base policies used by commands such as `semanage(8)` and `semodule(8)` to build a live policy called the Policy Store.
  They are all part of the reference policy and categorized into different categories such as the MLS and MCS we've seen, facilitating labeling the system and enforcing policy.
  When switching between policy, a relabeling is needed and issued through `fixfiles -F ↙ ↳ onboot` for example to do it on the next boot.

- Policy configuration files, which are the live/active policy. Only one policy can be active at a time, it's picked by the `SELINUXTYPE` we've seen in the global configuration files.
  The policy configuration files exist in a binary format loaded in the kernel, initially based on

the reference policy. The policy store contains the rules in one of two language: kernel policy language or common intermediate language (CIL), which are then compiled to policy package format and loaded unto the kernel. The global policy can then be modified on the fly, by issuing commands such as `semanage`. The current binary policy loaded in the kernel is found under `/etc/selinux/<SELINUXTYPE>/policy/policy.<ver>`. Internally, SELinux keeps the rules in a cache called the Access-Vector Cache or AVC. There's also the possibility to dump from the compiled policy module format, policy package `*.pp`, back into CIL format through the `/usr/libexec/selinux/hll/pp` command.

- The SELinux peudo-fs under `/sys/fs/selinux`, which reflects the current state of SELinux active policy. Usually, this isn't read directly but through utilities such as `apol(1)` to see currently loaded policy.

It would be impossible to write all the rules of a policy, this is why SELinux ships with a labeling database as a reference policy, a sort of path-based rule database. It contains usual roles such as `user_r` assigned to normal user login, and `system_r` for daemons and system services, it has the concept of `unconfined_<x>` labels that bypasses policies, usual users such as `user_u`, `staff_u`, and `system_u`, and much more.
The documentation for the reference policy found in `/usr/share/doc/selinux-policy/html/index.html` describes all the default rules, tunables, interfaces, type enforcement and others. There also exists on some systems `/usr/share/doc/selinux-base-<version>`. A version can be found online here

Yet, we're still left wondering how to write the policy ourselves, we haven't seen what they even look like, only that somehow they come built-in.

Let's take a look at the kernel policy language, meanwhile, the CIL is described in depth in this reference guide, will be skipped in this article. The first thing to understand is that it is a full-fledge language, with a wide-range of possibilities. It can exist either in a monolithic file, containing all the policy source/code file or in a combination of base policy (reference policy) with module (non-base) policy both needing to be compiled using `checkmodule(8)` or `checkpolicy(8)` or specific helper Makefile.

The modules can be listed using `semodule -l`:

```
> semodule -l
alsa    1.11.4
apache  2.6.10
apm     1.11.4
application     1.2.0
...
```

These also live as part of the live policy in files as compiled package format found in the `/etc/selinux/<SELINUXTYPE>/modules/active/modules` subdirectory.

For already compiled modules, the loading and unloading of `.pp` files is done through `semodule -i <module_name>.pp` and `semodule -r <module_name>` respectively. Disabling is done with the `-d` flag.

The source files, before compilation, are a series of statements, declaring and associating context and their transitions (user, role, type, boolean tunable, etc..), conditional and optional policies (in case a

tunable is turned on), access vector rules (`allow`, `neverallow`, `dontaudit`, `auditallow`), constraints (a wider vector using multiple parts of the context), labeling (file system, network, etc..), and more.

For example, this assigns the `user_r` role to users `user_u`:

```
user user_u roles { user_r };
```

Or allowing permission for `setgid chown` and`fowner`within the same domain`staff_t`‘:

```
allow staff_t self:capability { setgid chown fowner };
```

This would be the same as the above:

```
allow staff_t staff_t:capability { setgid chown fowner };
```

The following allows `user_t` execute permission over `bin_t` and `user_bin_t` type/domain files.

```
allow user_t bin_t:file { execute };
allow user_t user_bin_t:file { execute };
```

To allow transition from one role to another we can use this syntax:

```
allow from_role_id to_role_id;
role_transition current_role_id type_id new_role_id;
```

Example to allow `sysadm_r` to switch to `unconfined_r` and run processes with role type/domain `unconfined_exec_t`:

```
allow sysadm_r unconfined_r;
role_transition sysadm_r unconfined_exec_t:process unconfined_r;
```

Example, if all users have `user_home_t` by defaut, then this allows users to access another user’s home if POSIX DAC already allows it:

```
allow user_t user_home_t:dir { read write execute close open ... };
allow user_t user_home_t:file { read write execute close open ... };
```

Let’s see how writing our own module goes, there’s two methods of doing it. We can either automatically generate some skeleton files based on an executable or write them from scratch.
In the kernel policy language, we write the policy in multiple files, a `.te` (type enforcement), and optionally a `.fc` (file context), `.if` (interface), and others. As the name indicates, the type enforcement files contains the rules, while the file context contains how the labels will be applied to the system files, and the interface files defines functions.

Let’s create a local policy that contains an allow rule.

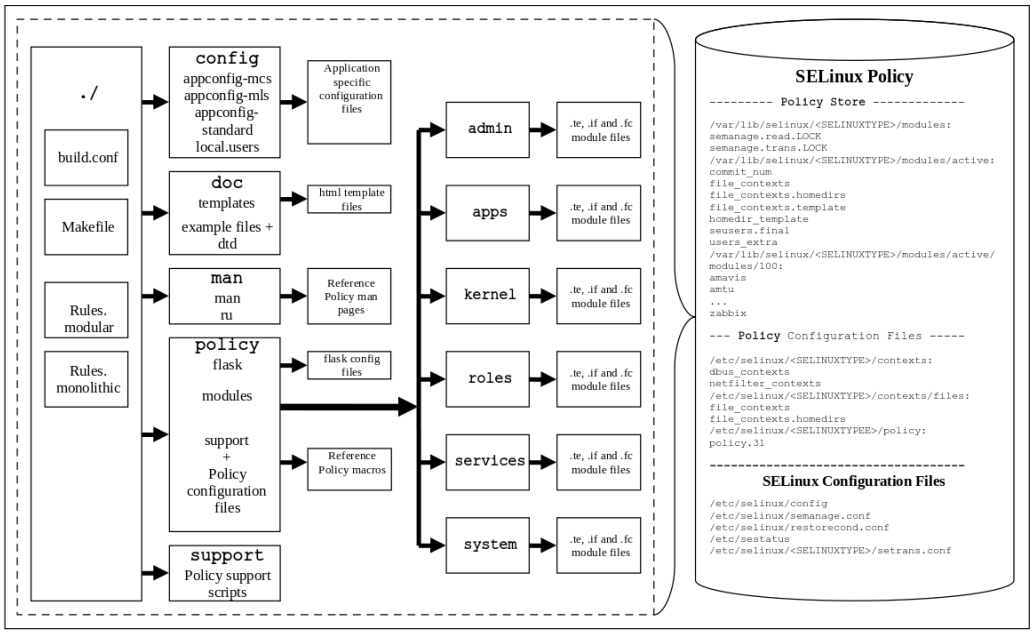
```
policy_module(my_new_module, 1.0)

gen_require(`
  type user_t;
  type var_log_t;
')

allow user_t var_log_t:dir { getattr search open read };
```

As you can note, the `gen_require` is a function/interface, it comes from the reference policy and allows to quickly define and require things. Here we are saying we require from other modules two types. There is a lot of reuse of interfaces defined by other modules but this can quickly get confusing. You can alway refer to the online reference policy documentation, here.

We can then compile this module using `selinux` custom compiler for kernel policy language into policy package format and load it using `semodule -i`:

```
> make -f /usr/share/selinux/strict/include/Makefile my_new_module.pp
> semodule -i my_new_module.pp
```

The location of the Makefile might differ, on some systems it is in `/usr/share/selinux/devel/Makefile`.



Another method is to rely on `sepolicy`, the SELinux policy inspection tool, to automatically generate the initial policy module template.

A simple example goes like this:

```
> sepolicy generate --init /usr/local/bin/mydaemon
Created the following files:
/home/example.user/mysepol/mydaemon.te # Type Enforcement file
/home/example.user/mysepol/mydaemon.if # Interface file
/home/example.user/mysepol/mydaemon.fc # File Contexts file
/home/example.user/mysepol/mydaemon_selinux.spec # Spec file
/home/example.user/mysepol/mydaemon.sh # Setup Script
```

The `mydaemon.sh` will both compile, load, and relabel the corresponding part of the file system. In the previous example, since we didn't have a `.fc` file we didn't have to relabel.

Relabeling can be done using `restorecon` or `fixfiles`. Keep in mind that if temporary changes are done using command line utility (which we'll see in a bit), these will be reverted back to the combined base/reference and module policy.

Example:

```
> restorecon /etc/resolv.conf
```

A simpler method to play with policy is offered by some systems such as Gentoo with the `selocal`. It allows to easily add or remove rules from the active policy as small incremental changes to a single module found in `~/.selocal` called `selocal`.

For instance, here's how to add a type enforcement policy:

```
> selocal --add "corenet_tcp_bind_generic_node(staff_t)"
> selocal --add "corenet_tcp_bind_generic_port(staff_t)"
> selocal --build --load
> selocal --list
12: corenet_tcp_bind_generic_node(staff_t)
13: corenet_tcp_bind_generic_port(staff_t)
> selocal --delete 13
```

This incremental approach is easier to manage than having to edit huge files, especially when things aren't working as expected. When starting with a policy it's good to know how to debug them without angrily being locked out of permissions. SELinux will log all access in auditd, which we'll see in a later section on logging and auditing, inspecting these logs will give us ideas why actions didn't work. For example with the `ausearch -m avc -ts recent` command (AVC: Access-Vector Cache of SELinux). Furthermore, there is also the `sealert -l "*"` which will give feedbacks on how to fix certain issues. A special tool called `audit2allow` takes logs from auditd `ausearch` and generates a policy that would remediate a permission deny issue and the `audit2why` explains the reason.

Since this can be annoying, running SELinux in the permissive mode will log to auditd while not enforcing the policy rules. The commands `getenforce` and `setenforce` are used to get and set this permissive mode. The permissive mode can also be applied specifically to a domain through the `semanage` command which will dynamically generate a new policy module and load it. For instance, setting permissive on `unconfined_t` domain/type will add a new module in: `/var/lib/selinux/<SELINUXTYPE>/active/modules/<order>/permissive_unconfined_t`.

```
semanage permissive -a unconfined_t
```

That's the main ideas about policy configuration, now we can take a look at the different utilities for SELinux administration.

We've seen the `sestatus` command, telling us the current SELinux state:

```
> sestatus

SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             strict
Current mode:                   enforcing
```

```
Mode from config file:          enforcing
Policy MLS status:              disabled
Policy deny_unknown status:     denied
Max kernel policy version:      28
```

The core-utilities have been enhanced, as we said, with the `-Z` flag, such as `id`, `ls`, `ps`, `netstat`. Another tool can be used to get similar details called `seinfo`. For example to get the context on a port `seinfo --portcon=80`. `seinfo` can be used to get descriptions of compiled policy, domain, users, etc..

```
> seinfo /etc/selinux/strict/policy/policy.24

Statistics for policy file: /etc/selinux/strict/policy/policy.24
Policy Version & Type: v.24 (binary, mls)

   Classes:            81    Permissions:       235
   Sensitivities:       1    Categories:       1024
   Types:            3508    Attributes:        277
   Users:               9    Roles:              12
   Booleans:          190    Cond. Expr.:       225
   Allow:          275791    Neverallow:          0
   Auditallow:         97    Dontaudit:      202153
   Type_trans:      24052    Type_change:        38
   Type_member:        48    Role allow:         20
   Role_trans:        292    Range_trans:      3995
   Constraints:        87    Validatetrans:       0
   Initial SIDs:       27    Fs_use:             22
   Genfscon:           81    Portcon:           426
   Netifcon:            0    Nodecon:             0
   Permissives:        59    Polcap:              2
```

We can rely on `sesearch(1)` to query the loaded policy, giving it fields in a rule.

```
> sesearch -s mozilla_t -t user_home_t -AC

Found 4 semantic av rules:
   allow application_domain_type user_home_t : file { getattr append ↙
       ↳ } ;
DT allow mozilla_t user_home_t : file { ioctl read getattr lock open ↙
    ↳ } ; [ mozilla_read_content ]
DT allow mozilla_t user_home_t : dir { ioctl read getattr lock ↙
    ↳ search open } ; [ mozilla_read_content ]
DT allow mozilla_t user_home_t : lnk_file { read getattr } ; [ ↙
    ↳ mozilla_read_content ]
```

We can get the raw extended attributes on files, the label binary format, similar to ACL and POSIX capabilities:

```
> getfattr -m . -d /etc/resolv.conf

security.evm=0sAoTjX3aOeDQdWxbOfOUV930tWoDA
```

```
security.ima=0sAYC508o0Lz4iAA9ucVAVsvK02tV/
security.selinux="system_u:object_r:net_conf_t:s0"
```

We mentioned that booleans can be defined and used in conditions of policy files for tunable run-time features. Their status can be queried using `getsebool` (`-a` for all), `semanage boolean -l`, or reading straight from the pseudo-fs in `/sys/fs/selinux/booleans`.

Example:

```
> semanage boolean -l | grep abrt_anon_write
abrt_anon_write  (off, off) Allow ABRT to modify public files
                            used for public file transfer services.
```

These booleans can be toggled using `semanage`, `setsebool`, or `togglesebool`. To set the change as persistent in the policy the `-P` option has to be passed to `setsebool`.
Combined with `sesearch` we can pass the `--bool` and `--show_cond` together to be sure of what will be influenced after a boolean is changed.

The `sepolicy` suite provides multiple features to query the installed SELinux policy. We've seen how it can be used to create templates, but it can also be used to query anything.
For example we can query booleans using `sepolicy`, like above using `sepolicy booleans -a`.

As for utilities that perform updates, we can temporarily change the security context of a file using `chcon`:

```
> chcon -t net_conf_t /etc/resolv.conf
```

Specifically we can change the security context using `chcat`, which can be accompanied with named values in `/etc/selinux/mcs/mcstrans.conf`:

```
> chcat +c12 metadata.xml
```

The `setfiles` utility is used when a file system is relabeled and the `restorecon` or `fixfiles` utility restores the default SELinux contexts, overriding temporary labels such as the ones set with `chcon`.

Similarly, we can temporarily change the security context of a user, if the transition is allowed, with `runcon` and `setcon`:

```
> id -Z
root:sysadm_r:sysadm_t:s0-s0:c0.c1023
> runcon -l s0-s0:c0.c10,c12 sh
>id -Z
root:sysadm_r:sysadm_t:s0-s0:c0.c10,c12
```

Additionally, there's a PAM module called `pam_selinux` to set the default security context when the session starts.

For permanent changes, the versatile tool to use is `semanage` with its countless subcommands.

We can use it to list the current file context of certain files:

```
> semanage fcontext -l | grep resolv

/etc/resolv\.conf.*                         regular file      ↙
    ↳ system_u:object_r:net_conf_t
/usr/libexec/polkit-resolve-exe-helper.*  regular file      ↙
    ↳ system_u:object_r:policykit_resolve_exec_t
```

And modify the ones of others, which will require a local-relabeling:

```
> semanage fcontext -a options file-name|directory-name
> semanage fcontext -a -t net_conf_t /etc/puppet-resolv\.conf
> restorecon -v file-name|directory-name
```

The SELinux users can also be listed, created and updated with `semanage`, as well as the login mappings.

The current mappings can be shown:

```
> semanage login -l

Login Name                SELinux User
__default__               user_u
root                      root
swift                     staff_u
system_u                  system_u
```

For instance, to map the Linux account “john” to the `staff_u` SELinux user:

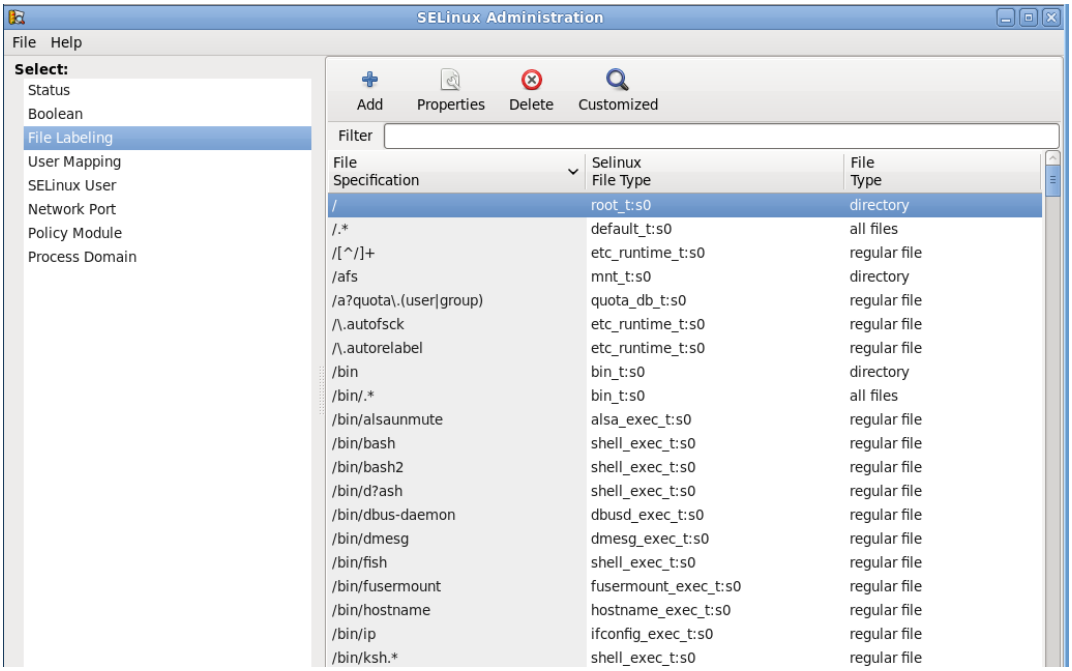
```
> semanage login -a -s staff_u john
```

Additional SELinux users can be created using `semanage` user, like so:

```
> semanage user -a -R "staff_r sysadm_r" myuser_u
```

Since the utilities can be confusing, there exists a few graphical tools to facilitate the management. The `sepolicy` suite mentioned above offers a GUI. There is an SELinux Manager that ships with RedHat, and others that comes with certain distributions such as a Policy Generation Tool.

Here’s a glimpse of what it looks like:



Before ending with an example of SELinux, let's mention that SEBSD module that is a port of SELinux FLASK and Type Enforcement to TrustedBSD/FreeBSD. It is an unmaintained project with a limited scope but shows that SELinux could live outside of Linux.

The last example we'll see is how to restrict root access with SELinux, not allowing the super-user to read a directory `/etc/private`.

We start with a sample policy defining the type/domain `etc_private_t` that can be associated with the files class (using the interface `fs_associate` from the reference policy, see it in the reference policy).

```
policy_module(myprivate, 1.0)

type etc_private_t;
fs_associate(etc_private_t)
```

Then we assign the type to the files and directory we want, however it's still not allowed, even as root, as no policy matches:

```
> chcon -t etc_private_t /etc/private
chcon: failed to change context of '/etc/private' to ↙
    ↳ 'system_u:object_r:etc_private_t:s0': Permission denied
```

We need to modify the policy to allow the `sysadm_t` domain/type to label files with `etc_private_t`.

```
allow sysadm_t etc_private_t:{dir file} relabelto;
```

As such, the super-user can relabel resources without being able to read them afterward. However, the super-user can still disable SELinux with `setenforce 0` and eventually read them.

A boolean in the reference policy (see) can prevent this. It will lock the current policy and any changes.

```
> setsebool secure_mode_policyload on
```

Yet this is only valid until reboot, and a super-user will probably still be able to find a way around this. This little puzzle, initially found here is a good exercise at our knowledge of SELinux.

Overall, SELinux theoretical concepts of labels is nice, however the practical application using compiled in-kernel policies with a full-fledge language is intimidating. There's a lot of tooling to manage the rules and search them, which facilitates the job but has so many redundant functionalities.

What you need to remember: *SELinux is a MAC that relies on labels called context. They include users (different than login name and mapped), roles, types/domains, security levels and categories. These context can independently be used to create different styles of policies, ranging from type enforcement, RBAC, MCS, and MLS. Policies are written in a textual format (kernel policy language or CIL) and consist of statements, interfaces/functions, tunables and booleans, conditions, and more. The statements associate context with permission over classes of objects on the system (defined in the pseudo-fs). The text policy needs to be compiled into policy package format* `.pp` *and loaded in the kernel to be applied, often also relabeling the system files. A base policy is always present called the reference policy and extended with modules. There exists a myriad of tools to check the current policy status, get the context of users/processes/files if permissible or not, search the policy (*`sesearch` *and* `sepolicy`*), check for potential security errors and how to fix them (*`audit2why` *and* `sealert`*), temporarily change the context of files and users (*`chcon`*,* `runcon`*, change the access-vector-cache), get/set tunables/booleans (*`setsebool`*), and manage the policy persistently (*`semanage`*).*

AppArmor AppArmor is a MAC relying on LSM, initially part of the defunct Immunix OS, and now maintained as part of SUSE. It was created as an easier-to-manage alternative to SELinux.

While SELinux is label-based, AppArmor is instead path-based. This means that AppArmor keeps track of a list of executable files that it needs to apply extra privilege checks on (after POSIX basic DAC permission checks) when invoked, these are called security profiles. When a process executes or tries to access a file listed in a profile, AppArmor will enforce appropriate behavior which can include the usual read-write-execute, resource usage limitation (`rlimit`), POSIX capabilities, and network access. This confinement can also apply to sub-process that the invoked file will spawn.

NB: AppArmor indirectly uses labels to keep track of files, but they are uninteresting to the end-user. Yet, one can check them by adding the `-Z` flag to standard core utilities. This is also found under `/proc/<pid>/attr/current`

This mindset is distinct from SELinux which disallows everything by default. With AppArmor, everything is under DAC until the path is explicitly mentioned in a security profile. Furthermore, the text-based rules stored on the file system use a readable language format, allowing newly installed services to set and enforce their own profiles without hassle. Similar to SELinux, these are compiled and loaded unto the kernel and cached.
On that note, AppArmor also comes with a number of default policies, however there aren't as many as SELinux and are meant as examples more than useful policies.

AppArmor can be used to implement a version of role-based access control (RBAC) using a feature called “hats”, which are subset profiles an application can switch to.

Let’s see how to define and load profiles, their syntax and their inner-working, then look at how to easily create profiles using the special “learning mode”, afterward we can have a glance at the “hat” feature for RBAC, and finally list a couple of utilities that are used to administer an AppArmor system.

The AppArmor tools configurations live in `/etc/apparmor` or locally in `${HOME}/.apparmor/`. Meanwhile the security profiles live under `/etc/apparmor.d` or locally under `${HOME}/.apparmor.d/`.
The profiles directory unsurprisingly contains all the enabled profiles but also has multiple sub-directories such as `abstractions` containing usable helpers that can be included in profiles, `cache` containing the currently binary cached profiles in the kernel, `disable` for a list of symlinks to profile that can be enabled in the parent directory, `namespaces` for sub-profiles, `pam` for PAM specific configurations, `tunables` for variables and aliases that can be used across profiles, and `local` that contains override to distributed profiles (by the package manager).
All the files directly present under the profiles directory (not sub-directories) will be loaded. There is no restrictions on the file name, however, by convention the name should relate to the profile it contains. For example a profile with rules pertaining to `/usr/bin/passwd` will be named `usr.bin.passwd`. In theory, all rules could be contained in a single profile, but that would be harder to manage.

A profile file is composed of some preamble, consisting of variables and aliases definition, inclusion of other files, conditional rules, followed by a series of profiles.
A profile begins with a name, describing to what executable it applies (optionally starting with the word `profile`), followed by optional flags, and opening `{` and closing brackets `}` enclosing the rules that will be enforced. For instance:

```
/usr/bin/ {
   # profile contents
}

profile user1 {
   # profile contents
}
```

The profile names can either be a file path, allowing globbing characters, or simple name. If a profile doesn’t refer to a file, which we call an unattached profile, then it will need to be referred explicitly to be used by another profile to actually be enforced.

A profile can include other files using the `include` directive, usually the files found in the `abstractions` directory are often included, along with the aliases in `tunables`.

Within the profile section different things can be defined: - Rules applying to the current running process executable such as file permission, execution permission, resource limits, network rules/ipc/D-Bus, capabilities. - Conditions, aliases, variables. - Inclusion of other profiles. (ex: `include ↙ ↳ <abstractions/base>`) - A child profile, when another executable is called. - A “hat”, a special child profile, starting with the `^` character, allowing reverting back to the parent.

Capabilities are defined using the `capability` keyword, as such:

```
/profile {
   capability sys_nice,
```

```
3     capability setgid,
4 }
```

The file rules consist of path name along with a permission set, the order in which the permission or path appear is irrelevant:

```
1 /profile {
2     /path/to/file  rw,   # file rule beginning with a pathname ↙
          ↳ (convention)
3     rw /path/to/file2,   # file rule beginning with permissions
4     /path/to/file3       # file rule split over multiple lines
5          rw,
6 }
```

Apart from the usual `rwx`, the permission also includes `a` for append, `m` for memory map executable, `k` for lock, `l` for link, and sub-categories of the executable permissions:

- `ux` - Execute unconfined (preserve environment) – WARNING: should only be used in very special cases
- `Ux` - Execute unconfined (scrub the environment)
- `px` - Execute under a specific profile (preserve the environment) – WARNING: should only be used in special cases
- `Px` - Execute under a specific profile (scrub the environment)
- `pix` - as px but fallback to inheriting the current profile if the target profile is not found
- `Pix` - as Px but fallback to inheriting the current profile if the target profile is not found
- `pux` - as px but fallback to executing unconfined if the target profile is not found
- `Pux` - as Px but fallback to executing unconfined if the target profile is not found
- `ix` - Execute and inherit the current profile
- `cx` - Execute and transition to a child profile (preserve the environment)
- `Cx` - Execute and transition to a child profile (scrub the environment)
- `cix` - as cx but fallback to inheriting the current profile if the target profile is not found
- `Cix` - as Cx but fallback to inheriting the current profile if the target profile is not found
- `cux` - as cx but fallback to executing unconfined if the target profile is not found
- `Cux` - as Cx but fallback to executing unconfined if the target profile is not found

Aliases are used to merge multiple path together or give them names:

```
1 alias /home/ -> /mnt/users/
```

In the above example when `/home/` is mentioned it will expand to `/mnt/users/` instead.

One thing that can be added to rules is a transition to another profile, this is done by adding an arrow `->` with the name of the new profile. Example:

```
1 /usr/bin/mutt {
2   ..
3   /bin/** px -> shared_profile,
4
5   /usr/*bash cx -> /bin/bash,
6   profile /bin/bash {
7     ..
```

```
  }
}
```

After any manual change to the profiles, the files need to be compiled. This is either done by restarting AppArmor service (it lives as a daemon that manages the kernel module), or by issuing the command `apparmor_parser`:

```
apparmor_parser -r /etc/apparmor.d/<changed_service>
```

We aren't going to dive into the gritty details of profile language syntax (it can be found here), instead let's move to a better approach.

As with SELinux, it can be tough to write your own rules, and similarly we have tools to help us find why our profiles aren't working as expected. On SELinux this was done through auditd logs along with a permissive mode that would allow everything but keeps warning logs.
On AppArmor it's similar, all actions are logged in auditd, analyzed with tools such as `aa-logprof` to scan the audit logs and interactively suggest updates to profiles, and the permissive mode called `complain` can be enabled on a per-profile basis. It can be set as an optional flag, or when moving the profile file to the `force-complain` sub-profile-directory under `/etc/apparmor.d`, or by loading the profile manually using `-C` argument of `apparmor_parser`, or even by dynamically changing it using `aa-complain` script. For instance:

```
/bin/foobash flags=(complain) {
 ...
}
```

A command called `aa-notify` can also be used to display desktop notification whenever it encounters logs for AppArmor access denied messages.

While the above is ok, it's still hard to come up with a good policy, or even start with one. There exists a tool called `aa-genprof` that is somewhat similar to SELinux `sepolicy generate`, but allows static analysis to automatically generate a learning-based policy. (We'll see later such tools also exist on TOMOYO Linux, and on OpenBSD `systrace`).
This learning mode can be used to secure complex application, and even though AppArmor doesn't apply profiles on all programs, it still provides some tools to find software that might need one. It has the `aa-unconfined` command that will output a list of processes with tcp or udp ports that do not have AppArmor profiles loaded, for instance.

We generate a profile for a script on a path:

```
aa-genprof <script>
```

Subsequently, the command will automatically set the profile to complain mode, write audit logs, and instruct the user to start the application in another window. `aa-genprof` will keep scanning the logs and interactively ask the user when a violation is encountered, relying on `aa-logprof` under the hood.

Let's now mention how to implement RBAC using profiles. We've seen previously that there can be child profiles within profiles and transitions allowing to reduce the access scope. A similar transition can be done using what's called a "hat", a child profile that starts with the character `^`. The main difference, is that the hat is applied on a user-basis, usually attributed on login through a PAM module

called `pam_apparmor`, or programmatically (`aa_change_hat`). The PAM module allows the creation of roles based on hats.

The name of the hat is attempted to be assigned by `pam_apparmor` depending on an order set in its loaded parameter, it can be either based on username, primary group, or the string `DEFAULT`. If a hat is present in the profile and it matches then the sub-profile with that hat will be used. For example:

```
session  optional pam_apparmor.so order=user,group,default
```

```
/tmp/example {
    /etc/locale/**      r,
    ...

    ^vnm {
        /tmp/example/*  rw,
    }
}
```

We can now review a few commands we haven't seen yet.

The current AppArmor status can be found, listing many useful info, through `aa-status`.

```
> aa-status

 apparmor module is loaded.
 11 profiles are loaded.
 11 profiles are in enforce mode.
    /usr/lib/connman/scripts/dhclient-script
    /usr/share/gdm/guest-session/Xsession
    /usr/bin/googleearth
    /usr/bin/evince-previewer
    /usr/sbin/tcpdump
    /usr/lib/cups/backend/cups-pdf
    /usr/bin/evince-thumbnailer
    /sbin/dhclient3
    /usr/bin/evince
    /usr/sbin/cupsd
    /usr/lib/NetworkManager/nm-dhcp-client.action
 0 profiles are in complain mode.
 2 processes have profiles defined.
 2 processes are in enforce mode :
    /usr/sbin/cupsd (1192)
    /sbin/dhclient3 (22378)
 0 processes are in complain mode.
 0 processes are unconfined but have a profile defined.
```

The `apparmor_parser` can be used to rebuild profiles, and launch them, optionally with debug options.

```
> apparmor_parser -Q --debug /etc/apparmor.d/usr.bin.firefox | head ↙
    ↳ -10
---- Debugging built structures ----
Name:       /usr/lib/firefox-4.0b7/firefox{,*[^s][^h]}
```

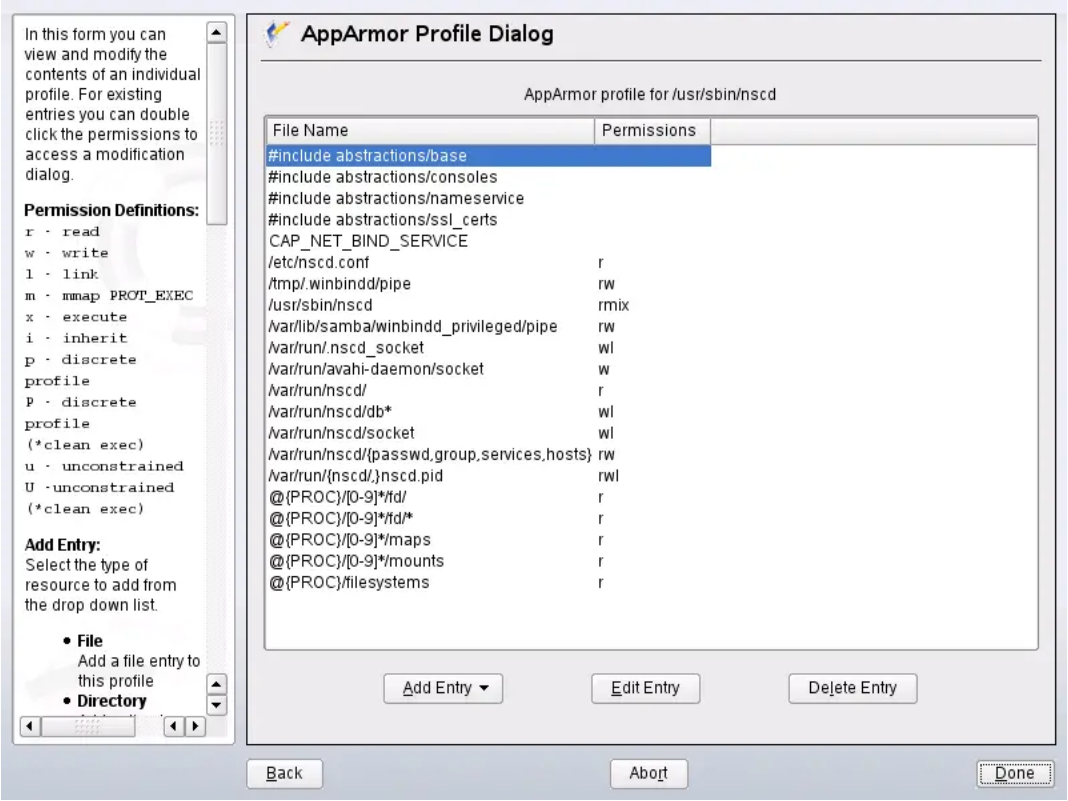
```
Profile Mode:   Enforce
Capabilities: net_bind_service
--- Entries ---
Mode:   r:r Name:   (/)
Mode:   r:r Name:   (/**/)
Mode:   rx:rx   Name:   (/bin/bash)
Mode:   rx:rx   Name:   (/bin/dash)
Mode:   rx:rx   Name:   (/bin/grep)
```

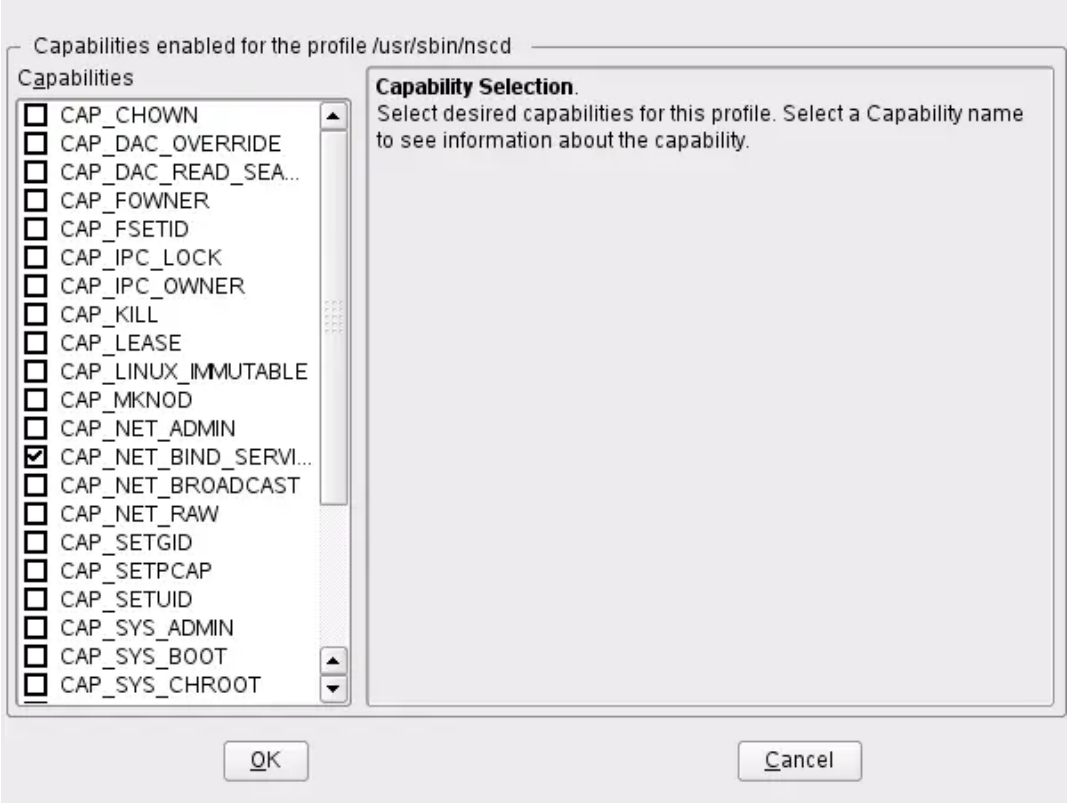
We can change the complain/enforce mode of profiles using `aa-complain` and `aa-enforce` respectively.

```
> aa-complain /bin/ping
> aa-enforce /bin/ping
```

We've taken a look at `aa-genprof` but there also exists `aa-autodep` to generate a minimal profile by looking at an executable.
In the same vein, another command that could be useful to test profiles that we haven't seen is the `aa-exec` command that will allow running a command using a particular profile.

Moreover, there exist graphical interfaces to manage AppArmor profiles such as AppArmor Admin and YaST.





That's it for AppArmor administration!

Practically it is used alongside `snapd` system, which we'll see in the isolation section, to simulate a containerised environment. Each snap package has a profile attached. Yet, most of the times, the rules are relaxed and useless.

While AppArmor sounds neater than SELinux, it still has weak points. It emphasizes the path of executables, and thus if the path is changed, the profile will stop working. Furthermore, the operations that it allows is considerably less granular than the class operations found on SELinux, and is mostly targetted at traditional DAC controls with MAC-level enforcement.

What you need to remember: *AppArmor is a MAC that relies on the file path of executable to restrict them. It achieves this using what's called a "profile", an association of path with rules, which can be hierarchical. The profile files are textual and compiled into a binary format using the `apparmor_parser`. The language syntax is straight forward: in the profile section it can include file permission, transition when invoking another executable, resource limits, network/ipc restriction, and POSIX capabilities. The utilities that come with AppArmor allow for the easy creation of profiles, using a learning-mode approach called "complain" in which the audit logs are followed and the user is asked whether the permission should be allowed. A minimal RBAC can be implemented using AppArmor in combination with a PAM module called `pam_apparmor`.*

Other LSMs There exist many other Linux security modules for MAC other than the popular SELinux and AppArmor. Some are pet projects and quickly went to the digital dustbin. This is the case of the Linux LOMAC project which then got ported to the FreeBSD's lomac we've seen, and the ZeroMAC project which has a simple system of labels on subject and objects with allowed and disallowed privileges such as read, write, append, execute, mount, etc.. ZeroMAC even includes a permissive mode allowing learning which privileges are needed.

Let's take a quick look at two more advanced LSMs: TOMOYO and Smack. In another section on RBAC we'll emphasize Grsecurity.

TOMOYO is a lightweight MAC developed by NTT Data Corporation and merged into the kernel mainline in June 2009. Like AppArmor, it uses a file path based approach to MAC. One particularity is that it separates security domains according to a process invocation history, learning the system behavior.
A security domain is a process call chain, an execution history, represented by a string. Every domain can run in one of 4 modes: disabled, learning, permissive, or enforcing. The learning mode, like the name implies, is analogous to AppArmor and also relies on auditd logs, easily creating policy by automatically analyzing which accesses occurred in the kernel.

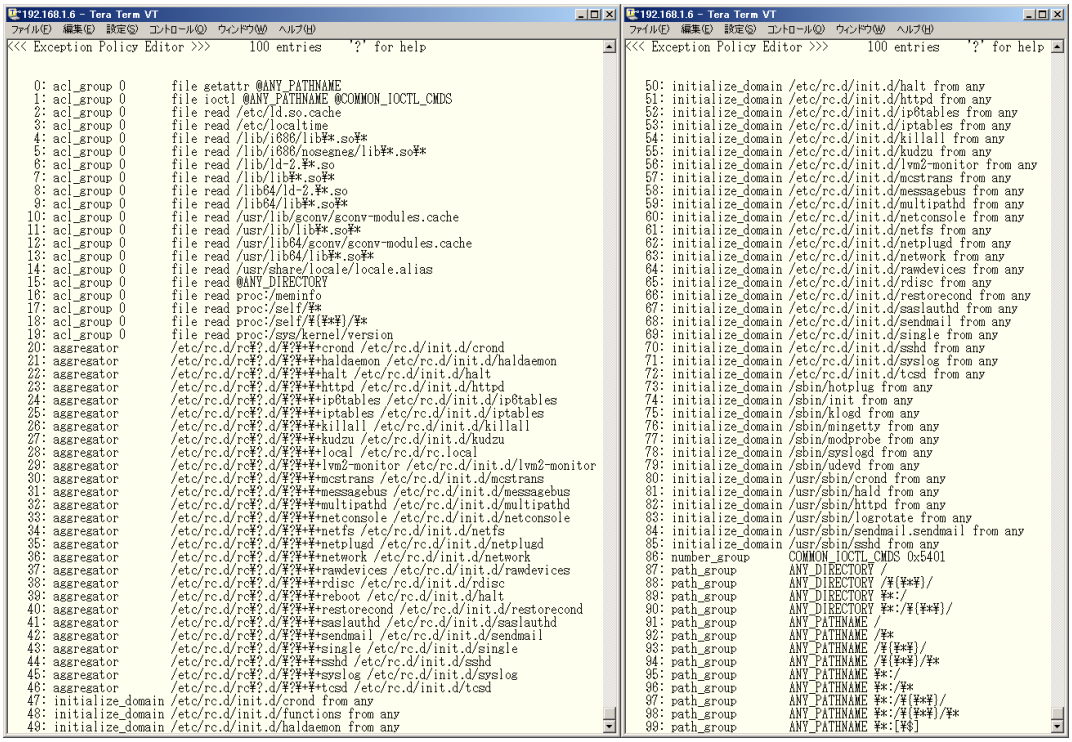
Practically, every time a process is executed, a new domain is created. Once another executable is invoked from that domain, a domain transition, then it is concatenated to the previous file path in the domain. This creates a string list of the process execution history.

There are two types of domains and of policies: kernel, for the starting kernel processes, and user-space. `<kernel>` is usually the start of the domain. Policies are assigned to domains, and they all live in the `/etc/tomoyo` directory.

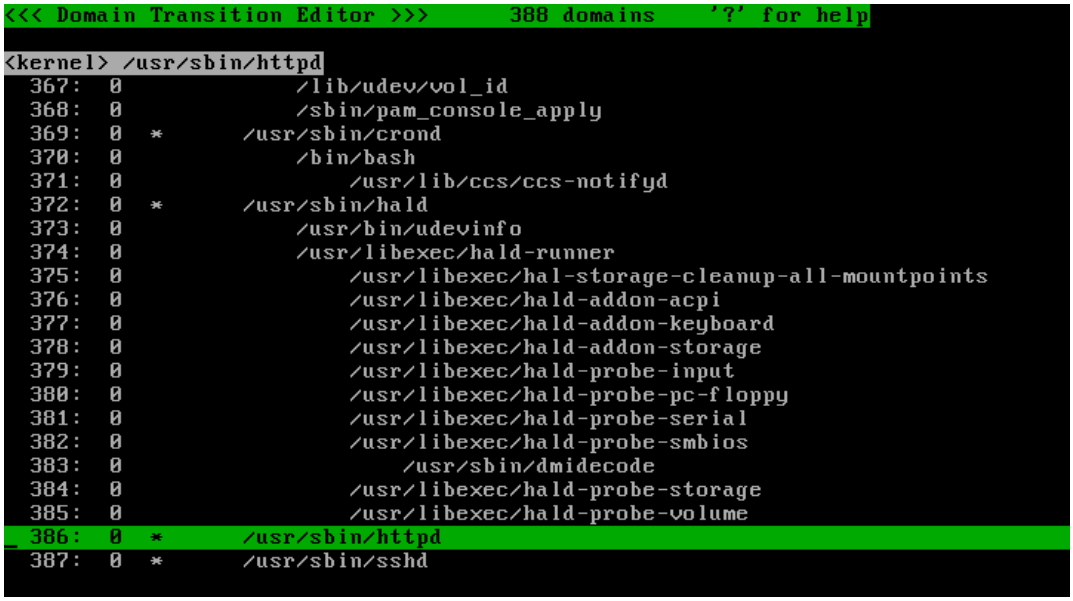
Here's an example of a domain for Apache, showing how it's as simple as appending the path of the executable.

If `/usr/sbin/httpd` is invoked by `<kernel> /usr/sbin/mingetty /bin/login /bin/bash`, then the domain name is `<kernel> /usr/sbin/mingetty /bin/login /bin/bash /usr/sbin/httpd`.

A policy editor `tomoyo-editpolicy` exists to facilitate the creation and modification of policies.



Furthermore, most of the interaction with TOMOYO happens in a curses interfaces, allowing easily adding restrictions and permissions to domain policy, which are represented as directives such as `file read` and `file write`, along with global policies and exception lists. TOMOYO also offers scripts to help edit policy files such as `tomoyo-diffpolicy`, `tomoyo-patternize`, `tomoyo-selectpolicy`, etc..



For example, `tomoyo-patternize` can help simplify policy from:

```
<kernel> /usr/sbin/httpd
file read /var/www/html/index.html
file read /var/www/html/alice/index.html
file read /var/www/html/alice/page1.html
file read /var/www/html/alice/page2.html
file read /var/www/html/alice/image1.jpg
file read /var/www/html/alice/image2.jpg
file read /var/www/html/bob/page2.html
file read /var/www/html/bob/image1.jpg
```

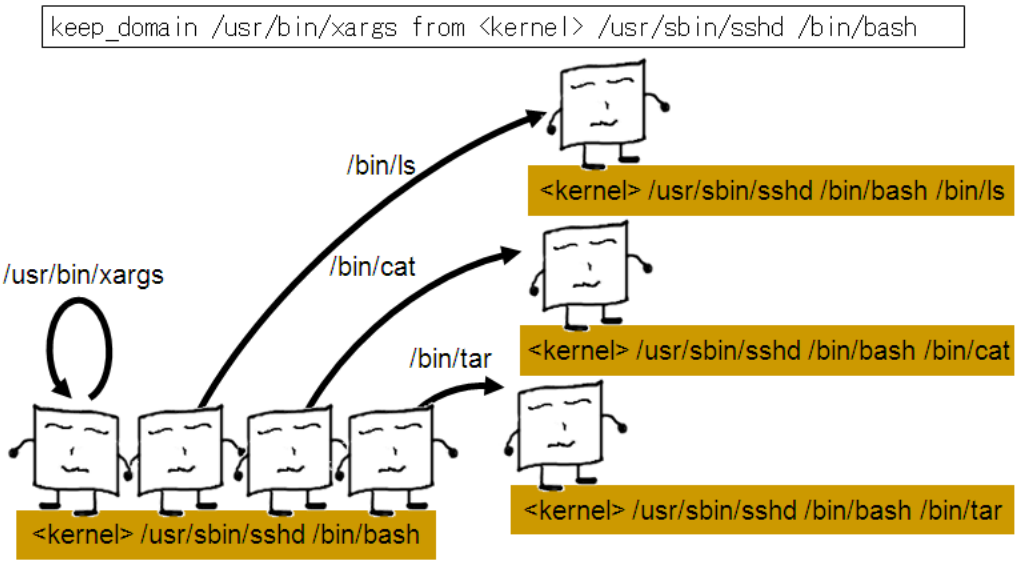
Unto:

```
<kernel> /usr/sbin/httpd
file read /var/www/html/\*.html
file read /var/www/html/\{\*\}/\*.html
file read /var/www/html/\{\*\}/\*.jpg
```

Lastly, just like AppArmor, TOMOYO can control domain transition, deciding whether to keep the permission of the current domain or not.

An example using `xargs`:

```
keep_domain /usr/bin/xargs from <kernel> /usr/sbin/sshd /bin/bash
```



Source: TOMOYO Linux - How do I manage domains

TOMOYO gives us AppArmor vibes but with its own concept of process historical behavior, emphasizing more the transition, which in AppArmor we called child profiles. Let's look at Smack now.

Smack, Simplified Mandatory Access Control Kernel, is a project from Tizen OS merged in Linux since the 2.6.25 release. The approach that Smack uses is label-based, relying on extended attributes.

One particularity is that it relies on a pseudo-fs to control and configure the MAC, it is usually mounted as such in `/etc/fstab`:

```
1 smackfs /sys/fs/smackfs smackfs defaults 0 0
```

The Smack labels exist as plain text extended attribute in the `security` namespace. As with any extended attributes, they can only be changed when the process has enough privileges, usually superuser or `CAP_MAC_ADMIN` capability. The possible extended attributes that can be set on objects are categorized depending on the type of file involved.

- `security.SMACK64`: Used for file system object access control.
- `security.SMACK64EXEC`: Used for processes that execute a program, when invoke their attributes will switch to those set here.
- `security.SMACK64TRANSMUTE`: Can only be set to `TRUE` and on a directory. When set, if the task creating an object in the directory has a `t` mode, the object created gets the label of the directory instead of the one of the creating process's attributes.
- `security.SMACK64IPIN` and `security.SMACK64IPOUT`: Used for file descriptors of sockets, controlling the access decision on packets and the outside world (controlled by `/sys/fs/smackfs/netlabel`).

As we said, Smack configuration are accomplished by writing to files in its pseudo-fs under `/sys/fs/smackfs`. This is also where you can associate labels on users and create access rules.

For instance, the `/sys/fs/smackfs/load2` and `change-rule` interfaces are used to add rules and modify them. Meanwhile `access2` interface is used to report whether a subject has particular access to an object. `load2` takes, as standard input text you can pipe in it, the following format:

```
subjectlabel objectlabel access
```

The `access` part takes the form of a combination of letters, similar to basic POSIX permission:

- `a`: Indicates that append access should be granted.
- `r`: Indicates that read access should be granted.
- `w`: Indicates that write access should be granted.
- `x`: Indicates that execute access should be granted.
- `t`: Indicates that the rule requests transmutation.
- `b`: Indicates that the rule should be reported for bring-up.

Indirectly, the starting process should get its first subject label from the init process or other means such as when executing an executable with the `security.SMACK64EXEC` label. The current label of a process can be read from `/proc/<pid>/attr/current`, like any extended attribute or `/proc/<pid>/attr/smack/current`, which can also be modified by writing to it.

Labels can be any string up to 255 chars, and usually they are only compared for equality (if they match a rule), however there exist a couple of special labels affecting the enforced rules. The general access check goes as follows:

- Any access requested by a task labeled `*` (star) is denied.
- A read or execute access requested by a task labeled `^` (hat) is permitted.
- A read or execute access requested on an object labeled `_` (floor) is permitted.
- Any access requested on an object labeled `*` (star) is permitted.
- Any access requested by a task on an object with the same label is permitted.
- Any access requested that is explicitly defined in the loaded rule set is permitted.
- Any other access is denied.

Furthermore, application that use the network can be labeled to be restricted too. This is done in the `/sys/fs/smackfs/netlabel` file where you can add white-listed rules, allowing access to specific IP in the form:

```
@IP1        LABEL1 or
@IP2/MASK   LABEL2
```

It means that your application will have unlabeled access to `@IP1` if it has write access on `LABEL1`, and access to the subnet `@IP2/MASK` if it has write access on `LABEL2`.

We did say that the `/sys/fs/smackfs` directory is created by the kernel. Yet, Smack still has a configuration file outside the pseudo-fs in `/etc/smack/accesses` containing the rules to be set at system startup and which will be directly written to `/sys/fs/smackfs/load2`.

Smack facilitates management of the pseudo-fs using only three commands:

- `chsmack`: Used to display or set Smack extended attribute values, instead of relying on `getfattr` and `setfattr`. (ex: `chsmack -a value path`)
- `smackctl`: Used to load Smack access rules

- `smackaccess`: Used to test if a subject label has access to an object label, similar to the `/sys/fs/smackfs/access2` interface.

Globally, Smack offers a simple MAC mechanism with a minimal set of permissions, which are not as granular as SELinux or AppArmor. Yet it is confusing how subjects will initially get assigned their labels, the documentation only mentions init system, `/etc/smack/accesses`, and upon execution of files with the `SMACK64EXEC` label.

What you need to remember: *There exists many Linux Security Modules other than SELinux and AppArmor, even the POSIX capabilities is an LSM as we said previously. TOMOYO is a MAC relying on the path of executable and the history of further executable they will call. It associates with that historical behavior certain access rules. Smack is a MAC relying on labels, extended attributes. The key of the labels control their behavior, some are only set on file system `SMACK64`, others will be given to the subject upon invoking the executable `SMACK64EXEC`, and others control the default label given to files created in a directory `SMACK64TRANSMUTE`. The access rules are controlled in a pseudo-fs in `/sys/fs/smackfs` and the permissions are sparse (read-write-execute-append). It also offers special labels, allowing subject global access, or object global access.*

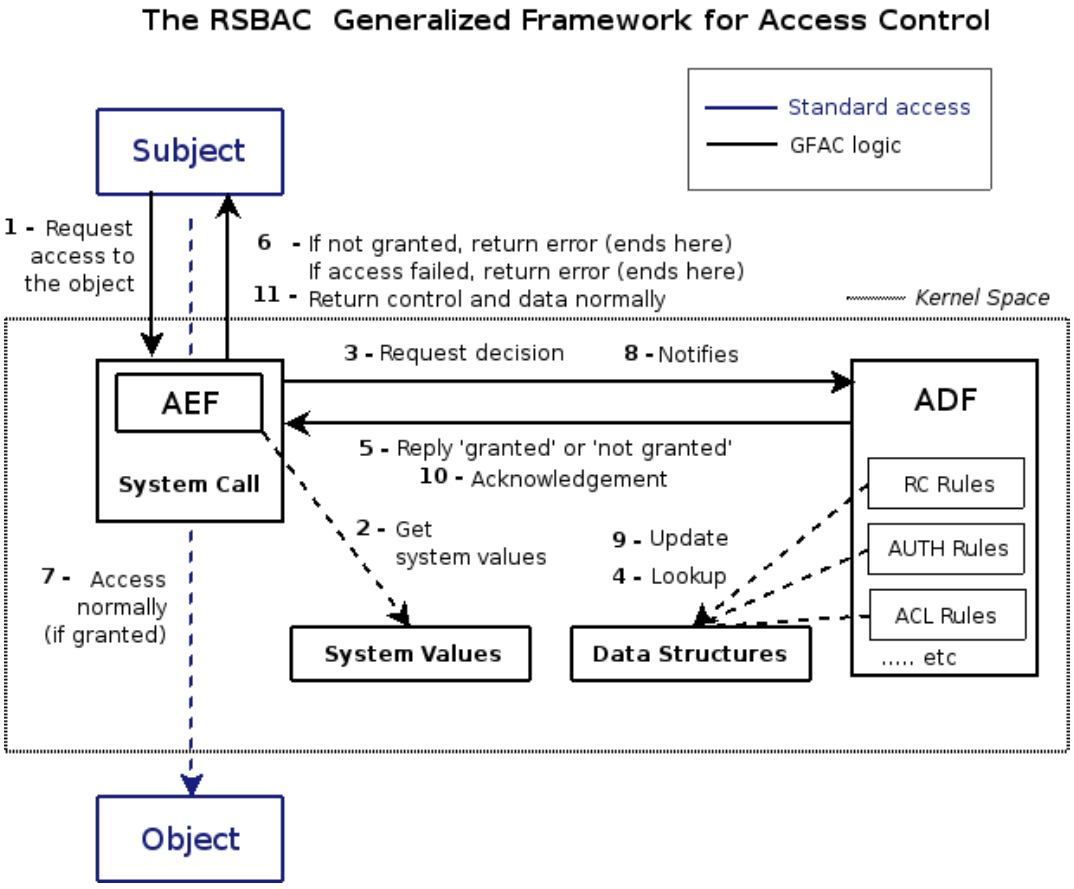
RSBAC Another Linux Modular Security Framework RSBAC, Rule-Set Based Access Control, a wink to the GFAC (Generalized Framework for Access Control) which it implements and extends, is a modular access control framework similar to SELinux but that has the particularity of not being a LSM.

It chose not to rely on LSM because it needed more hooks than were present at the time, didn't want to give modules direct access to kernel objects (instead passing copies of context information), wanted more control such as notification upon object change for logging/auditing, and wanted to allow multiple modules to co-exist which the LSM stacking only did on a per-module willingness to forward requests. Thus now it lives as a separate patch, "out of tree", of which the last update, as of this article, dates of 2021.

The logic is similar to SELinux: there are subject (processes), actions to be done on objects (requests for access on specific targets), and objects (targets).
For example, `READ` request can be done on targets `FILE`, `DIR`, `FIFO`, `DEV`, `IPC`. This is the equivalent of activities invoked on a class of object on SELinux.

The RSBAC framework is composed of two main components, the AEF, Access Control Enforcement Facility, and the ADF, Access Decision Facility.
The AEF is the piece that will catch system calls, gather the relevant kernel context and object status, and forwards it to the ADF to wait for its response to reply back to the subject on whether or not the action is allowed.
The ADF is a grouping of modules, or policies, all of them are passed the parameters from the AEF. Then, if a single module returns a negative reply, the ADF will deny access, all modules need to agree. A module can be an access control policy or any generic security feature.

Afterward, if the access is given and the object is modified, the ADF is notified of the change, which is used for logging and other purposes such as object attribute tagging.



Source: Framework Components

All modules manage their own attributes which they can assign to subjects and objects. With them, there's a kernel daemon managing the structures, `rsbacd`, implementing the ACI & ACC, Access Control Information & Context, which will periodically save any lists tagged as changed and apply them to file system objects.
These objects can also be network related, such as packets, ports, IP address, etc..

A particularity of RSBAC is that it has an optional, compile-time setting, to enable in-kernel user management, instead of shadow password suite and other file-based management. This includes the ability to have a virtual set of users, granular access control to user attributes per-user, and in-kernel password check and encryption.

This feature comes with a PAM module `pam_rsbac` and the usual set of administration tools prepended with the `rsbac_` prefix. This means `rsbac_useradd`, `rsbac_groupadd`, `rsbac_usermod`, `rsbac_groupdel`, `rsbac_login`, etc.. They also offers flags to convert all existing users:

```
rsbac_useradd -v -O
rsbac_groupadd -v -O
```

Additionally, `rsbac_usershow` and `rsbac_groupshow` give details about users and groups. They also

allow to make backups, if you have the necessary access rights to each individual user and group.

Virtual users and groups are part of virtual sets, prefixed with a number, that can be copied from set to set.

The in-kernel user management allows granularity in picking who can access what on the users, instead of having them all in the `passwd` file. These range from mapping UID to name with the `SEARCH` access, to `DELETE` access, and anything in between.

All the different attributes, used by the modules, are set using commands starting with `attr_set_` followed by the target, and read by `attr_get_<target>`. For example `attr_set_fd`. Otherwise, all modules have their specific commands starting with the module alias, example the ACL module has a command called `acl_grant`.
There are multiple curses interfaces ending in `menu` such as `rsbac_menu`, `rsbac_user_menu <user id>`, and `rsbac_fd_menu <file>`, however the full list is hard to find in the online documentation.

Furthermore, RSBAC has a love of kernel boot time parameters which are used to control certain options related to debugging. These include flags such as `rsbac_debug_adf_auth` and `rsbac_softmode` among many others.

The modules that RSBAC implements are turned on at compile time in kernel configuration. Some of these are mandatory, such as the AUTH, authenticated user, module, and others optional. Many of them are similar to LSM and SELinux features, such as the CAP module implementing Linux capabilities, and the MAC module implementing a multi-level security and clearance model.
Every module comes with its own set of tools and often curses interfaces to administer their configurations. Let's see a couple of them before closing this section, we'll pay more attention to a particular role-based access control module in a future section dedicated to it.

The "AUTH" module, is a support module for other modules, its main functionality is to define which UID a program or process can assume, the `CHANGE_OWNER`/setuid access. If none is defined, then all access is forbidden.
To be able to assume another UID, a process needs to be assigned the `auth_may_setuid` or be able to add the setuid bit by having the access `MODIFY_ATTRIBUTE` on the target `A_auth_add_f_cap` and `A_auth_remove_f_cap`, basically allowing modifying the file attributes.

Initially nothing is allowed, thus it's recommended to configure an administrative user, usually called `secoff`, the security office, with `UID=400` and allow it to login through kernel boot parameters such as `rsbac_auth_enable_login` or `rsbac_softmode`. A user can also be associated with an attribute `system_role=security_officer` to be able to manage the AUTH module.

For instance, after being able to login, you can allow the `/bin/login` executable to have setuid bit with either the curses interface or the `attr_set_fd` command:

```
> rsbac_fd_menu /bin/login
> attr_set_fd AUTH FILE auth_may_setuid 1 /bin/login
```

As you can see the syntax looks like, `attr_set_fd`, then the module name `AUTH`, where it applies `FILE`, and the attribute name and value `auth_may_setuid 1` along with the target, `/bin/login`.

RSBAC also offers a learning mode attribute which can be set in `rsbac_softmode`:

```
> attr_set_file_dir AUTH FILE `which sshd` auth_learn 1
```

```
> /etc/init.d/sshd start
> attr_set_file_dir AUTH FILE `which sshd` auth_learn 0
```

The attributes needed will be applied as it learns which UID it needs to access, or any other attributes from any other module that supports learning for that matter.
The learning mode can also be set globally with the command `rsbac_auth_learn`.

Another module that RSBAC implements is the POSIX capabilities we've seen. It can assign a minimum and maximum capability set to files and processes. Shortly said: `final set = (original & max_caps)| min_caps`. Note that `max_caps` is the upper-bound.
The list of the supported capabilities is found here and is mostly the same as under LSM

In softmode, only the maximum capability set is respected, but this is useful when the learning mode is activated.

Like other modules, the capabilities are set as attributes using the `attr_set_<target>` commands or `rsbac_<target>_menu`. However, they are only used to change the minimum and maximum set. This can either be done with `rsbac_user_menu` and `rsbac_fd_menu` or with the command line tools `attr_get_user`, `attr_set_user`, `attr_get_file_dir` and `attr_set_file_dir`. For instance:

```
> attr_set_user CAP secoff min_caps DAC_READ_SEARCH KILL
```

Yet another module is the ACL module, for an access control list management not to be confused with POSIX ACL. It can be used to specify, in a global ACL, which user, role, or group, is granted access to which object type and with which request (usual RSBAC access on target).
When there is not ACL for a subject on an object, then the rights of the parent object are inherited, mixed with a mask.

There is a default ACL for each object type. To change them, a user requires the necessary rights, unless it's the security officer, UID of 400.
Moreover, this module even allows to associate ACLs with a time-limit, removing them afterward.

Again, these are managed through either the curses menu `rsbac_acl_menu` and `rsbac_acl_group_menu`, or the tools `acl_grant`, `acl_group`, etc.. There even is a command called `linux2acl` which will convert the whole system to this ACL mechanism.

For example the `acl_grant` command has the form:

```
acl_grant [switches] subj_type subj_id [rights] target-type
    file/dirname(s)
```

Which looks like:

```
> acl_grant USER joe READ DIR /root
```

And `acl_tlist` can be used to show all ACLs at `/root`:

```
acl_tlist DIR /root
```

Let's finish by listing the rest of the modules:

- MAC: A mandatory access control mechanism, implementing Bell-La Padula and a bit more.

- FF (File Flags): Allow tagging files and directories with global attributes such as `execute_only`, `no_execute`, `read_only`, `append_only`.
- SIM (Security Information Modification): Only the security officers are allowed to modify data tagged as security information.
- FC (Functional Control): Restrict access to security information for security officers only and allows only administrators to access system information.
- MS (Malware Scan): A scanner relying on the notifications from the AEF to check for malware.
- JAIL (Process Jails): A clone of FreeBSD jail that adds a new system call `rsbac_jail`. Programs are launched in `chroot` and restricted. See also the isolation-based access control section for more information on jails.

Globally, RSBAC offers a lot of features with a neat and understandable architecture. However, the documentation is rough on the edges and the combination of curses and CLI to manipulate the attributes on files is a bit messy.

What you need to remember: *RSBAC, rule-set based access control, is a patch to the Linux kernel offering a modular approach to access control in the same way LSM does. It also uses the concept of subject, action/access, object/target. Modules are all consulted to take a decision on access and are notified when the access changes the object. There is an option to manage users in the kernel, allowing granular access to them. An administrative user exists called a security officer with `UID=400`. Multiple modules are present such as ACL (not POSIX ACL), AUTH (to fix who can change UID), POSIX capabilities, and more. The configuration allows a learning mode for most modules, along with kernel boot time params for better debugging. The tool set offers both curses and CLI interfaces to manipulate all target objects.*

Mandatory Access Control on Other Unix-Like Systems

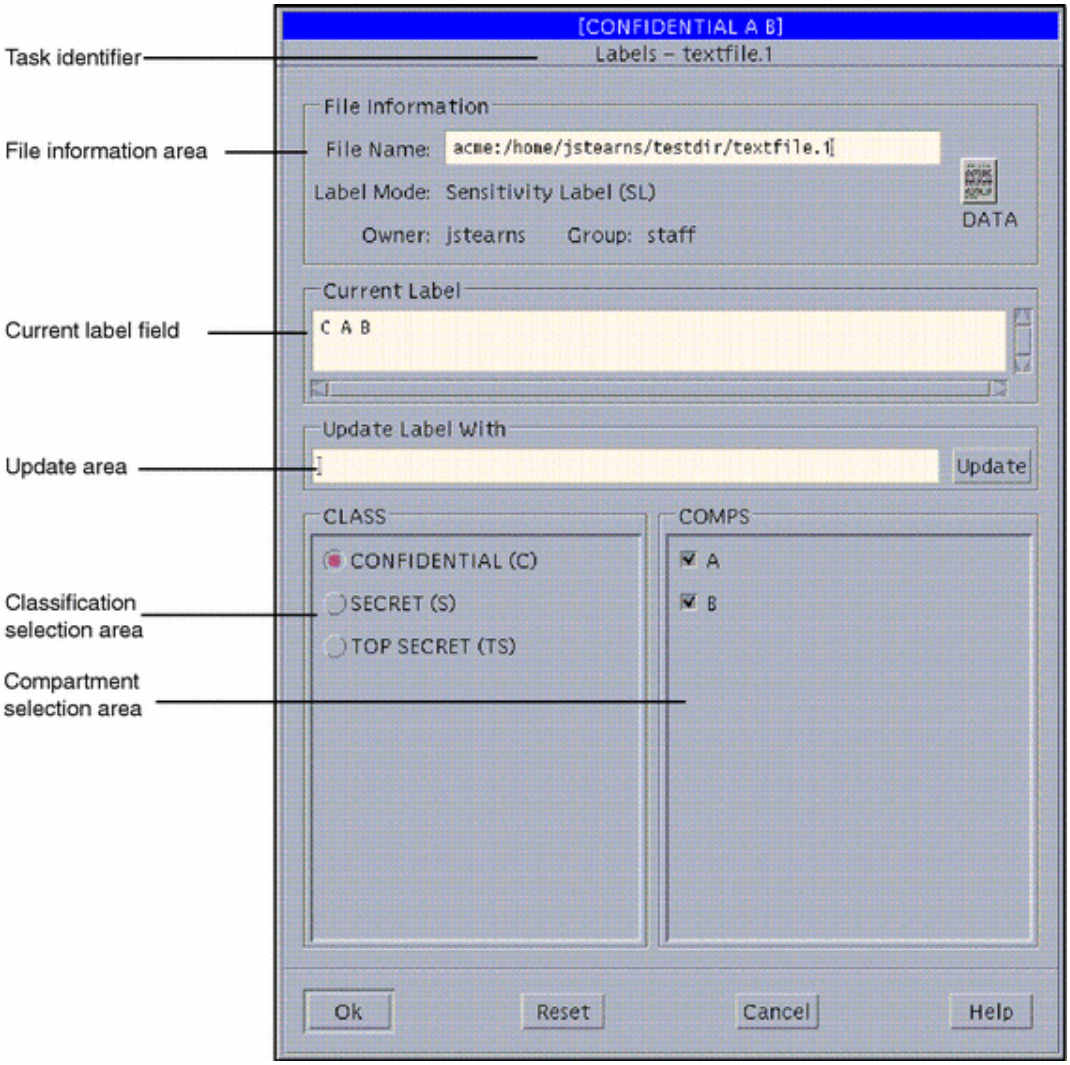
In this section, we'll have a glance at how some other Unix-like systems implement mandatory access control.

Apple's macOS has a MAC framework which is an implementation of the TrustedBSD's MAC framework and extends it using sandbox functionality which we'll cover in the isolation section. The security restrictions are created by application developers and can't be overridden, they are bytecode-compiled and loaded.

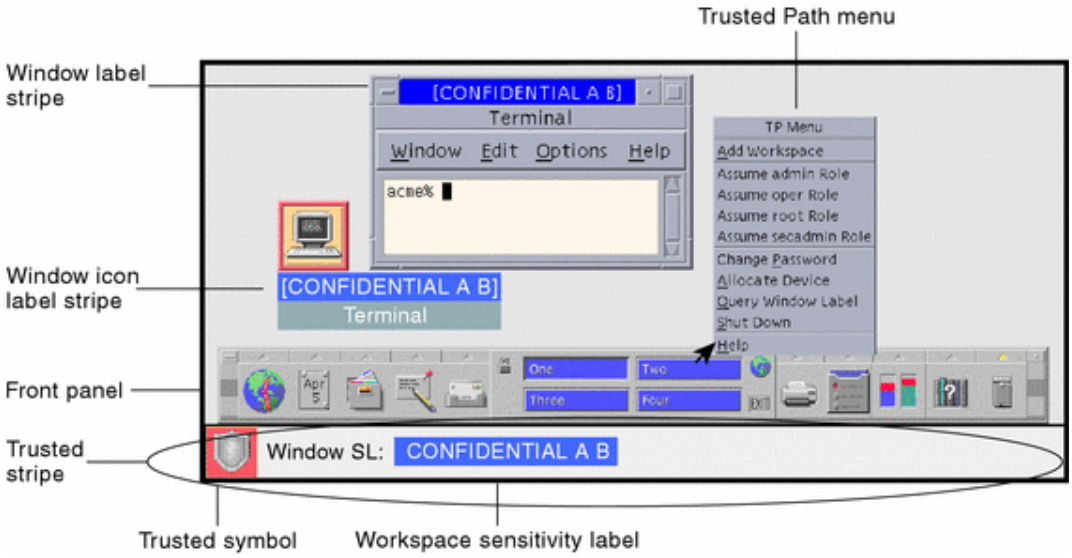
Android relies heavily on SELinux, also extending it with the concept of sandboxing. It had to dismiss the reference policy and create its own extensions and policies instead. Apps are launched by the `zygote` process and independently labeled by the Dalvik VM.

We'll see both macOS and Android sandboxing in another section.

TrustedSolaris has an interesting implementation of MAC where labels and clearance are easily managed through the file manager and graphical utilities. For instance, the file manager allows editing labels.



And is explicit in all the graphical utilities on which security label is assigned to what. So that the user can quickly drag and drop files, from one security level to another.



What you need to remember: *Many Unix-like OS implement MAC, some reusing existing pieces, and others going their own way*

OpenBSD relationship with POSIX.1e/2c

We can talk about the elephant in the room: Why doesn't OpenBSD, a Unix-like OS that is known for its security, implement any of the POSIX.1e/2c extensions?

OpenBSD had support for extended file attributes, added for POSIX ACL support, in a non-GENERIC branch, until 2005 when the lack of interest killed the project.

Multiple factors kept OpenBSD away from POSIX.1e/2c, starting with the lack of test and maintenance for the extended attributes.

Furthermore, OpenBSD has a different approach to security where it emphasizes minimizing the attack surface and exploit mitigation through programs correctness instead of system-wide rules. OpenBSD also has an aversion to complexity, which these solutions brings along, and favors keeping the kernel lean.

OpenBSD, as we'll see in the isolation section, prefers that programs voluntarily isolate themselves by adding patches to their codebase and relying on features such as `unveil` and `pledge`. This is also a reason why it offers "secure" alternatives to common pieces of software.
Meanwhile, on other systems that do have MAC, the mindset defaults to not trust programs running on the machine, especially third-party software that aren't part of the base OS.
Yet, one can argue that these options are not mutually exclusive, we can reduce the attack surface with quality and lean code while also offering a safety net by having globally enforced rules.

What you need to remember: *OpenBSD doesn't implement POSIX.1e/2c because it instead wants to keep its kernel lean and its software too. Instead it takes a voluntary approach where the maintainer patch software and write alternative that self-isolate and reduce the attack surface. Yet, nothing really justify not having both MAC and reduced attack surface at the same time.*

Particular Role-Based Access Control

We've observed how RBAC can be a subset of MAC, but some systems implement it separately.

RBAC on SunOS Derivatives

A feature from SunOS derivatives, including Solaris, OpenIndiana, Illumos, SmartOS and others, is a role-based access control implementation. It is used to split privileges and access across different administrative users that are not allowed to login, but that other users can access as "roles".

The roles get privileges and access through bundles called rights profiles. The same profiles we've seen in the SunOS profiles which can also be assigned to regular users.
In practice, roles are like usual accounts but made special through their functional responsibilities rather than because they represent an actual user.

Roles are like normal users, having their own password in the shadow file, however they are unable to log into a system as a primary user. Instead a user must first log in as a normal user and assume the role. This means that the "auth" actions (see in action-based access control), privileges (see in POSIX Capabilities on SunOS Derivatives), and executable profiles (see in SunOS Derivatives Profiles) are attributable to both normal user and roles. Furthermore, this implies that a user can assume a role and then launch a profile sell with privileges it didn't used to have before, or access functionalities in a program using the "auths" of the role.

For instance, on Solaris, the root user is by default a role that other users can assume, if allowed to. This means you cannot log into the system as root.

The roles can't be hierarchical, once a user assumes a role it cannot assume another one. But since they can be assigned profiles, which are hierarchical, then roles can indirectly have the same effect.

What differentiate a role from a normal user is its entry in the file `/etc/user_attr`, that contains the extended user attributes database, similar to `login.conf` and `login.defs` that we've seen. Its format was explained in the SunOS profiles section.
The relevant fields for us this time:

- `type`, can be either `normal` for normal accounts or `role` indicating the account can only be used as a role.

- `roles`, a comma separated list of role names that the account can switch to, these need to be of `type=role` and are only assignable to `type=normal`.
- `roleauth`, present in some systems such as Solaris 11 and above. It allows users to switch role by using their own password instead of the role's password when set to `roleauth=user`. When that is set, it acts similar to `sudo` and `doas`.

Editing the `attr` part of the `user_attr` file is how we enforce those, however, as we said earlier, it can also be done through a command, this time, instead of `usermod`, it's called `rolemod`.

For instance, we can change the `type` of the entry for root to a normal user.

```
> rolemod -K type=normal root
> getent user_attr root
root::::auths=solaris.*;profiles=All;audit_flags=lo\
  :no;lock_after_retries=no;
min_label=admin_low;clearance=admin_high
```

And switch back to `type` role.

```
> usermod -K type=role root
> getent user_attr root
root::::type=role...
```

Furthermore, we can directly create and remove roles with the `roleadd(8)` and `roledel(8)` commands.

The `roles` command prints the roles that the current, or passed users, have been granted.

```
> roles tester01 tester02
    tester01 : admin
    tester02 : secadmin, root
```

Roles are assumed through the usual login, be it `su`, `rlogin`, or any other service or program that supports the `PAM_RUSER` variable. For instance, to assume the "admin" role shown for the "tester01" user we can use:

```
> su admin
```

Whether the password of the "admin" role is asked or the password of "tester01" depends on whether the `roleauth` attribute in `user_attr` value is set to `user` or not.

Yet, all this isn't a real RBAC, as a user can have permissions that aren't assigned to roles. We don't only set permissions on roles, and then set roles to users.

What you need to remember: *SunOS have special accounts called "roles". They can't log in the system and can only be accessed by users that have them set as roles in their `user_attr` entry. Roles are accessed like any user through `su` and other commands. Since roles can also have execution profiles, "auths", and privileges, it allows the creation of granular access from a centralized place. Root is a role by default.*

We've uncovered the RSBAC Framework, rule-set access control, in a previous section.

There exist an RSBAC module called RC, role compatibility policy, that implements a simple RBAC-like mechanism.

Every process has a default role, one inherited by all processes, and then change role to get access to different permission sets. Processes can only be assigned one role at a time.

Like other access control in RSBAC, roles are assigned a list of object/target types along with access over them. The roles can either be assigned to users as attributes or on executable marked with the `rc_initial_role` or `rc_force_role` attributes.

A role is defined as an entry that has multiple fields including a name, a role compatibility (allowing to switch between roles without setuid), a list of objects it can access (`type_com_<target>`), its administrative role, which target it can create (`def_<target>_create_type`), whether it's allowed at boot, and finally if changing role requires the user to re-enter their password.

The administrative role is what decides the RC module administration. It can either be `none`, `system admin` for read-only, and `role admin` for full access.

As far as attributes goes, the targets get assigned their type in a separate `rc_type` attribute, the users get their roles in `rc_def_role`, and files and directory can additionally have the `rc_force_role`.
There exists a couple of special values assignable to role or types to allow more control, such as `role_inherit_parent` to inherit from parent object (ex: parent directory).

In the initial configuration, there's an optional default set of values that can be used to have predefined roles such as general user (role id `0`), role admin (role id `1`), and others. The root user gets assigned the system admin role (role id `2`), while the UID 400, the security officer, gets assigned the admin role (role id `1`).

Furthermore, just like ACL on RSBAC, roles can be assigned time limits.

When it comes to management, since a process can only be assigned one role at a time, some utilities are present to allow copying roles and types.

To get roles we can use `rc_get_item`/`rc_get_current_role`/`rc_get_eff_rights_fd` and to set or copy them we can use `rc_set_item`, `rc_copy_role` and `rc_copy_type`.
To launch a program with another role the command `rc_role_wrap` is used.

```
1 > rc_role_wrap role_id prog args
```

Two menus exist for the RC module: `rsbac_rc_role_menu` and `rsbac_rc_type_menu`.

What you need to remember: *RSBAC offers a role module called RC. It uses attributes assigned to users and targets to control access. A user can only have one role at a time. The administration of roles is a role in itself. A role contains which access on targets are permitted. Some default roles exist. A set of tools are used to manipulate them such as the curses menu `rsbac_rc_role_menu` and `rsbac_rc_type_menu`.*

Like RSBAC, GrSecurity/PaX is a set of patches, “out of tree”, to the Linux kernel adding security features that don’t rely on the LSM API. However, this time it isn’t modular and emphasizes mainly two things: enhanced hardened kernel protection and role-based access control as a MAC.

Since 2017 the patches aren’t publicly available anymore and the forked projects, such as the one by minipli, aren’t maintained.

GrSecurity keeps its RBAC system policy in a centralized system-wide file that has all the rules in `/etc/grsec/policy`. It contains the definition of roles, which in turn contains subjects and objects with their access rights such as read, write, capabilities, resources, IP ACLs, and PaX flags..
A role is given to a user based on whether it matches its UID, GID, or falls back to default role, in that order. The roles are essentially container of a set of subjects, acting in a specific scenario. The subjects represent executable paths on the system. This gives rise to the following role hierarchy:

```
user -> group -> default
```

Each role can have multiple subjects/file-path, and once executed as a process, these subjects can access the objects defined underneath.
The policy syntax goes as follows:

```
role <role1> <rolemode>
<role attributes>
subject / <subject mode>
<subject attributes>
    / <object mode>
    <extra objects>
    <capability rules>
    <IP ACLs>
    <resource restrictions>
subject <extra subject> <subject mode>
<subject attributes>
    / <object mode>
    <extra objects>
   ...
role <role2> <rolemode>
```

For instance:

```
role admin sA
subject / rvka
       / rwcdmlxi

role default G
role_transitions admin
subject /
       /               r
       /opt            rx
       /home           rwxcd
       /mnt            rw
```

```
        /dev
        /dev/grsec      h

role user1 u
  subject /
    / r
    /tmp rwcd
    /usr/bin rx
    /root r
    /root/test/blah r
   ...
  subject /usr/bin/specialbin
    /root/test rw
   ...
```

This is similar to the rules in the AppArmor section, however they are grouped by roles instead of path.

As you can see, the subject is either an executable path or a directory. Once executing that subject, the access rights of the objects underneath, along with other access restrictions will be enforced. This hierarchy of subjects and objects is always matched from the most specific to the less specific pathname (ie: it will match `/bin/ping` instead of `/bin` if both are present in subjects).
The rules can also allow glob/regex policy definition for objects with the usual characters such as `*`, `?` and `[]`.

Every role, subject, and objects are accompanied with a mode which decides either what it is, or additional restrictions and permissions.

When it comes to the modes that can be assigned to a role, the list is found here. It is used to decide how the match will take place. It can either be based on user, group, default, or some special role. Other than this, the role mode is used to control whether learning is turned on, if it's an administrative role, if authentication is needed, and if PAM should be involved, etc..

The subject modes are used to decide how the executable will be invoked, mostly related to kernel security features.

The objects modes include the usual read-write-execute along with more particular ones such as append, directory creation and deletion, access to hidden objects, allowing setting setuid/setgid on file, etc..

For example, the user role is defined with the mode `u`:

```
role user1 u
```

The group role with the mode `g`:

```
role group1 g
```

Both can have, as additional rules, a restriction on which IP can switch to these roles with the `role_allow_ip` attribute.

```
role user1 u
  role_allow_ip 192.168.1.5
  ...
```

The default role is defined as such:

```
role default
```

Meanwhile, the special roles, which are roles that aren't matched, but transitioned to using the command line `gradm -a/p/n <rolename>`, which we'll see, are defined with the `s` mode. These are often accompanied with the flags for authentication, whether its required or not, and using PAM, or not.

```
role specialauth s
role specialnoauth sN  # no auth
role specialpamauth sP # PAM auth
```

Moreover, roles can group multiple users or groups that don't share the same UID or GID using the concept of domain. The syntax is exactly the same, however, the word `domain` is used instead of `role`.

```
domain somedomainname u user1 user2 user3.. usern
domain somedomainname g group1 group2 group3.. groupn
```

Example:

```
domain somedomain u daemon bin www-data
subject /
    /    h
```

We know how to match roles to users and how to put underneath a path of an executable, now let's see what kind of rules we can set underneath.

We've seen we can have objects, which are path on the file-system, along with modes setting which permissions we have on them. Furthermore, we can merge different sets of objects together since grsecurity 2.x. We define objects separately, and then use mathematical set operators (`&`, `|`, `-`) underneath the subject.

```
define objset1 {
    /root/blah rw
    /root/blah2 r
    /root/blah3 x
}

define somename2 {
    /root/test1 rw
    /root/blah2 rw
    /root/test3 h
}

subject /somebinary o
    $object1 & $somename2
or
    $object1 | $somename2
```

```
or
    $object1 - $somename2
```

There's also the possibility of creating aliases using the keyword `replace`, and then referring to the alias as a variable `$(alias)`:

```
replace CVSROOT /home/cvs
replace PUBHTML public_html

subject $(CVSROOT)/bin/test o
      $(CVSROOT)/grsecurity r
      /home/spender/$(PUBHTML) r
     ...
```

A subject can have, apart from objects, POSIX capabilities, resource limitations, network access rules, and PaX flags. So far this is very similar to AppArmor.

The POSIX capabilities (listed here), are defined with either a `+` or `-` indicating if they will be allowed or not for the executable. The special `CAP_ALL` represents all capabilities.

Example:

```
 subject /
  ...
   -CAP_ALL
   +CAP_NET_RAW
   +CAP_NET_BIND_SERVICE
 subject /bin/ping
  ...
   -CAP_NET_BIND_SERVICE
```

Resource limitations (listed here) allow to restrict system resources such as memory, CPU, opened files and more. The restriction can either be soft or hard, relying on `setrlimit(2)`, which we'll see in the isolation section.

For instance, to only allow a process to open 3 files.

```
RES_NOFILE 3 3
```

The socket policies are related to which IP addresses, ports, and remote hosts the process can use and communicate with.

```
 connect <IP/host>/<netmask>:<port/portrange> <socket type ↙
    ↳ 1>..<socket type n> <proto 1>... <proto n>
 bind <IP/host>/<netmask>:<port/portrange> <socket type 1>..<socket ↙
    ↳ type n> <proto 1>... <proto n>

 or:

 connect disabled
 bind disabled
```

For example:

```
subject /usr/bin/ssh o
  ...
  connect 192.168.0.0/24:22 stream tcp
  connect ourdnsserver.com:53 dgram udp

  bind eth1:80 stream tcp
  bind eth0#1:22 stream tcp
```

The PaX flags are kernel security features, such as ASLR, which we'll briefly list in the last section of this article on general security.

Practically, GrSecurity is managed through the single command `gradm`, which makes it a breeze. Here's the result of the `--help` flag:

```
> gradm --help
gradm 3.1
grsecurity RBAC administration and policy analysis utility

Usage: gradm [option] ...

Examples:
    gradm -P
    gradm -F -L /etc/grsec/learning.logs -O /etc/grsec/policy
Options:
    -E, --enable    Enable the grsecurity RBAC system
    -D, --disable   Disable the grsecurity RBAC system
    -C, --check Check RBAC policy for errors
    -S, --status    Check status of RBAC system
    -F, --fulllearn Enable full system learning
    -P [rolename], --passwd
            Create password for RBAC administration
            or a special role
    -R, --reload    Reload the RBAC system while in admin mode
                    Reloading will happen atomically, preserving
                     special roles and inherited subjects
    -r, --oldreload Reload the RBAC system using the old method that
                        drops existing special roles and inherited ↙
                            ↳ subjects
    -L <filename>, --learn
            Specify the pathname for learning logs
    -O <filename|directory>, --output
            Specify where to place policies generated from
                        learning mode.  Should be a directory only if
                        "split-roles" is specified in learn_config and
                        full-learning is used.
    -M <filename|uid>, --modsegv
            Remove a ban on a specific file or UID
    -a <rolename> , --auth
            Authenticates to a special role that requires auth
```

```
    -u, --unauth    Remove yourself from your current special role
    -n <rolename> , --noauth
            Transitions to a special role that doesn't
                        require authentication
    -p <rolename> , --pamauth
            Authenticates to a special role through PAM
    -V, --verbose   Display verbose policy statistics when enabling ↙
        ↳ system
    -h, --help  Display this help
    -v, --version   Display version and GPLv2 license information
```

When enabled `gradm -E`, it will parse the policy file and check for security holes, if it finds one then it will refuse to start and list things to fix in the policy.
Once started, only roles that have the admin mode can access and modify the policy file.

```
> gradm -a admin
```

To facilitate the generation of policy, like AppArmor, RSBAC, TOMOYO, and others, grsecurity offers a learning mode which can either be applied as a mode on the subject or globally.

The global learning process is configured in `/etc/grsec/learn_config` with the files and directories that needs protection. Or if applied on a subject, the `l` flag needs to be added.

To enable full system learning, run gradm with the following options:

```
> gradm -F -L /etc/grsec/learning.logs
```

Then let gradm process and propose roles under `/etc/grsec/learning.roles`:

```
> gradm -F -L /etc/grsec/learning.log -O /etc/grsec/learning.roles
```

Similarly, for subject learning mode, the output will also go to the learning log files.

Largely, we can see that GrSecurity is a relatively simple but effective system. Somehow resembling AppArmor but using roles instead of path to perform access control. Yet, under the role matching by UID/GID, it seems to be a one-to-one mapping with AppArmor. The tooling and syntax are also extremely simple, allowing easy management, which is a great plus. Still, the granularity of access on object is rough and not as deep as SELinux and others.
One thing to note, is that GrSecurity seems to be a real RBAC system, where every user is always mapped to a role and only gets privileges through it, even if falling back to the default one. This makes it very solid.

What you need to remember: *GrSecurity is a patch to the Linux kernel adding kernel protection along with a MAC role-based access control mechanism. It has stopped being released in the open since 2017. The roles are a grouping mechanism, matching processed by UID or GID (or defaulting), that contains a list of executables and what files they will be able to access, along with restrictions such as POSIX capabilities, system resources (CPU, memory, ..), network, and more. A role can transition to other roles if specified. The policy syntax is straight forward and the system is managed through a single command* `gradm`.

Capability-Based Security

We talked about capability-based access control in a previous section, in this one, we'll see the concrete forms it can take.

As a reminder, this isn't to be confused with the POSIX capabilities we've seen. Instead, capabilities are abstract atomic, unguessable, and unforgeable objects that embody proof of coarse-grained privileges and are willingly transferred between processes. The capabilities are an inherent part of users/processes.
This assumes that these cannot be acquired out of thin air, but are passed from one process to another. Incidentally, this means that initially one process has all possible capabilities that will be present on the system at one time, otherwise they wouldn't exist on that system. We call the capabilities a process is born with an endowment.

In theory, this should completely remove the need for ACL, yet some systems are pure capability-based while others are hybrid and still contain other mechanisms for access control.

The motto of capability-based security, coming from Norm Hardy, is: *"don't separate designation from authority"*.
In which "designation" means "what we're talking about", and "authority" means "what we're allowed". This is another way to solve the confused deputy problem we talked about in the `su` and `newgrp` section. We shouldn't allow a program executed with certain privilege to do more than we intended it to do, misusing its authority. It's another way to formulate the principle of least privilege, which capability-based security called the Principle of Least Authority (POLA).
It also closely refers to ideas related to our next section: safety-through-compartmentalization.

For instance, in a classic ACL system, we'd open a text editor and ask it to save a file. It'll check, and use, our permission to know if it is allowed to write it on disk, and act accordingly. This means an application that is run by a user, can do anything that user can.
Meanwhile, in a capability-based system, the program has no access by default. When it opens a file, the user has to ask the OS to pass the program a file descriptor representing the file, and not the path, along with what it's allowed to do on it.

In effect, there are myriads of theories on how to apply this, and nobody really agree on what exactly the capability objects take form as, how they are passed between subjects, and how the OS will keep their integrity.
Some envision the capability as a key or token of authority, kind of like a certificate, others as a reference along with access rights, a non-modifiable file-descriptor, or even a label or attribute.

This last one reminds us of attributes on SELinux and RSBAC, attributes along with access rights, yet this time they are living inside processes only, transferable/derived, and not in-between files and processes and enforced globally.

A simple example of an implementation are file-descriptor.

```
1 int fd = open("/etc/passwd", O_RDWR);
```

In the above, the `fd` file descriptor is a capability, but not a very solid or unforgeable one.

Capability-based security is applied in multiple systems, from programming languages, CPU ISAs, web frameworks, network protocols, and operating system access control mechanisms.

On the programming language side, an abstract model has been devised called the object-capability model, or ocap for short, to allow a more standardized approach. It can be used for smart-contracts for instance.

Here's a couple of them, some maintained and others deprecated:

- Act 1
- Eden
- Emerald
- Trusty Scheme
- W7
- Joule
- Original-E
- Oz-E
- Joe-E
- CaPerl
- Emily
- Caja
- Monte
- Pony
- Wyvern
- Newspeak
- Hacklang
- Rholang
- Austral language

There is also some work to add ocap to WASM component model (such as WASI, and to Rust. See Awesome OCAP).

When it comes to networking protocol, the capnproto is a capability-based RPC format, basically allowing passing capabilities along data.
It is used within the sandstorm web application framework to implement capability-based security within a couple of example WYSIWYG applications.

Another cloud platform is the open source Tahoe-LAFS capability-based file system, a decentralized cloud system storage.

Indirectly, this concept is also applied in many web applications. For example, a Dropbox link has all the features of a capability system: permissions, unforgeable, transferable, revocation, etc..
OAuth2 can also allow such mechanism.

On the OS side, we'll see FreeBSD's capsicum soon, but let's mention a few notable examples first.

- Hydra is a capability-based system from the 70s.
- KeyKOS is a pure capability-based OS that has an emulator for POSIX, inspired by Hydra and EROS (another similar system).
- Genode, a capability-based security microkernel.

More modern approaches are Google's Fuchsia with its Zircon kernel, that tags objects with capabilities, and seL4, a high-assurance open-source microkernel providing capabilities.
The seL4 system initially starts by giving all capabilities to all resources to the root task, and then

through derivation and requested operation, other processes are given capabilities indirectly constructed from the root ones. This makes it another pure capability-based system.

In the CPU ISA world, there is research work on the Capability Hardware Enhanced RISC Instructions (CHERI), which, with a combination of hardware and software implements capabilities. It adds instructions to facilitate access control of OS and application code.
A real Unix-like application exist of this project through CheriBSD, a fork of FreeBSD adding support for CHERI-RISC-Vand Arm Morello in emulation and on hardware. The kernel and user space both support a pure or hybrid capability CHERI `C/C++` interface. It achieved this with a new ABI that is mainly used for memory-safety, extending system-call to implement pointers as "CHERI capabilities" instead of integers.
Yet so far, the project targets memory safety more than OS access control and is still in its early phase. The capability permissions are all related to vmmap, execute, load, and store operations, thus applied to CPU ops. Think of it as capability-based security but at the level of the CPU instruction set.

What you need to remember: *Capability-based security is hard to implement. There needs to be an object that is atomic, unforgeable, transferable, that represents the capability. Multiple current solutions exist in programming languages using object-capability (ocap), in CPU ISA (CHERI), and in different OSes such as seL4 and Fuchsia.*

Capability on FreeBSD using Capsicum

FreeBSD's Capsicum is a hybrid capability-based security system, present since the 9.0 release, that uses a refined form of file descriptors.

The extended file descriptors act as capabilities and have been grown with a rich set of permissions, allowing them to be manipulated and extracted from usual POSIX functions. The capabilities allows splitting normal permissions into a smaller set, and then transfer them through the file descriptors via socket and other usual message passing.

This means that file descriptors created by functions such as `open(2)`, `accept(2)`, `socket(2)`, etc.. Can be assigned capability rights. The list of rights can be found in the `rights(4)` man page. It includes a set of names that map to specific sets of functions. For instance, `CAP_READ` and `CAP_WRITE` related to whether it is allowed to read or write on the file descriptor. There are more granular access that can be set underneath certain rights, such as specifics to ioctl when `CAP_IOCTL` capability right is set, and specifics to file control when `CAP_FCNTL` capability right is set.

The rights are always reduced and never expanded.

FreeBSD's approach is hybrid, this means that processes have to willingly opt-in to enter Capsicum capability mode using the function `cap_enter(2)`. When a process enables it, it will stop having access to the global namespace (file system, process tree, networking, etc..), and instead will inherit, or will be delegated, only what is needed from the capability rights.

To enable this feature, the following kernel option need to be set:

```
options CAPABILITY_MODE
options CAPABILITIES
```

Since it's an opt-in feature, it requires source-code modification from programs. This is similar to OpenBSD's `unveil`/`pledge` which we'll see in the isolation section.

To facilitate this, the header `<sys/capsicum.h>` includes functions to create capability-aware software.
Functions such as `cap_enter(2)` (`cap_getmode(2)`) and `cap_rights_init(3)`, which initializes the `cap_rights_t` structure, and multiple functions to limit and fetch the current rights, such as `cap_rights_limit(2)` and `cap_rights_get(3)`.

Here's an example from the man pages, to limit the capability on the file descriptor to only allow reading:

```
cap_rights_t setrights;
char buf[1];
int fd;

// open a file with read-write
fd = open("/tmp/foo", O_RDWR);
if (fd < 0)
    err(1, "open() failed");

// enable capability mode
if (cap_enter() < 0)
    err(1, "cap_enter() failed");

// only allow reading
cap_rights_init(&setrights, CAP_READ);
if (cap_rights_limit(fd, &setrights) < 0)
    err(1, "cap_rights_limit() failed");

// try to write something, it will fail
buf[0] = 'X';
if (write(fd, buf, sizeof(buf)) > 0)
    errx(1, "write() succeeded!");

// but reading will still work
if (read(fd, buf, sizeof(buf)) < 0)
    err(1, "read() failed");
```

The failed operation on the file-descriptor will return `ENOTCAPABLE`.

There are easier libraries, such as `libcapsicum(3)`, with functions like `cap_init`, `cap_service_open`, `cap_wrap`, `cap_unwrap`, `cap_limit_get`, `cap_limit_set`, and more. This library relies on the `casperd(8)` daemon that hosts "services" that can be accessed by the capabilities. It acts as a sort of proxy for functionalities that needs to be accessed from the outside world from within an isolated environment, similar to what D-Bus and polkit do (with desktop portal), as we'll see in the action-based access control section.

The `casperd` comes with at least these services in FreeBSD 11 and above:

- `system.dns` - provides API compatible to:
  - `gethostbyname(3)`
  - `gethostbyname2(3)`
  - `gethostbyaddr(3)`
  - `getaddrinfo(3)`
  - `getnameinfo(3)`
- `system.grp` - provides `getgrent(3)`-compatible API
- `system.pwd` - provides `getpwent(3)`-compatible API
- `system.random` - allows to obtain entropy from `/dev/random`
- `system.sysctl` - provides `sysctlbyname(3)`-compatible API

We'll see more of this mindset in the isolation section.

Since Capsicum requires software modification, popular software need to be patched accordingly. A few notable software were used as conceptual tests: Chromium, `tcp-dump`, `gzip`, `dhclient`, and more. Still, it's hard to have all software in a base system capability-aware, this means the hybrid mechanism will stay in place.

Compared with MAC like SELinux, the opt-in mindset is more flexible and consistent, but limits itself to the programs that have it. Such hybrid systems is thus complementary with a MAC, and cannot be secure by itself.

Besides, Capsicum is subtitled: *"lightweight OS capability and sandbox framework"*, it also acts as a sandbox, which will be the aim of the next part of this article. Because it is hybrid and is used to compartmentalize applications on a need-basis, it isn't controlled system-wide, there's no global rule-book.
Unlike seL4, it doesn't inherit capability rights from an all-encompassing root task, it isn't pure.

A solution that arose is to have a Capsicum program manager that wraps other non-Capsicum programs in a sandbox. This is exactly what `capsicumizer` does.

`capsicumizer` is a sandbox launcher relying on Capsicum capability mode to restrict programs without performing any source code modification. All restrictions are done externally.
It allows writing profiles similar to AppArmor (see AppArmor section) to limit the scope of programs.

Lastly, let's see how Capsicum works across systems other than FreeBSD.

There exists patches to the Linux kernel (ported by Google, relying on `seccomp-bpf` we'll see in the next section), to NetBSD, and DragonFlyBSD porting the Capsicum mechanism. However, most of them are currently unmaintained.

One such system that is now deprecated but was multi-platform and based on Capsicum was Nuxi CloudABI. It was a mix of capability-based security and POSIX and removing everything incompatible with that. A more pure capability-based security system than what FreeBSD has.
The name comes from how it is useful to isolate networked services in a cloud environment.

We're on track to move to the topic of isolation.

What you need to remember: *FreeBSD's Capsicum augments normal file descriptors allowing to add capabilities on them via functions. Programs have to explicitly call function such as* `cap_enter`

to enable capability mode and restrict themselves. Afterward, they can limit what they can access with `cap_rights_limit` *and other functions. Since it acts on a opt-in basis, the FreeBSD capability is hybrid and is mostly used as a sandbox tech. Solutions such as* `capsicumizer` *allow isolating processes by relying on Capsicum.*

Putting in Boxes: Isolation and Constraints as Access Control

We've seen a substantial number of mechanisms to apply security policies over Unix-like systems, each with a different philosophy. In this section we'll emphasize OS features that are meant specifically to isolate software, contain and constraint them.
Undoubtedly, anything we've seen thus far could be used to "isolate" software, such as AppArmor, SELinux, Capsicum, POSIX capabilities, etc.. However, we'll emphasize on the idea of limiting the scope of what an application can see of the rest of the system, real isolation. That translates into facilities that make a process believe it is alone, or has limited view of the system.

This concept isn't new, we encounter something similar, away from the security world, when discussing process concurrency with virtual memory address space and concurrent tasks. Additionally, we could say that processes owned by one users are somewhat isolated from another user, as they can't be manipulated by others. Yet, they are usually visible, which isn't ideal. The same goes for files.
Isolation and constraint as access control goes further than this.

The intention to use such mechanism instead of the system-wide ones comes from a pragmatic place. In today's world, user systems are exposed to hundreds of thousands of packages written by a legion of authors, leading to an increase in complexity when it comes time to administer a system security policies. The word "supply chain" attack has surfaced to describe a security issue that emerged from a dependency in an unchecked package.
For that reason, a simpler solution has materialized: isolate specific software, and rely on a system-wide policy for the rest. In that scenario, if that particular isolated software is breached, it shouldn't have a big impact.

There are three words that need to be defined in the lingo around the topic of isolation: **virtualisation**, **container**, and **sandbox**.

The word **virtualisation** is used to describe anything that is a virtual version of something physical. Away from security, we have the concept of virtual memory, for example.

The word **container**, or containerization, is a case of OS-level virtualisation. Instead of having the hardware virtualised, containers virtualise the user-space environment in an OS. An application in a container will think it is the only application running. A distinct aspect of containers is that they share the underlying kernel for efficiency, and rely on OS features for the isolation.
Nowadays containers are used as lightweight standalone environments to run microservices. Almost all the common solutions follow the OCI standard, the open container initiative.

The word **sandbox** has multiple meaning, all of them security related.
The first definition is a test environment in which security analyst can monitor potential security issues.
The second definition is a simulated environment, which if broken, would not let the attacker break into the wider machine, but still make them believe they are within a real environment.
The third definition, which is the definition we'll use, is about placing a process within virtual walls to prevent breaking into the system. The walls are the sandbox.

Various logic embody this new isolation philosophy.
A typical description would refer to it as a Domain Type Enforcement (DTE), categorizing users/pro-

grams/data into domains which are protected from one another. Somewhat similar to the type labels used in SELinux.
Rather than DTE, the most notorious names for this scheme is *safety-through-compartmentalization*.

Within all this, a question of design comes up: who should perform the isolation? To which there are two answers: **self-isolation**, or smart-sandboxing, and **oblivious-isolation**, aka sysadmin-style isolation, aka external isolation, aka dumb-sandboxing.
In the self-isolation case, software need to be updated to include in their code features that allow them to limit what they are capable to do on the whole system. Meanwhile, in oblivious-isolation, a separate program is called that will invoke the one actually wanted and wrap it in a box, a sandbox.

This mindset is also popular on the OpenBSD Unix-like OS under the name "privsep & privdrop", privilege separation and privilege dropping/revocation. A motto that says that programs should always self-reduce/isolate their attack surface, dropping privileges as soon as they're not needed (usually accomplished by switching UID, but no only), otherwise separate/split them into different programs performing sub-functions. Indirectly, this creates isolated security domains.

Nonetheless, all the progress made with multitasking and software reuse would go to waste if we went back to old-school systems on which programs can't interact with each others. That's why, even though software are in boxes, there should still be dedicated and formal openings, IPC (inter-process communication) with the outside. This is a topic we'll cover in the action-based access control.

What you need to remember: *It is hard to manage a system-wide policy, instead an approach to isolate specific software in sandboxes is easier. The software should be isolated, either willingly (self-isolation) or via another software (oblivious-isolation), within the confine of virtual walls that prevent it from accessing anything other than what it is intended to.*

Classic Constraints

Resource Limits

We'll start with classic configurations and methods to limit the resources used, be it for a user, process, group, project, and others. Some of the following can either be set system-wide or on a per-process basis. These aren't necessarily related to isolation, but this should get us going in the right direction.

Niceness

The niceness of a process is a value used by the OS scheduler to decide how it will prioritize tasks. It ranges between -20 and 19, the lower the value, the more it will be prioritized.
In general, only the super-user can increase and decrease the priority of processes, whether it owns them or not. For other users, they can only decrease the priority of processes they own, and this

change is irreversible. There even exist flexibility regarding this, as we'll see in the next section, if a processes has a limit of the category `RLIMIT_NICE`.

The interpretation of the priority depends on the scheduler currently in use by the OS. However, in general if the niceness is 19, it means that the process will only run when nothing else in the system needs to.

The commands `nice(1)` is used to launch a process with a niceness level, `renice(1)` to modify the niceness of a currently running processes. These rely on the functions `nice(2)` (C library) and `setpriority(2p)` (POSIX).

There also exist other priority schemes such as the one offered by the "Real-time Extensions" of POSIX 1b standard, manipulated through commands such as `chrt(1)`. However, we'll skip that particular topic here.

What you need to remember: *Niceness, a value between -20 and 19, lets the scheduler decide how to prioritize tasks. -20 is the highest priority and 19 the lowest. A normal user can only lower the priority of processes they own.*

ulimit, rlimit, and sysctl Tunables The POSIX and C library functions `ulimit`, `setrlimit`, `getrlimit` (ancient version of `vlimit`) allow to set per-process resource limit consumption.

The POSIX `ulimit` function, and related utility of the same name, is used to impose a limit on the maximum file size that can be written by a process, and only that.
However, the POSIX version of `ulimit` is barely used and instead the `getrlimit`, `setrlimit`, and Linux specific `prlimit` have replaced it with a wider range of resource limitations.

These functions control the maximum resource consumption through soft and hard limits. The hard limit is the ceiling enforced by the kernel that cannot be changed by a process, while the soft limit allows a process to have some wiggle room under the hard limit. A process can lower its hard limit, but it is usually irreversibly.
The same rule related to the niceness applies here, only privileged users can raise their hard limits and change the limit of processes they don't own using `setrlimit`. On Linux this takes the form of a POSIX capability named `CAP_SYS_RESOURCE`.

The resources that can be controlled are passed as flags, starting with `RLIMIT_<resource>`, to `setrlimit`. There's a multitude of them, often varying between systems. However, they usually include the following:

- `RLIMIT_CORE`: Maximum size of a core file.
- `RLIMIT_CPU`: Maximum amount of CPU time.
- `RLIMIT_DATA`: Maximum size of data segment of a process.
- `RLIMIT_RSS`: Maximum size of a process RSS.
- `RLIMIT_STACK`: Maximum size of the initial stack of a process.
- `RLIMIT_AS`: Maximum size of total memory for a process.
- `RLIMIT_FSIZE`: Maximum size of file in bytes that can be created. (instead of `ulimit`)

- `RLIMIT_MEMLOCK`: Maximum size of locked-in-memory address space.
- `RLIMIT_NOFILE`: Maximum number of file descriptors
- `RLIMIT_NPROC`: Maximum number of simultaneous process for a user.

Note that on Linux, the manpages related to limits such as `getrlimit(3p)` aren't up-to-date and it's instead better to look directly in headers such as `/usr/include/bits/resource.h`.

These functions are accompanied by command line tools that allow to set these values on processes.

Generally, `ulimit` is a shell built-in command, which instead of only controlling the maximum file size, allows to set any of the above mentioned resources.

On Linux, the command `prlimit` can be used to get/set resource limitation while invoking an executable, or on already running processes by specifying its PID.

```
> prlimit
RESOURCE   DESCRIPTION                             SOFT      HARD UNITS
AS         address space limit                unlimited unlimited bytes
CORE       max core file size                 unlimited unlimited bytes
CPU        CPU time                           unlimited unlimited seconds
DATA       max data size                      unlimited unlimited bytes
FSIZE      max file size                      unlimited unlimited bytes
LOCKS      max number of file locks held      unlimited unlimited locks
MEMLOCK    max locked-in-memory address space unlimited unlimited bytes
MSGQUEUE   max bytes in POSIX mqueues            819200    819200 bytes
NICE       max nice prio allowed to raise            31        31
NOFILE     max number of open files                1024    524288 files
NPROC      max number of processes                21353     21353 processes
RSS        max resident set size              unlimited unlimited bytes
RTPRIO     max real-time priority                    98        98
RTTIME     timeout for real-time tasks        unlimited unlimited microsecs
SIGPENDING max number of pending signals          21353     21353 signals
STACK      max stack size                       8388608 unlimited bytes
```

The equivalent command on FreeBSD is called `limit(1)`, it achieves the same thing but with different parameters.
On SunOS derivatives, the equivalent utility is `prctl(1)`, however, it does much more than this, as we'll see in the SunOS projects section.

Another place where resource limitations can be set, this time system-wide, is through kernel tunables via `sysctl`. The type of resource limit that is configurable depends on the OS in use.

On Linux, issuing the command `sysctl -a` will list all the current tunables, which include resource limitations such as: `fs.file-max`, `kernel.pid_max`, etc.. The same command can be used to dynamically change these tunables, or they could also be changed through the pseudo-fs `/proc/sys`, or via `sysctl.conf`.

The situation is similar on BSDs such as FreeBSD and OpenBSD with the `sysctl` utility but also with a function of the same name.
For instance on FreeBSD:

- `kern.maxvnodes`
- `kern.maxproc`
- `kern.maxprocperuid`
- `kern.maxfiles`
- `kern.maxfilesperproc`

And on OpenBSD:

- `kern.maxfiles`
- `kern.maxlocksperuid`
- `kern.maxpartitions`
- `kern.maxproc`
- `kern.maxthread`
- `kern.maxvnodes`

For all the above to be useful, it would be neat to be able to easily set them per-user or per-login. On Linux this is achieved with PAM, using the `pam_limits` plugin, while on BSDs the `login.conf` capability database is used.

The `pam_limits` plugin is configured in the `/etc/security/limits.conf` file, and allows to set any of the resource limit we've previously mentioned, either as soft or hard limit.

Similarly, the capability database `login.conf` that we've seen in the BSD Auth section, includes attributes to assign resource limitations to classes. For instance:

```
default:\
    :nologin=/var/run/nologin:\
    :filesize=unlimited:\
    :openfiles=unlimited:\
    :maxproc=unlimited:\
    :umask=022:\
    ...
```

We need to mention `rctl(8)`, a neat flexible runtime resource control mechanism present on FreeBSD that is used to more easily manage what we've cited.
It relies on the `/etc/rc.d/rctl` service, that applies resource limitations configured within the `/etc/rctl.conf` configuration file to specific subjects (users, login class, jails). The rules set in the `rctl.conf` files are accompanied by action to take if a resource limit is reached, which can be either about denying, logging, notifying, and more.
To enable it, the kernel needs to have the options `RACCT` and `RCTL`.

Lastly, Solaris offers an analogous mechanism to the above FreeBSD one with its resource control, also referred to as `rctl`, with its `rctladm(8)`. While Solaris does offer functions such as `getrlimit`/`setrlimit` it extends them with a new `getrctl(2)`/`setrctl(2)` that allows more flexibility in the assignment of resources. This allows to associate resources not only to processes but also to "tasks" and Solaris "projects", which we'll discover in another section (a project is a set of tasks), and assign actions such as "allow", "deny", and "signal" when a resource limit is reached.

The resources are specified as strings that are flagged with levels: basic, privileged, and system controls. These flags specify what access rights are needed to control their values, and they can be attached to the resource value as needed. These strings are often prefixed with the `idtypes` which specify to what the resource is applied to: `process.`, `task.`, `project.`, or `zone.`.
The type of resources include most of the ones we've seen above, such as `max-cpu-time`, `max-shm-memory`, etc.. The list can be found in `resource_controls(7)`.

These can be set in multiple ways, either programmatically with `setrctl(2)`, system-wide in `/etc/rctladm.conf`, or with the command line `prctl(1)`. For example, here's a truncated output showing current resources limits of different `idtypes`:

```
> prctl -i process 136150
136150: /bin/ksh
NAME    PRIVILEGE       VALUE    FLAG   ACTION              RECIPIENT
task.max-cpu-time
        usage            8s
        system          18.4Es    inf   none                 -
task.max-processes
        usage              30
        system          2.15G     max   deny                 -
project.max-tasks
        usage               2
        system          2.15G     max   deny                 -
project.max-processes
         usage             30
        system          2.15G     max   deny                 -
zone.max-processes
        system          2.15G     max   deny                 -
zone.max-locked-memory
        usage              0B
        privileged       508MB       -   deny                 -
```

Solaris also offers something called a resource pool, which is used to group cpuset with scheduler, calling them together a pool. We'll see this in another section.

What you need to remember: *There exist resource limitation functions such as `ulimit` and `setrlimit`. What they limit depends on the OS but that usually contains memory, files, and cpu usage as resources. Resources limitations can also be done at the kernel level via tunables set with `sysctl`, this also depends on the OS. All these can be managed either with a PAM plugin (`pam_limits`) to set limits on login, or via `login.conf` capability database on BSDs. FreeBSD includes a neat runtime management of these with `rctl(8)`, a similar mechanism exists on Solaris.*

File System Quotas The last classic resource control we'll take a look at, that is more or less standard across Unix-like OS are file system quotas. Quotas are a feature that can limit the number of files (inodes) or disk space (block) used by users, groups, or "projects", with a soft and hard limit. The soft limit in that case is used as a grace period/ceiling.

On FreeBSD this should be enabled with both the kernel option `QUOTA` and with the `rc.conf` configuration `quota_enable="YES"`. Additionally, it needs to be set on a per-file-system basis in the `/etc/fstab` entries, adding a line for the related quota. For example, to enable user and group quotas:

```
/dev/da1s2g    /home    ufs rw,userquota,groupquota 1 2
```

The quota files will be stored in `quota.user` and `quota.group` in the root directory `/`.

In the same vibe, on Linux, this feature can be enabled per-file-system, either on creation during `mkfs` with `-O quota`, or on existing file system (after unmounting) with `tune2fs -O quota`.
The quotas are also stored in files with the same name as FreeBSD, but sometimes prepended with an `a`, like `aquota.user`.

On both these systems, the `quotaon`/`quotaoff` commands exist to perform the above enabling and disabling, instead of performing the changes manually.

They also share the following quotas-related commands:

- `quota(1)`: display quota and limits
- `edquota(8)`: edit user/group/project quota (editor)
- `setquota(8)`: set disk quotas for user/group/project
- `repquota(8)`: report quota usage
- `quotacheck(8)`: scan an fs for quota
- `warnquota(8)`: perform action when a quota (`/etc/quotatab`,`/etc/warnquota.conf`)
- `quotastats`: query quota statistics

```
> quotastats
Kernel quota version: 6.5.1
Number of dquot lookups: 0
Number of dquot drops: 0
Number of dquot reads: 0
Number of dquot writes: 0
Number of quotafile syncs: 38
Number of dquot cache hits: 0
Number of allocated dquots: 0
Number of free dquots: 0
Number of in use dquot entries (user/group): 0
```

A less used, but useful, Linux feature of quota is its concept of project quotas. Its support depends on the file system in use.
Projects are defined by names associated with IDs. These IDs can subsequently be assigned to directories to tag them as part of a project. This allows setting a quota on a particular directory or group of directories.

This needs to be configured at multiple levels. First of all, the file system needs to have project quotas enabled, either via `tune2fs` or with `mkfs`. Then be sure to mount the file system with the project quota option. (example from SO)

```
> tune2fs -Q prjquota  /dev/loop0
> tune2fs -E mount_opts=prjquota /dev/loop0
```

Secondly, the projects need to be added in the file `/etc/projects` (mapping ID to names) and `/etc/projid` (mapping names to ID), this isn't mandatory as no real tool seems to use them. Here we create a project called `testproj` with id `51`.

```
> echo testproj:51 >> /etc/projid
```

Thirdly, we need to assign the project ID to some directory as an extended attribute.

```
> chattr +P -p 51 abc
```

Finally, we can set a hard block usage limit of 1024 on the file system we mounted for the `testproj` we just created.

```
> setquota -P testproj 0 1234 0 0 /mnt/loop/
```

```
> dd if=/dev/zero of=someoutput oflag=append

loop0: write failed, project block limit reached.
dd: writing to 'someoutput': Disk quota exceeded
2471+0 records in
2470+0 records out
1264640 bytes (1.3 MB, 1.2 MiB) copied, 0.00985608 s, 128 MB/s
```

Yet, this can trivially be escaped by changing the project attribute on the directory. So quotas are useless if you can break from them.

What you need to remember: *File system quotas are limits on the disk space or inode usage on a per-user, per-group, or per-project basis that should be enabled on each mounted disk. Quotas projects is a method of tagging specific directories with quotas.*

`chroot`

The `chroot` function, with its command of the same name, changes the apparent root directory `/` of a process and its children to one picked by the invoker. The modified environment the process runs in is called a "chroot jail".
This mechanism, after calling `chroot`, translates into a process having a different file system hierarchy, one that is a sub-directory (sub-tree) of the initial process that called `chroot`. It hijacks how the path-resolution is done by that chrooted process. Indirectly, this means that a process from outside the chroot jail can always access files that are in-use within the chroot jail.

Historically, `chroot` dates from Unix V7 and was used to run programs in a compatibility mode with another system, in this case V6. It was part of the `chdir` code, changing what field is acted on.

In general this can be used for system maintenance, during the booting of a system, for containerization, for running untrusted programs (with limitations as we'll see), to have a clean environment for testing, to try different versions of an OS, to test different architectures (on Linux using `personality`), and more.

Nevertheless, `chroot` does not virtualise any other aspect of the system, such as the memory, networking, the process tree, or devices, and thus might be less secure than other solutions we'll see later.

Only a privileged user, root-privileges, can invoke chroot. On Linux that takes the form of a required POSIX capability named `CAP_SYS_CHROOT`. This is intended as a weak security measure to prevent users inadvertently crafting chroot jails that contain malicious setuid programs, leading to privilege escalation.

Yet, this isn't the only security issue with `chroot`. While the name "jail" implies a process cannot get out of the `chroot` environment, this is a misconception. The only thing that it does is to change the path resolution, and nothing else.
Software outside the `chroot` jail can interact with files from within, and move them outside the new root, leading to processes from within the `chroot` jail trivially bypassing it. In other words, if a process within a `chroot` jail waits in a directory that was meant to be moved until it is actually moved, and then issues calls to change its path or reads files outside, it will have access to the parent system.

Another issue, particular to Linux, is that the `chroot(2)` function doesn't change the current directory of the parent process. Thus `.` can be outside the rooted `/` right after the call. For example:

```
1 mkdir foo; chroot foo; cd ..
```

That means we haven't moved within `foo` yet, and `.` still points to the current parent directory, so we can escape from it.
Thus, with `chroot` we always have to keep in mind the file descriptors that are still opened and accessible from outside the new root.

Nonetheless, if privileges are dropped properly after entering the new root, and the environment is clean of any potential `setuid`/`setgid` executable that could create issues, then the `chroot` jail isn't such a bad file system compartmentalization solution.
Indeed, this is exactly what `chrootuid(1)` does. This command is a mix of `chroot` and `su`, entering the new root and dropping privileges to the user specified (it also must be an account that exists in the new environment).

Alternative tools try to avoid the need of root privileges, either wrapping `chroot`, or simulating its behavior.

For example, this is the case of `fakeroot` on Linux, which relies on hijacking `LD_PRELOAD` `chroot` calls and fakes the result to simulate `chroot` as a regular user.

Another tool called `schroot`, secure `chroot` the successor of `dchroot`, is a utility that allows `chroot` as a normal user.
It manages the permission checking and the setup of the `chroot` environment, mounting additional file systems and setup configuration for the new root. The configuration `schroot.conf(5)` stored in `/etc/schroot/schroot.conf` or `/etc/schroot/chroot.d/` contains the location of the `chroot`, along with which users and groups can access it, the architecture involved (`personality` aka process execution domain or how to map system calls number to action), initialization scripts, and more.

Lastly, there are two `chroot`-related tools used mostly during boot on Linux systems that are called `switch_root` (for `initramfs`) and `pivot_root` (`initrd` and anything after the system is mounted).

The command `pivot_root` is used with `docker` to avoid certain privilege escalation methods. What it does is sort of like a double `chroot`, it moves the current root file system within a new root, and sets the new root as the current root. This means it keeps the parent process' root present within the `chroot` jail within a directory (Here it's not really a `chroot` jail yet, but a pivoted root).
Hence, if the old root is unmounted (it is mounted in a bind namespace, kind of like symlinks as mount points, something we'll see in another section), and `chroot` is called afterward, it makes the outside world inaccessible. Removing most of the issues with breaking out of the `chroot` jail.

Even so, it's only the root of the file system that is abstracted, everything else is still shared, including the process tree and memory. This is why we need more than that to isolate processes.

What you need to remember: *`chroot` is a function and command that changes the root of the file system by hijacking how the path is resolved for the processes invoked afterward. The environment they run in is called a `chroot` jail, yet it isn't a real jail and could trivially be bypassed. It only virtualises the root of the file system and nothing else. Only root can call `chroot` but there are solutions to allow normal users to use it such as `schroot`. For better isolation privileges need to be dropped properly, such as with `chrootuid`.*

Isolation on OpenBSD

`systrace`

An interesting early oblivious/dumb-isolation software attempt on OpenBSD was called `systrace`. It was dropped after being unmaintained, and in favor of self-isolation solutions such as `unveil`/`pledge`, from OpenBSD 6.0 in 2016 (the current version of OpenBSD as of this article is 7.2).
The project also has compatibility with Linux, however, as with the rest of the project, it isn't maintained.

The `systrace` framework is made to act as a wrapper to executable, enforcing policies on system calls. It achieves this by using a special device `/dev/systrace` which interfaces between processes, the policy, and the kernel.

Additionally, `systrace` can also be used to generate and trace the behavior of programs, kind of like one of the definitions of sandbox that we've seen: to trace untrusted applications.

Furthermore, this tracing behavior can be used as a learning mode to interactively generate access policies. It works by having an agent wait for notifications/alarms from `systrace` and ask the user to take a decision: to allow or not the system call. A graphical agent exist called `xsystrace(1)` which can also work in text mode with the `-t` parameter.
This is all very similar to learning modes we've seen with TOMOYO, AppArmor, GrSecurity, and RSBAC.

The `systrace` policies are defined either system-wide in `/etc/sytrace` or in the user's home in `$HOME/.systrace`.

A pofile consists of a series of system calls (ex: `native-fsread`) followed by a colon `:` and a filter, along with a condition/predicate and subject it will be executed as.

Here's an excerpt from the grammar of policies filter:

```
filter = expression "then" action errorcode logcode
expression = symbol | "not" expression | "(" expression ")" |
    expression "and" expression | expression "or" expression
symbol = string typeoff "match" cmdstring |
    string typeoff "eq" cmdstring | string typeoff "neq" cmdstring |
    string typeoff "sub" cmdstring | string typeoff "nsub" cmdstring |
    string typeoff "inpath" cmdstring | string typeoff "re" ↙
        ↳ cmdstring |
    "true"
typeoff = /* empty */ | "[" number "]"
action = "permit" | "deny" | "ask"
errorcode = /* empty */ | "[" string "]"
logcode = /* empty */ | "log"
```

Basically, the filter is composed of an expression on the left capturing a certain parameter of the system call (ex: `filename eq "/tmp"`), then on the right the action to take when the expression is true, whether to `permit`, `deny` or `ask` what to do.

The condition that can be added afterward starts with `if` and can apply on users, groups and others, while the execute-as effective uid and gid are specified with:

```
as user
as user:group
as :group
```

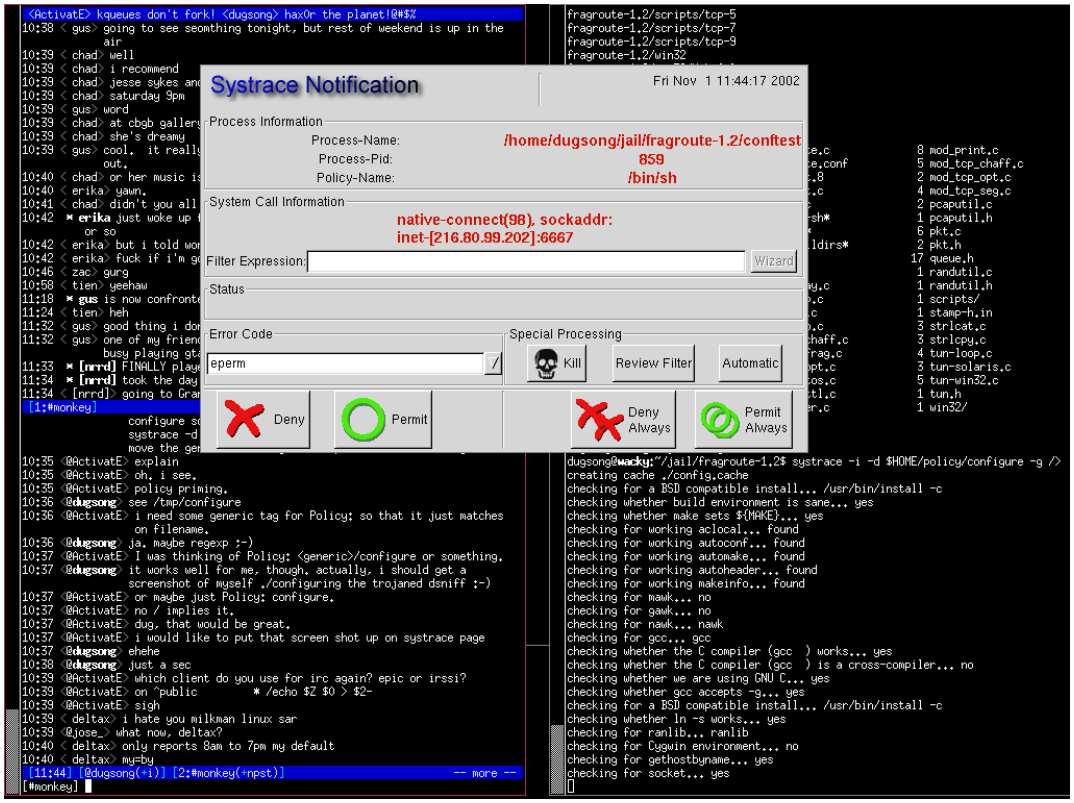
Keep in mind that these will only be available for the duration of the system call and the effective values will be reverted afterward. This feature can be used to replace `setuid`/`setgid` as a precise temporary privilege elevation feature.

Here's an example of a policy from the manpage.

```
Policy: /bin/ls, Emulation: native
[...]
   native-fsread: filename eq "$HOME" then permit
   native-fchdir: permit
[...]
   native-fsread: filename eq "/tmp" then permit
   native-stat: permit
   native-fsread: filename match "$HOME/*" then permit
   native-fsread: filename eq "/etc/pwd.db" then permit
[...]
   native-fsread: filename eq "/etc" then deny[eperm], if group != ↙
       ↳ wheel
```

And another example with network sockets:

```
native-bind: sockaddr eq "inet-[0.0.0.0]:22" then permit as root
```



Source: Systrace - Interactive Policy Generation for System Calls

As you can see, `systrace` is a sandbox with an approach that is dynamic and relatively easy. However, a need arose from users to have a centralized repository of pre-generated user-suggested policies for common software, the equivalent of SELinux reference policy. It took the form of something called the Hairy eyeball Project. However, `systrace` and its hairy reference policy quickly lost traction and the project became unmaintained as it took too much effort to recreate policies on every application change.

In 2015 an attempt was made to revamp `systrace` to perform privilege separation, however it didn't last long.

Instead, OpenBSD took another turn with self-isolated software. The mindset being that software should always be segregated to only what they require, regardless of external policies being enabled or not. This is what we'll see with `unveil` and `pledge` in the next section.

What you need to remember: *`systrace` is a now unmaintained oblivious-isolation software, a wrapper that applies policies on system calls to sandbox software. Its policy file can be generated either manually or with the help of an interactive learning mode. The syntax is relatively simple, mapping system calls with parameters expression and actions to take. Each line can also be accompanied with a predicate and a uid or gid to execute the system call as. This last feature can be used to replace `setuid`/`setgid`.*

`unveil` & `pledge`

While `systrace` was enforcing policies externally, `unveil` and `pledge` enforces them through code, with self-isolation.
Self-isolation implies that it is an integral part of the application, tied with it, and thus cannot be disabled or removed. Practically this takes the form of two system calls named `unveil` and `pledge`.

Let's note that `pledge` and `unveil` were ported to Linux by Justine Tunney as a command line utility and a C API, by relying on similar Linux features such as `seccomp` and `landlock` which we'll see later.

The `unveil` system call is used to restrict the view of the file system by creating a whitelist of paths, while `pledge`'s job is to restrict system calls and features of the OS.

The signature of `unveil(2)` is as follows:

```
1 int unveil(const char *path, const char *permissions);
```

The first call to `unveil` activates its feature and makes anything else, apart from what is set in its arguments, invisible to the process. The permissions are the usual read-write-execute along with a `c` for creation rights. Any file outside of the path specified will be seen as non-existent and access will be denied.
A last call to `unveil` can be performed with two `NULL` argument to disallow further `unveil`ing.

In a way it is similar to `chroot`, however it should be deliberately done by the programmers themselves, trusting them.

For example:

```
1 unveil("/tmp/file", "r"); # activate unveil,
2                           # can only see /tmp/file
3 unveil(NULL, NULL);       # no more unveil afterward
```

As you can guess, it's a terrible idea to `unveil` arguments passed from a user such as `argv[]`.

The signature of `pledge(2)` is also simple:

```
1 int pledge(const char *promises, const char *execpromises);
```

A pledge is a promise that only the feature set found in the `promises` list will be used by the program. Whenever anything else is accessed, the program dies. Once a `pledge` is made, no more abilities can be gained, they can only be restricted more.

The promise takes the form of a space-separated string that contains named-sets of predefined groups of system calls. These features are categorized into computation, memory management, read-write operations, opening files, networking, and more. For example, there are sets such as `rpath` related to reading path, `wpath` related to writing to a path, `audio` to manipulate audio input-output, etc..

The second parameter of `pledge` called `execpromises` only makes sense when the `exec` promise is put in place, and it will contain the inherited promises that the child promises will have.

If two `NULL` values are passed to `pledge`, nothing happens.

Here's an example of a `pledge` usage:

```
1 pledge("stdio rpath", NULL)
```

Finally, the `ps(1)` utility does offer keywords allowing to display the current pledges via `pledge`, and can display in the state (`stat`) column info about the unveil and pledge locking state.

Overall, `unveil` and `pledge` are straight forward self-isolating solutions, making privilege separation easier and reducing the attack-surface (OpenBSD's security motto). However, the programmer needs to sandbox everything themselves. The programs need to manipulate their own future runtime. Meanwhile, having them as system call makes it efficient for kernel processing.

Since these are features that are willingly implemented by each software, the number of them having adopted it is still relatively sparse. Some notable examples are the Chromium browser, OpenSSH, go, spamd, mount, ping, openssl, rsync, tmux, etc.. Yet, this approach can become increasingly complex with certain software.
Another method would be to have `unveil` and `systrace` exist as command line tools taking as parameter their argument for the `execpromises` and whitelisted path, then calling an executable in a sandbox. Indirectly recreating an oblivious-isolation environment.

What you need to remember: *`unveil` and `pledge` are system calls used to whitelist file paths, and restrict system calls sets. This is a self-isolation approach in which software developers have to edit their programs to create privilege separation in the future runtime of the process. `unveil(2)` takes the path to whitelist while `pledge(2)` takes a list of set-features to allow.*

Isolation on FreeBSD

Capsicum as a Sandbox

The Capsicum, hybrid capability-based security implementation, we've dealt with in an earlier section can be seen under a new pair of eyes as a self-isolation mechanism.

We won't dwell on it too much other than mentioning that it is familiar with the `pledge` system call of OpenBSD. However, instead of applying a family of rights, with Capsicum we apply these permissions on file descriptors themselves. Thus the constraints are limited to these file descriptors, and not the view of the whole system.

As with `pledge`, the adoption heavily depends on the software package maintainers and the patches they can apply. With Capsicum it is even more complex to add these features than with OpenBSD's `pledge` as the rights aren't grouped into sets.

This leads us to the same conclusion as with `unveil` and `pledge`, that instead another wrapper could be used as an oblivious-isolation solution. In this case `capsicumizer` exists exactly for this purpose.

What you need to remember: *Capsicum, previously seen in another section, can be viewed as a self-isolation technology similar to OpenBSD `pledge(2)` but instead applied to extended file descriptors. Likewise, `capsicumizer` can be used as an oblivious-isolation wrapper.*

FreeBSD jail extends on top of `chroot` by virtualising much more than just the file system root. It creates sophisticated segregated environments for processes where they have their own process tree, users, networking stack, and limitations on system resources and capabilities, a real "inescapable" jail and not a "`chroot` jail".

In general, there are two categories of jails: either full systems or isolated services. Yet, this distinction only matters when it comes to building the jail environment.

The first step is to fetch the files needed and set them in a directory that will be used as the jail file system. In a way, this step is identical to `chroot`: we need to recreate a file system for what we want to run.

On FreeBSD there are multiple ways to do that. Since it's a source-based distributions, the system installer can be used to build a base tree:

```
1 bsdinstall jail /locationofjail
```

These specific files will need to be built, since they are only source:

```
1 > make buildworld
2 > make installworld DESTDIR=/locationofjail
3 > make distribution DESTDIR=/locationofjail
```

Similarly, any other mean can be used to achieve this, such as an extracted ISO, or skeleton tree, or an online project, etc..

This also means that the user will have to maintain this sub-system up-to-date, just like they keep their main system up-to-date. For that reason, it's better to use the usual FreeBSD base and keep relying on the system update facilities.

```
1 > freebsd-update -b /here/is/the/jail fetch
2 > freebsd-update -b /here/is/the/jail install
```

A problem that might arise from having so many similar directories with full systems in them is the amount of redundancy and space they will take. An easy solution to this would be to keep a read-only symlink farm for these jails. Another would be to use union/overlay-mount of mount-bind such as `nullfs(5)`.

Once the directory is setup with a file system, it can be used as a jail, just like it could've been used as a `chroot` environment.
There are three ways to start a jail, either manually on the command line by passing all the params we require, either with a command line but putting the params in a configuration file `/etc/jail.conf`, or as a service at boot time relying on `rc.conf`.

The `jail(8)` administration utility is used for the first two launching methods. Its `-c` flag is used to create new jails, `-m` to modify, `-r` to remove, and `-e` to exhibit a list of all jails.

The `-c` creation flag requires at least the following 4 parameters, the path of the jail, the hostname given, the ip address, and the command that will be executed at the start of the jail.

```
> jail -c path=/data/jail/testjail mount.devfs \
  host.hostname=testhostname ip4.addr=192.0.2.100 \
  command=/bin/sh
```

Apart from these, there are an enormous amount of configurations that can be picked, from the jail identifier `jid`, the name of the jail `name`, the `path` of the jail, the ip (v4 or v6) and networking options such as hostname, device rules (`devfs_ruleset` pointing to rules in `/etc/devfs.rules` and `/etc/defaults/devfs.rules`), specific features allowed such as mounting device, the actions taken whenever the jail pre-start/starts/stops/post-stop, the user to run the commands as, and much more.

This frenzy of parameters makes it a pain to manage on the command line, this is why it is easier to have the jail configurations set in `/etc/jail.conf`. This file is composed of global parameters, and jail specific ones within `name { ... }`. For instance:

```
exec.start = "/bin/sh /etc/rc";
exec.stop = "/bin/sh /etc/rc.shutdown";
exec.clean;
exec.consolelog = "/var/log/jail_${name}_console.log";
mount.devfs;
host.hostname = ${name};
path = /jail/${name};

firefox {
 devfs_ruleset = 30; # from /etc/devfs.rules
 ip4.addr = 10.0.0.200;
 interface = wlan0;
 allow.raw_sockets;
 allow.sysvipc;
 mount.fstab = "/jail/firefox/etc/fstab";
}

www {
 host.hostname = www.example.org;       # Hostname
 ip4.addr = 192.168.0.10;               # IP address of the jail
 path = "/usr/jail/www";                # Path to the jail
 mount.devfs;                           # Mount devfs inside the jail
 exec.start = "/bin/sh /etc/rc";        # Start command
 exec.stop = "/bin/sh /etc/rc.shutdown";# Stop command
}
```

And the `devfs.rules` file with the rule `30` pointe to by firefox jail that gives access to audio devices.

```
[sound=30]
add path 'mixer*' unhide
add path 'dsp*'   unhide
```

To run these we can either use `jail(8)` pointing to the jail id or name set in `jail.conf` or we can set them as isolated services that are started at boot time in `rc.conf`. For example, to run the “firefox” jail from above we can do:

```
jail_enable=YES
```

```
jail_parallel_start=YES
jail_list="firefox" # entry in /etc/jail.conf
```

Afterward, jails can be managed through the usual `service(8)` utility, which will trigger the start/stop/restart commands set in `jail.conf`.

```
> service jail start www
> service jail stop www
```

To manage jails, `jail(8)` can be used to add/create/modify/remove, the `jls(8)` command to list jails, and `jexec(8)` to execute a command within an existing jail.

```
> jls
   JID  IP Address      Hostname                      Path
     3  192.168.0.10    www                           /usr/jail/www
> jexec 3 /etc/rc.shutdown
```

FreeBSD also offers a hierarchy of kernel options `security.jail.*` to fine-tune and restrict even more what is allowed within jails. This can be used instead of global parameters in the `jail.conf`.

```
security.jail.set_hostname_allowed: 1
security.jail.socket_unixiproute_only: 1
security.jail.sysvipc_allowed: 0
security.jail.enforce_statfs: 2
security.jail.allow_raw_sockets: 0
security.jail.chflags_allowed: 0
security.jail.jailed: 0
```

All of this administration can be painful and helper tools such as `ezjail` and `bastille` try to alleviate the process.

The `ezjail` automatically allows to create a base FreeBSD system with commands such as:

```
ezjail-admin install
ezjail-admin create jailname jailip
ezjail-admin create larry 192.168.0.100
```

Instead of `jexec(8)`, which can still be used, the `console` sub-command can be used to enter a jail environment, somewhat like `ssh`.

```
ezjail-admin console jailname
```

Meanwhile, `bastille` goes even further, by simplifying the bootstrapping process to make the creation of containers seamless with straight forward sub-commands such as `bootstrap`, `update`, `upgrade` and `verify`. It even allows running jails with a Linux emulation layer. Here's an example from the `README.md`:

```
> bastille create alcatraz 11.4-RELEASE 10.17.89.7
> bastille start alcatraz
> bastille console alcatraz
```

There is an abundance of similar jail managers, containers, and virtualisation wrappers on FreeBSD such as cbsd, pot, and iocage.

On the whole, FreeBSD jails are a good way to upgrade `chroot` environment. They are advertised as reducing administration overhead and risk of compromise, however they also need their own administration training and mindset. Furthermore, they are an oblivious-isolation solution and thus don't require any modification to programs.

What you need to remember: *FreeBSD jails are an upgrade over `chroot`, virtualising many more features of the OS such as networking, process tree, users, and resource usage. Like `chroot` it requires a directory set with a full file system tree, which can be facilitated using FreeBSD's system utilities. The jail can then either be created on the command line, or with a configuration file `/etc/jail.conf` as a one-time jail or a service in `rc.conf`. When run as a service it can be managed like a service with `start`/`stop` and other operations which will call the appropriate command set in `jail.conf`. There are wrappers to this procedure such as `ezjail`, `bastille`, `cbsd`, `pot`, and `iocage`.*

Isolation on Linux

Linux Control Groups

Control groups, or cgroups for short, are a Linux kernel feature that gives control of the hardware resource usage of processes, this includes memory, block IO, cpu usage and sets, number of processes, and more.

Compared to other resource constraints solutions we've seen such as niceness, `ulimit`/`setrlimit`, and file system quotas, cgroups are more flexible and can be set on a per-process basis with an inheritance grouping mechanism. It attaches cgroups to process hierarchies, indirectly letting it be inherited through the process tree, bounding resources while never gaining more as we approach a leaves. Additionally, it offers a simple way to monitor resource usage across a group of processes part of the same hierarchy.

There currently exist two versions of cgroups: v1 and v2. While they can currently somewhat coexist, and v1 has more "controllers"/subsystems than v2, we'll only focus on version 2 in this article.

Subsystems, or also called resource controllers, are what cgroups calls the sets of resources of the same type, in v2 this includes:

- `cpu`: used for number of allowed cpus, cpu accounting, etc..
- `cpuset`: used to bind process to specific cpu and NUMA
- `memory`: used to limit and report on memory usage
- `freezer`: used to suspend and restore processes
- `perf_event`: allow `perf` monitoring for the cgroup
- `hugetlb`: used to limit the use of huge pages
- `io`: used to control and limit access to block devices
- `pids`: used to limit the number of processes created

- `rdma`: used to limit the use of RDMA/IB-specific resources

Practically, these controllers exist under a pseudo-file system in `/sys/fs/cgroup/`. This directory is the root cgroup and contains files, which filenames are prefixed, thus categorized, with the name of controllers available. These files contain the current limit applied on the resource indicated by its name (ex: `memory.max`). Within this directory there are also files that describe the behavior of cgroup itself, their names are prefixed with `cgroup.` (ex: `cgroup.procs`).
Sub-groups are managed by creating and removing subdirectories within this cgroup virtual file system. This arrangement gives rise to a hierarchy of cgroups directories, one within the other. In v2, processes can only be parts of either the root cgroup hierarchy, or one of the leaves, and no where in between. This is referred to as the "no internal process" rule, processes only residing in leaf nodes. (It is more subtle than that though, as a cgroup can have subgroups as long as its `cgroup.subtree_control` is empty)

Example of the files under the root cgroup:

```
> ls -1 /sys/fs/cgroup/
cgroup.controllers
cgroup.max.depth
cgroup.max.descendants
cgroup.pressure
cgroup.procs
cgroup.stat
cgroup.subtree_control
cgroup.threads
cpu.pressure
cpuset.cpus.effective
cpuset.mems.effective
cpu.stat
init.scope/
io.cost.model
io.cost.qos
io.pressure
io.prio.class
io.stat
irq.pressure
memory.numa_stat
memory.pressure
memory.reclaim
memory.stat
misc.capacity
system.slice/
user.slice/
```

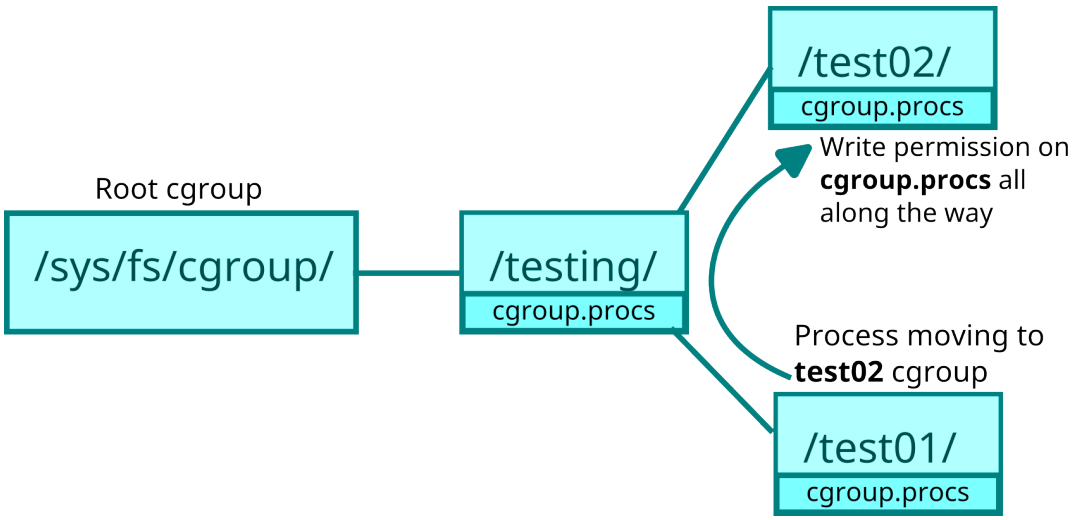
As we said, the controllers and limits enabled and set at one level can only be reduced as we dive deeper in the hierarchy, the limits cannot be exceeded by descendants. For example, a limit set at `/sys/fs/cgroup/some-group/` will be inherited by `/sys/fs/cgroup/some-group/sub-group/`, the files automatically being copied from the parent when the sub-directory is created.

To move a process to a particular cgroup, its PID needs to be written to the cgroup's `cgroup.procs` file, which contains the list of all processes that are part of this cgroup.

The value `0` can be written instead of the PID to move the current process to the group. However, beware that only one entry can be added at a time, and that a process can be a member of only one cgroup.

```
> echo $$ > /sys/fs/cgroup/testing/cgroup.procs
```

There are other restrictions to this procedure too, such as processes only being able to be added to leaf cgroups, the no internal process rules. This can be checked by looking at the `cgroup.stat` file which lists the `nr_descendants`, if it is 0 then this is a leaf node.
Additionally, the user writing to this directory should have the permission to do so on the `cgroup.procs` file. Furthermore, a process can only be moved to a sibling nodes if the user has both write access in the parent's (nearest common ancestor) `cgroup.procs` and the sibling, target node, `cgroup.procs`.
Giving access to certain files in a cgroup is called "delegating".



A process' current cgroup can be checked in the `/proc` file system in the form: `hierarchy-ID:controller-list:cgroup-path` :

```
> cat /proc/self/cgroup
0::/user.slice/user-1000.slice/session-3.scope
> echo $$ > ↙
    ↳ /sys/fs/cgroup/user.slice/user-1000.slice/testing/cgroup.procs
> cat /proc/self/cgroup
0::/user.slice/user-1000.slice/testing
```

For instance we can then limit the maximum number of child processes:

```
> echo 3 > pids.max
> sleep 10 &
> sleep 10 &
> sleep 10 &
zsh: fork failed: resource temporarily unavailable
```

Let's take a look at other `cgroup.` special files.

The `cgroup.max.depth` and `cgroup.max.descendants` are used to, obviously, limit the number of sub-cgroup and number of "live" descendants. Both of these files default to "max".

The `cgroup.controllers` and `cgroup.subtree_control` are used to decide which controllers are enabled at the current level, and which controller will be enabled in sub-cgroups. The `cgroup.controllers` of one level is equal to the `cgroup.subtree_control` of the parent.
Adding or modifying values in these files is done by writing `+` or `-` followed by the name of the controller, to add or remove a controller.

```
> echo '+pids -memory' > /sys/fs/cgroup/testing/cgroup.subtree_control
```

The `cgroup.type` file is used to decide the mode the cgroup is in. This can be either `domain`, for process granularity, `threaded` for thread granularity, `domain threaded` for the root of a threaded subtree, and `domain invalid` for an invalid state. We won't dive into how to create threaded sub-cgroups.

There are also a couple of files that can be used, along with `inotify`, as a notification or statistic mechanism. For example, `cgroup.events` can be used to know if a subgroup is populated or frozen.

Other files include the `/proc/cgroups` file which contains information about all the controllers currently enabled and the number of hierarchies using them. The directory `/sys/kernel/cgroup` contains which cgroups can be delegated and which features are currently enabled in the kernel.

Lastly, `ps` can be used to interrogate the current cgroups of running processes.

```
> ps -eo pid,user,args,cgroup --sort user

 869102 vnm      /usr/lib/firefox/firefox -c ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
 911471 vnm      vim newsletter.md           ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
 940268 vnm      /usr/lib/firefox/firefox -c ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
 940854 vnm      /usr/lib/firefox/firefox -c ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
 942171 vnm      /usr/lib/speech-dispatcher/ ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
 942174 vnm      /usr/bin/speech-dispatcher  ↙
    ↳ 0::/user.slice/user-1000.slice/session-3.scop
    480 root     /usr/lib/iwd/iwd            ↙
        ↳ 0::/system.slice/iwd.service
    482 root     /usr/lib/systemd/systemd-lo ↙
        ↳ 0::/system.slice/systemd-logind.service
    485 root     dhcpcd: [privileged proxy]  ↙
        ↳ 0::/system.slice/dhcpcd.service
    611 root     /usr/bin/lightdm            ↙
        ↳ 0::/system.slice/lightdm.service
    640 root     /usr/lib/Xorg :0 -seat seat ↙
        ↳ 0::/system.slice/lightdm.service
    821 root     lightdm --session-child 15  ↙
        ↳ 0::/user.slice/user-1000.slice/session-3.scop
```

```
 32174 root     /usr/lib/udisks2/udisksd    ↙
    ↳ 0::/system.slice/udisks2.service
509281 root     gpg-agent --homedir /etc/pa ↙
   ↳ 0::/user.slice/user-1000.slice/session-3.scop
```

Other than manipulating all these manually, a few different tools can be used instead to facilitate the cgroups management. For example, there is systemd and libcgroup tools which we'll take a look at.

In systemd, the tools `systemctl status`, `systemd-cgtop` (like top but for cgroups usage), and `systemd-cgls` can be used to introspect the state of cgroups on the system.

```
> systemd-cgls
Control group /:
-.slice
├─user.slice (#185)
│ → user.invocation_id: 0841542f9c6c4034a8100e3769abcba3
│ └─user-1000.slice (#1953)
│   → user.invocation_id: c34a74cc011a416fa5ffece28154cefa
│   ├─user@1000.service … (#2087)
│   │ → user.delegate: 1
│   │ → user.invocation_id: d8faa4886d484718ab2d73bd4b2eaedc
│   │ ├─session.slice (#2228)
│   │ │ ├─pipewire-pulse.service (#3651)
│   │ │ │ └─910 /usr/bin/pipewire-pulse
│   │ │ ├─wireplumber.service (#3611)
│   │ │ │ └─908 /usr/bin/wireplumber
│   │ │ ├─gvfs-daemon.service (#3851)
│   │ │ │ ├─  957 /usr/lib/gvfsd
...
```

systemd calls a sub-tree within a cgroup a "slice" (`systemd.slice(5)`) and offers special unit files to create them or associate them with services and other units.

A `my.slice` file:

```
[Slice]
CPUQuota=30%
```

Or restriction directly mentioned from a service file (see `systemd.resource-control(5)`):

```
[Service]
```

```
MemoryMax=1G
AllowedCPUs=0-5
MemoryHigh=6G
 # or
Slice=my.slice
```

Furthermore, delegation can be explicitly mentioned in units:

```
[Service]
User=%i
Slice=user-%i.slice
Delegate=cpu cpuset io
```

When units or commands are launched, a slice name can be added to specify the restriction:

```
> systemd-run --slice=my.slice command
```

Another helper that can be used are the set of utilities that come with libcgroup.
It offers command line tools such as `cgget` (given a path, without `/sys/fs/cgroup`, it prints the current configs), `cgset`, `cgcreate`, `cgdelete`, `cgclassify` (to move task to cgroup), `cgexec`, and others.

```
> cgcreate -a user -t user -g memory,cpu:groupname
> cgexec  -g memory,cpu:groupname/foo bash
> cgclassify -g memory,cpu:groupname/foo `pidof bash`
```

libcgroup comes with a daemon called `cgrulesengd` that manages a set of rules in `/etc/cgrules.conf` to automatically associate processes with control groups, and a configuration in `/etc/cgconfig.conf` to automatically set up control groups with their restrictions.

For example, here's a `cgconfig.conf` with an entry for the testing cgroup.

```
group testing {
  perm {
    admin {
      uid = username;
    }
    task {
      uid = username;
    }
  }
  cpuset {
    cpuset.mems="0";
    cpuset.cpus="0-5";
  }
  memory {
    memory.limit_in_bytes = 5000000000;
  }
}
```

Meanwhile, the `cgrules.conf` takes the form of the following (the user field being the same form as the `sudoers` file):

```
<user>                  <controllers>  <destination>
<user>:<process name>  <controllers>  <destination>
```

For example:

```
peter  cpu  testing/
```

That's about it for cgroups. It is a rather particular, and initially non-obvious, way to associate resource constraints to a group of processes. The pseudo-file system manipulation can be quite flimsy, however the set of tools around it such as libcgroup and systemd makes it a breeze.

What you need to remember: *Linux' cgroups is a pseudo-file system `/sys/fs/cgroup/` used to limit the hardware resources a process can use. Processes are associated with a sub-directory, always a leaf, within this file system along with files that describe what is limited. Wrappers exist to facilitate the management of the pseudo-fs such as systemd and libcgroup.*

Linux Namespaces

Linux namespaces is a kernel feature providing similar functionality as FreeBSD jails, however giving more control over which part of the system is virtualised. It provides a mechanism to create a per-process view of the system, partly inspired by Plan9 namespaces and layering.

We've seen that the Linux control groups were about constraining hardware resources, meanwhile, namespaces are for compartmentalizing OS resources between processes. This can either be done as self-isolation by relying on functions such as `unshare(2)`, `clone(2)`, and `setns(2)`, or as oblivious/dumb-isolation by using command line wrappers such as `unshare(1)` and `nsenter(1)`.

The virtualisation touches anything OS-related, such as: cgroups, inter-process communication (IPC), the networking stack, mount points, the process tree, time, the users/groups and their mapping, and the domain name (hostname).

When a process is in a namespace and spawns another process, this new process will inherit its parent's namespace, just like cgroups. However, unlike cgroups, namespaces can be nested. Additionally, unlike cgroups, namespaces only live as long as the last process alive in it, they are automatically destroyed when the last process terminates.

The man page `namespace(7)` gives a never-ending description of what namespaces are and do, and we'll try to summarize it in a simpler and approachable way. However, refer to the man page and its subsection for every namespace category in case more details are needed.

Namespaces each have their specificities, however, when used in the APIs, only the flag mask name is specified and can be combined as a bitwise-OR operation. Here's the list of namespaces:

- Cgroup (`CLONE_NEWCGROUP`): creates a new root cgroup
- IPC (`CLONE_NEWIPC`): isolates System V IPC, POSIX message queues
- Network (`CLONE_NEWNET`): create a new network stack (devices, stacks, ports, etc..)
- Mount (`CLONE_NEWNS`): isolate mount points

- PID (`CLONE_NEWPID`): create a new isolated process tree
- Time (`CLONE_NEWTIME`): isolate the clock
- User (`CLONE_NEWUSER`): create and isolate new user and group tree
- UTS (`CLONE_NEWUTS`): isolate and create new hostnames

The flag itself isn't enough to set everything needed for a new namespace, thus one has to look in the intricacies of each man page, which are found in `<type>_namespaces(7)`, for example: `ipc_namespaces(7)`, `mount_namespaces(7)`, `user_namespaces(7)`, etc..

As we said, each namespace is separate from one another, additionally, all processes are part of at least one namespace of each type. In other words, at any given moment, any process belongs to exactly one instance of each namespace. Whenever a namespace of a type is created, the process instantly moves into it. As soon as it leaves the namespace, then it is destroyed. A trick allows keeping the namespace alive by assigning it to a file and bind mounting `/proc/pid/ns/type`.
Bind mount, similar to FreeBSD nullfs, is a way to remount part of the file hierarchy somewhere else as if it was a device (instead of symbolic links).

Hence, the namespaces are a per-process attribute and live along them. Like all process-related attributes, they live in `/proc`, more precisely in `/proc/<pid>/ns` as abstract file descriptor, or symlink, pointing to the namespace.
Under this directory, you'll find the following files, one for every namespace:

```
cgroup
ipc
mnt
net
pid
pid_for_children
time
time_for_children
user
uts
```

The identifier of the namespace can be found by following the symbolic link and reading the inode number:

```
> readlink /proc/self/ns/user
user:[4026531837]
```

If two processes are in the same namespaces of a particular type, then they'll get the same identifier.

Two special files above exist: `pid_for_children` and `time_for_children`, which exist because the original `pid` and `time` namespaces are permanent.

There exists 4 functions used to manipulate namespaces and self-isolate, which can be used with the above flags we mentioned:

- `clone(2)`: create a new process, if one of the above flag is used then the child process will be spawned in the new namespace.
- `unshare(2)`: move the calling process to a new namespace
- `setns(2)`: allow a process to join an existing namespace (by specifying a file descriptor).

- `ioctl(2)`: This can be used to discover namespace information when the file descriptor of the namespace is passed(see `ioctl_ns(2)`). It is mainly used for the PID and USER namespaces types.

The difference between the `clone` and `unshare` functions is that `clone` will spawn a new process inside the new namespace, while `unshare` will move the current process within the new namespaces. This matters because some namespaces can only be started along a new process and thus only `clone` will work.
All of these functions require the capability `CAP_SYS_ADMIN`, with the exception of the creation of user namespaces which don't require privileges.

Meanwhile, there are two main commands used to do oblivious-isolation: `unshare(1)` and `nsenter(1)`, to run a program in new namespaces and run a program in existing namespaces, respectively.

Both commands are relatively simple, taking arguments mapping to the flags we've seen above, and mixed with the bind-mount of procfs trick, can easily allow to create the namespaces desired and reuse them. By default these commands will launch `/bin/sh`.
For example, `unshare(1)` has the following flags: `--ipc`, `--mount`, `--net`, etc.. It also offers wrapper options such as `--mount-proc`, `--map-users`, `--map-root-user`, and others that are helpful for certain namespace types (here mount and user namespaces).

Here's an example of the UTS namespace used with `unshare` and `nsenter` to bind a hostname namespace to a file.

```
> touch /root/uts-ns
> unshare --uts=/root/uts-ns hostname FOO
> nsenter --uts=/root/uts-ns hostname
FOO
> umount /root/uts-ns
```

Let's now have a quick look at a couple of namespaces and see some of their peculiarities.

The PID namespace is used to isolate the system process tree. As we've seen, there always exists the file descriptor in `/proc/<pid>/ns/` for the initial PID and the new namespace process identifier will be in `pid_for_children`. This is apparent when looking at `/proc/self/status`, the namespaces identifiers are prefixed with `NS`:

```
> cat /proc/self/status | gre -i NS
NStgid: 1330184
NSpid:  1330184
NSpgid: 1330184
NSsid:  1330128
```

Keep in mind that namespaces can be nested, that means we an have multiple `PID=1`, each perceiving themselves as the root of a subtree.

One caveat with the PID namespace is that it won't work with `unshare(1)` as it requires the process to be new at the time of spawning, and thus can only be created with `clone(2)`. To make this work with `unshare(1)` the `--fork` parameter needs to be passed. Furthermore, the `--mount-proc` also needs

to be passed because most tools such as `ps` look at procfs to see the process tree, even though these processes can't interact with processes in the other PID namespace. Example:

```
> unshare --pid --fork --mount-proc ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 20:20 pts/11   00:00:00 ps -ef
```

Let's take a look at the NET namespace which is used to limit the view of network interfaces, firewalls, and routing rules.
When we `unshare --net`, and try to have a look at the network interfaces available, we'll see that we only have access to the loopback and with an empty MAC address.

```
> ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group ↙
    ↳ default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

The `ip` toolset and other commands have been augmented with the `netns` option to manipulate and make networks available for the network namespace (`ip-netns(8)`). Virtual interfaces for the namespaces are by convention objects living under `/var/run/netns/NAME` that can be opened, or listed with `ip`:

```
> ip netns list
netnstest
```

For example, to enter a network namespace:

```
nsenter --net=/var/run/netns/NAME
```

Let's look at the USER namespace, used to isolate the UID, GID, and other user-related attributes. Within these namespaces, a user can be root, have full capabilities in the namespace, but have no privileges outside.

This namespace is the only one that doesn't require `CAP_SYS_ADMIN` to be created:

```
> unshare -U /bin/bash
```

The user mapping relies on configuration files and special files. Without mentioning any UID, the default UID and GID used in the user namespaces are the ones in `/proc/sys/kernel/overflowuid` and `overflowgid`, which is usually `65534`, or the "nobody" user.

For a more precise mapping of users, the `/proc/<PID>/uid_map` and `/proc/<PID>/gid_map` files can be used. The values returned by these files also depends on whether the processes are in the same namespace. The files contain 3 values, the login name or UID/GID, the subordinate/lower values for the UID/GID, and the count of UID/GID.

These can be manipulated in the `/proc/` filesystem, or within `/etc/subuid` and `/etc/subgid` files, or with the commands `newuidmap(1)` and `newgidmap(1)`.

Let's move on and check the MNT (mount) namespace, which creates a per-process file system tree, sort of like `chroot`. The main difference with `chroot` is that the mount namespace is not bound to the current file system structure and changes, and thus it is entirely virtualised and isolated, avoiding the

security issues of `chroot` jails.
We can examine the current root and working directory of a process in procfs: `/proc/<PID>/root` and `/proc/<PID>/cwd`, there is additional information in `/proc/<PID>/mounts`, `/proc/<PID>/mountinfo`, and `/proc/<PID>/mountstats`.
Initially, the mount namespace is the same as the process invoking or parent, in the case of `clone(2)`. The difference, is that any further changes, by default, like umounting a file system, is private, that means it won't affect the other namespaces. For further control, each mount can be tagged with a type of propagation, on whether it will be shared, private, slave, or unbindable. This is done on the command line with `mount(8)` using te `--make-<type>` argument (ex: `--make-shared`). Examples can be found in `mount_namespaces(7)` and the `findmnt(8)` command can also be used to get more info.

We'll cut it short for now and move to another namespaces feature. It is possible to impose a limit on the number of namespaces of a specific type by using the files used `/proc/sys/user/` directory (ex: `max_cgroup_namespaces`).

Lastly, the `pam_namespace` module can be used to facilitate creating a mount namespaces as soon as sessions start. Its configuration file `/etc/namespace.conf` and `namespace.init` can be of great use to compartmentalize users by creating new instance of the same directory, which will globally appear "normal". This mechanism is called polyinstantiated directories. For example, each user can have their own `/tmp` which will indirectly be mapped to another directory in the parent namespace.

To conclude, Linux namespaces are a really grainy way to divide the operating system resources between process groups. It's reliance of file descriptors in `/proc` makes it a bit hard, however with the help of oblivious-isolation tools such as `unshare(1)` it's easier to manipulate.

What you need to remember: *Linux' namespaces is a virtualisation of different OS resources such as mount points, networking, process tree, associated to inode/file descriptors. All processes are in at least one namespace of every type, a namespace is destroyed with the last process in it (there's a bind mount trick to avoid this). They are manipulated through the `/proc` file system, either using functions (`clone(2)`, `unshare(2)`, ...) for self-isolation, or with wrappers (`nsenter(1)`, `unshare(1)`). Every namespace has its own manpage (`<type>_namespaces(8)`) and particularities to how it is configured. The only namespace that can be created without `CAP_SYS_ADMIN` is the user namespace.*

`landlock` & `seccomp`

`landlock` and `seccomp` are Linux' equivalent to OpenBSD's `unveil` and `pledge`, they achieve more or less the same functionalities. `landlock` is used to reduce the view of a process of the file system and `seccomp` is used to limit which system calls are allowed by a process.

The `landlock` project is an LSM (Linux Security Module) to restrict which file system operations are allowed on which files. It initially relied on eBPF to achieve this, but isn't requiring it anymore.
It is available in Linux 5.13 and above if the kernel is compiled with the `CONFIG_SECURITY_LANDLOCK` option. You can confirm this by taking a look at the loaded LSM in `/sys/kernel/security/lsm`.

In practice, it is a self-isolation solution, and thus needs programmers to create their own rulesets in their software.

A process using `landlock` doesn't require elevated privileges, and once the rules are in place, they are also inherited by child processes. This means only more constraints can be added, and never removed.

Essentially, the `landlock` policy restriction takes the form of a ruleset, and aggregation of rules, rules taking the form of allowed access rights (action) on files and directories (object, the file descriptor/inodes).
The current list of file system actions is as follows:

- `LANDLOCK_ACCESS_FS_EXECUTE`: Execute a file.
- `LANDLOCK_ACCESS_FS_WRITE_FILE`: Open a file with write access.
- `LANDLOCK_ACCESS_FS_READ_FILE`: Open a file with read access. (applies to directory and sub-directories)
- `LANDLOCK_ACCESS_FS_READ_DIR`: Open a directory or list its content. (applies to directory and sub-directories)
- `LANDLOCK_ACCESS_FS_REMOVE_DIR`: Remove an empty directory or rename one. (applies only to directory)
- `LANDLOCK_ACCESS_FS_REMOVE_FILE`: Unlink (or rename) a file. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_CHAR`: Create (or rename or link) a character device. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_DIR`: Create (or rename) a directory. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_REG`: Create (or rename or link) a regular file. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_SOCK`: Create (or rename or link) a UNIX domain socket. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_FIFO`: Create (or rename or link) a named pipe. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_BLOCK`: Create (or rename or link) a block device. (applies only to directory)
- `LANDLOCK_ACCESS_FS_MAKE_SYM`: Create (or rename or link) a symbolic link. (applies only to directory)

As you can see, this is more granular than OpenBSD's `unveil`.

After a ruleset is applied, if a program tries to perform an action that isn't in the list, the system call that failed will return `EPERM` instead, and the program will continue execution (similar to what OpenBSD does with `unveil`). An exception to this are all the files and directories that were opened before the ruleset is applied.

The creation of policies is done in three steps. First of all the ruleset is created with the list of possible actions that can be used in upcoming rules that will be added (`landlock_create_ruleset(2)`). Second of all, rules are created and added to the ruleset, mapping an action from the ruleset to a (inode) file descriptor (`landlock_add_rule(2)`). Thirdly, and finally, the ruleset is applied, restricted to not gain anymore privileges, and the program goes into enforcing mode (`landlock_restrict_self(2)` and some `prctl(2)` for `PR_SET_NO_NEW_PRIVS`).

Multiple languages have support for `landlock`, including C, Python, Rust, Go. Let's have a look at a C example from the manpage and annotate it. Even though it would obviously be much easier to write it in Python.

```
1 #include <linux/landlock.h>
```

```
#include <sys/syscall.h>

struct landlock_ruleset_attr attr = {0};
int ruleset_fd;

// In this ruleset these are the only allowed permissions
attr.handled_access_fs =
LANDLOCK_ACCESS_FS_EXECUTE |
LANDLOCK_ACCESS_FS_WRITE_FILE |
LANDLOCK_ACCESS_FS_READ_FILE |
LANDLOCK_ACCESS_FS_READ_DIR |
LANDLOCK_ACCESS_FS_REMOVE_DIR |
LANDLOCK_ACCESS_FS_REMOVE_FILE |
LANDLOCK_ACCESS_FS_MAKE_CHAR |
LANDLOCK_ACCESS_FS_MAKE_DIR |
LANDLOCK_ACCESS_FS_MAKE_REG |
LANDLOCK_ACCESS_FS_MAKE_SOCK |
LANDLOCK_ACCESS_FS_MAKE_FIFO |
LANDLOCK_ACCESS_FS_MAKE_BLOCK |
LANDLOCK_ACCESS_FS_MAKE_SYM;


ruleset_fd = landlock_create_ruleset(&attr, sizeof(attr), 0);
if (ruleset_fd == -1) {
    perror("Failed to create a ruleset");
    exit(EXIT_FAILURE);
}

// Using the file descriptor ruleset_fd we can add rules
// currently there's only one rule type available: ↙
    ↳ LANDLOCK_RULE_PATH_BENEATH
// the landlock_path_beneath_attr has two attributes: ↙
    ↳ allowed_access, parent_fd

struct landlock_path_beneath_attr path_beneath = {0};
int err;

path_beneath.allowed_access =
LANDLOCK_ACCESS_FS_EXECUTE |
LANDLOCK_ACCESS_FS_READ_FILE |
LANDLOCK_ACCESS_FS_READ_DIR;

// this rule are on files and directories via a file descriptor
// parent_fd is either a directory or a file
path_beneath.parent_fd = open("/usr", O_PATH | O_CLOEXEC);
if (path_beneath.parent_fd == -1) {
    perror("Failed to open file");
    close(ruleset_fd);
    exit(EXIT_FAILURE);
}
```

```
// the last argument (flag) is unused
err = landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,
        &path_beneath, 0);
close(path_beneath.parent_fd);
if (err) {
    perror("Failed to update ruleset");
    close(ruleset_fd);
    exit(EXIT_FAILURE);
}

// we use prctl to disallow more privileges
if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("Failed to restrict privileges");
    close(ruleset_fd);
    exit(EXIT_FAILURE);
}

// and finally apply landlock ruleset
if (landlock_restrict_self(ruleset_fd, 0)) {
    perror("Failed to enforce ruleset");
    close(ruleset_fd);
    exit(EXIT_FAILURE);
}
close(ruleset_fd);
```

There currently aren't that many wrapper tools relying on `landlock` that would allow dumb/oblivious-isolation apart from the `unveil` port to Linux by Justine Tunney. Since `landlock` is like `unveil`, it also has the same limitations, namely that developers have to define their own threat models. Moreover, some system calls are still allowed and not covered by the previously mentioned actions, these are: `chdir(2)`, `truncate(2)`, `stat(2)`, `flock(2)`, `chmod(2)`, `chown(2)`, `setxattr(2)`, `utime(2)`, `ioctl(2)`, `fcntl(2)`, `access(2)`. However, the project hopes to cover them in the future.

Let's switch course and move on to discover `seccomp`, the Linux secure computing state.

It is similar to OpenBSD `pledge`, reducing the attack surface by restricting which system calls are allowed. However it is much more granular, it offers two modes: a strict mode, `SECCOMP_SET_MODE_STRICT` often simply called `seccomp`, and a filter mode, `SECCOMP_SET_MODE_FILTER` often referred to as `seccomp-bpf`.
The filter mode, as the name implies, relies on dynamic BPF (Berkeley Packer Filter) rules. This is the "classic" BPF virtual machine, and not the newer extended BPF (eBPF), allowing to load assembly-like programs in the kernel. There is currently no plan to switch from BPF to eBPF.

Like OpenBSD `pledge`, `seccomp` is a self-isolation mechanism, letting the programmers decide how to drop privileges in their applications. The wrappers, for oblivious-isolation, exist too, usually as part of container solutions, as we'll see in the next section.

The functions needed to interact with `seccomp` are `prctl(2)`, process control, and `syscall(2)`, indirect system call, using `SYS_seccomp` as the first parameter. Both are mostly equivalent and only differ in the way `seccomp` is launched.
There also are less dreadful approaches to `seccomp` such as using `libseccomp` (`-lseccomp`),

easyseccomp, and Kafel. We'll see an example of each.

The first mode of operation that seccomp can run in is the SECCOMP_SET_MODE_STRICT, strict mode. It denies access to all system calls except read, write, exit, and sigreturn. Any other call will terminal the process with a SIGILL signal.
To enter this mode no elevated privileges are needed. For example, here are two ways to initiate it:

```
// using process control
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
// using syscall, the flags and args are 0
syscall(SYS_seccomp, SECCOMP_SET_MODE_STRICT, 0, 0);
```

The second mode of operation that seccomp can run in is SECCOMP_SET_MODE_FILTER, the seccomp-bpf filter mode. It uses a BPF program to filter and decide what to do with system calls. In this mode, we'll need to write a program to observe another program, which might sound redundant.
Thus, using prctl and syscall we'll have to pass a pointer to a BPF program which will need to be loaded into the kernel. For this reason, the seccomp-bpf mode requires CAP_SYS_ADMIN capabilities. If multiple filters are loaded, then they are all executed in the reverse order in which they were loaded.

Upon loading a filter, a flag can be passed for specific behavior such as notification upon successful loading. The available flags are:

- SECCOMP_FILTER_FLAG_LOG
- SECCOMP_FILTER_FLAG_NEW_LISTENER
- SECCOMP_FILTER_FLAG_SPEC_ALLOW
- SECCOMP_FILTER_FLAG_TSYNC

The BPF program is a series of BPF instructions, which are assembly-like low-level instructions. These could be painful to manually write.

```
struct sock_fprog {
  unsigned short      len;    /* Number of BPF instructions */
  struct sock_filter *filter; /* Pointer to array of
                                 BPF instructions */
};
```

Here are two different ways to load a BPF seccomp filter:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog);
syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, flag, &prog)
```

The BPF filter program will return a value that contains an action and additional data. The action will decide what will happen to the system call it filtered, these are, in decreasing order of precedence:

- SECCOMP_RET_KILL_PROCESS
- SECCOMP_RET_KILL_THREAD
- SECCOMP_RET_TRAP
- SECCOMP_RET_ERRNO
- SECCOMP_RET_USER_NOTIF
- SECCOMP_RET_TRACE
- SECCOMP_RET_LOG

- SECCOMP_RET_ALLOW

The SECCOMP_RET_USER_NOTIF can be particularly useful as it lets the seccomp filter be passed to a user-space program to intercept and decide on the outcome and behavior of the system call.

After loading a BPF program, it is mandatory to set PR_SET_NO_NEW_PRIVS with process control, otherwise all operations will fail.

```
prctl(PR_SET_NO_NEW_PRIVS, 1);
```

Before having a look at a couple of example of seccomp filter programs, let's look at how we can get more information on the current state.

The syscall interface of seccomp has two information-related actions: SECCOMP_GET_ACTION_AVAIL, to get all possible BPF filter actions, and SECCOMP_GET_NOTIF_SIZES to get the size of BPF notifications into user-space. This information also exist under /proc/sys/kernel/seccomp/action_avail and /proc/sys/kernel/seccomp/action_logged.

```
> cat /proc/sys/kernel/seccomp/actions_avail
kill_process kill_thread trap errno user_notif trace log allow
> cat /proc/sys/kernel/seccomp/actions_logged
kill_process kill_thread trap errno user_notif trace log
```

We can introspect whether a process has seccomp enabled and in which mode using prctl(2) with PR_GET_SECCOMP, or simply by looking in procfs.

```
> cat /proc/self/status | grep -i secc
Seccomp: 0
Seccomp_filters: 0
```

- 0: Seccomp is not enabled
- 1: Seccomp "strict mode" is enabled
- 2: Seccomp-bpf is enabled

Now let's see a few examples.

With easyseccomp, we can define BPF programs using a simple DSL.

```
#ifndef CAP_AUDIT_WRITE
$syscall == @socket && $arg0 == 16 && $arg2 == 9 => ERRNO(EINVAL);
#endif
$syscall == @socket => ALLOW();
```

It can then be compiled into a BPF filter and used with an OCI compliant container runtime environment such as podman, or used for self-isolation in a program:

```
> easyseccomp < config > /path/to/the/filter.bpf
> podman run --annotation ↙
    ↳ run.oci.seccomp_bpf_file=/path/to/the/filter.bpf ...
```

Kafel is very similar to easyseccomp, the policies are written in a DSL.

```
POLICY a {
    ALLOW {
        write, execve, brk,
        access, mmap, open,
        newfstat, close, read,
        mprotect, arch_prctl,
        munmap, getuid, getgid,
        getpid, rt_sigaction,
        geteuid, getppid, getcwd,
        getegid, ioctl, fcntl, newstat,
        clone, wait4,
        rt_sigreturn, exit_group
    }
}
USE a DEFAULT KILL
```

With `libseccomp`, the programmer has to rely on functions such as `seccomp_init(3)`, `seccomp_rule_add(3)`, and `seccomp_load(3)`, to manipulate the BPF instructions. However, it's still significantly easier than having to write them manually. Here's an example:

```
/* initialize the libseccomp context, default action is KILL */
scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

/* allow exiting */
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

/* allow getting the current pid */
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);

/* allow changing data segment size, as required by glibc */
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);

/* allow writing up to 512 bytes to fd 1,
   syscall order of params 1 and 2 for write */
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 2,
    SCMP_A0(SCMP_CMP_EQ, 1),
    SCMP_A2(SCMP_CMP_LE, 512));

/* if writing to any other fd, return -EBADF */
seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(read), 0);

/* load and enforce the filters */
seccomp_load(ctx);
seccomp_release(ctx);
```

Finally, we could torture ourselves and write BPF instructions manually, nonetheless we have a smarter approach: rely on chatGPT. (Beware, this might be full of typos)

```
// This program uses seccomp-bpf to apply a filter that only allows ↙
    ↳ read
```

```
// access to the /usr directory and denies all other system calls. The filter
// rules are defined in the filter array and loaded into the kernel using
// the prctl function. The program then sleeps for a while to demonstrate
// that the filter is applied. You can modify the filter rules to allow
// other system calls or file access patterns as needed.

#include <stddef.h>
#include <linux/audit.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>

int main() {
  // Define the BPF filter rules
  struct sock_filter filter[] = {
    // Load the syscall number into the accumulator
    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, nr)),
    // Allow read system calls on the /usr directory (14 = __NR_access)
    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 14, 0, 1),
    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, (uintptr_t) "/usr", 0, 1),
    // Allow read system calls with an argument length less than or equal to 4KB
    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, args[1])),
    BPF_JUMP(BPF_JMP | BPF_JGT | BPF_K, 4096, 1, 0),
    // Deny all other system calls
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO | (AUDIT_SYSCALL << 16) | EPERM),
    // Allow the system call to proceed
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
  };
  struct sock_fprog prog = {
    .len = (unsigned short) (sizeof(filter) / sizeof(filter[0])),
    .filter = filter,
  };

  // Load the filter into the kernel
  if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) < 0) {
    perror("prctl(PR_SET_NO_NEW_PRIVS)");
    return 1;
  }
  if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) < 0) {
    perror("prctl(PR_SET_SECCOMP)");
    return 1;
```

```
43   }
44
45   // Execute the desired action (in this case, just sleep for a while)
46   sleep(60);
47   return 0;
48 }
```

Lastly, systemd units can rely on a simple seccomp filtering through the `SystemCallFilter` directive (see `systemd.exec(5)` manpage, or `systemd.directives(7)` for all other directives). There are many more sandboxing features we'll see in the next section.

That's all there is to seccomp, obviously, we only skimmed the topic. Both `landlock` and `seccomp` are deep and intricate facilities on Linux that allow fine-grained self-isolation, especially with BPF filters.

What you need to remember: *`landlock` and `seccomp` are the Linux self-isolation equivalent of OpenBSD `unveil` and `pledge`, yet have way more intricacies and fine-tuning. `landlock` let the programmer create a ruleset (`landlock_create_ruleset`) which can then be used to apply file system rules (`landlock_add_rule`), and finally launch the restriction (`prctl` for `PR_SET_NO_NEW_PRIVS` and `landlock_restrict_self`). `landlock` still has a few drawback as it doesn't cover all system calls. Meanwhile, `seccomp` has two modes: a strict one only allowing `read`, `write`, `exit`, `sigreturn`, and a filter mode aka `seccomp-bpf` that relies on BPF (not eBPF) programs. These programs need `CAP_SYS_ADMIN` to be loaded in the kernel. The BPF return an action, which can be used to intercept system calls in userspace `SECCOMP_RET_USER_NOTIF`. Both `landlock` and `seccomp` have programming language support in multiple languages, however it is `seccomp` that needs the most helpers as it is difficult to write BPF programs. The `libseccomp` and `easyseccomp` can be used instead of manually writing BPF programs.*

Linux Software Relying on Isolation

In this section we'll have a quick look at a few Linux programs that rely on many of the isolation, limitation, and compartmentalization features we've explored. This is going to be more of a listing than a deep dive into each software.
There are two big categories of such software: container managers and sandbox wrappers (oblivious-isolation).

Most of the container managers and runners are compliant with something called OCI, the open container initiative. These technologies combine cgroups for resource control, bind mount (and union/overlay-mount), Linux POSIX capabilities, `seccomp-bpf`, `chroot`, and namespaces to forge a containerised environment. Such solutions include, but aren't limited to:

- CRI-O
  Open Container Initiative-based implementation of Kubernetes Container Runtime Interface
- RKT (project ended)
  rkt is a pod-native container engine for Linux. It is composable, secure, and built on standards

- runC
  CLI tool for spawning and running containers according to the OCI specification
- podman
  Podman is a daemonless container engine for developing, managing, and running OCI Containers on your Linux System. Containers can either be run as root or in rootless mode. Simply put: `alias docker=podman`.
- LXC
  LXC is a user-space interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers
- docker
  A sandboxed process on your machine that is isolated from all other processes on the host machine. That isolation leverages kernel namespaces and cgroups, features that have been in Linux for a long time. Docker has worked to make these capabilities approachable and easy to use
- `systemd-nspawn(1)`
  Spawn a command or OS in a light-weight container
- nsbox
  nsbox is a multi-purpose, nspawn-powered container manager.
- unbox
  An wrapper using namespaces.
- folderbox
  Workspaces using containers which can be executed against a project folder. Allowing the development environment to be separate from the host, while still providing sandbox escapes.
- distrobox
  Use any Linux distribution inside your terminal by relying on `docker` or `podman`.

Many of the above allow running `seccomp-bpf` filters, and easily load them with hooks such as this one.

Let's take a closer look at one of the container approach: `systemd-nspawn` which is managed through `machinectl(1)`.
In general systemd offers all the security features we've seen thus far, allowing them to be set in their directives (see `systemd.directives(7)`).

On the same note, `system-nspawn` give access to namespaces, `seccomp-bpf`, `pivot_root`, POSIX capabilities, bind mounts, `prctl` with `PR_SET_NO_NEW_PRIVS`, and `cgroups`.
For example a new container can be launched in a directory, along with the security options desired, with:

```
systemd-nspawn -D ~/containerdir
systemd-nspawn -b -D ~/containerdir #-b to boot
```

Then the launched containers can be see with `machinectl`:

```
> machinectl list
MACHINE  CLASS     SERVICE        OS VERSION ADDRESSES
newroot2 container systemd-nspawn -  -       -

1 machines listed.
```

To make this permanent and more easily manipulate settings, `systemd` offers container files settings `/etc/systemd/nspawn/machine.nspawn` and storage for machine skeletons `/var/lib/machines/machine.nspawn`. This allows to manage containers like services (similar to FreeBSD jails services).
Furthermore, most `systemd` tools allow for an additional flag `-M <container-name>` to interface with running containers.

Indeed, any units can use isolation features to reduce privileges and access rights via directives such as:

- `ProtectSystem`, `ProtectHome`, `Protect*`: Makes read-only certain aspect of the system (`systemd.exec(5)`)
- `ReadWritePaths`, `ReadOnlyPaths`, `InaccessiblePaths`, `ExecPaths`, `NoExecPaths`: Control which path is accessible for what usage (`systemd.exec(5)`)
- `RestrictSUIDSGID`: Doesn't allow changing UID/GID.
- `CapabilityBoundingSet`, `AmbientCapabilities`: Control the POSIX capabilities. (`systemd.exec(5)`)
- `SystemCallFilter`: Uses `seccomp-bpf` to restricts access to system calls. (`systemd.exec(5)`.
- `MemorySwapMax`: resource control to limit swap usage (`systemd.resource-control(5)`)
- etc..

The list of directives is extensive, hence to make this easier a tool called `systemd-analyze` has the `security` option to help isolate units. It'll parse the unit files and see if it contains enough security directives, give recommendations, and score them accordingly.

```
> systemd-analyze security --json=pretty adsuck.service | less
[
   {
      "set" : false,
      "name" : "RootDirectory=/RootImage=",
      "json_field" : "RootDirectoryOrRootImage",
      "description" : "Service runs within the host's root directory",
      "exposure" : "0.1"
   },
   {
      "set" : null,
      "name" : "SupplementaryGroups=",
      "json_field" : "SupplementaryGroups",
      "description" : "Service runs as root, option does not matter",
      "exposure" : null
   },
   {
      "set" : null,
      "name" : "RemoveIPC=",
      "json_field" : "RemoveIPC",
      "description" : "Service runs as root, option does not apply",
      "exposure" : null
   },
   {

    ...

   {
```

```
        "set" : true,
        "name" : "NotifyAccess=",
        "json_field" : "NotifyAccess",
        "description" : "Service child processes cannot alter service ↙
            ↳ state",
        "exposure" : null
    },
    {
        "set" : false,
        "name" : "UMask=",
        "json_field" : "UMask",
        "description" : "Files created by service are world-readable ↙
            ↳ by default",
        "exposure" : "0.1"
    }
]
```

The output when grading:

```
> systemd-analyze security
UNIT                                     EXPOSURE PREDICATE HAPPY
adsuck.service                                9.6 UNSAFE    😨
alsa-state.service                            9.6 UNSAFE    😨
archlinux-keyring-wkd-sync.service            2.0 OK        🙂
atd.service                                   9.6 UNSAFE    😨
…
```

Some containerization solutions choose to not rely on Linux isolation features, such as proot, instead hijacking system calls by relying on ptrace, so called “ptrace sandbox”. This is used by Arts for instance. Another similar but more secure approach is gVisor, which uses ptrace but interprets them itself, kind of like User-Mode Linux (UML).

When it comes to oblivious-isolation wrappers, there are also quite a few tools doing similar jobs.

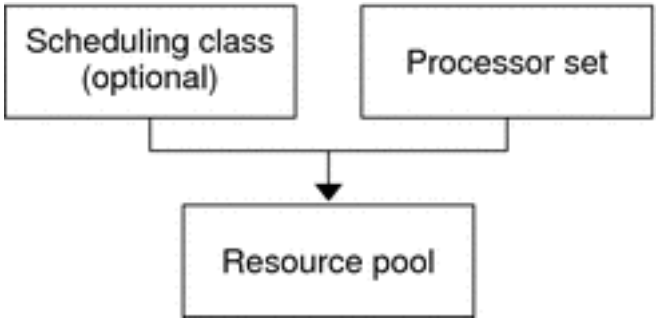
- bubblewrap
  A tool (`bwrap(1)`) used mainly by the flatpak project to do unprivileged sandboxing.
- firejail
  a SUID program that wraps program in sandboxes by relying on namespaces and `seccomp-bpf`.
- minijail
  A sandbox and containment tool mainly used in ChromeOS, Android, and Google internal teams. It regroups most of the Linux security features in a single command line utility. One of the perks is that it offers and easier `seccomp` filter policy syntax (see.
- nsjail
  A tool very similar to `minijail` (also a Google project) but it relies on the Kafel for the `seccomp-bpf` syntax, which is also much easier.

What you need to remember: *There's an explosion of Linux tools intertwining multiple isolation and security features. They are either used for creating containers or to sandbox programs.*

Isolation on SunOS Derivatives

Solaris Projects & Pools

In addition to the resource controls, `rctl`, we've seen in a previous section, Solaris also offers something called a resource pool. It is used to bind a scheduler along with a set of processors, and then assign these to processes, "tasks", or "projects". It is used to efficiently divide the workload on the system when multiple users are using it for different purposes.
A CPU can only be part of one processor set at a time.



The pool of CPU is managed with the `pooladm(8)` command and configured using the `poolcfg(8)` command or manually in `/etc/pooladm.conf`. This can also be done programmatically with `libpool(3LIB)`. A default pool exist as `pool.default`, and is assigned a default `pset.default` processor set.

These are either configured manually in `/etc/pooladm.conf` or using the command line tools `poolcfg(8)`. For example, here's an example creating a CPU set called `queen_set` and associate it with the pool `queen_pool`, and then refreshing the configuration of `pooladm`.:

```
1 > poolcfg -c 'create pset queen_set (uint pset.min=1 ; uint ↙
    ↳ pset.max=2)'
2 > poolcfg -c 'create pool queen_pool'
3 > poolcfg -c 'associate pool queen_pool (pset queen_set)'
4 > pooladm -c
```

Here's another example associating a scheduling class:

```
1 > poolcfg -c 'modify pool pool_queen (string pool.scheduler="FSS")'
```

We can interrogate the current pool configuration using the following:

```
> poolcfg -c info
...
pool pool_queen
  boolean pool.default false
  boolean pool.active true
  int pool.importance 1
  string pool.scheduler FSS
  pset batch

pset pset_queen
  int pset.sys_id -2
  string pset.units population
  boolean pset.default true
  uint pset.max 10
  uint pset.min 2
  boolean pset.escapable false
  uint pset.load 0
  uint pset.size 0

  cpu
          int     cpu.sys_id 5
          string  cpu.comment
          string  cpu.status on-line

  cpu
          int     cpu.sys_id 4
          string  cpu.comment
                  string  cpu.status on-line
```

These come into practice when assigning them to processes, "tasks", and "project". We'll see what tasks and projects are but let's first take a look at process pool assignment.

This can be done using the `poolbind` command, it binds projects, tasks and processes to a pool. With the `-e` option, it can execute a command, move the target to a pool, or determine which pool they are currently associated with. For example, we can bind the running shell to the `pool_queen` (`-i pid` is the default behavior):

```
> poolbind -i pid -p pool_queen $$
```

Resource pools are more useful when assigned to projects, which are an aggregation of related "tasks", which represents a workload.

Projects, like users, groups, roles, and profiles, are defined in a colon-separated file, the project database `project(5)`, `/etc/project` (note that this has nothing to do with quotas). It contains the project name, along with other fields related to resource control. If the project name starts with `user.` or `group.` followed by an actual user or group name, then this project will automatically be assigned to these subjects. Pools can either be assigned directly in this file, or with the `poolbind` utility we've just seen:

```
user.vnm:2001:Venam:::project.pool=queen_pool
```

Equivalent to doing it dynamically:

```
> poolbind -i project -p pool_queen user.vnm
```

Instead of manually editing this file, the commands `projadd(8)` and `projmod(8)` can be used:

A task is anything that is launched under a project using the `newtask` command. This is used to either invoke a new command, or move an already started process to a project.

```
> newtask -p projectname <command>
```

For instance, to launch a process in the "important" project.

```
> newtask -l -p important
```

Obviously, to move tasks or launch in a project, the user needs to belong to it. The command `id(1)` and `project(1)` can get this information.

```
> id -p
uid=565(gh) gid=10(staff) projid=10(default)
```

Projects can additionally have special resource control attributes that start with `rcap`, these will be managed by a user-space daemon called `rcapd`, the capping daemon, instead of the in-kernel `rctl`. For example, `rcap.max-rss`.
The capping daemon is configured using `rcapadm` and its statistics are monitored with `rcapstat(1)`. It can also be used for processes.
Yet, the capping daemon only has one resource value described in its man page: `rcap.max-rss`, the total amount of physical memory that is available to the subject.

What you need to remember: *Solaris pools are a way to divide workload by assigning cpu to sets along with a scheduling algorithm, and then bind these to running processes (`poolbind`). This can be combined with projects and tasks, tasks being processes that are launched with `newtask` under a projects. Users are assigned to projects in the project database using `projadd`/`projmod` or manually editing the file. Additionally, projects and processes can rely on a user-space daemon `rcapd`, the capping daemon, but it's limited to `rcap.max-rss` attribute only.*

Solaris Zones

Solaris zones are roughly akin to FreeBSD jails, which we've seen in another section. They're used to compartmentalize processes that give the appearance of being run on a separate system. The isolation includes, process tree, networking, file system, resources, and more. However the resemblance with FreeBSD jails stops here.

One difference with FreeBSD jails, is that the system by default is considered to run in a zone called the "global" zone (zone id is always `0`). This is the zone that can boot, access to system hardware, networking, and everything else, the one which all processes are assigned to if not already assigned to another zone.

Zones are identified by their name, id, and a root directory path on the global zone system.

They are managed sort of like virtual machines. They need to be configured, installed, booted, and then login, etc…

The interactive `zonecfg` command is used to initially set them up, defining all their configuration, this includes network interfaces, the root directory, resource control, and more.

```
> zonecfg -z zonequeen
 zonequeen: No such zone configured
 Use 'create' to begin configuring a new zone.
 zonecfg:zonequeen> create
 zonecfg:zonequeen> set zonepath=/opt/zones/zonequeen
 zonecfg:zonequeen> add net
 zonecfg:zonequeen:net> set physical=lo0
 zonecfg:zonequeen:net> set address=127.0.0.100
 zonecfg:zonequeen:net> end
 zonecfg:zonequeen> verify
 zonecfg:zonequeen> commit
 zonecfg:zonequeen>
```

This is where we can assign zones to pools `set pool=`, or resource control `rctl`, and others. Additionally, there are specific sub-commands to `zonecfg` creation tool such as `capped-cpu`, `capped-memory`, and others. These can also be changed later on.

After the zone configuration, it needs to be installed on the file system using `zoneadm`. This command is also used to boot, halt, list, reboot, shutdown, and other general zone management activities. Like a physical machine, the zone is in a state, which can be: `CONFIGURED`, `INCOMPLETE`, `INSTALLED`, `READY`, `RUNNING`, `SHUTTING_DOWN/DOWN`.

```
> zoneadm -z zonequeen install
```

Now that the zone is installed, we can boot it and login to it:

```
> zoneadm -z zonequeen boot
> zlogin zonequeen
```

Once in a zone, we can use the `zonename` to print the name of the current zone. If we're not in any yet, this means we're in the global zone:

```
> zonename
global
```

What you need to remember: *Solaris zones are similar to FreeBSD jails in the way they isolate processes, however they are managed more like virtual machines than `chroot` file systems. The zones need to be configured via `zonecfg`, installed, booted, and administered with `zoneadm`, and used (`zlogin`). Zones, like most things on Solaris and SunOS derivatives, can be assigned resource limitations.*

macOS and Android Sandboxes

macOS and iOS (XNU) offer a sandbox solution that provides programmatic self-isolation and oblivious/dumb/external-isolation. The feature was introduced in Mac OS X Leopard under the codename Seatbelt and was based on the kernel hooks of the TrustedBSD MAC framework we've explored in another section. It adds on top of the MAC framework a kernel extension called Sandbox.kext to enforce sandboxing profiles decision making. Additionally, macOS also provides a System Integrity Protection mechanism, SIP, that applies to every process.

The documentation for the sandboxes is scarce as Apple considers it private and subject to change. A few people have reversed engineered the inner-workings by analyzing the publicly available information. This is what the information here is based on.

The sandbox allows restricting access to multiple parts of the system, which can include file operations, IPC, Mach, networking, executable invocation, `sysctl` changes, system calls, and others.

When a process is in a sandbox, all its children will also indirectly inherit the sandbox. The current sandbox a process is in will be stored in a process MAC label. There also exists a daemon `sandboxd(8)` which purpose is to perform user-space management on behalf of the kernel extension.

As we said, the sandbox can be either created programmatically or by using a wrapper called `sandbox-exec(1)`.

The manpage documentation for the self-isolation method, relying on `sandbox_init(3)`, only gives us predefined profiles to prohibit certain aspects of the system:

- `kSBXProfileNoInternet`
- `kSBXProfileNoNetwork`
- `kSBXProfileNoWrite`
- `kSBXProfileNoWriteExceptTemporary`
- `kSBXProfilePureComputation`

Yet, newer sources mention that `sandbox_init(3)` isn't required anymore, but that a signature mechanism is instead the norm. The self-isolation could also be done directly from Xcode.

The other method relies on `sandbox-exec(1)` utility, which takes as argument a profile policy file and applies it to run an executable. The file passed with the `-f` argument can be either a full path, or simply the name of the profile, if it is present in one of the following directories:

- `/Library/Sandbox/Profiles`
- `/System/Library/Sandbox/Profiles`
- `/usr/share/sandbox`

The profiles use a language called SBPL, SandBox Profile Language, derived from TinyScheme to describe what the sandbox will allow or deny. These files, with the `.sb` extension, are then compiled to a binary form and automatically loaded with the Sandbox.kext (in iOS these are already compiled and built-in).
Apple also offers a container mechanism to bundle these together, created in the `~/Library/Containers/{CFBundleIdentifier}`, directory as a subfolder. A `Container.plist` file will have the compiled SBPL file as a base64 hex value in the `SandboxProfileData` entry.

No real documentation exist for the language definition, however people have reverse engineered it and contributed their findings in a compilation such as here, listing all the primitives available (In case the link is down, a backup is found here).
The language is composed of actions, such as "allow" or "deny", followed by operations on which the action is applied, such as `file*` for all file-related operations, along with more modifiers and filters depending on the operation. Profiles also have the ability to include extra rules from other profiles with the `import` directive.

For example, here's a small profile:

```
;; Comments start with ;
(version 1)
(deny default)

;; Allow read on /usr
(allow file-read*
    (literal "/usr/*")
)
```

Meanwhile, Android's approach to sandboxing relies on both POSIX basic DAC and SELinux policies.

On a first level, every application is assigned a different unique UID/GID, which by itself provides basic process isolation.
On another level, it assigns SELinux labels to different resources on the system and applications, providing access policies.
On a third level, applications run with `seccomp-bpf` enabled, limiting the system calls they are allowed to use, creating an app and kernel boundary.

These are mostly controlled by the Dalvik VM on process creation (`zygote`), assigning and wrapping applications, only allowing them access to data they own and allowing permission to system features based on what is defined in their manifest file.
Applications only communicate together through intents. In the next section, we'll see more about this type of action-based access control.

A more stringent sandboxing capability can be used for services when the `isolatedProcess` feature is turned on. This will run the process in a separate SELinux domain, along with its own sensitivity, basically restricting communication to only the service API (binding and starting).
When looking at the process tree they will look like this:

```
u:r:isolated_app:s0:c512,c768
```

What you need to remember: *macOS/iOS and Android provide sandbox functionalities. macOS sandbox are either self-isolation or oblivious-isolation. It isn't publicly documented. macOS sandbox uses a layer on top of TrustedBSD MAC called Sandbox.kext that will process binary policies compiled from SBPL files (SandBox Profile Language), a TinyScheme language. These can be loaded with the wrapper `sandbox-exec(1)`. Android relies on POSIX DAC, along with SELinux labels and `seccomp-bpf` to wrap applications as soon as they're launched from the Dalvik VM. These applications will then only be able to access what their permissions in the manifest file allows, and to*

communicate through well-defined interfaces. An `isolatedProcess` flag also exist in services for stricter isolation.

Virtual Machines as Sandboxes

One can wonder, if isolation is mostly about virtualisation and not allowing breaking away from this isolation, then why not use virtual machines for this.

This discussion led to the idea of virtual machines as antivirus. Thinking of them as a way to define a domain of operation in which the application will think it is running standalone, with its own users, processes, hardware resources, etc..
The main difference with what we've seen previously is that in VMs, the hardware and hardware instructions are also virtualised.

It is a more totalitarian approach that isn't granular, sort of like a huge wrapper. In a way it is like Solaris zones and FreeBSD jails, needing a whole setup for the environment.

While this type of access control makes sense in a data center environment, it's drawback is that it is resource intensive for smaller use-cases. Furthermore, the management and security boundaries with the host heavily depends on the virtualisation technology in use.
Yet, this is a possibility that exists and has existed for a long time.

One Unix-like OS that took advantage of this idea is Qubes OS which uses the Xen hypervisor to run applications in their own "qubes" which are stripped-down (or even full-fledged) virtual machines based on Fedora, Debian, or Windows.

Other container solutions also rely on something similar such as Kata containers and Firecracker.

What you need to remember: *Virtual machines, VMs, can be thoughts of as isolation mechanism, creating their own domains where even hardware and hardware instructions are virtualised. It makes sense in a data center environment. The drawback is that it is resource intensive. Qubes OS actually implements this idea.*

Image-Based OS & Immutable Distro

Since the advent of containers, a trend has emerged of Unix-like OS, predominantly Linux distributions, that are based around the concept. These take the form of an immutable core system files and packages being installed in sandboxes by a universal/standalone package manager. These immutable distributions are often referred to as "image-based" OS.

There are quite a few of these distributions around, and the package manager of choice is usually Flatpak, and the container manager in-use varies a lot, ranging from docker to distrobox, nsbox, and

a hundred of others that mostly perform the same tasks relying on the Linux isolation features we've covered.
The base/core of some of these distros have somewhat got a standardized way to be upgraded, with a "git for file system tree", called ostree.

Thus, in these operating systems, the users only modifies home directory files, along with the software and files that these software use. The rest of the system can be upgraded without affecting what the user currently interfaces with.

Here's a couple of these systems, you can find more on the awesome-immutable Github project:

- vanillaOS
- Fedora Silverblue
- EndlessOS
- rlxos
- AshOS

An idea that goes hand-in-hand with image-based OS are reproducible builds, which are *a set of software development practices that create an independently-verifiable path from source to binary code*. This is achieved by having software live in their own file-system tree, combining them using union/overlay-mount and bind-mount. This also allows for easy rollback to previous states.
While this doesn't rhyme with more access control mechanism, it is still worth mentioning. Some of the popular distributions using these are nixOS and Guix System.

Now, let's move to another type of access control based on "actions".

What you need to remember: *Many Unix-like OS, mostly Linux distros, have taken the idea of containers at heart and run the home directory and packages in isolated environments, we call them image-based distros. The base/core of the system is immutable and managed in a git-like fashion.*

Action-Based Access Control

In the previous part of this article we've looked at isolation, resource limitations, and constraints, however there still need to be a way for processes to interface with one another. While these could include standard system calls being allowed, in this part we'll focus more on specific actions that are well-defined by a program and of which the access is controlled by another system-wide policy. This is what we've chosen to refer to as "action-based" access control.

These actions could include allowing to mount a device, rebooting, restarting a specific service, manipulating a very particular feature within an application such as changing a color theme, and more.

To achieve this there needs to be a common accepted protocol that programs can use to verify whether the subject is allowed the action, how to request it, a system-wide mechanism to enforce access control, and a strict way to define the interfaces of these actions that a program provides.

Presumably, action-based access control can be achieved like any custom programmatic authentication done today: with a user login via token, and checks on the user permissions from within the application by relying on its own DB. As we mentioned in another section, RBAC and ABAC can be implemented at this level too, binding a role to a software action. Centralized IdM also come to mind, such a OPA, and Cedar.

In this part, we'll take a look at a few implementations of these action-based access control on Unix-like systems: polkit, SunOS derivatives "auths", macOS extension points, and Android intents.

What you need to remember: *We define action-based access control as a system-wide mechanism that enforces control over software-specific features. The system should have a standard protocol and ways to define the interfaces supported by software.*

SunOS derivatives auths

Authorizations, or SunOS auths, are rights that are programmatically checked at run-time by programs to determine whether a user may perform a functionality.

These coarse-grained rights can be assigned to users, profiles, and roles. Authorizations are represented by fully-qualified names, which identify the organization that created them and the functionality that it controls. The components of the authorization string follows the Java convention, it's a reversed hierarchical series of classed separated by dots `.`. Furthermore, the glob/asterisk `*` character can be used to indicate all authorizations in a class. Example: `solaris.printer.postscript`, or `solaris.admin.usermgr.*`.

SunOS derivatives have multiple pre-defined auths related to system management, for example, here's a list of auths related to printer management:

```
solaris.admin.printer.grant
solaris.admin.printer.delete
solaris.admin.printer.modify
solaris.admin.printer.read
```

The list of all auths on a system are defined in the `auth_attr(5)` file, `/etc/security/auth_attr`, the authorization description database. Like other databases on SunOS derivatives, it is a colon-separated list of attributes. Most notably, it contains the name of the auth, a description, and optionally an HTML help page. For instance:

```
solaris.admin.usermgr.:::User Accounts::help=AuthUsermgrHeader.html
solaris.admin.usermgr.pswd:::Change ↙
    ↳ Password::help=AuthUserMgrPswd.html
solaris.admin.usermgr.write:::Manage Users::help=AuthUsermgrWrite.html
```

After defining the auths, they are then assigned to users and roles in the `user_attr(5)` file that we've seen in another section via the `auths` attribute, which is a list of comma separated auths.
For profiles the auths are assigned in `/etc/security/prof_attr` file, within the `attr` entry as an `auths` key.
Both of these are merged with the default policy found in `policy.conf(5)`, `/etc/security/policy.conf`, key `AUTHS_GRANTED`.

The list of auths that a user is assigned can be checked using the `auths(1)` command.

```
> auths vnm queen
vnm : solaris.system.date,solaris.jobs.admin
queen : solaris.system.*
```

The `auths(1)` command also include sub-commands such as `add`, `check`, `info`, and `list` to manage the authorizations (`getent auth_attr` NSS utility can also be used).

```
> auths add -t "manage foo"\
    -h /home/abc/AuthFoo.html solaris.foo.manage
```

Programs then have to programmatically call `getauthattr()` or `chkauthattr()` functions to get access to the information found in the above databases and verify if the subject has access to the functionality.

One useful administration command that relies on auths is `pfedit(8)`. It reads the `solaris.admin.edit</path_to_file>` and allows editing capability to the specified file. For example: `auths=solaris.admin.edit/etc/syslog.conf` allows editing `/etc/syslog.conf` by invoking:

```
> pfedit /etc/syslog.conf
```

The full list of pre-defined system auths is hard to find, however these are some of the common ones:

- `solaris.device.*`: Device-related auths
- `solaris.network.*`: Network-related auths
- `solaris.account.*`: Account-related auths
- `solaris.zone.*`: Zone-related auths
- `solaris.admin.*`: Admin-related auths
- `solaris.profmgr.*`: Rights-related auths
- `solaris.system.*`: System management related auths (reboot/shutdown)
- `solaris.jobs.*`: Cron-related management auths

In combinations with roles and profiles, this can be a tremendous way to discretely split what certain subjects are allowed to do. However, it can also quickly get messy as profiles and roles get intermixed, and nobody knows where, and what, permissions are set.

What you need to remember: *SunOS derivatives authorizations, auths, are names given to functionalities within programs. The programs have to programmatically check whether the subject is allowed or not (`chkauthattr`). They are defined in the authorization description database (`auth_attr`), and assign to user, roles, and profiles, in their respective database (`user_attr`, `prof_attr`). A couple of pre-defined auths exists too for system management.*

Polkit/D-Bus

D-Bus is a message bus which runs system-wide or user-wide, mostly on Linux distributions. In simple terms, that means programs register as services that fulfill particular actions on this centralized bus, which are then requested by other software. The action is performed on behalf of the program, in an RPC-fashion.

The actions/methods that can be requested on "objects", can either be introspectively checked from the dbus service, or analyzed by looking at pre-defined XML interfaces files found in `/usr/share/dbus-1` or `/etc/dbus`, depending on the installation details.
Thus, applications send signals or messages to this bus for the methods exposed by the services. Meanwhile, other programs act as services, implementing things that other programs can ask for.

There's a couple of tools for monitoring and debugging dbus such as:

- `dbus-send`
- `gdbus`
- `qdbus`
- `d-feet`

More info can be found in one of my previous article entirely dedicated to the topic of dbus and polkit. Some parts of the article are reproduced here.

Hence, Polkit, formerly PolicyKit, is one such service running on dbus, `polkitd`, that offers to clients a way to perform granular system-wide authorization for specific actions. These programs rely on one of the policykit library, such as libpolkit-gobject-1, raw dbus api (or here) interfacing with polkit, or any of the many implementations, to add checks right before the action. In polkit parlance, we talk of `MECHANISMS`, privileged services, that offer actions to `SUBJECTS`, which are unprivileged programs. Polkit will then perform the appropriate checks that are defined, and ask an "authentication agent" if needed. The authentication agent is another service attached to dbus that has as role to ask the user/subject to authenticate themselves. Here's a couple of possible authentication agents:

- `pkttyagent`: the default textual agent coming with polkit
- `lxqt-policykit` - which provides `/usr/bin/lxqt-policykit-agent`
- `lxsession` - which provides `/usr/bin/lxpolkit`
- `mate-polkit` - which provides `/usr/lib/mate-polkit/polkit-mate-authentication-agent-1`

- `polkit-efl` - which provides `/usr/bin/polkit-efl-authentication-agent-1`
- `polkit-gnome` - which provides `/usr/lib/polkit-gnome/polkit-gnome-authentication-agent-1`
- `polkit-kde-agent` - which provides `/usr/lib/polkit-kde-authentication-agent-1`
- `ts-polkitagent` - which provides `/usr/lib/ts-polkitagent`
- `xfce-polkit` - which provides `/usr/lib/xfce-polkit/xfce-polkit`

Services/mechanisms have to define the set of actions for which clients require authentication. This is done through defining a policy XML files in the `/usr/share/polkit-1/actions/` directory. The actions are defined in a namespaced format, and there can be multiple ones per policy file. These include a wide-array of things such as mounting disks, configuring network interfaces, rebooting, suspending, etc..
These files contain the policies telling polkit whether to allow, deny, or prompt the user for a password.

These files define metadata information for each action, such as the vendor, the vendor URL, the icon name, the message that will be displayed when requiring authentication in multiple languages, and the description. The important sections in the `action` element are the `defaults` and `annotate` elements.

The `defaults` element is the one that polkit inspects to know if a client is authorized or not. It is composed of 3 mandatory sub-elements: `allow_any` for authorization policy that applies to any client, `allow_inactive` for policy that apply to clients in inactive session on local console, and `allow_active` for client in the currently active session on local consoles.
These elements take as value one of the following:

- `no` - Not authorized
- `yes` - Authorized.
- `auth_self` - The owner of the current session should authenticate (usually the user that logged in, the user password)
- `auth_admin` - Authentication by the admin is required (root)
- `auth_self_keep` - Same as `auth_self` but the authentication is kept for some time that is defined in polkit configurations.
- `auth_admin_keep` - Same as `auth_admin` but also keeps it for some time

NB: The timeout is currently hardcoded as 5min.

The admin identity is anyone defined as a `pklocalauthority(8)` in the configuration file, but it can also be defined through rules, as we'll see.

```
1 AdminIdentities=unix-user:0
2 AdminIdentities=unix-group:sudo;unix-group:admin
```

The `annotate` element is used to pass extra key-value pair to the action. There can be multiple key-value that are passed. Some annotations/key-values are well known, such as the `org.freedesktop.policykit.exec.path` which, if passed to the pkexec program that is shipped by default with polkit, will tell it how to execute a certain program.
Another defined annotation is the `org.freedesktop.policykit.imply` which will tell polkit that if a client was authorized for the action it should also be authorized for the action in the `imply` annotation.
One last interesting annotation is the `org.freedesktop.policykit.owner`, which will let polkitd know who has the right to interrogate it about whether other users are currently authorized to do certain actions or not.

Other than policy actions, polkit also offers a rule system that is applied every time it needs to resolve authentication. The rules are defined in two directories, `/etc/polkit-1/rules.d/` and `/usr/share/polkit-1/rules.d/`. As users, we normally add custom rules to the `/etc/` directory and leave the `/usr/share/` for distro packages rules.
Rules within these files are defined in javascript and come with a preset of helper methods that live under the `polkit` object.

The `polkit` javascript object comes with the following methods, which are self-explanatory.

- `void addRule( polkit.Result function(action, subject){...});`
- `void addAdminRule( string[] function(action, subject){...});` called when administrator authentication is required
- `void log( string message);`
- `string spawn( string[] argv);`

The `polkit.Result` object is defined as follows:

```
polkit.Result = {
    NO               : "no",
    YES              : "yes",
    AUTH_SELF        : "auth_self",
    AUTH_SELF_KEEP   : "auth_self_keep",
    AUTH_ADMIN       : "auth_admin",
    AUTH_ADMIN_KEEP  : "auth_admin_keep",
    NOT_HANDLED      : null
};
```

Note that the rule files are processed in alphabetical order, and thus if a rule is processed before another and returns any value other than `polkit.Result.NOT_HANDLED`, for example `polkit.Result.YES`, then polkit won't bother continuing processing the next files. Thus, file name convention does matter.

The functions `polkit.addRule`, and `polkit.addAdminRule`, have the same arguments, namely an action and a subject. Respectively being the `action` being requested, which has an `id` attribute, and a `lookup()` method to fetch annotations values, and the `subject` which has as attributes the `pid`, `user`, `groups`, `seat`, `session`, etc, and methods such as `isInGroup`, and `isInNetGroup`.

Here are some examples taken from the official documentation:

Log the action and subject whenever the action `org.freedesktop.policykit.exec` is requested.

```
polkit.addRule(function(action, subject) {
    if (action.id == "org.freedesktop.policykit.exec") {
        polkit.log("action=" + action);
        polkit.log("subject=" + subject);
    }
});
```

Allow all users in the admin group to perform user administration without changing policy for other users.

```
polkit.addRule(function(action, subject) {
    if (action.id == "org.freedesktop.accounts.user-administration" &&
```

```
        subject.isInGroup("admin")) {
        return polkit.Result.YES;
    }
});
```

Define administrative users to be the users in the wheel group. This is one of the default rules that comes with polkit installation, it means that when `auth_admin` is present the user will be prompted for the user password and not the root password.

```
polkit.addAdminRule(function(action, subject) {
    return ["unix-group:wheel"];
});
```

Run an external helper to determine if the current user may reboot the system:

```
polkit.addRule(function(action, subject) {
    if (action.id.indexOf("org.freedesktop.login1.reboot") == 0) {
        try {
            // user-may-reboot exits with success (exit code 0)
            // only if the passed username is authorized
            polkit.spawn(["/opt/company/bin/user-may-reboot",
                          subject.user]);
            return polkit.Result.YES;
        } catch (error) {
            // Nope, but do allow admin authentication
            return polkit.Result.AUTH_ADMIN;
        }
    }
});
```

The following example shows how the authorization decision can depend on variables passed by the pkexec(1) mechanism:

```
polkit.addRule(function(action, subject) {
    if (action.id == "org.freedesktop.policykit.exec" &&
        action.lookup("program") == "/usr/bin/cat") {
        return polkit.Result.AUTH_ADMIN;
    }
});
```

Keep in mind that polkit will track changes in both the policy and rules directories, so there's no need to worry about restarting polkit, changes will appear immediately.

A tool that comes pre-installed with polkit is `pkexec(1)`. This program allows executing a command as another user, by default executing it as root. It is a sort of `sudo` replacement but that may appear confusing to most users who have no idea about polkit. However, the integration with authentication agent is quite nice.
`pkcheck(1)` can be used to check if a process is authorized to perform an action, when the authentication is kept for some time, it can be used as a dummy authentication agent.

Polkit also offers some excellent manpages that are extremely useful, be sure to check `polkit(8)`, `polkitd(8)`, `pkcheck(1)`, `pkaction(1)`, `pkexec(1)`.
The following tools are of help:

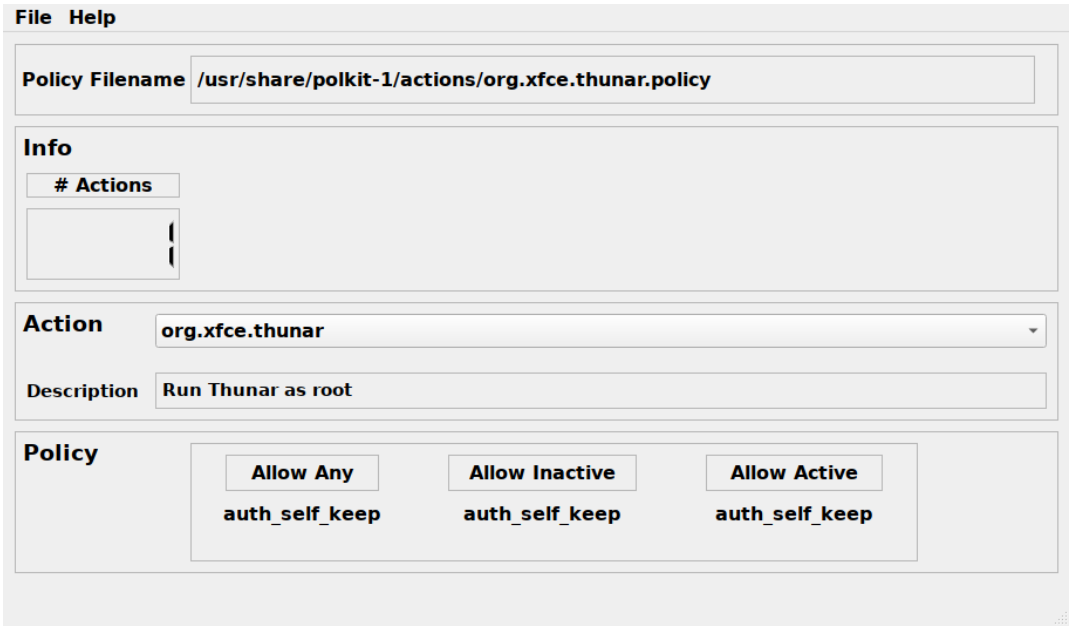
- `polkit-explorer` or `polkitex` - a GUI to inspect policy files
- pkcreate - a WIP tool to easily create policy files, but it seems it is lacking
- `pkcheck` - Check whether a subject has privileges or not
- `pkexec` - Execute a command as another user

Let's test through some examples.

First `pkaction(1)`, to query the policy file.

```
> pkaction -a org.xfce.thunar -v

org.xfce.thunar:
  description:       Run Thunar as root
  message:           Authentication is required to run Thunar as root.
  vendor:            Thunar
  vendor_url:        https://xfce.org/
  icon:              system-file-manager
  implicit any:      auth_self_keep
  implicit inactive: auth_self_keep
  implicit active:   auth_self_keep
  annotation:        org.freedesktop.policykit.exec.path -> ↙
      ↳ /usr/bin/thunar
  annotation:        org.freedesktop.policykit.exec.allow_gui -> true
```

Compared to `polkitex`:



We can get the current shell PID.

```
> ps
    PID TTY          TIME CMD
 421622 pts/21   00:00:00 zsh
 421624 pts/21   00:00:00 ps
```

And then give ourselves temporary privileges to `org.freedesktop.systemd1.manage-units` permission.

```
> pkcheck --action-id 'org.freedesktop.systemd1.manage-units' ↙
    ↳ --process 421622 -u
> pkcheck --list-temp
authorization id: tmpauthz10
action:           org.freedesktop.systemd1.manage-units
subject:          unix-process:421622:195039910 (zsh)
obtained:         26 sec ago (Sun Jun 28 10:53:39 2020)
expires:          4 min 33 sec from now (Sun Jun 28 10:58:38 2020)
```

As you can see, if the `auth_admin_keep` or `auth_self_keep` are set, the authorization will be kept for a while and can be listed using `pkcheck`.

You can try to exec a process as another user, just like sudo:

```
> pkexec /usr/bin/thunar
```

If you want to override the currently running authentication agent, you can test having `pkttyagent` running in another terminal passing it the `-p` argument for the process it will listen to.

```
# terminal 1
> pkttyagent -p 423619
# terminal 2
> pkcheck --action-id 'org.xfce.thunar' --process 423619 -u
# will display in terminal 1
polkit\56temporary_authorization_id=tmpauthz13
polkit\56retains_authorization_after_challenge=true
==== AUTHENTICATING FOR org.xfce.thunar ====
Authentication is required to run Thunar as root.
Authenticating as: vnm
Password:
==== AUTHENTICATION COMPLETE ====
```

Dbus also offers integration with SELinux, for SELinux-aware applications that can interpret security context. It can be used to make authentication more strenuous.
Two SELinux class exist related to d-bus, one is `acquire_svc`, that allows binding as a service, and the other `send_msg`, which allows sending a message to a service. These can be used while defining services with an SELinux context, in the `busconfig` section. After having a context on dbus, SELinux policies can be constructed such as:

```
<busconfig>
  <selinux>
    <associate
```

```
      own="com.example.dnsmasq"
      context="system_u:object_r:dnsmasq_t:s0" />
  </selinux>
</busconfig>
```

```
allow dnsmasq_t self:dbus { acquire_svc send_msg };
allow sysadm_t dnsmasq_t:dbus send_msg;
allow dnsmasq_t sysadm_t:dbus send_msg;
```

Here we can see that `sysadm_t` domain can send dbus messages to `dnsmasq_t` domain, and that the `dnsmasq_t` domain can bind to dbus as a service and send messages.

Lastly, dbus is widely used by sandboxed applications to request well-known desktop features through the `xdg-desktop-portal` interfaces defined in `/usr/share/xdg-desktop-portal`.

What you need to remember: *Polkit is a dbus authentication and authorization service. Dbus is a message bus technology, allowing to define interfaces with object methods that any software binding to the bus can implement, polkit being one of them. Software rely on the polkit API to perform checks before a privileged action. They also define these action in an XML format within polkit configuration directory `/usr/share/polkit-1/actions/`, which will decide how to authenticate the user. Additionally, polkit has a dynamic javascript rule mechanism found in `/usr/share/polkit-1/rules.d/` which is consulted when authentication is required. Let's note that dbus messages and service binding can have SELinux context attached to them and a rule can be used to consider them as an SELinux class.*

macOS Extension Points & Android Intents

We've seen in another section that Android offers sandboxing between applications. These applications are only allowed access to the permissions declared in their manifest file. The user is then explicitly asked, at install-time and runtime, whether they agree on these permissions, and is also able to granularly manipulate them in the OS settings.

Apart from this permission mechanism to access OS features, Android also offers a way for applications to communicate together called Intents. An Intent is a message requesting an action from another application offering it, by passing it to the Android system as intermediary, just like message bus. To say that an application supports an Intent message, an intent-filter should be defined in the manifest file. This is referred to as the "implicit" way to declare intents, by letting the Android system resolve them. Another way involves directly asking it from another application, without having it defined in the manifest.

```
<activity android:name="ShareActivity" android:exported="false">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

However, Intents don't enforce any access control mechanism, and thus each applications have to deal with this internally. Yet, it's the primary method for IPC between applications. Some research have tried adding such access control on top, such as: Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook.

Meanwhile, on macOS (iOS, iPadOS), there's support for something called "extension points", which are the way one application can provide functionalities to another one. These are very similar to Android Intents, however, they live as their own part away from the main/host application, having their own life cycle and are sandboxed apart from the rest. However, they share the same access to privacy controls as the host application.
Yet, just like Android, this is mostly used to fulfill common tasks on a system, and doesn't include any type of access control by default.

What you need to remember: *Android Intents and macOS extensions points are a way for applications to register actions/functionalities that they can fulfill, similar to a message bus. These, by themselves don't include any access control.*

After the Facts: Logging & Auditing

We've seen a substantial amount of methods to perform access control. Yet, this makes no sense if there's no way to make sure of what happened on a system. This is where logs come in, after every activity or decision taken from an access control mechanism, it needs to be logged, and these logs need to be protected from attempted manipulation.
Since this isn't the core of this article, we'll not dive into gritty details in the following sections.

Classic System Logger

Logs are simply traces of actions left in a file (or any storage), and as such, they could be generated haphazardly. A software developer can create their own logging format and storage mechanism.

However, on Unix-like system, a centralized system-based log standard called syslog has emerged. There are two views of centralization. It could mean that a single system takes all the messages and logs, or it could mean that all the log files are in the same location (here usually `/var/log/`). For the RFC 5424 de-facto standard of syslog it means both.
Since this is an RFC, every Unix-like OS can chose to have their own implementation while keeping a bare minimum that is coherent across them. Indeed most Unix-like OSes have it including Solaris, OpenBSD, FreeBSD, Linux, etc.. Often referring to the service simply as `syslogd`. For example, on Linux, the main implementation is called `rsyslog`, but there also exist others such as `syslogng`.

In general, the daemon is configured in a file such as `/etc/syslog.conf`, a configuration which will have lines deciding where to route logs based on a tagging system. For instance, this can be useful to separate authentication logs.

```
1 authpriv.*        /var/log/secure
```

Additionally, syslog offers other metadata to attach to logs, such as priority level (`NOTICE`, `WARNING`, `ERROR`, ..), "term", and "selector". The configuration can also allow to route logs to other syslog servers, pipes, sockets, email, and more.

Programs that want to rely on this have to use the `syslog(3)` library, calling functions such as:

```
1 void syslog(int priority, const char *format, ...);
```

It is also possible to use the utility `logger(1)` to log events. Example:

```
1 logger -p local0.notice -t host "hello world"
```

A recent replacement, coming from systemd, has taken over syslog in the Linux world: journald. It uses its own logging system called a journal, a binary format stored in `/run/systemd/journal`, but can still act as an in-place syslog too if need-be.

Since it uses a binary format, the command `journalctl` is required to be able to search through the logs. It can both be used in user mode by passing the `--user` flag or in system-wide mode.

The journal is configured through `/etc/systemd/journald.conf`, which has options for compression, splitting, syncing interval (for when it will actually write to disk), max use, maximum runtime, if forwarded to syslog, max storage, level of priority, etc..

When it comes to actual usage of these logs for accounting, PAM and many others rely on `syslog(3)`.

Meanwhile, the shadow password suite, relies on `lastlog(8)`, manually writing to a database of previous logins in `/var/log/lastlog`, which is configured in `login.defs`.

Another file related to `login(1)` records is the `utmp`/`utmpx`/`wtmp`/`btmp` used to record all logins and logouts in a binary format from POSIX (or mostly inspired by it because not officially part of the standard). It is accompanied with functions such as `logwtmp(3)`, and command line utilities to process the files such as `last(1)`, `lastb(1)`, `utmpdump(1)`, `who(1)`, and `w(1)`. It is the responsibility of the application itself to use the function and write to these logs (`login`, `ssh`, ..)

Finally, OpenBSD offers a form of particular accounting via the `acct(5)` function to notify of the misbehavior and termination status of processes. These are written by the kernel to `/var/account/acct` in a binary format, and can be read using the `lastcomm(1)` utility.

What you need to remember: *syslog is the de-facto logging standard on Unix-like systems, with many implementations. Journald is a Linux systemd alternative. Most programs rely on syslog (like PAM), however a few are manually writing important login files such as `utmp` and `lastlog`.*

POSIX.1e/2c Auditing

The POSIX.1e/2c draft defines auditing constructs and functions, however, just like information labeling, these end up not being used in practice.

The draft expands on the definition of "audit", capturing, storing, analyzing, maintaining, managing events that concern security activities. Along with this it defines what audit log, audit records, and audit events are, namely the destination of the records, the discrete unit of data in a log, and an actual activity written to the log, respectively.
Each event is said to be accountable to a user through its event ID, and is also attached a type, predefined for system events, and freely chosen for application events.

The predefined events are related to changes on the system, usually system calls, such as `AUD_AET_CHDIR`, `AUD_AET_CHMOD`, `AUD_AET_EXEC`, and are accompanied with the related parameters passed to the system call.

The draft specifies both functions to manipulate and reading the audit logs, along with a front-facing standard format for the records in them. The internal format of the audit log is unspecified/opaque and left to implementers, however an audit record should at least contain a description of events, with the goal that they should be able to hold accountable the subject of the event, and pinpoint what was affected by it. Thus, the following is recommended:

- A headers with the event type and a timestamp
- The subject attributes, describing who performed the action

- Zero or more sets of event-specific data, related to the parameter used to perform the action
- Zero or more sets of objects attributes, describing what is affected by the action

The functions defines by POSIX.1e to read, write, control, construct, analyze, save the audit logs all start with the prefix `aud_...` and found in the header `<sys/audit.h>`, along with user-facing structures such as `aud_info_t`. For instance, `aud_write`, `aud_read`, `aud_get_subj`, …

No actual user-land utilities are defined for auditing in POSIX.2c.

What you need to remember: *POSIX.1e/2c draft defines an auditing interface along with a user-facing format and particular sets of system events that will generate audit logs. However, this draft went unused.*

Basic Security Module Auditing

The Sun's Basic Security Module, BSM, along with its open source copy-cat OpenBSM, is a binary event auditing file format, API, along with a set of user-land utilities. The open source version, OpenBSM was created by McAfee Research, sponsored by Apple, then extended by TrustedBSD, and now sponsored by multiple organizations.
This auditing framework is the one used by SunOS derivatives, BSDs, and macOS.

It defines a few auditing-related terms such as:

- Auditing: any log of security-relevant system events that can be monitored for intrusion detection.
- Audit Record: an audit log entry describing a single security event, it contains info such as subject, date, object, etc…
- Audit Tokens: The type of information saved in each audit event, the parts of the record.
- An Audit Trail: A log consisting of a series of audit records describing security events.
- An Audit Class: A named set of related events, used as a selection expression (file creation, exec, login_logout), usually abbreviated.
- Selection Expression: A string containing a list of prefixes, class, to match events, basically search criteria.
- Reduction: The process by which records from existing audit trails are selected for preservation, printing, or analysis.
- Preselection: The process to select which event are of interest to admin, a term used when configuring which events will be auditable (`audit_event(4)`).

The service responsible for auditing is called the `auditd` daemon. It usually stores its configuration and run-time data in `/etc/security/`, such as its runtime PID and current audit file in `/etc/security/audit_data`. For instance, to enable it on FreeBSD, `rc.conf` should have: `auditd_enable="YES"` and the service should be started `service auditd start`.

The `auditd` daemon is configured through different files. The first of which `/etc/security/audit_class`, `audit_class(4)` defines the possible classes of events that can be audited, a table with a list of abbreviated class name and their description. For example:

```
0x0000000000001000:lo:login or logout
0x0000000000100000:ps:process start/stop
0x0000000000200000:pm:process modify
0x0000000020000000:io:ioctl
0x0000000040000000:ex:exec
```

These are then mapped to event numbers and names in the `/etc/security/audit_event` file, basically picking which system events will be part of a class. For example, here are a couple of events related to `lo`:

```
6152:AUE_login:login - local:lo
6153:AUE_logout:logout:lo
6154:AUE_telnet:login - telnet:lo
6155:AUE_rlogin:login - rlogin:lo
```

These all then take effect in the `/etc/security/audit_control`, `audit_control(4)` file, which controls how the audit trail binary file will be created, which classes to look at, the maximum size of audit trail, etc.. For instance:

```
dir:/var/audit
dist:off
flags:lo,aa
minfree:5
naflags:lo,aa
policy:cnt,argv
filesz:2M
expire-after:10M
```

The administrator can also set specific per-user auditing configurations in `/etc/security/audit_user`, for instance:

```
other:lo,am:io,cl
fred:lo,ex,+fc,-fr,-fa:io,cl
ethyl:lo,ex,nt:io,cl
```

There are more files, but we'll cut it short and move on to the utilities part of BSM.
Since the audit trails are stored in a binary format, a couple of tools are require to inspect them.

- `auditreduce(8)`: Audit trail reduction tool, merging multiple records
- `audump(8)`: Debugging tool to parse and print audit databases
- `praudit(8)`: Tool to print audit trails

For example, `auditreduce` can be used to filter by token "user" with the value "queen" in the file "AUDITFILE", this will be output in binary format and thus needs to be piped to `praudit` to "print" them.

```
auditreduce -u queen /var/audit/AUDITFILE | praudit
```

The OpenBSM implementation additionally comes with a pseudo-device file `/dev/auditpipe` that can be used for live audit event tracking. It can be interfaced with using `ioctl` (see `auditpipe(4)`), and is restricted to users part of the `audit` group.

What you need to remember: *Sun's BSM, and OpenBSM, is an auditing facility used by Sun derivatives, macOS, and BSDs. It is configured by using classes/aliases (`/etc/security/audit_class`), mapped to pre-defined system events (`/etc/security/audit_event`), and controlled for which of these classes get audited (`/etc/security/audit_control`), along with more granular per-user options (`/etc/security/audit_user`). Since it uses a binary format to store the audit trail, utilities have to be used to search and parse logs (`praudit`, `auditreduce`).*

Linux Auditing System

Linux auditing system is composed of a kernel part (`CONFIG_AUDIT=y`), and a user-space daemon called `auditd`. It is used, like all the above auditing framework, to perform event log of system calls and any security actions on the system. This can be used for incident detection, forensic, post-mortem analysis, or even the learning-mode we've seen in some MAC such as AppArmor and SELinux.

While the kernel part is loaded with rules and generates the logs, the user-space daemon `auditd` is responsible for helping load the rules in the kernel, and write the audit records to the disk. A set of accompanying utilities are used to manage them.

The daemon is configured via files found in the `/etc/audit` directory. By default it will audit all events happening on the system, however additional rules can be loaded at the start in `audit.rules(7)`, or having this file generated by `augenrules(8)` which will read the sub-directory `rules.d` instead. The rules can also be modified on the fly by issuing them from `auditctl(8)`.
The `auditd` daemon also has configurations that pertains to itself found in `auditd.conf(5)` for things such as the location of the audit file (`log_file`), the format the records will be written in (`log_format`), how log rotation will work, which plugins to load (see `auditd-plugins(5)for plugins such as` audisp-syslog' forwarding audit to syslog), and more.

Note that the audit trail, like all other kernel events, are also logged in `dmesg`.

The `audit.rules(7)` file is a series of `auditctl(8)` commands to modify the behavior that `auditd` will have when encountering certain events. The `auditctl` command can also be used to change the behavior of `auditd` on-the-fly and listing the currently active rules (with `-l`). A set of example rules are found in `/usr/share/audit/sample-rules/`.
The rules are composed of an action, a list, and a set of fields to match against or system call.

The actions can be either: `never`, to not generate the record, or `always`, to allocate an audit context.
The list can either be `task`, `exit`, `user`, `exclude`, or `filesystem`, each list has its specificities. For instance, the `user` list are for messages originating from user-space, while the `exclude` list is a separate exclusion list to filter events, it ignores the action and defaults to `never`.
The set of fields and system calls contains anything imaginable that can be captured by the kernel. This includes things such as `success`, whether the action was successful, `euid`, the effective UID, `exe`, the absolute path of the executable, and more.

The `-w` parameter can be passed to watch a particular directory, this is useful when audit is disabled system-wide.

Here's a couple of interesting rules:

To see unsuccessful `openat` calls:

```
> auditctl -a always,exit -S openat -F success=0
```

To suppress events originating from the file system:

```
> auditctl -a never,filesystem -F fstype=tracefs
> auditctl -a never,filesystem -F fstype=debugfs
```

These can then be listed using:

```
> auditctl -l
-a never,filesystem -F fstype=tracefs
-a never,filesystem -F fstype=debugfs
```

The `aureport(8)` command can be used to generate a summary report of the system:

```
> aureport

Summary Report
======================
Range of time in logs: 09/10/2022 16:24:57.620 - 02/23/2023 ↙
    ↳ 18:44:03.570
Selected time for report: 09/10/2022 16:24:57 - 02/23/2023 ↙
    ↳ 18:44:03.570
Number of changes in configuration: 2
Number of changes to accounts, groups, or roles: 0
Number of logins: 0
Number of failed logins: 0
Number of authentications: 24
Number of failed authentications: 18
Number of users: 3
Number of terminals: 10
Number of host names: 2
Number of executables: 13
Number of commands: 8
Number of files: 0
Number of AVC's: 0
Number of MAC events: 0
Number of failed syscalls: 0
Number of anomaly events: 6
Number of responses to anomaly events: 0
Number of crypto events: 0
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 99
Number of events: 800
```

Since the audit system keeps track of all system calls, an equivalent to `strace(1)` and `lastlog(8)` exist as `autrace(8)` and `aulastlog(8)` respectively.

However, the most useful too is `ausearch(8)`, used to search the audit logs. It has many options allowing to filter by fields, using a checkpoint-file, using start-end timestamps, and more.

An example of searching for events by user-id.

```
> ausearch -ua vnm
----
time->Thu Feb 23 18:48:38 2023
type=USER_ACCT msg=audit(1677170918.278:931):
pid=680257 uid=1000 auid=1000 ses=3 msg='op=PAM:accounting
grantors=pam_unix,pam_permit,pam_time acct="vnm" exe="/usr/bin/sudo"
hostname=? addr=? terminal=/dev/pts/10 res=success'
----
time->Thu Feb 23 18:48:38 2023
type=CRED_REFR msg=audit(1677170918.291:932):
pid=680257 uid=1000 auid=1000 ses=3 msg='op=PAM:setcred
grantors=pam_faillock,pam_permit,pam_faillock acct="root"
exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/10 ↙
    ↳ res=success'
----
time->Thu Feb 23 18:48:38 2023
type=USER_START msg=audit(1677170918.311:933):
pid=680257 uid=1000 auid=1000 ses=3 msg='op=PAM:session_open
grantors=pam_systemd_home,pam_limits,pam_unix,pam_permit acct="root"
exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/10 ↙
    ↳ res=success'
```

A similar example but relying on the session id (see `loginctl`):

```
> ausearch --session 3 -m USER_AUTH --success no
----
time->Wed Feb 22 21:12:55 2023
type=USER_AUTH msg=audit(1677093175.088:778): pid=663799 uid=1000
auid=1000 ses=3 msg='op=PAM:authentication grantors=? acct="vnm"
exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/10 res=failed'
----
time->Thu Feb 23 18:15:02 2023
type=USER_AUTH msg=audit(1677168902.416:863): pid=672948 uid=1000
auid=1000 ses=3 msg='op=PAM:authentication grantors=? acct="vnm"
exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/10 res=failed'
```

Lastly, if the Linux auditing system should be used in a CAPP (Common Criteria) environment, a couple of additional things need to be kept in mind. The framework should be enabled at boot time through a boot parameter `audit=1`, and the system should be denied access as soon as it cannot write to the audit trail.

Let's move to another important part: general security.

What you need to remember: *Linux audit framework is composed of a kernel component with a user-space daemon `auditd` that will receive, filter, and write them to disk. The daemon is configured in `/etc/audit` for how it will handle the writing of log and plugins (`auditd.conf`), and for the rules that*

will be applied to them (`audit.rules` and `rules.d`). A couple of utilities exist to manage (`auditctl`), search the audit trail (`ausearch`), and do much more.

General Security & Trusted Computing Base

"A system is as secure as its weakest point". Thus, in this last part we'll wrap up by looking at things outside the scope of access control decision-making. Unsurprisingly, A compromise at the operating system level quickly shows the limitations of any framework. We'll swiftly go over some of the practices such as exploit mitigation, general security, threat modeling, and the trusted computing base.

Let's start by going over the term trusted computing base, or TCB for short. It refers to the set of all parts and components of a system that must not be compromised to keep it "secure", all the critical parts, whatever the definition of security is (see introduction section). A bug, misbehavior, or vulnerability in these critical parts would jeopardize the security properties of the entire system. Hence, it is mandatory to keep the core pieces of the system under such rigid criteria.

Let's examine a few ways to achieve this, apart from keeping the code error-free and out of supply-chain attacks (BOM, bill of material, SDLC, secure development life-cycle, …).

One aspect involves keeping the integrity of the system, making sure it originates from the right place, and isn't modified afterward. This is akin to Biba security model but applied outside access control. This can be done through file integrity system and code-signing mechanisms, and can be applied at different levels.

The concept of trusted platform, trusted base, trusted execution environment (TEE), and trusted path come mind. These can either refer to completely isolated environment, code that is signed by a hardware cryptographic key (HSM), or any way that a user can interface with a "trusted" software.
This can also be applied at boot time, and is often called secure boot, verified boot, or trusted boot, allowing authentication of boot partition files through hashing and file integrity mechanisms.

Yet, this only covers trust, and, as we said in the introduction, a system that is trustworthy is not necessarily a system we must trust or that is following our definition of security.

Hence, attack surface reduction and mitigation are also important. This can be achieved at the level of the processor, such as with the CHERI ISA we've seen.
Or it can be done, as is often the case, at the level of the kernel. This can take the form of ASLR (address randomization), prevention of stack-smashing attacks, all kinds of memory/process protection, kernel lockdown features, kernel guard, ELF hardening, "secure levels", and much more.

Yet, the surface area is larger than this.
For instance, we could rely on better methods for multi-factor authentication, such as HSM USBs with fingerprint recognition. And while discussing external devices, we should mention policies regarding the insertion of USB modules (USBGuard is an example).
The information, as the most valuable assets on systems, should be kept secure through encryption mechanisms, be it at the hardware-level or software level (LUKS, HSMs, `dm-crypt`, …).
Furthermore, the system should also be secure on its external face, not only through devices, but also on the network. Thus, firewalls should be put in place, secure protocols should be used (`ssh`, `ipsec`, …), (`nftables`, `Iptables`, `PF`, …). The system could possibly be put out of service, breaking the Availability part of CIA, hence it should be reactive with mechanisms such as `fail2ban`, traffic shaping to control bandwidth, and smart traffic inspection.

Finally, if an attacker ever breaks into the last bastion, they should be met with an alarm system, often called intrusion detection and prevention systems (Open Scap, ClamAV, …).

Covering all aspects of security is tough, it requires a lot of time to do threat modeling, studying all part of a system that could be a possible threat. Everyone has their own philosophy and approach, be it attacker-centric, asset-centric, or system-centric (STRIDE, Trike, PASTA). Yet as the "Threat Modeling Manifesto" puts it, it comes down to the following questions:

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good enough job?

> All models are wrong, some models are useful. - George Box

It is more about the realization that there's a need to cover ourselves, than about the different theoretical models in use.

```
Risk = Likelihood * Impact
```

What you need to remember: *Security is much more than access control, if an important part of the system falls, it will lead to all of it falling. The Trusted Computing Base, is the critical core that needs to be protected. Security can be applied to check that the code is trusted, reducing the attack surface, thwarting intrusion at different levels (hardware devices, networks, files, ...). Studying all possible attacks, threat modeling, is mandatory to have a secure system.*

Conclusion

The road was long but we've finally reached the end of this article. We've covered a whole lot regarding access control on at least a few Unix-like OSes, skipping QNX, IBM z/OS, IBM AIX, OpenVMS, Haiku, etc.. Even though these also have interesting access control features such as IBM AIX *RBAC system*, z/OS *System Authorization Facility and Resource Access Control Facility*, and OpenVMS *Authorization DB*.
What we've seen ranged from theoretical models, to how to prove a subject's identity (identification/authentication/authorization), then we divided access control into three parts - system-wide, with usual permissions and MACs, isolation and constraint, with container and sandbox, and action-based - and finally we finished by looking at auditing, logs, and generic security aspects such as the trusted computing based and threat modeling.

The world of security is moving fast and everyone is asking: what should I do to keep my system secure, which access control mechanism should I put in place?
As we've amply discussed, this isn't a straight forward answer. The territory is immense, and the goal of this article was to draw a map to better understand it. Nonetheless, the tech-space is in constant amnesia, with a fear of missing out, reinventing the wheel (NIH syndrome). Consequently, there's a need to stop and stare at what is already present, to look at the past and learn from it, otherwise we're bound to recreate everything, to add more complexity on top of the already huge decaying pile.
As time passes, these access control mechanisms, like many present in this article, will probably be forgotten.

One thing is clear though, for this hectic and connected world, the simpler the approach, the more it'll be used. This explains the rise of isolation and constraint as access control. Yet this doesn't mean it is the appropriate solution for everyone, and no mechanism, policy, or model, has really become dominant over the years either. The diversity of ideas found in this article is proof of that.

We're now left with this wall of knowledge behind us and have to live with the ambiguity that nothing is ever secure nor perfect. This imposing "compendium" is only the tip of a colossal iceberg.

Thank you for reading.
Patrick Louis, aka venam

Bibliography

Generic and Models

Access Control. https://people.cs.rutgers.edu/~pxk/419/notes/access.html. Accessed 27 Feb. 2023.

"Access Control Matrix." Wikipedia, 8 Nov. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Access_control_matrix&oldid=1120722943.

Adam. "What's the Difference between Graham-Denning Model and Harrison, Ruzzo, Ullman (HRU) Model." Information Security Stack Exchange, 11 July 2018, https://security.stackexchange.com/q/189341.

"Bell–LaPadula Model." Wikipedia, 30 Nov. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Bell%E2%80%93LaPadula_model&oldid=1124772158.

"Biba Model." Wikipedia, 22 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Biba_Model&oldid=1140939029.

"Clark–Wilson Model." Wikipedia, 19 Dec. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Clark%E2%80%93Wilson_model&oldid=1061054543.

Common Criteria. https://www.redhat.com/en/blog/common-criteria. Accessed 27 Feb. 2023.

"—." Wikipedia, 5 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Common_Criteria&oldid=1131737682.

"Controlled Access Protection Profile." Wikipedia, 30 May 2018. Wikipedia, https://en.wikipedia.org/w/index.php?title=Controlled_Access_Protection_Profile&oldid=843611818.

"Evaluation Assurance Level." Wikipedia, 1 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Evaluation_Assurance_Level&oldid=1136820643.

Garrison, William C., et al. "On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud." 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 819–38. DOI.org (Crossref), https://doi.org/10.1109/SP.2016.54.

"Graham–Denning Model." Wikipedia, 6 Aug. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Graham%E2%80%93Denning_model&oldid=1037470296.

"HRU (Security)." Wikipedia, 9 Dec. 2019. Wikipedia, https://en.wikipedia.org/w/index.php?title=HRU_(security)&oldid=929976454.

"Introduction To Classic Security Models." GeeksforGeeks, 2 July 2020, https://www.geeksforgeeks.org/introduction-to-classic-security-models/.

"LOMAC." Wikipedia, 8 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=LOMAC&oldid=1138266468.

"Mandatory Integrity Control." Wikipedia, 27 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Mandatory_Integrity_Control&oldid=1112595809.

"Multi Categories Security." Wikipedia, 9 July 2020. Wikipedia, https://en.wikipedia.org/w/index.php?title=Multi_categories_security&oldid=966832847.

"Multilevel Security." Wikipedia, 17 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Multilevel_security&oldid=1110762509.

"NIST RBAC Model." Wikipedia, 2 July 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=NIST_RBAC_model&oldid=1096094233.

Operating Systems: Protection. https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/14_Protection.html. Accessed 27 Feb. 2023.

Red Hat Adds Common Criteria Certification for Red Hat Enterprise Linux 8. 2 Mar. 2021, https://www.redhat.com/en/about/press-releases/red-hat-adds-common-criteria-certification-red-hat-enterprise-linux-8.

Security Models | CISSP Exam Cram: Security Architecture and Models | Pearson IT Certification. https://www.pearsonitcertification.com/articles/article.aspx?p=1998558&seqNum=4. Accessed 27 Feb. 2023.

The Orange Book. http://ftp.ntu.edu.tw/pub/linux/libs/security/Orange-Linux/refs/Orange.html. Accessed 27 Feb. 2023.

"Trusted Computer System Evaluation Criteria." Wikipedia, 22 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Trusted_Computer_System_Evaluation_Criteria&oldid=1135094760.

"Trusted Computing." Wikipedia, 24 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Trusted_Computing&oldid=1141291904.

"Trusted System." Wikipedia, 18 Feb. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Trusted_system&oldid=1072662427.

"Type Enforcement." Wikipedia, 25 Jan. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Type_enforcement&oldid=1067865240.

Proving who we are

Access.Conf(5): Login Access Control Table File - Linux Man Page. https://linux.die.net/man/5/access.conf. Accessed 27 Feb. 2023.

Alston. "Answer to 'In Linux, What Is /Etc/Security?' " Super User, 4 Nov. 2016, https://superuser.com/a/1142562.

"Apache Fortress." Wikipedia, 15 Aug. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Apache_Fortress&oldid=1104536773.

Authenticate(3) - OpenBSD Manual Pages. https://man.openbsd.org/authenticate.3. Accessed 27 Feb. 2023.

"Authentication." Wikipedia, 25 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Authentication&oldid=1141574934.

"Authentication Methods in OpenBSD." Linux.Com, 29 Sept. 2004, https://www.linux.com/news/authentication-methods-openbsd/.

Auth_subr(3) - OpenBSD Manual Pages. https://man.openbsd.org/auth_subr.3. Accessed 27 Feb. 2023.

"AWS Creates New Policy-Based Access Control Language Cedar." InfoQ, https://www.infoq.com/news/2023/02/aws-policy-language-cedar/. Accessed 27 Feb. 2023.

BestGuestNA. "In Linux, What Is /Etc/Security?" Super User, 10 Mar. 2017, https://superuser.com/q/962683.

"BSD Authentication." Wikipedia, 14 Jan. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=BSD_Authentication&oldid=1000192439.

Cap_mkdb(1). https://man.freebsd.org/cgi/man.cgi?query=cap_mkdb&sektion=1&apropos=0&manpath=FreeBSD+13.1-RELEASE+and+Ports. Accessed 27 Feb. 2023.

Catalfamo, Dante. How BSD Authentication Works. 18 Oct. 2021, https://blog.lambda.cx/posts/how-bsd-authentication-works/.

Cedar: A New Policy Language – One Cloud Please. https://onecloudplease.com/blog/cedar-a-new-policy-language. Accessed 27 Feb. 2023.

Cedar Language Playground. https://www.cedarpolicy.com/en. Accessed 27 Feb. 2023.

Chapter 1. Introduction to Red Hat Identity Management Red Hat Enterprise Linux 7 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/linux_dom
Accessed 27 Feb. 2023.

Chapter 18. Pluggable Authentication Modules (PAM). http://www.netbsd.org/docs/guide/en/chap-pam.html. Accessed 27 Feb. 2023.

Cheeseman, David. Intro. 2021. 18 Feb. 2023. GitHub, https://github.com/nuvious/pam-duress.

Doas - ArchWiki. https://wiki.archlinux.org/title/Doas. Accessed 27 Feb. 2023.

—. https://wiki.archlinux.org/title/Doas. Accessed 27 Feb. 2023.

Doas.Conf(5) - OpenBSD Manual Pages. https://man.openbsd.org/OpenBSD-6.4/doas.conf.5. Accessed 27 Feb. 2023.

Enforce an Oso Policy. https://docs.osohq.com/guides/enforcement.html. Accessed 27 Feb. 2023.

FreeIPA. https://www.freeipa.org/page/Main_Page. Accessed 27 Feb. 2023.

Getcap(3) - OpenBSD Manual Pages. https://man.openbsd.org/OpenBSD-5.9/getcap.3. Accessed 27 Feb. 2023.

Group(5): User Group File - Linux Man Page. https://linux.die.net/man/5/group. Accessed 27 Feb. 2023.

IBM Developer. https://developer.ibm.com/tutorials/l-pam/. Accessed 27 Feb. 2023.

"Identity Management." Wikipedia, 21 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Identity_management&oldid=1134966227.

Illumos: Manual Page: Pam.3pam. https://illumos.org/man/3PAM/pam. Accessed 27 Feb. 2023.

"Lightweight Directory Access Protocol." Wikipedia, 14 Oct. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Lightweight_Directory_Access_Protocol&oldid=1115993034.

Linux PAM and OpenPam Diffs. https://pam-list.redhat.narkive.com/L5LkNbU0/linux-pam-and-openpam-diffs. Accessed 27 Feb. 2023.

Login.Conf(5). https://man.freebsd.org/cgi/man.cgi?query=login.conf&sektion=5&format=html. Accessed 27 Feb. 2023.

—. https://man.freebsd.org/cgi/man.cgi?query=login.conf&sektion=5&format=html. Accessed 27 Feb. 2023.

Login.Conf(5) - OpenBSD Manual Pages. https://man.openbsd.org/OpenBSD-5.9/login.conf.5. Accessed 27 Feb. 2023.

Mind Drops :: Assorted Bits of Wit. https://dmitry.khlebnikov.net/2015/07/18/should-we-use-sudo-for-day-to-day-activities/. Accessed 27 Feb. 2023.

nKn. "Answer to 'In Linux, What Is /Etc/Security?' " Super User, 25 Aug. 2015, https://superuser.com/a/962702.

"OAuth." Wikipedia, 1 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=OAuth&oldid=1136876936.

Open Policy Agent. https://openpolicyagent.org/. Accessed 27 Feb. 2023.

Open Source Silicon Root of Trust (RoT) | OpenTitan. https://opentitan.org/. Accessed 27 Feb. 2023.

OpenPAM. "OpenPAM." Gitea, https://git.des.dev/OpenPAM/OpenPAM. Accessed 27 Feb. 2023.

PAM - ArchWiki. https://wiki.archlinux.org/title/PAM. Accessed 27 Feb. 2023.

PAM for Slackware. http://www.slackware.com/~vbatts/pam/. Accessed 27 Feb. 2023.

Pam(8): Pluggable Authentication Modules for - Linux Man Page. https://linux.die.net/man/8/pam. Accessed 27 Feb. 2023.

Pam_access(8) - Linux Man Page. https://linux.die.net/man/8/pam_access. Accessed 27 Feb. 2023.

Pam_limits(8): PAM Module to Limit Resources - Linux Man Page. https://linux.die.net/man/8/pam_limits. Accessed 27 Feb. 2023.

Passwd(5): Password File - Linux Man Page. https://linux.die.net/man/5/passwd. Accessed 27 Feb. 2023.

Problems with PAM (Pluggable Authentication Modules). https://www.dtucker.net/pam/. Accessed 27 Feb. 2023.

Pwd_mkdb(8) - OpenBSD Manual Pages. https://man.openbsd.org/pwd_mkdb.8. Accessed 27 Feb. 2023.

Pwd_mkdb(8) [Freebsd Man Page]. https://www.unix.com/man-page/FreeBSD/8/pwd_mkdb/. Accessed 27 Feb. 2023.

Security - ArchWiki. https://wiki.archlinux.org/title/Security. Accessed 27 Feb. 2023.

SSSD - System Security Services Daemon - Sssd.Io. https://sssd.io/. Accessed 27 Feb. 2023.

"Superuser." Wikipedia, 17 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Superuser&oldid=1139935618.

TrouSerS - The Open-Source TCG Software Stack. https://trousers.sourceforge.net/. Accessed 27 Feb. 2023.

"Understanding PAM." Linux.Com, 11 Feb. 2004, https://www.linux.com/news/understanding-pam/.

Vipw(8) - OpenBSD Manual Pages. https://man.openbsd.org/vipw.8. Accessed 27 Feb. 2023.

What Is Root? – Definition by The Linux Information Project (LINFO). http://www.linfo.org/root.html. Accessed 27 Feb. 2023.

Why I Prefer To Use Doas Over Sudo. https://i-bsd.com/doas/. Accessed 27 Feb. 2023.

"Zanzibar: A Global Authorization System - Presented by Auth0." Zanzibar Academy, https://zanzibar.academy. Accessed 27 Feb. 2023.

http://cr.yp.to/daemontools/setuidgid.html. Accessed 27 Feb. 2023.

System-Wide Access Control

5.11. Multi-Level Security (MLS) Red Hat Enterprise Linux 6 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/mls. Accessed 27 Feb. 2023.

"604.Pdf on Egnyte." Egnyte, https://sansorg.egnyte.com/dl/GBObLgJl6E. Accessed 27 Feb. 2023.

A Decade of OS Access-Control Extensibility - ACM Queue. https://queue.acm.org/detail.cfm?id=2430732. Accessed 27 Feb. 2023.

About RBAC and Using Pfexec. https://docs.tritondatacenter.com/public-cloud/instances/infrastructure/managing/rbac. Accessed 27 Feb. 2023.

Access(2) - OpenBSD Manual Pages. https://man.openbsd.org/access.2. Accessed 27 Feb. 2023.

"Access-Control List." Wikipedia, 2 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Access-control_list&oldid=1130995616.

Adaephon. "Answer to 'Why Are 666 the Default File Creation Permissions?' " Unix & Linux Stack Exchange, 21 Nov. 2013, https://unix.stackexchange.com/a/102085.

"AppArmor." AppArmor, https://apparmor.net/. Accessed 27 Feb. 2023.

AppArmor - Community Help Wiki. https://help.ubuntu.com/community/AppArmor. Accessed 27 Feb. 2023.

arif. "Core Difference between SELinux and Apparmor." Unix & Linux Stack Exchange, 12 Jan. 2018, https://unix.stackexchange.com/q/411853.

"Attribute-Based Access Control." Wikipedia, 3 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Attribute-based_access_control&oldid=1131333280.

Bits and Words. https://venam.nixers.net/blog/unix/2017/06/04/bits-and-words.html. Accessed 27 Feb. 2023.

Borretti, Fernando. "Introducing Austral: A Systems Language with Linear Types and Capabilities." Fernando Borretti, 28 Dec. 2022, https://borretti.me/article/introducing-austral.

Capabilities(7) - Linux Manual Page. https://man7.org/linux/man-pages/man7/capabilities.7.html. Accessed 27 Feb. 2023.

"Capabilities-101.Pdf." Dropbox, https://www.dropbox.com/s/g1gpwz2vbas2zdb/Capabilities-101.pdf?dl=0. Accessed 27 Feb. 2023.

Capability Based Operating Systems. http://www.cap-lore.com/CapTheory/KK/OperatingSystems.html. Accessed 27 Feb. 2023.

Capability Theory by Sound Bytes. http://www.cap-lore.com/CapTheory/index.html. Accessed 27 Feb. 2023.

"Capability-Based Security." Wikipedia, 29 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Capability-based_security&oldid=1113013984.

Capsicum Object-Capabilities on Linux. 2013. Google, 27 Jan. 2023. GitHub, https://github.com/google/capsicum-linux.

"Capsicum (Unix)." Wikipedia, 4 June 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Capsicum_(Unix)&oldid=1026880441.

Capsicum(4). https://man.freebsd.org/cgi/man.cgi?query=capsicum&sektion=4. Accessed 27 Feb. 2023.

[Cap-Talk] "Capability Myths Demolished" Review. https://archive.ph/20130414162939/http://www.eros-os.org/pipermail/cap-talk/2003-March/001133.html. Accessed 27 Feb. 2023.

Changing Whether Root Is a User or a Role - Securing Users and Processes in Oracle® Solaris 11.4. https://docs.oracle.com/cd/E37838_01/html/E61023/rbactask-21.html. Accessed 27 Feb. 2023.

Chapter 3. SELinux Contexts Red Hat Enterprise Linux 6 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/chap-security-enhanced_linux-selinux_contexts. Accessed 27 Feb. 2023.

"Chapter 15. Security." FreeBSD Documentation Portal, https://docs.freebsd.org/en/books/handbook/security/. Accessed 27 Feb. 2023.

"Chapter 17. Mandatory Access Control." FreeBSD Documentation Portal, https://docs.freebsd.org/en/books/handbook/mac/. Accessed 27 Feb. 2023.

Chapter 24. Policy: Defining SELinux User Maps Red Hat Enterprise Linux 6 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/identity_manage mapping. Accessed 27 Feb. 2023.

"Chattr." Wikipedia, 25 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Chattr&oldid=1141530053.

Chattr and Lsattr Usage - Ononoki's Blog. https://blog.ononoki.org/chattr-and-lsattr-usage/. Accessed 27 Feb. 2023.

Chcon(1) - Linux Manual Page. https://www.man7.org/linux/man-pages/man1/chcon.1.html. Accessed 27 Feb. 2023.

Chflags(1) [Freebsd Man Page]. https://www.unix.com/man-page/freebsd/1/chflags/. Accessed 27 Feb. 2023.

Computer Security Division, Information Technology Laboratory. "Role Based Access Control | CSRC | CSRC." CSRC | NIST, 21 Nov. 2016, https://csrc.nist.gov/Projects/Role-Based-Access-Control.

"Confused Deputy Problem." Wikipedia, 5 Aug. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Confused_deputy_problem&oldid=1102474902.

"Context-Based Access Control." Wikipedia, 16 Dec. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Context-based_access_control&oldid=1060665718.

Creating Roles and Users in Trusted Extensions - Trusted Extensions Configuration and Administration. https://docs.oracle.com/cd/E37838_01/html/E61029/txconf-14.html. Accessed 27 Feb. 2023.

CVS Log for Src/Sys/Ufs/Ufs/Attic/Extattr.h. https://cvsweb.openbsd.org/src/sys/ufs/ufs/Attic/extattr.h. Accessed 27 Feb. 2023.

dannSWashko. Episode 028 – Extended Attributes – Lsattr and Chattr | Linux In The Shell. https://www.linuxintheshell.com/2013/04/23/episode-028-extended-attributes-lsattr-and-chattr/. Accessed 27 Feb. 2023.

dbrown. "TrustedBSD - MAC BIBA Policy." NetworkSynapse, 4 Aug. 2007, https://networksynapse.net/freebsd/trustedbsd-mandatory-access-control-part-3-policy-mac-biba/.

—. "TrustedBSD - MAC Processes." NetworkSynapse, 6 July 2007, https://networksynapse.net/freebsd/trustedbsd-mac-processes/.

—. "TrustedBSD - Mandatory Access Control." NetworkSynapse, 5 July 2007, https://networksynapse.net/freebsd/mandatory-access-control-mac-part-1-bsd-extended-tutorial/.

"Discretionary Access Control." Wikipedia, 12 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Discretionary_access_control&oldid=1109849127.

Documentation [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Architecture_implementation:Framework_components [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/ Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models. Accessed 27 Feb. 2023.

—. https://www.rsbac.org/documentation/rsbac_handbook/security_models. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Acl [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/acl. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Auth [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/auth. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Cap [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/cap. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Jail [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/jail. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Mac [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/mac. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:Security_models:Rc [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/security_models/rc. Accessed 27 Feb. 2023.

Documentation:Rsbac_handbook:User_management [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/rsbac_handbook/user_management. Accessed 27 Feb. 2023.

Documentation:Why_rsbac_does_not_use_lsm [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/documentation/why_rsbac_does_not_use_lsm. Accessed 27 Feb. 2023.

“E2fsprogs.” Wikipedia, 7 Apr. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=E2fsprogs&oldid=1081475354.

“Extended File Attributes.” Wikipedia, 7 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Extended_file_attributes&oldid=1137966929.

File Permissions and Attributes - ArchWiki. https://wiki.archlinux.org/title/File_permissions_and_attributes. Accessed 27 Feb. 2023.

“FLASK.” Wikipedia, 30 Sept. 2020. Wikipedia, https://en.wikipedia.org/w/index.php?title=FLASK&oldid=981183806.

Flask: Flux Advanced Security Kernel. https://www-old.cs.utah.edu/flux/fluke/html/flask.html. Accessed 27 Feb. 2023.

Fully Capable. https://sites.google.com/site/fullycapable/Home. Accessed 27 Feb. 2023.

Fully Capable - Pam_cap.So. https://sites.google.com/site/fullycapable/pam_cap-so. Accessed 27 Feb. 2023.

Gite, Vivek. “Linux Kernel Security (SELinux vs AppArmor vs Grsecurity).” NixCraft, 27 May 2009, https://www.cyberciti.biz/tips/selinux-vs-apparmor-vs-grsecurity.html.

Gohman, Dan. “What Is a Capability?” Sunfishcode’s Blog, 21 Nov. 2022, https://blog.sunfishcode.online/what-is-a-capability/.

Grsecurity - Features - RBAC. https://grsecurity.net/featureset/rbac. Accessed 27 Feb. 2023.

Grsecurity/The RBAC System - Wikibooks, Open Books for an Open World. https://en.wikibooks.org/wiki/Grsecurity/The_RBAC_System. Accessed 27 Feb. 2023.

Habitat Chronicles: What Are Capabilities? http://habitatchronicles.com/2017/05/what-are-capabilities/. Accessed 27 Feb. 2023.

“Hardenedbsd/2023-01_decade/Article.Md · Master · Shawn Webb / Articles · GitLab.” GitLab, 2 Jan. 2023, https://git.hardenedbsd.org/shawn.webb/articles/-/blob/master/hardenedbsd/2023-01_decade/article.md.

Hardened/Grsecurity2 Quickstart - Gentoo Wiki. https://wiki.gentoo.org/wiki/Hardened/Grsecurity2_Quickstart. Accessed 27 Feb. 2023.

heath. “Answer to ‘How Do I Use the PAM Capabilities Module to Grant Capabilities to a Particular User and Executable?’ ” Stack Overflow, 26 Apr. 2016, https://stackoverflow.com/a/36873841.

—. “How Do I Use the PAM Capabilities Module to Grant Capabilities to a Particular User and Executable?” Stack Overflow, 3 Jan. 2018, https://stackoverflow.com/q/36755412.

Home | SeL4. https://sel4.systems/. Accessed 27 Feb. 2023.

“Home · Wiki · AppArmor / Apparmor · GitLab.” GitLab, 22 Nov. 2022, https://gitlab.com/apparmor/apparmor/-/wikis/home.

Home [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/. Accessed 27 Feb. 2023.

How to Restrict Root User to Access or Modify a File and Directory in Linux | GoLinuxCloud. 23 Mar. 2019, https://www.golinuxcloud.com/restrict-root-directory-extended-attributes/.

IBM Documentation. 8 Mar. 2021, https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-security-unix.

—. 8 Mar. 2021, https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-security-facilities.

Illumos: Manual Page: Privileges.7. https://www.illumos.org/man/7/privileges. Accessed 27 Feb. 2023.

—. https://illumos.org/man/7/privileges. Accessed 27 Feb. 2023.

Illumos: Manual Page: Roles.1. https://illumos.org/man/1/roles. Accessed 27 Feb. 2023.

k3a. “Linux Capabilities in a Nutshell.” K3A, 24 Apr. 2019, https://k3a.me/linux-capabilities-in-a-nutshell/.

Kennaway, Robert N. M. Watson, Jonathan Anderson, Ben Laurie, Kris. A Taste of Capsicum: Practical Capabilities For Unix. https://cacm.acm.org/magazines/2012/3/146252-a-taste-of-capsicum/abstract. Accessed 27 Feb. 2023.

Linux, Playing With. “Open-Sys-Services: How ACL & MASK Work in Linux?” Open-Sys-Services, 4 Sept. 2011, https://kmaiti.blogspot.com/2011/09/acl-and-mask-in-linux.html.

“Linux Security Module Interface” - MARC. https://marc.info/?l=linux-kernel&m=98695004126478&w=2. Accessed 27 Feb. 2023.

“Linux Security Modules.” Wikipedia, 7 Sept. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Linux_Security_Modules&oldid=1042962670.

“—.” Wikipedia, 7 Sept. 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Linux_Security_Modules&oldid=1042962670.

Linux Security Modules: General Security Hooks for Linux — The Linux Kernel Documentation. https://docs.kernel.org/security/lsm.html. Accessed 27 Feb. 2023.

MAC(4) - Mandatory Access Control. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac. Accessed 27 Feb. 2023.

MAC_BIBA(4) - Biba Data Integrity Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_biba. Accessed 27 Feb. 2023.

MAC_IFOFF(4) - Interface Silencing Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_ifoff. Accessed 27 Feb. 2023.

MAC_LOMAC(4) - Low-Watermark Mandatory Access Control Data Integrity Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_lomac. Accessed 27 Feb. 2023.

MAC_MLS(4) - Multi-Level Security Confidentiality Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_mls. Accessed 27 Feb. 2023.

MAC_NONE(4) - Null MAC Policy Module. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_none. Accessed 27 Feb. 2023.

MAC_PARTITION(4) - Process Partition Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_partition. Accessed 27 Feb. 2023.

MAC_PORTACL(4) - Network Port Access Control Policy. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_portacl. Accessed 27 Feb. 2023.

MAC_SEEOTHERUIDS(4) - Simple Policy Controlling Whether Users See Other Users. https://www.gsp.com/cgi-bin/man.cgi?section=4&topic=mac_seeotheruids. Accessed 27 Feb. 2023.

"Mandatory Access Control." Wikipedia, 21 Oct. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Mandatory_access_control&oldid=1117371527.

minipli. Minipli/Linux-Unofficial_grsec. 2017. 20 Feb. 2023. GitHub, https://github.com/minipli/linux-unofficial_grsec.

Morgan, Andrew G. "Answer to 'How Do I Use the PAM Capabilities Module to Grant Capabilities to a Particular User and Executable?' " Stack Overflow, 2 Oct. 2021, https://stackoverflow.com/a/69413616.

MrVaykadji. "Answer to ' "Dmask" and "Fmask" Mount Options.' " Ask Ubuntu, 5 Mar. 2014, https://askubuntu.com/a/429858.

NOTE: This Project Is Unmaintained. 2016. Nuxi, 26 Feb. 2023. GitHub, https://github.com/NuxiNL/cloudabi.

—. 2016. Nuxi, 26 Feb. 2023. GitHub, https://github.com/NuxiNL/cloudabi.

"Organisation-Based Access Control." Wikipedia, 28 Feb. 2019. Wikipedia, https://en.wikipedia.org/w/index.php?title=Organisation-based_access_control&oldid=885573488.

Packett, Val. The Super Capsicumizer 9000. 2018. 27 Jan. 2023. GitHub, https://github.com/valpackett/capsicumizer.

Paradigm Regained: Abstraction Mechanisms for Access Control. http://www.erights.org/talks/asian03/. Accessed 27 Feb. 2023.

Peter. "Why Are 666 the Default File Creation Permissions?" Unix & Linux Stack Exchange, 21 Nov. 2013, https://unix.stackexchange.com/q/102075.

POSIX Access Control Lists on Linux. https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher_html/main.html. Accessed 27 Feb. 2023.

Privileges Man Page on OpenIndiana. http://www.polarhome.com/service/man/?qf=privileges&af=0&sf=0&of=OpenIndiana&tf=2. Accessed 27 Feb. 2023.

PRIVILEGES(7). https://man.omnios.org/privileges.7. Accessed 27 Feb. 2023.

Progress in Security Module Stacking [LWN.Net]. https://lwn.net/Articles/635771/. Accessed 27 Feb. 2023.

Project:RSBAC - Gentoo Wiki. https://wiki.gentoo.org/wiki/Project:RSBAC. Accessed 27 Feb. 2023.

Project:RSBAC/Introduction - Gentoo Wiki. https://wiki.gentoo.org/wiki/Project:RSBAC/Introduction. Accessed 27 Feb. 2023.

Project:SELinux - Gentoo Wiki. https://wiki.gentoo.org/wiki/Project:SELinux. Accessed 27 Feb. 2023.

R, Joseph. "Answer to 'Why Are 666 the Default File Creation Permissions?' " Unix & Linux Stack Exchange, 21 Nov. 2013, https://unix.stackexchange.com/a/102080.

Rbac(5) [Freebsd Man Page]. https://www.unix.com/man-page/freebsd/5/rbac/. Accessed 27 Feb. 2023.

Reconsidering Unprivileged BPF [LWN.Net]. https://lwn.net/Articles/796328/. Accessed 27 Feb. 2023.

Redirecting to Google Groups. https://groups.google.com/forum/. Accessed 27 Feb. 2023.

"Relationship-Based Access Control." Wikipedia, 6 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Relationship-based_access_control&oldid=1137698111.

"Role-Based Access Control." Wikipedia, 13 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Role-based_access_control&oldid=1139093053.

RSBAC/Overview - Gentoo Wiki. https://wiki.gentoo.org/wiki/RSBAC/Overview. Accessed 27 Feb. 2023.

RSBAC/Quickstart - Gentoo Wiki. https://wiki.gentoo.org/wiki/RSBAC/Quickstart. Accessed 27 Feb. 2023.

sebasth. "Answer to 'Core Difference between SELinux and Apparmor.' " Unix & Linux Stack Exchange, 20 Dec. 2017, https://unix.stackexchange.com/a/411979.

"Security Modes." Wikipedia, 25 Nov. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Security_modes&oldid=1123713280.

"Security-Enhanced Linux." Wikipedia, 23 Nov. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Security-Enhanced_Linux&oldid=1123337877.

SELinux - Gentoo Wiki. https://wiki.gentoo.org/wiki/SELinux. Accessed 27 Feb. 2023.

SELinux Concepts - but for Humans – LearntEmail. 15 Mar. 2017, https://web.archive.org/web/20170315162651/https://learntemail.sam.today/blog/selinux-concepts-but-for-humans/.

SELinux History - SELinux [Book]. https://www.oreilly.com/library/view/selinux/0596007167/ch01s04.html. Accessed 27 Feb. 2023.

SELinux/Commands - Fedora Project Wiki. https://fedoraproject.org/wiki/SELinux/Commands. Accessed 27 Feb. 2023.

SELinux/Constraints - Gentoo Wiki. https://wiki.gentoo.org/wiki/SELinux/Constraints. Accessed 27 Feb. 2023.

SELinux/Quick Introduction - Gentoo Wiki. https://wiki.gentoo.org/wiki/SELinux/Quick_introduction. Accessed 27 Feb. 2023.

Setfacl(1): Set File Access Control Lists - Linux Man Page. https://linux.die.net/man/1/setfacl. Accessed 27 Feb. 2023.

Setgid(2) - Linux Manual Page. https://man7.org/linux/man-pages/man2/setgid.2.html. Accessed 27 Feb. 2023.

“Setuid.” Wikipedia, 13 Dec. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Setuid&oldid=1127199036.

Setuid() Isn’t Good Enough. http://www.cap-lore.com/CapTheory/ConfusedDeputyM.html. Accessed 27 Feb. 2023.

Shamim, Uzair. “The Comprehensive Guide To AppArmor: Part 1.” Information & Technology, 5 Apr. 2019, https://medium.com/information-and-technology/so-what-is-apparmor-64d7ae211ed.

Smack — The Linux Kernel Documentation. https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/Smack.html. Accessed 27 Feb. 2023.

“SmartOS.” Wikipedia, 22 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=SmartOS&oldid=1141020873.

The Confused Deputy. http://www.cap-lore.com/CapTheory/ConfusedDeputy.html. Accessed 27 Feb. 2023.

The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. 21 June 2007, https://web.archive.org/web/20070621155813/http://jya.com/paperF1.htm.

“The Insecurity of OpenBSD.” All That Is Wrong with the World…, 21 Jan. 2010, https://allthatiswrong.wordpress.com/2010/01/20/the-insecurity-of-openbsd/.

The Meaning of Posix.1e. http://wt.tuxomania.net/topics/1999_06_Posix_1e/. Accessed 27 Feb. 2023.

The Role Compatibility Security Model. https://www.rsbac.org/doc/media/rc-nordsec2002/index.html. Accessed 27 Feb. 2023.

—. https://www.rsbac.org/doc/media/rc-nordsec2002/index.html. Accessed 27 Feb. 2023.

The SELinux Notebook. 2020. SELinux Project, 24 Feb. 2023. GitHub, https://github.com/SELinuxProject/selinux-notebook.

TOMOYO Linux Home Page. http://tomoyo.osdn.jp/. Accessed 27 Feb. 2023.

Truck Brakes. http://www.cap-lore.com/CapTheory/Truck.html. Accessed 27 Feb. 2023.

TrustedBSD - SEBSD. http://www.trustedbsd.org/sebsd.html. Accessed 27 Feb. 2023.

TrustedBSD - TrustedBSD POSIX.1e Capabilities. http://www.trustedbsd.org/cap.html. Accessed 27 Feb. 2023.

TrustedBSD - TrustedBSD POSIX.1e Privileges. http://www.trustedbsd.org/privileges.html. Accessed 27 Feb. 2023.

Ubuntu Manpage: Aa-Status - Display Various Information about the Current AppArmor Policy. https://manpages.ubuntu.com/manpages/jammy/en/man8/aa-status.8.html. Accessed 27 Feb. 2023.

Udica - Generate SELinux Policies for Containers! 2018. Containers, 25 Feb. 2023. GitHub, https://github.com/containers/udica.

Understanding Capabilities in Linux - Tinker, Tamper, Alter, Fry. 1 Dec. 2014, https://blog.ploetzli.ch/2014/understanding-linux-capabilities/.

User Rights Management - Securing Users and Processes in Oracle® Solaris 11.4. https://docs.oracle.com/cd/E37838_01/html/E61023/rbac-1.html. Accessed 27 Feb. 2023.

user2650973. “ ‘dmask’ and ‘Fmask’ Mount Options.” Ask Ubuntu, 24 Feb. 2017, https://askubuntu.com/q/429848.

Vermeulen, Sven. “Restricting Even Root Access to a Folder.” Simplicity Is a Form of Art…, 14:09:00+02:00, https://blog.siphos.be/2015/07/restricting-even-root-access-to-a-folder/.

Vfs_acl_xattr. https://www.samba.org/samba/docs/current/man-html/vfs_acl_xattr.8.html. Accessed 27 Feb. 2023.

Vrabec, Lukas. “Technologies for Container Isolation: A Comparison of AppArmor and SELinux.” Enable Sysadmin, Red Hat, Inc., https://www.redhat.com/sysadmin/apparmor-selinux-isolation. Accessed 27 Feb. 2023.

What Are Capabilities? http://www.cap-lore.com/CapTheory/What.html. Accessed 27 Feb. 2023.

What Could Have Been Posix 1003.1e. http://wt.tuxomania.net/topics/1999_06_Posix_1e/download.html. Accessed 27 Feb. 2023.

Why [RSBAC: Extending Linux Security Beyond the Limits]. https://www.rsbac.org/why. Accessed 27 Feb. 2023.

Working with Profile Shells. https://docs.tritondatacenter.com/public-cloud/instances/infrastructure/managing/rbac/working-with-profile-shells. Accessed 27 Feb. 2023.

Your Visual How-to Guide for SELinux Policy Enforcement | Opensource.Com. https://opensource.com/business/13/11/selinux-policy-guide. Accessed 27 Feb. 2023.

https://www.alanhkarp.com/Capabilities-101.html. Accessed 27 Feb. 2023.

https://wiki.c2.com/?PosixCapabilities. Accessed 27 Feb. 2023.

2.10. Generating the /Etc/Cgconfig.Conf File Red Hat Enterprise Linux 6 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-generating_the_etc-cgconfig.conf_file. Accessed 27 Feb. 2023.

19.13. File System Quotas. https://people.freebsd.org/~blackend/doc/el/books/handbook/quotas.html. Accessed 27 Feb. 2023.

A Seccomp Overview [LWN.Net]. https://lwn.net/Articles/656307/. Accessed 27 Feb. 2023.

"About." Minijail, https://google.github.io/minijail/. Accessed 27 Feb. 2023.

Abusing Access to Mount Namespaces through /Proc/Pid/Root. https://labs.withsecure.com/publications/abusing-the-access-to-mount-namespaces-through-procpidroot. Accessed 27 Feb. 2023.

"Add an Extra Layer of Security with Systrace." Linux.Com, 24 Jan. 2006, https://www.linux.com/news/add-extra-layer-security-systrace/.

aditya, boni. "Answer to 'Is There a Sandboxing Program like Sandboxie for Mac?' " Ask Different, 11 Nov. 2018, https://apple.stackexchange.com/a/342336.

Administering Resource Pools (Task Map) - System Administration Guide: Virtualization Using the Solaris Operating System. https://dlc.openindiana.org/docs/20090715/SYSADRM/html/gentextid-6182.html. Accessed 27 Feb. 2023.

Akif, Oion. "Is There a Sandboxing Program like Sandboxie for Mac?" Ask Different, 22 Oct. 2016, https://apple.stackexchange.com/q/258318.

Aleksandersen, Daniel. Systemd Service Sandboxing and Security Hardening 101. 14 Jan. 2020, https://www.ctrl.blog/entry/systemd-service-hardening.html.

"Apple Sandbox Guide v1.0." Reverse Engineering, 14 Sept. 2011, https://reverse.put.as/2011/09/14/apple-sandbox-guide-v1-0/.

"Application Sandbox." Android Open Source Project, https://source.android.com/docs/security/app-sandbox. Accessed 27 Feb. 2023.

AshOS (Any Snapshot Hierarchical OS). 2022. AshOS, 26 Feb. 2023. GitHub, https://github.com/ashos/ashos.

Baker, Pam. "Minijail: Google's Tool To Safely Run Untrusted Programs." Linux.Com, 29 Sept. 2016, https://www.linux.com/news/minijail-googles-tool-safely-run-untrusted-programs/.

"Basics of Seccomp for Docker." Tbhaxor, 15 June 2022, https://tbhaxor.com/basics-of-seccomp-for-dockers/.

Bastille. 2018. BastilleBSD, 26 Feb. 2023. GitHub, https://github.com/BastilleBSD/bastille.

Bastille — Bastille 0.9.20220714-Beta Documentation. https://bastille.readthedocs.io/en/latest/. Accessed 27 Feb. 2023.

Bubblewrap. 2016. Containers, 27 Feb. 2023. GitHub, https://github.com/containers/bubblewrap.

Bubblewrap - ArchWiki. https://wiki.archlinux.org/title/Bubblewrap. Accessed 27 Feb. 2023.

Cano, Florencio. Introduction to Landlock – Infosec Adalid. 19 Aug. 2021, https://infosecadalid.com/2021/08/19/introduction-to-landlock/.

Castro, Jorge O. Awesome Immutable. 2021. 26 Feb. 2023. GitHub, https://github.com/castrojo/awesome-immutable.

CBSD Project. 2011. CBSD Project, 13 Feb. 2023. GitHub, https://github.com/cbsd/cbsd.

Chapter 32. Limiting Storage Space Usage on Ext4 with Quotas Red Hat Enterprise Linux 8 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_file_systems/assembly_limiting-storage-space-usage-on-ext4-with-quotas_managing-file-systems. Accessed 27 Feb. 2023.

Chris's Wiki :: Blog/Unix/ChrootHistory. https://utcc.utoronto.ca/~cks/space/blog/unix/ChrootHistory. Accessed 27 Feb. 2023.

Chromium OS Sandboxing. https://www.chromium.org/chromium-os/developer-guide/chromium-os-sandboxing/. Accessed 27 Feb. 2023.

ChromiumOS Docs - Sandboxing ChromeOS System Services. https://chromium.googlesource.com/chromiumos/docs/+/HEAD/sandboxing.md. Accessed 27 Feb. 2023.

Chroot - ArchWiki. https://wiki.archlinux.org/title/Chroot. Accessed 27 Feb. 2023.

"Controlling Resource Limits with Rctl in FreeBSD." Klara Inc, 16 Mar. 2022, https://klarasystems.com/articles/controlling-resource-limits-with-rctl-in-freebsd/.

"CVS: Cvs.Openbsd.Org: Src" - MARC. https://marc.info/?l=openbsd-cvs&m=146161167911029&w=2. Accessed 27 Feb. 2023.

—. https://marc.info/?l=openbsd-cvs&m=146161509612179&w=2. Accessed 27 Feb. 2023.

D, Tim. "FreeBSD — Managing Jails with Ezjail." Medium, 1 Nov. 2020, https://medium.com/@tdebarbora/freebsd-managing-jails-with-ezjail-b2b1b9e1bd7a.

davmac. "Cgroups v2: Resource Management Done Even Worse the Second Time Around." Software Is Crap, 14 Oct. 2016, https://davmac.wordpress.com/2016/10/14/cgroups-v2-resource-management-done-even-worse-the-second-time-around/.

Dchroot(1): Enter Chroot Environment - Linux Man Page. https://linux.die.net/man/1/dchroot. Accessed 27 Feb. 2023.

Digging into Linux Namespaces - Part 1. https://blog.quarkslab.com/digging-into-linux-namespaces-part-1.html. Accessed 27 Feb. 2023.

Digging into Linux Namespaces - Part 2. https://blog.quarkslab.com/digging-into-linux-namespaces-part-2.html. Accessed 27 Feb. 2023.

"Docker Resource Management in Detail." Tbhaxor, 13 June 2022, https://tbhaxor.com/docker-resource-management-in-detail/.

E.2.7. /Proc/Execdomains Red Hat Enterprise Linux 6 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-execdomains. Accessed 27 Feb. 2023.

EBPF Seccomp() Filters [LWN.Net]. https://lwn.net/Articles/857228/. Accessed 27 Feb. 2023.

Edwards, Christer. “BastilleBSD.” BastilleBSD, https://bastillebsd.org/. Accessed 27 Feb. 2023.

“Endless OS from the Endless OS Foundation.” Endless Computers, https://endlessos.com/home/. Accessed 27 Feb. 2023.

frigo. “Answer to ‘Project Quotas in Ext4.’ ” Stack Overflow, 27 Apr. 2020, https://stackoverflow.com/a/61465057.

Gajjaria, Manish. “Project Quotas in Ext4.” Stack Overflow, 5 Oct. 2018, https://stackoverflow.com/q/52665377.

Generate SECCOMP Profiles for Containers Using Podman and EBPF. https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html. Accessed 27 Feb. 2023.

Giovanni Bechis. Linux Seccomp(2) vs OpenBSD Pledge(2). https://www.slideshare.net/GiovanniBechis/linux-seccomp2-vs-openbsd-pledge1.

Google Code Archive - Long-Term Storage for Google Code Project Hosting. https://code.google.com/archive/p/seccompsandbox/wikis/overview.wiki. Accessed 27 Feb. 2023.

Home - Rlxos Linux. https://rlxos.dev. Accessed 27 Feb. 2023.

How Sandstorm Works: Containerize Data, Not Services. https://sandstorm.io/how-it-works. Accessed 27 Feb. 2023.

Illumos: Manual Page: Newtask.1. https://illumos.org/man/1/newtask. Accessed 27 Feb. 2023.

Illumos: Manual Page: Pooladm.8. https://illumos.org/man/8/pooladm. Accessed 27 Feb. 2023.

Illumos: Manual Page: Poolbind.8. https://illumos.org/man/8/poolbind. Accessed 27 Feb. 2023.

Illumos: Manual Page: Prctl.1. https://illumos.org/man/1/prctl. Accessed 27 Feb. 2023.

—. https://illumos.org/man/1/prctl. Accessed 27 Feb. 2023.

Illumos: Manual Page: Projadd.8. https://illumos.org/man/8/projadd. Accessed 27 Feb. 2023.

Illumos: Manual Page: Project.5. https://illumos.org/man/5/project. Accessed 27 Feb. 2023.

Illumos: Manual Page: Projects.1. https://illumos.org/man/1/projects. Accessed 27 Feb. 2023.

Illumos: Manual Page: Projmod.8. https://illumos.org/man/8/projmod. Accessed 27 Feb. 2023.

Illumos: Manual Page: Rctladm.8. https://illumos.org/man/8/rctladm. Accessed 27 Feb. 2023.

Illumos: Manual Page: Resource_controls.7. https://illumos.org/man/7/resource_controls. Accessed 27 Feb. 2023.

Illumos: Manual Page: Setrctl.2. https://illumos.org/man/2/setrctl. Accessed 27 Feb. 2023.

Illumos: Manual Page: Zlogin.1. https://illumos.org/man/1/zlogin. Accessed 27 Feb. 2023.

Illumos: Manual Page: Zoneadm.8. https://illumos.org/man/8/zoneadm. Accessed 27 Feb. 2023.

Illumos: Manual Page: Zonecfg.8. https://illumos.org/man/8/zonecfg. Accessed 27 Feb. 2023.

Illumos: Manual Page: Zonename.1. https://illumos.org/man/1/zonename. Accessed 27 Feb. 2023.

Illumos: Manual Page: Zones.7. https://illumos.org/man/7/zones. Accessed 27 Feb. 2023.

—. https://illumos.org/man/7/zones. Accessed 27 Feb. 2023.

Introduction to Solaris Zones - System Administration Guide: Virtualization Using the Solaris Operating System. https://dlc.openindiana.org/docs/20090715/SYSADRM/html/zones.intro-1.html. Accessed 27 Feb. 2023.

Iocage. 2017. iocage, 18 Feb. 2023. GitHub, https://github.com/iocage/iocage.

joe425g. “Firejail.” Firejail, 17 July 2022, https://firejail.wordpress.com/.

“Kafel.” Kafel, https://google.github.io/kafel/. Accessed 27 Feb. 2023.

Landlock LSM: Kernel Documentation — The Linux Kernel Documentation. https://docs.kernel.org/security/landlock.html. Accessed 27 Feb. 2023.

Landlock: Unprivileged Access Control — Landlock Documentation. https://landlock.io/. Accessed 27 Feb. 2023.

Landlock: Unprivileged Access Control — The Linux Kernel Documentation. https://docs.kernel.org/userspace-api/landlock.html. Accessed 27 Feb. 2023.

Limits - FreeBSD. https://nixdoc.net/man-pages/FreeBSD/man1/limits.1.html. Accessed 27 Feb. 2023.

MagicPoint Presentation Foils. https://www.openbsd.org/papers/bsdcan2019-unveil/mgp00001.html. Accessed 27 Feb. 2023.

Merino, Julio. “A Quick Glance at MacOS’ Sandbox-Exec.” Julio Merino (Jmmv.Dev), https://jmmv.dev/2019/11/macos-sandbox-exec.html. Accessed 27 Feb. 2023.

Minijail. 2019. Google, 13 Feb. 2023. GitHub, https://github.com/google/minijail.

Mitigations | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/. Accessed 27 Feb. 2023.

Nachum, Yotam. “Linux Namespaces Are a Poor Man’s Plan 9 Namespaces.” Yotam’s Blog, 5 Nov. 2022, https://yotam.net/posts/linux-namespaces-are-a-poor-mans-plan9-namespaces/.

Namespace Page from Section 4 of the Plan 9 Manual. http://man.cat-v.org/plan_9/4/namespace. Accessed 27 Feb. 2023.

Namespaces(7) - Linux Manual Page. https://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed 27 Feb. 2023.

OpenBSD FAQ: Virtualization. https://www.openbsd.org/faq/faq16.html. Accessed 27 Feb. 2023.

Ozymandias42. MacOS Sandbox-Exec Profiles. 2017. 27 Jan. 2023. GitHub, https://github.com/Ozymandias42/macOS-Sandbox-Profiles.

Personality(2): Set Process Execution Domain - Linux Man Page. https://linux.die.net/man/2/personality. Accessed 27 Feb. 2023.

Peter’s Solaris Zone. http://www.petertribble.co.uk/Solaris/pools.html. Accessed 27 Feb. 2023.

—. http://www.petertribble.co.uk/Solaris/minizone.html. Accessed 27 Feb. 2023.

PJT. “Answer to ‘Is There a Sandboxing Program like Sandboxie for Mac?’ ” Ask Different, 12 Sept. 2018, https://apple.stackexchange.com/a/336276.

Pledge | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/pledge/. Accessed 27 Feb. 2023.

Porting OpenBSD Pledge() to Linux. https://justine.lol/pledge/. Accessed 27 Feb. 2023.

Pot. 2017. bsdpot, 26 Feb. 2023. GitHub, https://github.com/bsdpot/pot.

Privilege Drop, Privilege Separation, and Restricted-Service Operating Mode in OpenBSD. https://sha256.net/privsep.html. Accessed 27 Feb. 2023.

Privilege Separated OpenSSH. http://www.citi.umich.edu/u/provos/ssh/privsep.html. Accessed 27 Feb. 2023.

“Privilege Separation.” Wikipedia, 1 July 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Privilege_separation&oldid=1095889062.

Privsep and Privdrop | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/privsec_privdrop/. Accessed 27 Feb. 2023.

Project vs Pool vs Use. https://www.unix.com/solaris/38152-project-vs-pool-vs-use.html. Accessed 27 Feb. 2023.

Rentzsch.Com: Virtualization as an Antivirus. 4 May 2006, https://web.archive.org/web/20060504192215/http://rentzsch.com/notes/virtualizationAsAnAntivirus.

Resource Pools Used in Zones - System Administration Guide: Virtualization Using the Solaris Operating System. https://dlc.openindiana.org/docs/20090715/SYSADRM/html/rmpool-114.html. Accessed 27 Feb. 2023.

S, Jan. “Introduction to Privilege Dropping in C.” Medium, 26 Sept. 2016, https://medium.com/@jan.schreib/introduction-into-privilege-dropping-in-c-b0dca6f47b82.

Sandbox(7) [Osx Man Page]. https://www.unix.com/man-page/osx/7/sandbox/. Accessed 27 Feb. 2023.

Sandbox-Exec(1) [Osx Man Page]. https://www.unix.com/man-page/osx/1/sandbox-exec/. Accessed 27 Feb. 2023.

SBPL - SandBox Policy Language. https://www.romab.com/ironsuite/SBPL.html. Accessed 27 Feb. 2023.

Schroot - Debian Wiki. https://wiki.debian.org/Schroot. Accessed 27 Feb. 2023.

Schroot(1): Securely Enter Chroot Environment - Linux Man Page. https://linux.die.net/man/1/schroot. Accessed 27 Feb. 2023.

Scrivano, Giuseppe. Easyseccomp. 2021. 2 Sept. 2022. GitHub, https://github.com/giuseppe/easyseccomp.

—. “Seccomp Made Easy.” *scratch*, https://www.scrivano.org/posts/2021-01-30-easyseccomp/. Accessed 27 Feb. 2023.

“Search | Packt Subscription.” Packt, https://subscription.packtpub.com/search. Accessed 27 Feb. 2023.

“Seccomp and Seccomp-BPF.” Alex Chapman’s Blog, 31 Aug. 2016, https://ajxchapman.github.io/linux/2016/08/31/seccomp-and-seccomp-bpf.html.

Seccomp BPF (SECure COMPuting with Filters) — The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. Accessed 27 Feb. 2023.

—. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. Accessed 27 Feb. 2023.

“Seccomp Filter in Android O.” Android Developers Blog, https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html. Accessed 27 Feb. 2023.

Seccomp/Libseccomp. 2015. The libseccomp Project, 25 Feb. 2023. GitHub, https://github.com/seccomp/libseccomp.

Secure Containerized Browser | Lobsters. https://lobste.rs/s/ftkfnu/secure_containerized_browser. Accessed 27 Feb. 2023.

Setarch(8) - Linux Man Page. https://linux.die.net/man/8/setarch. Accessed 27 Feb. 2023.

softwareguru. “MacOS: How to Run Your Applications in a Mac OS X Sandbox to Enhance Security.” Paolo Fabio Zaino’s Blog, 3 Aug. 2015, https://paolozaino.wordpress.com/2015/08/04/how-to-run-your-applications-in-a-mac-os-x-sandbox-to-enhance-security/.

Solaris Resource Manager Basics : Understanding Resource Pools – The Geek Diary. https://www.thegeekdiary.com/solaris-resource-manager-basics-understanding-resource-pools/. Accessed 27 Feb. 2023.

Sysctl(2) - OpenBSD Manual Pages. http://man.openbsd.org/sysctl.2. Accessed 27 Feb. 2023.

—. http://man.openbsd.org/sysctl.2. Accessed 27 Feb. 2023.

Sysctl(3) [Freebsd Man Page]. https://www.unix.com/man-page/freebsd/3/sysctl/. Accessed 27 Feb. 2023.

System Administration Guide: Solaris Containers-Resource Management and Solaris Zones. https://docs.huihoo.com/opensolaris/solaris-containers-resource-management-and-solaris-zones/html/p37.html. Accessed 27 Feb. 2023.

Systemd Service Hardening. https://ruderich.org/simon/notes/systemd-service-hardening. Accessed 27 Feb. 2023.

Systemd-Nspawn - ArchWiki. https://wiki.archlinux.org/title/Systemd-nspawn. Accessed 27 Feb. 2023.

Systems Administration - OpenIndiana Docs. http://docs.openindiana.org/handbook/systems-administration/. Accessed 27 Feb. 2023.

“Systrace.” Wikipedia, 27 July 2021. Wikipedia, https://en.wikipedia.org/w/index.php?title=Systrace&oldid=1035742005.

Systrace - Interactive Policy Generation for System Calls. http://www.citi.umich.edu/u/provos/systrace/. Accessed 27 Feb. 2023.

Systrace in OpenBSD | Introduction | InformIT. https://www.informit.com/articles/article.aspx?p=363731. Accessed 27 Feb. 2023.

Systrace(1) - OpenBSD Manual Pages. https://man.openbsd.org/OpenBSD-5.9/systrace. Accessed 27 Feb. 2023.

—. https://man.openbsd.org/OpenBSD-5.9/systrace. Accessed 27 Feb. 2023.

The Flatpak Security Model – Part 1: The Basics – Alexander Larsson. 18 Jan. 2017, https://blogs.gnome.org/alexl/2017/01/18/the-flatpak-security-model-part-1-the-basics/.

Ubuntu Manpage: Chrootuid - Run Command in Restricted Environment. https://manpages.ubuntu.com/manpages/xenial/man1/chrootuid.1.html. Accessed 27 Feb. 2023.

Ubuntu Manpage: Lscgroup - List All Cgroups. https://manpages.ubuntu.com/manpages/bionic/man1/lscgroup.1.html. Accessed 27 Feb. 2023.

Unveil | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/unveil/. Accessed 27 Feb. 2023.

Using Landlock to Sandbox GNU Make. https://justine.lol/make/. Accessed 27 Feb. 2023.

Vanilla OS. https://vanillaos.org/. Accessed 27 Feb. 2023.

vermaden. “Secure Containerized Browser.” vermaden, 14 Dec. 2021, https://vermaden.wordpress.com/2021/12/15/secure-containerized-browser/.

What Chroot() Is Really for [LWN.Net]. https://lwn.net/Articles/252794/. Accessed 27 Feb. 2023.

What Does Security Mean #3: What Does Security Mean for Your Linux Kernel? https://www.linkedin.com/pulse/what-does-security-mean-3-your-linux-kernel-roland-gharfine. Accessed 27 Feb. 2023.

“Why Pivot Root Is Used for Containers.” Tbhaxor, 1 July 2022, https://tbhaxor.com/pivot-root-vs-chroot-for-containers/.

Action-Based Access Control

Louis, Patrick. D-Bus and Polkit, No More Mysticism and Confusion. https://venam.nixers.net/blog/unix/2020/07/06/dbus-polkit.html. Accessed 27 Feb. 2023.

“Intents and Intent Filters.” Android Developers, https://developer.android.com/guide/components/intents-filters. Accessed 27 Feb. 2023.

“Manage Flatpak Permissions Graphically With Flatseal.” It’s FOSS, 23 Nov. 2021, https://itsfoss.com/flatseal/.

“Permissions on Android.” Android Developers, https://developer.android.com/guide/topics/permissions/overview. Accessed 27 Feb. 2023.

Pfedit - Man Pages Section 8: System Administration Commands. https://docs.oracle.com/cd/E88353_01/html/E72487/pfedit-8.html. Accessed 27 Feb. 2023.

Redirect to the New Docs Location. https://flatpak.github.io/xdg-desktop-portal/portal-docs.html. Accessed 27 Feb. 2023.

“Supporting Extensions in IOS, IPadOS, and MacOS.” Apple Support, https://support.apple.com/en-lb/guide/security/secabd3504cd/web. Accessed 27 Feb. 2023.

Vermeulen, Sven. “D-Bus and SELinux.” Simplicity Is a Form of Art…, 20:07:00+02:00, https://blog.siphos.be/2014/06/d-bus-and-selinux/.

—. “D-Bus, Quick Recap.” Simplicity Is a Form of Art…, 19:16:00+02:00, https://blog.siphos.be/2014/06/d-bus-quick-recap/.

"Why Polkit (or, How to Mount a Disk on Modern Linux)." Collabora | Open Source Consulting, https://www.collabora.com/about-us/blog/2015/06/08/why-polkit-(or,-how-to-mount-a-disk-on-modern-linux)/. Accessed 27 Feb. 2023.

"XDG Permissions Stores Should Be Configurable with Snapd." Snapcraft.Io, 2 July 2021, https://forum.snapcraft.io/t/xdg-permissions-stores-should-be-configurable-with-snapd/25048/4.

Yagemann, Carter, and Wenliang Du. "Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook." Computer Security – ESORICS 2016, edited by Ioannis Askoxylakis et al., Springer International Publishing, 2016, pp. 383–400. Springer Link, https://doi.org/10.1007/978-3-319-45744-4_19.

After the Facts: Logging & Auditing

Azad, Usama. Auditd Linux Tutorial. https://linuxhint.com/auditd_linux_tutorial/. Accessed 27 Feb. 2023.

Brown, Paul. "Linux System Monitoring and More with Auditd." Linux.Com, 25 May 2016, https://www.linux.com/topic/desktop/linux-system-monitoring-and-more-auditd/.

"Chapter 18. Security Event Auditing." FreeBSD Documentation Portal, https://docs.freebsd.org/en/books/handbook/audit/. Accessed 27 Feb. 2023.

Illumos: Manual Page: Audit.2. https://www.illumos.org/man/2/audit. Accessed 27 Feb. 2023.

Illumos: Manual Page: Audit.8. https://www.illumos.org/man/8/audit. Accessed 27 Feb. 2023.

Illumos: Manual Page: Audit_binfile.7. https://www.illumos.org/man/7/audit_binfile. Accessed 27 Feb. 2023.

Illumos: Manual Page: Auditconfig.8. https://www.illumos.org/man/8/auditconfig. Accessed 27 Feb. 2023.

Illumos: Manual Page: Auditd.8. https://www.illumos.org/man/8/auditd. Accessed 27 Feb. 2023.

Illumos: Manual Page: Audit.Log.5. https://www.illumos.org/man/5/audit.log. Accessed 27 Feb. 2023.

Illumos: Manual Page: Praudit.8. https://www.illumos.org/man/8/praudit. Accessed 27 Feb. 2023.

OpenBSM. 2014. OpenBSM, 23 Feb. 2023. GitHub, https://github.com/openbsm/openbsm.

—. 2014. OpenBSM, 23 Feb. 2023. GitHub, https://github.com/openbsm/openbsm.

Preface (SunSHIELD Basic Security Module Guide). https://docs.oracle.com/cd/E19455-01/806-1789/6jb25l4a2/index.html. Accessed 27 Feb. 2023.

Secure Levels | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/secure_levels/. Accessed 27 Feb. 2023.

SunSHIELD Basic Security Module Guide. https://docs.oracle.com/cd/E19455-01/806-1789/index.html. Accessed 27 Feb. 2023.

Teleport. What You Need to Know About Linux Audit Framework. https://goteleport.com/blog/linux-audit/. Accessed 27 Feb. 2023.

TrustedBSD - OpenBSM. http://www.trustedbsd.org/openbsm.html. Accessed 27 Feb. 2023.

—. http://www.trustedbsd.org/openbsm.html. Accessed 27 Feb. 2023.

"Utmp." Wikipedia, 11 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Utmp&oldid=1138716570.

General Security & Trusted Computing Base

"Chapter 32. Firewalls." FreeBSD Documentation Portal, https://docs.freebsd.org/en/books/handbook/firewalls/. Accessed 27 Feb. 2023.

Department of Computer Science and Technology: Capability Hardware Enhanced RISC Instructions (CHERI). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/. Accessed 27 Feb. 2023.

Yu, Jason Zhijingcheng, et al. Capstone: A Capability-Based Foundation for Trustless Secure Memory Access (Extended Version). arXiv, 2023, https://doi.org10.48550/ARXIV.2302.13863.

"ELISA - Advancing Open Source Safety-Critical Systems." ELISA, https://elisa.tech/. Accessed 27 Feb. 2023.

Firewalld - ArchWiki. https://wiki.archlinux.org/title/Firewalld. Accessed 27 Feb. 2023.

Hardened/Grsecurity2 Quickstart - Gentoo Wiki. https://wiki.gentoo.org/wiki/Hardened/Grsecurity2_Quickstart. Accessed 27 Feb. 2023.

Home | OpenSCAP Portal. https://www.open-scap.org/. Accessed 27 Feb. 2023.

Iptables - ArchWiki. https://wiki.archlinux.org/title/Iptables. Accessed 27 Feb. 2023.

Kernel_lockdown(7) - Linux Manual Page. https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html. Accessed 27 Feb. 2023.

“Linux Unified Key Setup.” Wikipedia, 14 Dec. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Linux_Unified_Key_Setup&oldid=1127324514.

LKRG - Linux Kernel Runtime Guard. https://lkrg.org/. Accessed 27 Feb. 2023.

Love, Sol. “My First 5 Minutes On A Server; Or, Essential Security for Linux Servers.” Sol Love, https://sollove.com/2013/03/03/my-first-5-minutes-on-a-server-or-essential-security-for-linux-servers/. Accessed 27 Feb. 2023.

Lutefisk. “Answer to ‘What Is the Difference between a HIDS/HIPS and an Anti Virus?’ ” Information Security Stack Exchange, 1 Jan. 2016, https://security.stackexchange.com/a/109445.

Mjg59 | Integrating Linux with Okta Device Trust. https://mjg59.dreamwidth.org/64311.html. Accessed 27 Feb. 2023.

Mjg59 | PKCS#11. Hardware Keystores, and Apple Frustrations. https://mjg59.dreamwidth.org/64968.html. Accessed 27 Feb. 2023.

Mjg59 | Trying to Remove the Need to Trust Cloud Providers. https://mjg59.dreamwidth.org/63261.html. Accessed 27 Feb. 2023.

Nftables - ArchWiki. https://wiki.archlinux.org/title/nftables. Accessed 27 Feb. 2023.

OpenBSD PF: User’s Guide. https://www.openbsd.org/faq/pf/. Accessed 27 Feb. 2023.

PASTA Threat Modeling - Threat-Modeling.Com. 24 July 2022, https://threat-modeling.com/pasta-threat-modeling/.

Popov, Alexander. Linux Kernel Defence Map. 2018. 23 Feb. 2023. GitHub, https://github.com/a13xp0p0v/linux-kernel-defence-map.

“Project Verona.” Wikipedia, 25 Jan. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=Project_Verona&oldid=1135498577.

Secure Boot and Trusted Boot | Is OpenBSD Secure? https://isopenbsdsecu.re/mitigations/secure_boot/. Accessed 27 Feb. 2023.

Security - ArchWiki. https://wiki.archlinux.org/title/Security. Accessed 27 Feb. 2023.

Security Features and Hardening. http://wiki.netbsd.org/security/. Accessed 27 Feb. 2023.

Status. 2019. Microsoft, 26 Feb. 2023. GitHub, https://github.com/microsoft/verona.

“STRIDE (Security).” Wikipedia, 20 Feb. 2023. Wikipedia, https://en.wikipedia.org/w/index.php?title=STRIDE_(security)&oldid=1140600943.

The Trusted Platform Module Key Hierarchy | Posts. https://ericchiang.github.io/post/tpm-keys/. Accessed 27 Feb. 2023.

Threat Modeling | OWASP Foundation. https://owasp.org/www-community/Threat_Modeling. Accessed 27 Feb. 2023.

Threat Modeling Manifesto. https://www.threatmodelingmanifesto.org/. Accessed 27 Feb. 2023.

trikers. “Trike.” Octotrike.Org, http://www.octotrike.org/. Accessed 27 Feb. 2023.

“Trusted Computing Base.” Wikipedia, 16 Oct. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Trusted_computing_base&oldid=1116356882.

“TrustedGRUB.” SourceForge, 5 June 2013, https://sourceforge.net/projects/trustedgrub/.

whatever489. “What Is the Difference between a HIDS/HIPS and an Anti Virus?” Information Security Stack Exchange, 1 Jan. 2016, https://security.stackexchange.com/q/109442.

WhiteWinterWolf. “Answer to ‘What Is the Difference between a HIDS/HIPS and an Anti Virus?’ ” Information Security Stack Exchange, 1 Jan. 2016, https://security.stackexchange.com/a/109450.