LIDS 2.0 Specifications
Version 0.2pre1

Philippe Biondi
philippe.biondi@webmotion.com

November 8, 2000

Contents

[1] Direct Network Access : code used for LIDS to access network directly from the kernel, without any user space code support, to send mails or logs to remote syslog daemons.

Part I

General concepts

Chapter 1

Introduction

The goal of this article is to define what may become LIDS 2.0. LIDS means Linux Intrusion Detection System. “Linux” because it works on Linux[1]. The aim is to prevent intruders from doing anything on a protected machine. Because it has several ways for warning you about unauthorized actions, we can say it is an intrusion detection system.

Ideally, an intrusion detection system must be invulnerable. However, this would result in a box that offers virtually no access to its administrator. The next best thing is to drastically limit vulnerability factors.

As in a scientific experiment, the best way to achieve this is to begin by removing all access privileges to any entity in the box and work one’s way up from there.

The starting point is a box where each process has no rights at all. That means that it can’t happen anything on the box as there isn’t even the right to execute something. But this is pointless, the box has a job to fulfill (web server, firewall, ...). Then we give each entity the rights it needs. The final result is a box where each process running has the exact number of rights it needs to run and no one more.

1.1 Inside the box

As a first approach, I will briefly describe what happens in a computer from the power on to the working state.

First, the POST[2] is issued. Then the machine try to boot on a bootable medium[3]. Then, one or more boot loaders[4] are loaded, the kernel is loaded and booted. The kernel boot initializes the different devices of the computer, creates task 0 and forks and execs `init`.

Until this moment, if you secured the two ore three physical vulnerabilities

[1] and more exactly : it runs *in* Linux. It’s a kernel patch

[2] Power On Self Test. Auto-test of different physical components of the computer : chips, processor(s), memory, etc.

[3] This is an often forgotten vulnerability. You must be sure that the computer can’t boot on another medium (disk, cd-rom) than the one you wanted, for example by using the correct BIOS parameters, or by removing the floppy driver.

[4] Don’t forget to put a password to lilo!

of your box, no interaction can compromise the correct boot[5].

Then the problems begin with foreign interactions[6]. `init` has some network interfaces and routes mounted, some daemons executed, etc. The first question that must be asked are "who `init` is able to execute" [7]. Then comes "which rights the new processes have", "which binaries they can execute", "which operations they can perform", "which files they can read", etc.

1.2 Needs

The classic unix model is insufficient for a fine-tuned access control. New models have appeared such as capabilities and ACL[8]. Other problems are caused by the existence of a superuser, that have all the rights, and whose account is not so hard to compromise.

The ACL approach is to control the access to files with the uid of the user. The difference with the classic unix model is that you can give a list of uids that can access to a precise file, without having to play with unix groups.

We choose to have an ACL-like approach. In addition to control the access to files with uid, we add a control on the program which want to access the file.

To control access to hardware and to some kernel data, we choose a sort of melt with existant capabilities support and the previously introduced ACL-like approach. That is to say we give a capabilities set to a program depending upon the program itself, upon the uid of the user and upon the inheritance of the parent process.

1.3 Comparison with previous LIDS

We used to have the following philosophy : root is not powerful anymore. The new philosophy is : root is not needed anymore. The aim is to shape as accurately as possible the normal users for them to have enough privileges to run for example a daemon, and use the fact you can have multiple users to begin a sort of compartment with users, so that if you break into httpd, you can't access mysqld.

1.4 A new security model

Let F be the set of files on the system. In F, we can find `/sbin/init`, `/etc/modules.conf`, `/usr/bin/`, etc.

Let R be the set of rights that can be given to a program. In R we can find «mount a partition» or «read `/this/file`» or «execute `/that/program` if uid=0». We will see more details on this set later.

[5] We suppose that the boot loader, the kernel and the `init` process are not compromised. If we prove that they can't be compromised while the box is working, we prove by induction that they won't be compromised for every future boot sequence. The initial point where these part are not compromised is the moment after the securisation of the box and before its first working day.

[6] We can't elimate them as the major part makes the whole job of the box : serving web pages, firewalling, ...

[7] And not "who `init` must not execute"

[8] Access Control Lists

Then we can consider two functions σ and μ from F to $\mathcal{P}(R)$ (the set of parts of R), that map a sub-set of R to each file on the system.

The set of rights mapped by σ to a file are the rights that will be given to a process exec'ed with this file. Those mapped with μ are those the process will be able to transmit to a child.

Let X be the property to be executable. For $f \in F$, $X(f)$ will mean that f is executable. X is defined by induction :

- $X($`/sbin/init`$)$
- $X(f) \Leftrightarrow \exists g \in F \mid$ «execute f»$\in \sigma(g)$ and $X(g)$

which mean that `/sbin/init` is executable and that the file f is executable if there exists another file g that is executable and that has the right to execute f.

The files can't use by themselves the rights they are given. They must be executed to give their process their rights. A file that can't be executed and that have some rights won't be able to use them, but who cares ?

When a file is exec'ed, the new process inherits its rights from the rights mapped to the file but also from its parent process rights.

Note that for $f \in F$, there can exist $r \in R$ so that we have $r \notin \sigma(f)$ and $r \in \mu(f)$, i.e. f can have the right to give as inheritage a right it doesn't own. That means that if its process obtain this right by inheritage, it can transmit it.

We can't consider a static set of processes. Let $P(t)$ be the set of running processes at the time t. t is a discrete virtual time variable that is incremented when a process is forked, is exec'ed or dies.

Let τ_t be a function from $P(t)$ to $\mathcal{P}(R)$ that maps a sub-set of R to each running processes, which will be the rights of this process.

Let ν_t be a function from $P(t)$ to $\mathcal{P}(R)$ that maps a sub-set of R to each running processes, which will be the rights the process can give to its lineage by inheritage.

The initial state, $P(0)$, is a singleton and the unique element is the `init` process.

So, if $i \in F$ is `/sbin/init` and $P(0) = \{\iota\}$ then $\phi_0(\iota) = i$. Moreover, we define $\tau_0(\iota) = \sigma(i)$ and $\nu_0(\iota) = \mu(i)$. This is the initial state.

The transition rules are :

A process p forks : a new task p' is created from p.

$$\begin{aligned}
P(t+1) &= P(t) \cup \{p'\} \\
\phi_{t+1}(x) &= \begin{cases} \phi_t(x) & \text{for } x \in P(t) \\ \phi_t(p) & \text{for } x = p' \end{cases} \\
\tau_{t+1}(x) &= \begin{cases} \tau_t(x) & \text{for } x \in P(t) \\ \tau_t(p) & \text{for } x = p' \end{cases} \\
\nu_{t+1}(x) &= \begin{cases} \nu_t(x) & \text{for } x \in P(t) \\ \nu_t(p) & \text{for } x = p' \end{cases}
\end{aligned}$$

A process p execs a file f : if «execute f»$\in \tau_t(p)$ (with possible uid checks) then

$$\begin{aligned}
P(t+1) &= P(t) \\
\phi_{t+1}(x) &= \begin{cases} \phi_t(x) & \text{for } x \in P(t) - \{p\} \\ f & \text{for } x = p \end{cases} \\
\tau_{t+1}(x) &= \begin{cases} \tau_t(x) & \text{for } x \in P(t) - \{p\} \\ (\tau_t(p) \cap \nu_t(p)) \cup \sigma(f) & \text{for } x = p \end{cases} \\
\nu_{t+1}(x) &= \begin{cases} \nu_t(x) & \text{for } x \in P(t) - \{p\} \\ \nu_t(p) \cup \mu(f) & \text{for } x = p \end{cases}
\end{aligned}$$

A process p dies : $P(t+1) = P(t) - \{p\}$ and ϕ_{t+1}, τ_{t+1} and ν_{t+1}, are the restrictions of respectively ϕ_t, τ_t and ν_t over $P(t+1)$.

Let Σ be the set of couple of functions (σ, μ) that enable the computer to fulfill its work. Each element s of Σ is a complete set of access controls that can rule the behaviour of the machine. Let define the partial order $\prec$ over Σ :

$$(\sigma_1, \mu_1) \prec (\sigma_2, \mu_2) \Leftrightarrow \forall f \in F \begin{cases} \sigma_1(f) \subset \sigma_2(f) \\ \mu_1(f) \subset \mu_2(f) \end{cases} \quad \text{and } (\sigma_1, \mu_1) \neq (\sigma_2, \mu_2)$$

If s_1 and s_2 are two elements of Σ and $s_1 \prec s_2$, we can affirm that s_1 is more secure than s_2 because s_2 give at the very least one unnecessary right to a file. As the order is partial[9], there can be more than one minimal element[10].

There is no rule to decide which of two minimal elements is the most secure, but I think there will generally be only one minimal element.

1.5 Adopted approach

The adopted approach is to implement in the kernel everything needed to find, given the program executed under a given uid, a set of capabilities, and, in the case of the access to a specified file, a set of permissions to this file.

Thus, we need an easy and efficient way to join each program with his privileges set. Each privileges set must enclose capabilities sets for each uid and permissions for access on each file, depending on the uid.

The problem now is how to generate and store all the information that represent an element of Σ. A solution is to rely on the underlaying structure of F : the filesystem hierarchy.

[9] $s_1 \neq s_2 \not\Rightarrow (s_1 \prec s_2 \text{ or } s_2 \prec s_1)$
[10] s_m is minimal $\Leftrightarrow \{s \in \Sigma \mid s \prec s_m\} = \varnothing$

Chapter 2

Privileges and permissions

2.1 Filesystem permissions

We used to work with the read/write/access permissions. These are not adapted to what we want. That is why we use new ones. These apply also to directories, which are some kind of files.

Basically, four kinds of accesses can be granted or denied to a file :

- the permission to read data (`R`)
- the permission to append data (`A`)
- the permission to truncate the file (`T`)
- the permission to execute the file (`X`)

The association of the three former can be associated with the more used following permissions :

`---`	No access granted
`R--`	Read-only
`-A-`	Append-only (can't read)
`RA-`	Read and append-only (often called append-only)
`-T`	Truncate only
`R-T`	Read and Truncate only
`-AT`	Write-only
`RAT`	Read/Write

2.2 Privileges

The privileges that can be granted or refused to an executable are a superset of the usual linux capabilities.

Indeed, to the usual linux capabilities[1], we added some LIDS ones, for LIDS special features.

Thus, to the following ones :

[1] Please see `linux/include/linux/capability.h` for a complete and up to date description of these capabilities

CAP_CHOWN chown/chgrp

CAP_DAC_OVERRIDE DAC access

CAP_DAC_READ_SEARCH DAC read

CAP_FOWNER owner ID not equal user ID

CAP_FSETID effective user ID not equal owner ID

CAP_KILL real/effective ID not equal process ID

CAP_SETGID setgid

CAP_SETUID set*uid

CAP_SETPCAP transfer capability

CAP_LINUX_IMMUTABLE immutable and append file attributes

CAP_NET_BIND_SERVICE binding to ports below 1024

CAP_NET_BROADCAST broadcasting/listening to multicast

CAP_NET_ADMIN interface/firewall/routing changes

CAP_NET_RAW raw sockets

CAP_IPC_LOCK locking of shared memory segments

CAP_IPC_OWNER IPC ownership checks

CAP_SYS_MODULE insertion and removal of kernel modules

CAP_SYS_RAWIO ioperm/iopl access

CAP_SYS_CHROOT chroot

CAP_SYS_PTRACE ptrace

CAP_SYS_PACCT configuration of process accounting

CAP_SYS_ADMIN tons of admin stuff

CAP_SYS_BOOT reboot

CAP_SYS_NICE nice

CAP_SYS_RESOURCE setting resource limits

CAP_SYS_TIME setting system time

CAP_SYS_TTY_CONFIG tty configuration

We add these ones :

LIDS_LFS_MASTER can be a LFS[2] master.

[2] LIDS-free session. See chapter 6.2

`LIDS_UMOUNT` to unmount partitions

`LIDS_UNKILLABLE` to be unkillable (god mode)

`LIDS_KILL` to kill unkillable processes (hehe!)

`LIDS_HIDE` to be hidden

`LIDS_OUTLAW` to ignore permissions rules (ie everything that concern fs accesses)

The capabilities set a process is given is the union of the capabilities its parent have and can transmit and the capabilities set it is given in `lids.conf` (which can depend upon uid).

In addition there are some useful sets of capabilities, for a more efficient use :

`FULL_SET` all the capabilities

`EMPTY_SET` no capability at all

`SET_ALL_CAPS` everything that begin with `CAP`

`SET_NET` everything that begin with `CAP_NET`

`SET_SYS` everything that begin with `CAP_SYS`

`SET_IPC` everything that begin with `CAP_IPC`

`SET_DAC` everything that begin with `CAP_DAC`

and other ones whose need will appear with the utilisation of LIDS.

2.3 Insight into the set of rights

The set of rights R can be broken into two subsets : $R = R_C \cup R_F$ where R_C is the set of capabilities and R_F is the set of permissions that can be given to a file, for each possible uid. Let U be the set of available uids plus a special element that will mean that the uid check is not needed. Let C be the set of capabilities seen in section 2.2. Let $A = \{r, a, t, x\}$ be the set of file accesses.

We can write $R_C = U \times C$ and $R_F = U \times A \times F$. Thus an insight of R could be the union of the two following tables where each row is an element of R. Let's have a very reduced disk with three files : $F = \{$`/`,`/bin/`,`/bin/vi`$\}$, and $U = -1, 0, 1000$ (no uid test, id=root and first user).

-1	`CAP_CHOWN`
0	`CAP_CHOWN`
1000	`CAP_CHOWN`
-1	`CAP_DAC_OVERRIDE`
0	`CAP_DAC_OVERRIDE`
1000	`CAP_DAC_OVERRIDE`
...	...
-1	`LIDS_OUTLAW`
0	`LIDS_OUTLAW`
1000	`LIDS_OUTLAW`

-1	`r`	`/`
0	`r`	`/`
1000	`r`	`/`
-1	`a`	`/`
0	`a`	`/`
1000	`a`	`/`
-1	`t`	`/`
0	`t`	`/`
1000	`t`	`/`
-1	`x`	`/`
0	`x`	`/`
1000	`x`	`/`
-1	`r`	`/bin/`
0	`r`	`/bin/`
1000	`r`	`/bin/`
...	...	...
1000	`x`	`/bin/vi`

Chapter 3

Enhancing rights mapping

3.1 Gathering files

The aim of this section is to decrease the size needed to store all the information contained by an element of Σ. We will make groups of files in F, using the structure induced by the filesytem. The fact that files stored in the same directory often share the same properties or same function make it easy to use this structure.

Let's call $\psi : F \to F$ the function that maps to the file f its parent. For example :

$$\begin{aligned} \psi(\texttt{/usr/bin/vi}) &= \psi(\texttt{/usr/bin/}) \\ \psi(\texttt{/bin/}) &= \psi(\texttt{/}) \\ \psi(\texttt{/}) &= \psi(\texttt{/}) \end{aligned}$$

Let G be a subset of F so that "/"$\in G$.

Let's call $\alpha_G(x) = \min\{n \in \mathbb{N} \mid \psi^n(x) \in G\}$. Let's call ψ_G the function that map to a file f the nearest parent in G :

$$\begin{array}{rccl} \psi_G & : F & \to & G \\ & x & \mapsto & \psi^{\alpha_G(x)}(x) \end{array}$$

Let $\mathcal{R}_G$ be the binary relation so that

$$x\mathcal{R}_G y \Leftrightarrow \psi_G(x) = \psi_G(y)$$

We can easily verify that $\mathcal{R}_G$ is an equivalence relation[1].

If $x \in F$, we write $\bar{x} = \{y \in F \mid y\mathcal{R}_G x\} \in \frac{F}{\mathcal{R}_G}$ the equivalence class of x. $\frac{F}{\mathcal{R}_G}$ is the quotient set, i.e. the set of all the equivalence classes.

[1] An equivalence relation must verify

$$\begin{array}{rcl} \forall x & & x\mathcal{R}x \\ x\mathcal{R}y & \Rightarrow & y\mathcal{R}x \\ x\mathcal{R}y \text{ and } y\mathcal{R}z & \Rightarrow & x\mathcal{R}z \end{array}$$

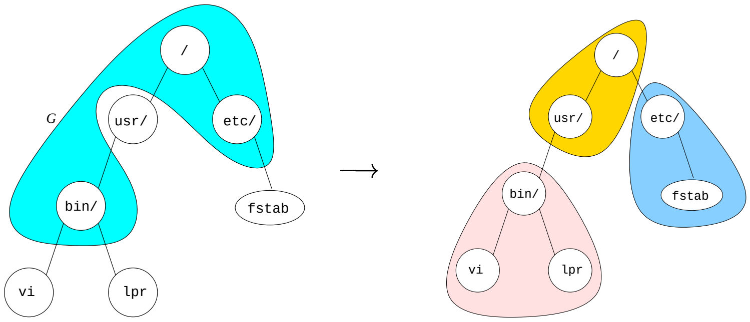

Figure 3.1: From G to the equivalence classes of $\frac{F}{\mathcal{R}_G}$.

We can now reduce the quantity of informations used in σ or μ by using functions whose starting set is G. We have to chose correctly the set G so that we can factorize σ or μ and can write $\sigma = \sigma' \circ \psi_G$ and $\mu = \mu' \circ \psi_G$.

This amounts to the same thing as to group files that are given the same rights and to choose G so that the equivalence classes match the groups.

3.2 Gathering rights

As we saw in section 2.3, we can write $R = R_C \cup R_F$ with $R_C = U \times C$ and $R_F = U \times A \times F$.

We saw in section 3.1 that we can reduce the size of the starting set, and thus the size of the function, using a subset of F and the file system structure. We can do the same here.

For each file $f \in F$, we will adapt the set of rights. let H_f be a subset of F so that "/"$\in H_f$. We have, as defined in section 3.1,

$$\begin{array}{rccl} \psi_{H_f} & : F & \to & H_f \\ & x & \mapsto & \psi^{\alpha_{H_f}(x)}(x) \end{array}$$

The binary relation $\mathcal{R}_{H_f}$ defined by

$$x \mathcal{R}_{H_f} y \Leftrightarrow \psi_{H_f}(x) = \psi_{H_f}(y)$$

is an equivalence relation.

If $x \in F$, we write $\tilde{x} = \{y \in F \mid y \mathcal{R}_{H_f} x\} \in \frac{F}{\mathcal{R}_{H_f}}$ the equivalence class of x. $\frac{F}{\mathcal{R}_{H_f}}$ is the quotient set, i.e. the set of all the equivalence classes.

Let's define the function

$$\begin{array}{rrcl} \Psi_{H_f} & : U \times C \cup U \times A \times H_f & \to & \mathcal{P}(R) \\ & (u, c) \in U \times C & \mapsto & \{(u, c)\} \\ & (u, a, h) \in U \times A \times H_f & \mapsto & \{(u, a)\} \times \tilde{h} \end{array}$$

With a given $f \in F$, if we correctly choose H_f, we can factorize σ' and μ'. Let's call $\beta(f)$ the best set H_f that can be found for f. Let's call $\gamma : f \mapsto \Psi_{\beta(f)}$. And let's call $\delta(f) : G \to \beta(f)$ so that $\forall f \in F \quad \sigma(f) = (\gamma(f) \circ \delta(f) \circ \psi_G)(f)$. Here is our factorization.

Thus we have replaced the data of the graph of $\sigma : F \to R$ by the data of the graphes of $\beta : F \to \mathcal{P}(F)$ and

$$\begin{array}{rcl} \delta : F & \to & \mathfrak{F}\mathrm{onc}(G, \beta(f)) \\ f & \mapsto & \delta(f) : G \to \beta(f) \end{array}$$

which is smaller in practical cases.

Part II

LIDS 2.0 description

Chapter 4

Access lists

4.1 Mapping elements of $\frac{F}{\mathcal{R}_G}$ to those of $\mathcal{P}(R)$

For each program, we must be able to give a permissions set. We gather programs into equivalence classes of $\frac{F}{\mathcal{R}_G}$. Thanks to the canonical bijection between G and $\frac{F}{\mathcal{R}_G}$, which maps $x \in G$ to $\bar{x} \in \frac{E}{\mathcal{R}_G}$, this amount to giving to each element of G a permissions set. This permission set is a subset of R, which can be made by selecting rows in the table in section 2.3.

We saw in section 3.2 that we can also reduce the number of rows of the table using the underlaying structure of the filesystem hierarchy.

We will carry on with the example of the figure 3.1, where $G = \{\texttt{/}, \texttt{/usr/bin/}, \texttt{/etc/}\}$. Let's define β's graph for example as following:

$$\begin{aligned}
\beta(\texttt{/}) = H_{\texttt{/}} &= \{\texttt{/}, \texttt{/usr/}\} \\
\beta(\texttt{/usr/bin/}) = H_{\texttt{/usr/bin/}} &= \{\texttt{/}, \texttt{/etc/}, \texttt{/usr/bin/lpr}\} \\
\beta(\texttt{/etc/}) = H_{\texttt{/etc/}} &= \{\texttt{/}, \texttt{/usr/}, \texttt{/usr/etc/fstab}\}
\end{aligned}$$

This lead to figures 4.1, 4.2 and 4.3.

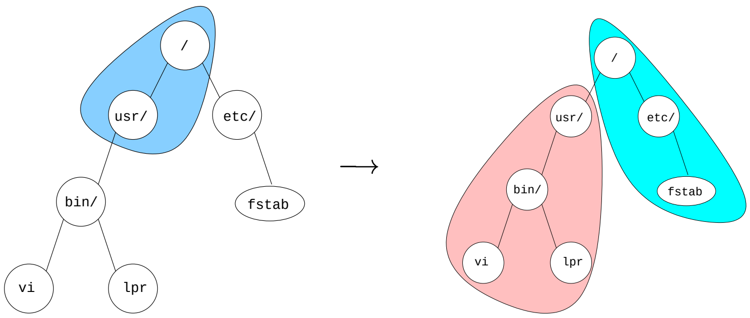

Figure 4.1: From $H_{\texttt{/}}$ to the equivalence classes of $\frac{F}{\mathcal{R}_{H_{\texttt{/}}}}$.

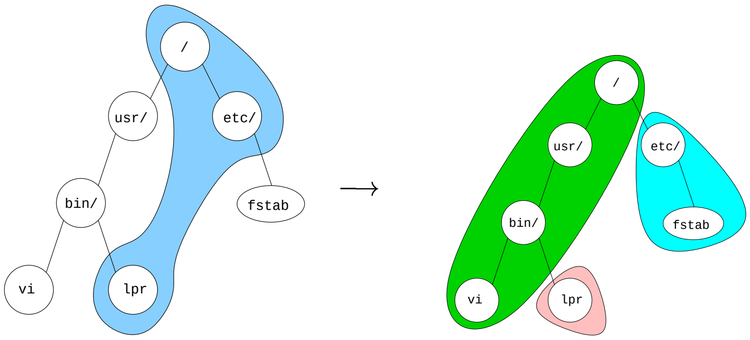

Figure 4.2: From $H_{\texttt{/usr/bin/}}$ to the equivalence classes of $\frac{F}{\mathcal{R}_{H_{\texttt{/usr/bin/}}}}$.

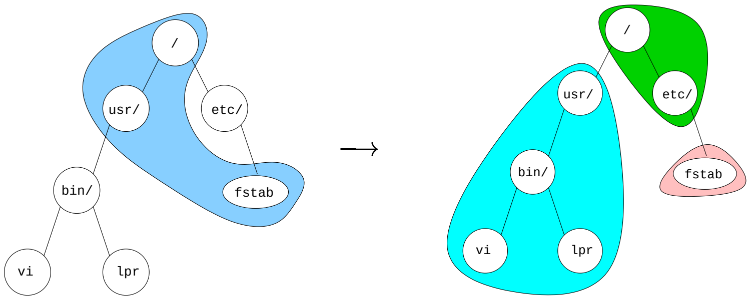

Figure 4.3: From $H_{\texttt{/etc/}}$ to the equivalence classes of $\frac{F}{\mathcal{R}_{H_{\texttt{/etc/}}}}$.

When `/usr/bin/vi` is exec'd, it is given (in addition to its inherited capabilities) the capabilities given by

$$\begin{array}{ccccccc} F & \xrightarrow{\bar{\cdot}} & \frac{F}{\mathcal{R}_G} & \xrightarrow{\text{canonical bij.}} & G & \longrightarrow & \mathcal{P}(U \times C) \\ \texttt{/usr/bin/vi} & \xrightarrow{\bar{\cdot}} & \overline{\texttt{/usr/bin/vi}} & \xrightarrow{\text{canonical bij.}} & \texttt{/usr/bin/} & \longrightarrow & \text{capabilities set} \end{array}$$

If it wants to access, say, `/etc/fstab`, the following operations are done :

$$\begin{array}{ccccccc} F & \xrightarrow{\bar{\cdot}} & \frac{F}{\mathcal{R}_G} & \xrightarrow{\text{canonical bij.}} & G & \longrightarrow & \mathcal{P}(F) \\ \texttt{/usr/bin/vi} & \xrightarrow{\bar{\cdot}} & \overline{\texttt{/usr/bin/vi}} & \xrightarrow{\text{canonical bij.}} & \texttt{/usr/bin/} & \longrightarrow & H_{\texttt{/usr/bin/}} \end{array}$$

Now that we obtained $H_{\texttt{/usr/bin/}}$, we can carry on :

$$\begin{array}{ccccccc} F & \xrightarrow{\tilde{\cdot}} & \frac{F}{\mathcal{R}_{H_{\texttt{/usr/bin/}}}} & \xrightarrow{\text{canonical bij.}} & H_{\texttt{/usr/bin/}} & \longrightarrow & \mathcal{P}(U \times A) \\ \texttt{/etc/fstab} & \xrightarrow{\tilde{\cdot}} & \widetilde{\texttt{/etc/fstab}} & \xrightarrow{\text{canonical bij.}} & \texttt{/etc/} & \longrightarrow & \text{permissions} \end{array}$$

4.2 Inheritance

To avoid repeating all the rights each time a new class of $\frac{F}{\mathcal{R}}$ must be detailed, classes can inherit some rights, eitheir from their parent class (the class $\overline{\psi \circ \psi_G(x)}$) or from a specified class. (See fig. 4.4)

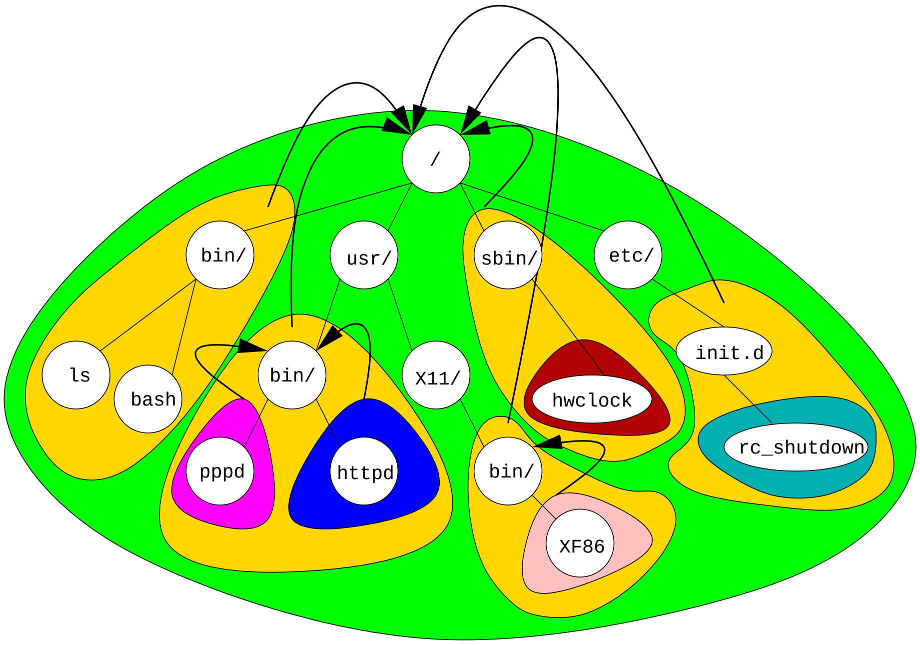

Figure 4.4: Inheritance of the rights

4.3 Effective rights

Let's explain it with an example. We suppose we have `/usr/bin` that have some permissions, and that does not have any inheritance, and `/usr/bin/httpd` that have some other permissions and that inherit the permissions of `/usr/sbin` (see fig. 4.5). The effective permissions for `/usr/bin/httpd` will be a merge of all permissions, as seen in fig. 4.6. We can notice that the permissions on `/home/httpd/index.html` are those inherited from `/usr/bin` and not those from `/usr/bin/httpd` for `/home/httpd`.

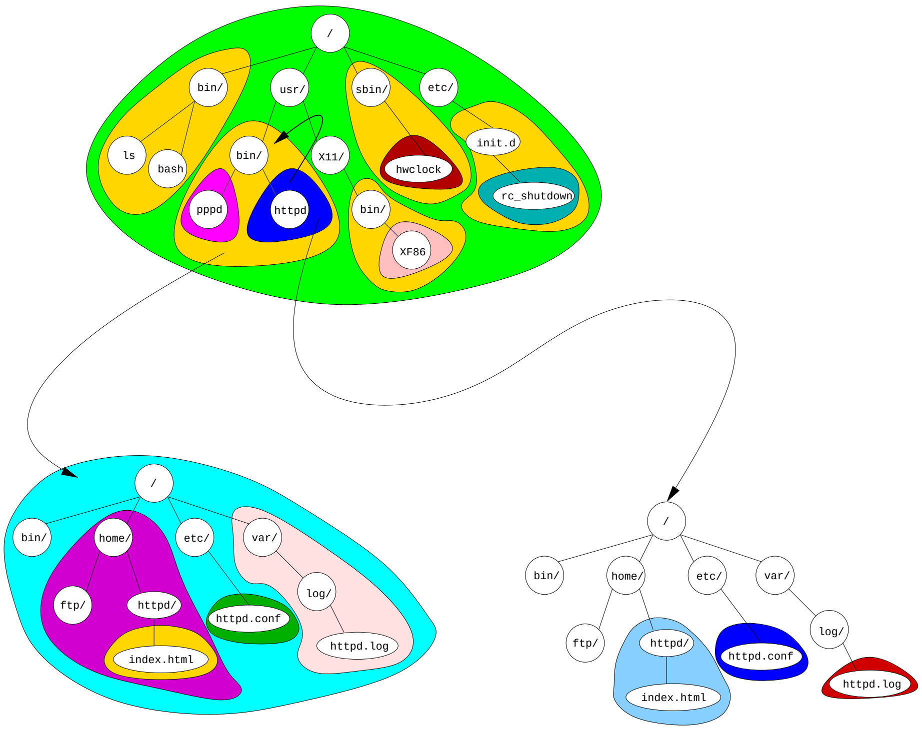

Figure 4.5: Inheritance of permissions

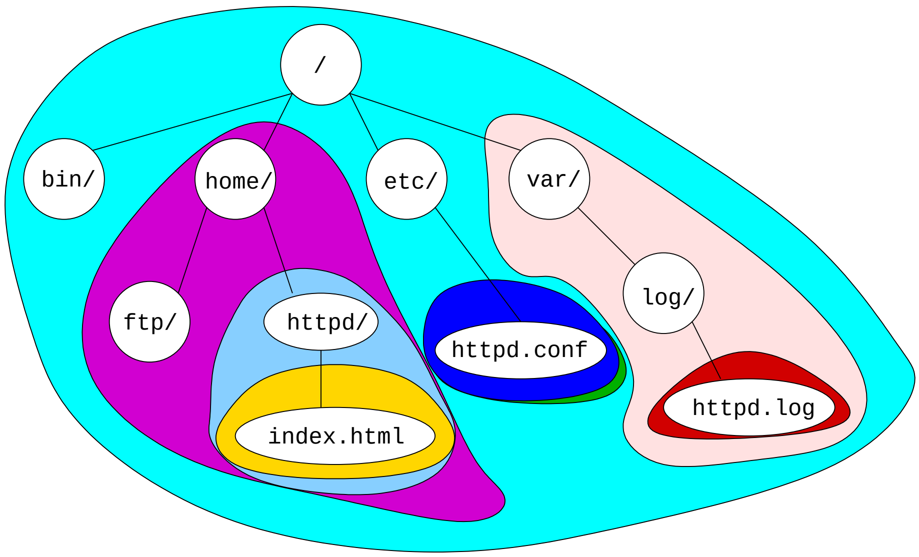

Figure 4.6: Effective permissions

The permissisons also depends upon the real uid of the process. Each file or directory representing an equivalence class is tagged with an uid number or a default flag. When a process wants to access a file, LIDS will check for the presence of this file in his rules (the rules are thoses of the file and those inherited), tagged with the correct uid or the default flag. If both are found, the one tagged with the uid will be used. If none are found, the same check is done for the parent directory of the file, and so on until a rule is found. If no rules are found, the permission is refused.

Chapter 5

Capabilities

5.1 Assigning a capability to a subtree of the executables tree

Each time a program is executed, we must be able to give him a capabilities set. This capabilities set include the LIDS capabilities, and an inheritance mask to know which capabilities must be transmitted to the children of this process. This set will depend upon the program itself, upon the user executing the program and upon the capabilities and mask of its parent.

This amounts to giving each equivalence class executable or directory we choosed in section **??** to represent a partition of the executables set a list of uid and capabilities/mask.

5.2 Inheritage

To avoid repeating all the permissions each time a new subgroup of executables must be detailed, groups can inherit some capabilities/masks, eitheir from their parent subtree or from a specified subtree.

5.3 How this works

When a program is executed, we check if we have some capabilities sets related to it (directly or inherited). If we have, we check for set tagged with the correct uid or the default flag. If we have both, we take the one tagged with the uid. If we have none, the same check is done with the parent directory, and so on until a capabilities set is found. If none are found, the capabilities set will be empty.

Chapter 6

Everyday life administration

6.1 On the fly changes

If you need to install a new program, do a `fsck`, have `httpd` temporary access some files, you could change the capabilities set and permissions of a process modifying `lids.conf`, having LIDS reload it and relaunching the process. This is not really effective.

This is why you can overload the capabilities set or the permission rules of a specified process (by its pid). Each time you overload a capabilities set or you add a permissions rule, you give a tag to the overload. Then you can remove overloads by pid or by tags, or you can remove all the overloads. You can overload as many times as you want an overloaded process.

6.2 LIDS-free sessions

To improve the administration, let's introduce the LIDS-free session (or LFS) concept. When you type `lidsadm -S`, you request the opening of a LFS. If your shell have the capability of beeing a LFS master[1], you will be asked for the LIDS password and, once authenticated, it will be marked as so.

Now, `lidsadm` commands won't ask you for a password anymore when they are launched from this shell or a child of its.

Moreover, each time you request an overload without the `-a` option, the overload will be linked to the LFS used, so that when the shell dies or when the LFS is closed, every overload done in this session and inherited from overloads linked to this session will disappear.

You can make an overload linked to your LFS only if the process also is in your LFS, ie have the same LFS master as your command. There is a command to adopt a process into a LFS, and one do delete his link to the LFS.

There can be as many opened session as you want, which can be closed only by a process of the session, or by a timer created by a process of the session.

A process cannot have more than one LFS master.

[1] See `LIDS_LFS_MASTER` in section 2.2

6.3 Use of these options

Let's see, for example, the emulation of the old `lidsadm -S - -LIDS` and `lidsadm -S - -LIDS_GLOBAL`.

This is the same as `lidsadm -S - -LIDS`

```
lidsadm -S
lidsadm -O -m --caps =FULL_SET -LIDS_HIDE -LIDS\_UNKILLABLE --mask =FULL_SET -LIDS_HIDE
```

This is the same as `lidsadm -S - -LIDS_GLOBAL`

```
lidsadm -O -a --caps =FULL_SET -LIDS_HIDE -LIDS\_UNKILLABLE --mask =FULL_SET
```

But, you guessed it, these are now rough methods !

Chapter 7

Scan port detection

Each time a machine try to open a lot of closed ports in a small amount of time, an alert is raised.

Chapter 8

Logging

8.1 Kinds of logs

There will be different kinds of logs.

- Logs when a LIDS' flag is switched.
- Logs when a program want to do a thing whose needed capability it doesn't have.
- Logs when a program want to access a file in a way that it is not authorized to.
- Logs when a port scan is detected.

Each kind of logs will have the following attributes :

- His own kernel logging priority, from `KERN_EMERG` to `KERN_DEBUG`, or none (`-1`).
- A pointer to a function that will send the alert by mail, or `NULL`
- A pointer to a function that will send the alert in an udp packet (for remote syslogs) or `NULL`
- A pointer to a function that will hang the terminal up, or `NULL`

8.2 DNA[1] and klids

The DNA code is run in a kernel thread[2]. Its role is to serialize the access to the network for the log messages. Each time a log message is generated by LIDS and must be sent through the network, it is placed in a message queue. And klids send one message at time through network, sending for example one UDP packet to a remote host, port 514 for a remote syslog daemon, or sending a whole set of packets, to establish a TCP connection, and to dialog with a mail relay, to finnaly send the message by mail. All these operation are done in kernel space, so that even with an unreliable user space, mail can be send.

[1] Direct Network Access : code used for LIDS to access network directly from the kernel, without any user space code support, to send mails or logs to remote syslog daemons.

[2] a process that is created by the kernel, and that run in kernel space

Part III

Specifications

Chapter 9

The configuration tool

The configuration tool will be `/sbin/lidsadm` The configuration tool will have the following options:

`-A` to add a node in the configuration file.

```
-A node [--caps_inherit (NONE|[PARENT:]/path/to/node)]
        [--caps (uid|-1) ((+|-|=)(FULL_SET|EMPTY_SET|
                                  CAP_SYS...|LIDS_UMONT|..))*
         --mask ((+|-|=)(FULL_SET|EMPTY_SET|
                         CAP_SYS...|LIDS_UMONT|..))* ]*
        [--perm (uid|-1) (RATX|hex) /path/to/object]*
        [--perm_inherit (NONE|[PARENT:]/path/to/node)]
```

The capabilities sets are computed as follow : the initial value is the previous one, or an empty set if there was no previous. Then, each time a keyword is read, his associated value[1] either is added to the set if the keyword is preceded by a `+`, or is removed from the set if the keyword is preceded by a `-` or replace the current value of the set if the keyword is preceded by a `=`.

`-C` To sanity check the configuration file.

```
-C -p|-u|-a
```

The following points are checked :

- The files have correct inode and device numbers
- The files tagged as `PARENT` are the right ones[2]
- The general options are consistents
- There is a mail address if a log through mail is asked for
- The mail address is correct[3]

[1] an empty set for `EMPTY_SET`, a full set for `FULL_SET`, and the corresponding bit to 1 for any other

[2] ie the closest parent which is a `NODE` in `lids.conf`

[3] maybe using `vrfy`

- There is a syslog address if a log to a remote syslog daemon is asked for
- Check wether the remote syslog daemon is listening
- Any other tests

`-p` print updates needed but do nothing

`-u` update all without asking

`-a` ask before updating

`-P` to get a RipeMD-160 encoded password.

```
-P [-f]
```

`-P` ask for a password and display its Ripe-MD encoded form

`-P -f` ask for a password and update the `PASSWORD` field in `lids.conf`

`-S` to open or close a LIDS-free session

```
-S [-c|-t sss|hh:mm[:ss]]
-S (-a|-r) pid [pid...]
```

`-S` to open the session. This has no effect if we are already in a LFS.

`-S -c` to close the session

`-S -t 10` to open the session if it wasn't already opened, and to create a timer to close the session 10 minutes later.

`-S -t 7:30` to open the session if it wasn't already opened, and to create a timer to close the session when it is half past seven.

`-S -a 12 45 648` to add the three given processes to the LFS. This means that their LFS master becomes the current LFS master. If they have already one, an error is issued.

`-S -r 12 648` to remove the two given processes from the LFS. The overloads taht were linked to the LFS are removed.

`-O` to overload capabilities sets.

```
-O [-p pid[ pid[ ...]]] [-m] [-t tag]
   (-r [-a|-A]|[-a]
               [--caps ((+|-|=)(FULL_SET|EMPTY_SET|CAP_SYS...|LIDS_UMONT|..))*]
               [--mask ((+|-|=)(FULL_SET|EMPTY_SET|CAP_SYS...|LIDS_UMONT|..))*]
               [--perm (uid|-1) (RATX|hex) /path/to/object]*)
```

You first give a selector: every processes, or those with specified pids, or those with specified tag or those with specified pids and tag. Then you choose between removing the selected overloads or adding or modifying them.

-m select also the LFS master, if we are in a LFS.

-a when overloading is to overload without making a dependance to the possible current LFS. This change will persist when the possible LFS close or the shell die. If removing tags, only those processes which don't have a LFS master will be processed.

-A only when removing tags, remove those wanted even for processes which have a LFS master.

`-I` to seal the kernel.

```
-I
```

It just signal LIDS that the boot sequence is over, so that it check permissions and capabilities. If you configure LIDS to check them even in the boot sequence[4], you don't need to signal the end of the boot sequence

[4] See the `BOOT_TIME` option in section 10.1.

Chapter 10

The configuration file

The configuration will be saved in the `/etc/lids/lids.conf` file.

It will be made of declarations of the type

```
OPTION1=value
 OPTION2  = value
```

There could have spaces between the words. Any line whose first non-space character is a "`#`" will be ignored. The file is case-sensitive.

The file will begin with general configuration options, which will be followed by the rules about permissions and capabilities.

10.1 General configuration options

`VERBOSE=0-4`: give the verbosity of LIDS (this is not related to the security alerts).

- **`0`:** Absolutely quiet
- **`1`:** LIDS version and warnings only
- **`2`:** Verbose
- **`3`:** Debug
- **`4`:** Heavy debug

`LOG_SWITCH=`*priority*`,`*mail*`,`*syslog*`,`*hang up* : Give the logging methods when a LIDS switch is switched

- ***priority* :** the logging priority from 0 (`KERN_EMERG`) to 7 (`KERN_DEBUG`), or `-1` for no logs through syslog.
- ***mail* :** `0` or `1`, according to wether a mail must be sent by LIDS.
- ***syslog* :** `0` or `1`, according to wether a line must be sent to a remote syslog daemon.
- ***hang up* :** `0` or `1`, according to wether the terminal must be hung up.

There is the same syntax for `LOG_CAP_VIOL`, `LOG_PERM_VIOL`, `LOG_SCAN`

`MAIL_RELAY=`***hex IP:port*** **:** IP[1] of the machine that will be directly connected by LIDS for relaying its mails. Port is usually 25, but who knows. . .

`MAIL_SOURCE=`***source machine*** **:** Name of the source machine, used for the `ehlo` identification. Note that a bad name here could make the mail relay refuse your mails.

`MAIL_FROM=`***sender address*** **:** Sender address, which will also be in the "from" field.

`MAIL_TO=`***recipient address*** **:** Recipient address.

`MAIL_SUBJECT=`***subject*** **:** Subject of the mail.

`REMOTE_SYSLOG=`***hex IP:port*** **:** IP[1] of the machine that will receive the UDP syslog packets. Port is usually 514, but who knows. . .

`BOOT_TIME=`***perm, caps*** **:** give the way LIDS will behave before sealing

- ***perm*** **:** `0` or `1`, according to wether permission checks will be performed only after sealing or also during the boot sequence.
- ***caps*** **:** `0` or `1`, according to wether capabilities checks will be performed only after sealing or also during the boot sequence.

`FLOOD_PROTECTION=0|1` **:** decide wether the same security alert can be raised a lot of times in a short amount of time.

`ALLOW_SWITCH=0|1` **:** decide wether LIDS stuffs can be switched.

`REMOTE_SWITCH=0|1` **:** if LIDS stuffs can be switched, decide wether remote users can do it.

`RELOAD_CONF=0|1` **:** decide wether the configuration file can be reloaded.

`SCAN_DETECTOR=0|1` **:** decide wether to activate the ports scan detector. This option must also be activated in the kernel compilation time options.

`HIDE_KLIDS=0|1` **:** In conjuntion with `VERBOSE=0`, can make LIDS be invisible.

`NET_RETRIES=`***number*** **:** number of retries after a failed connection.

`NET_SLEEP=`***time*** **:** number of seconds between two failed connections.

`PASSWORD=`***password*** **:** RipeMD-160 twice[2] encoded password.

`LOGIN_RETRIES=`***number*** **:** number of times a password can be submitted before disabling the switch for the `FAIL_DELAY` time.

`FAIL_DELAY=`***time*** **:** number of seconds to wait after too many fails to be able to switch again.

Some options need to have the necessary code compiled into the kernel. For example, you need to select the scan port detector option in the compilation options of the kernel before `SCAN_DETECTOR=1` can work.

[1] in hexadecimal format. For example, 127.0.0.1 will be `0100007f`

[2] This permit `lidsadm` to encode the password once internally before sending it to the kernel, so that an interception out of lidsadm does not reveal it.

10.2 Permissions and capabilities

Then will follow the nodes description. A node is an executable or a directory of the filesystem which is linked to capabilities and/or permissions.

A node description begin with a `NODE=` declaration :

```
NODE=dev:inode:/path/to/node
CAPS_INHERIT=NONE|dev:inode:[PARENT:]/path/to/parent
CAPS=(uid|-1),caps,lids_caps,mask,lids_mask
CAPS=(uid|-1),caps,lids_caps,mask,lids_mask
...
PERM_INHERIT=NONE|dev:inode:[PARENT:]/path/to/parent
PERM=perm:dev:inode:/path/to/object
PERM=perm:dev:inode:/path/to/object
...
```

There must not be any spaces arround the colons ("`:`"). -1 stands for default capability.

When the keyword "`PARENT`" is used, this mean that the indicated `dev:inode` and path may change, so that you need to correct this with the configuration tool.

The `caps`, `lids_caps`, `mask`, `lids_mask` are 32 bits integers written in hexadecimal, without the leading `0x`, with lower or upper case.

Chapter 11

Logging specifications

Each call to the logging function will provide the kind of log :

- LIDS_LOG_SWITCH
- LIDS_LOG_CAP_VIOL
- LIDS_LOG_PERM_VIOL
- LIDS_LOG_SCAN

Each format string will be written with correct bounds so that it can't exceed 2047 bytes long when been substitued (never use "%s" but "%.1024s")

Chapter 12

Compilation time options

Some options can/must be chosen at compilation time :

- wether to compile the port scans detection code
- wether to compile the DNA[1] code[2]. It is used for the mail and syslog options (see section 10.1).
- wether to compile the switching code
- wether to compile the code to reload configuration file
- wether to trust[3] the `lids.conf` password.

[1] Direct Network Access : code used for LIDS to access network directly from the kernel, without any user space code support, to send mails or logs to remote syslog daemons.

[2] ask also for its message queue size

[3] if not, the password field must be given as a compilation time option, as it used to be before

Chapter 13

Most important data structures

13.1 New types

```
typedef kernel_cap_t lids_cap_t;

typedef int lids_perm_t;
```

13.2 task_struct

```
struct task_struct {
...
        lids_perm_s *perm;
        lids_caps_s *caps;
lids_lfs_s *lfs;
struct task_struct *lfs_master;
};
```

13.3 lids_prog_s

```
typedef struct lids_prog_struct {
        lids_prog_s *next;
        ino_t ino;
        dev_t dev;
        lids_perm_s *perm;
        lids_caps_s *caps;
} lids_prog_s;
```

13.4 lids_perm_s

```
typedef struct lids_perm_struct {
        lids_perm_s *next;
```

```
        ino_t ino;
        dev_t dev;
        uid_t uid;
        lids_perm_t perm;
} lids_perm_s;
```

13.5 lids_caps_s

```
typedef struct lids_caps_struct {
        lids_caps_s *next;
        union {
                uid_t uid;
                int tag;
        } id;
        kernel_cap_t caps,mask
        lids_cap_t lids_caps,lids_mask;
} lids_caps_s;
```

13.6 lids_lfs_s

```
typedef struct lids_prog_struct {
        lids_lds_s *next;
task_struct *task;
int tag;
} lids_prog_s;
```

We can see how these structures are linked in memory on figure 13.1.

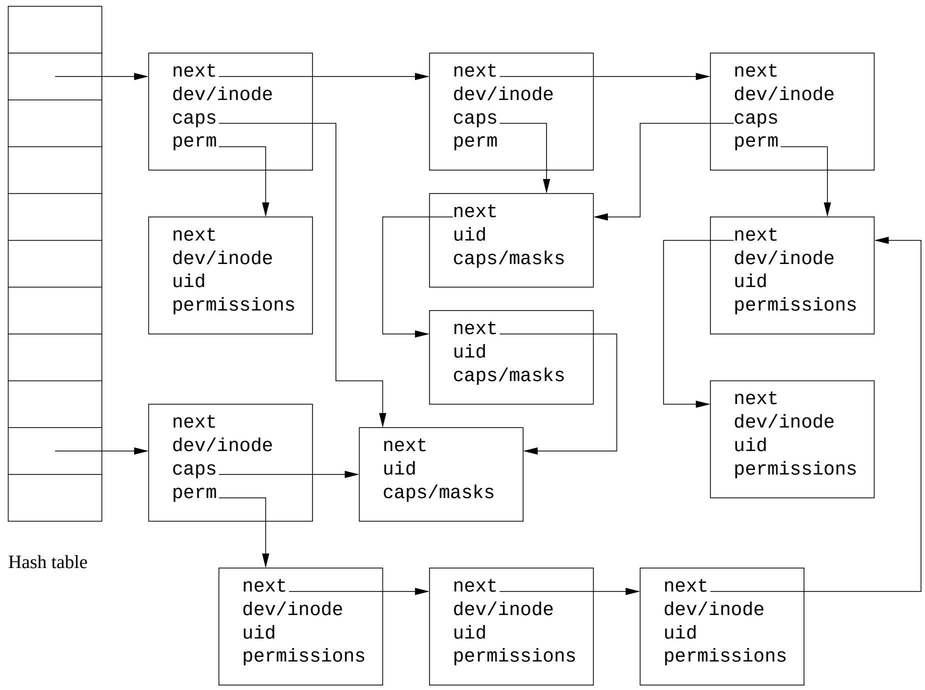

Figure 13.1: Link between data structures

The figure 13.2 shows only the `lids_prog_s` structure links, for the example of the figure ??. We imagine that `/usr/bin`, `/etc/init.d` and `/sbin` have the same image through the hash function. Idem for `/usr/X11/bin/XF86` and `/bin`.

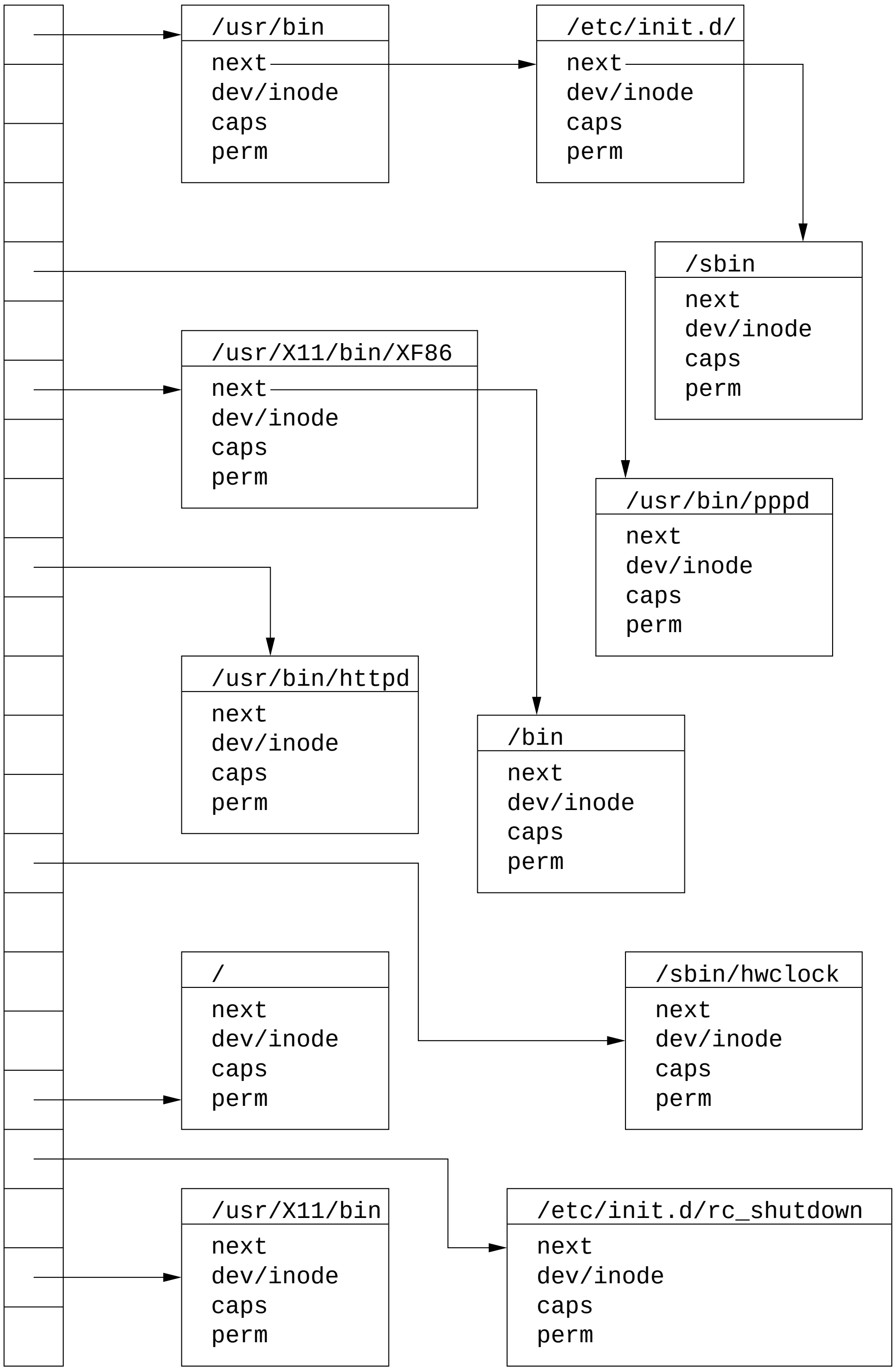

Figure 13.2: Program structures in memory for fig. ??

If we focus on the two permissions sets of the figure 4.5, and take in account the inheritance of `/usr/bin` permissions by `/usr/bin/httpd`, we can have the structures arrangement in memory as shown in figure 13.3. The inheritance is effective thanks to the bold arrow.

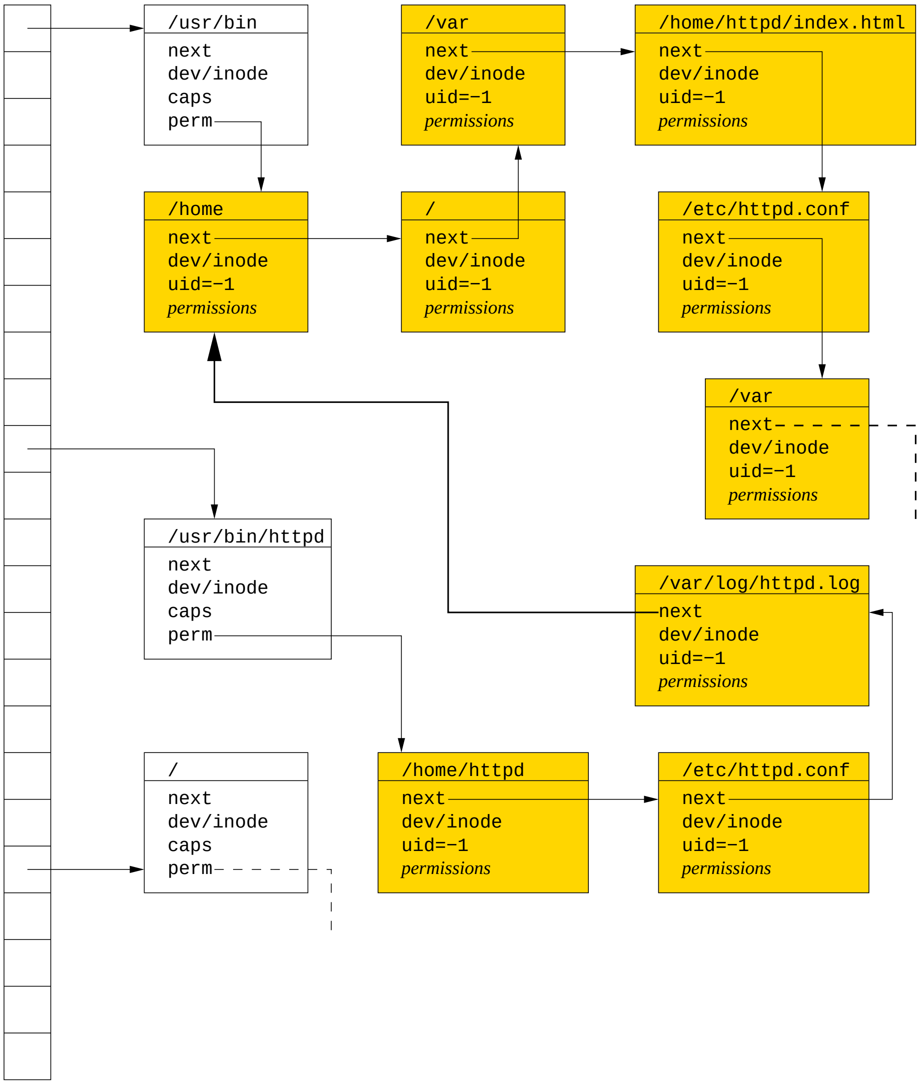

Figure 13.3: Program structures in memory

Chapter 14

LIDS-free sessions implementation

The structures under the dashed horizontal line are those which existed before any LFS. The permission arrows pointing to nothing point in fact to the chained lists allocated when reading the configuration file.

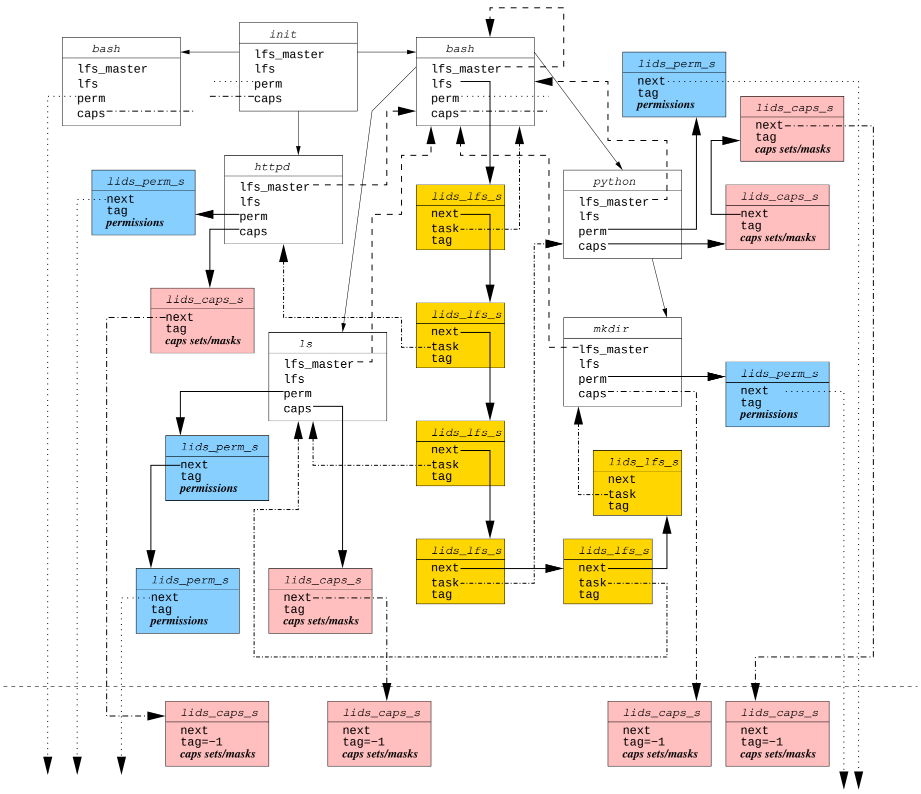

Figure 14.1: LFS strutures in memory

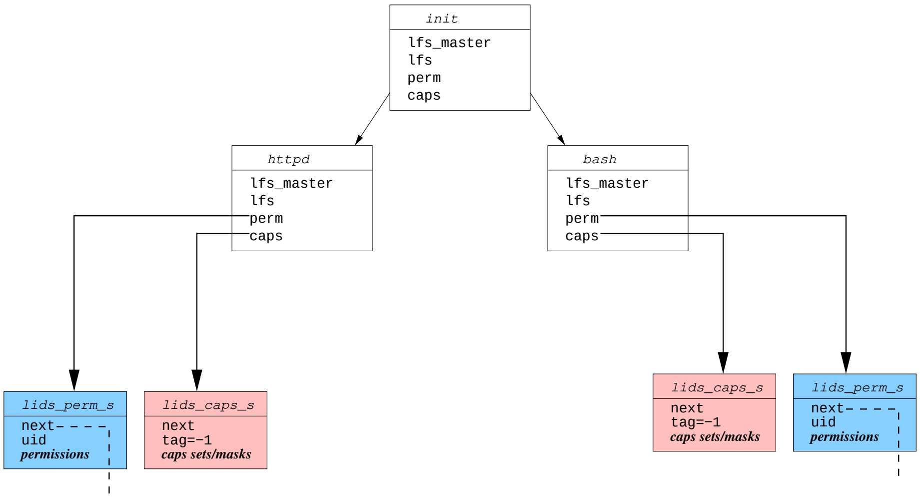

Figure 14.2: LFS step 1 : Initial state

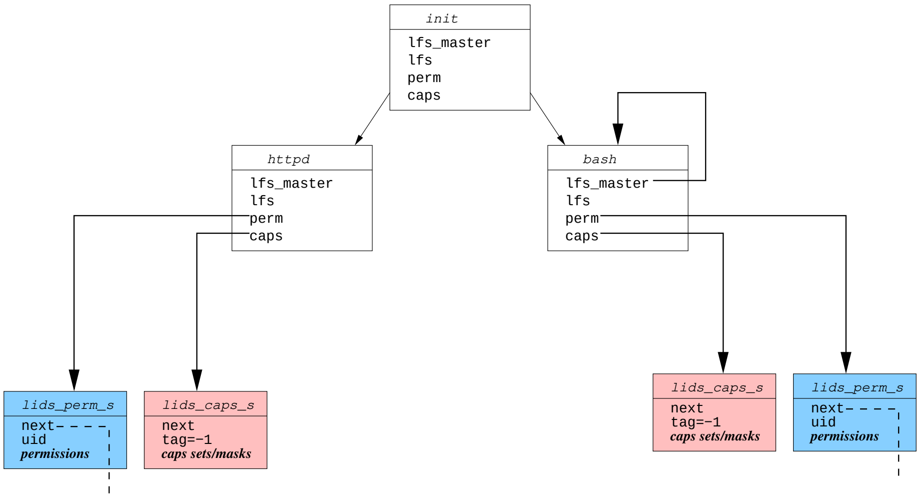

Figure 14.3: LFS step 2 : LFS opened

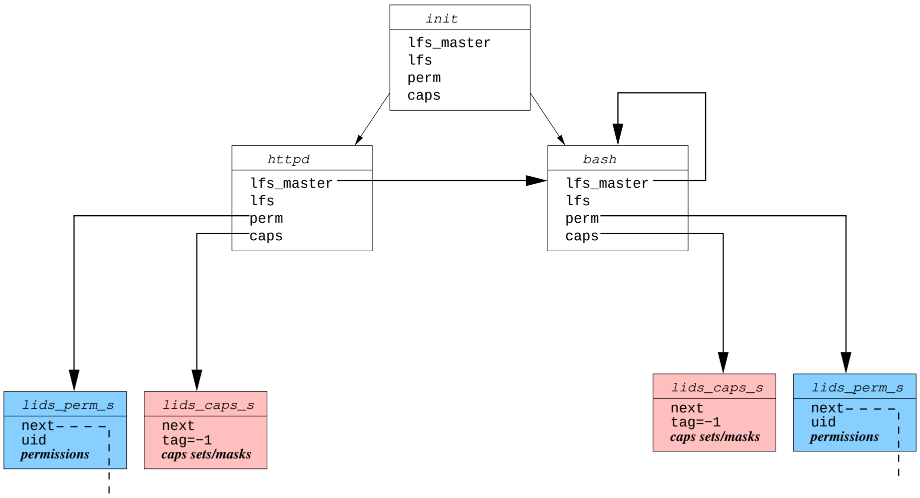

Figure 14.4: LFS step 3 : `httpd` adopted into the LFS

The dotted line between the two tags is to say that they have the same value.

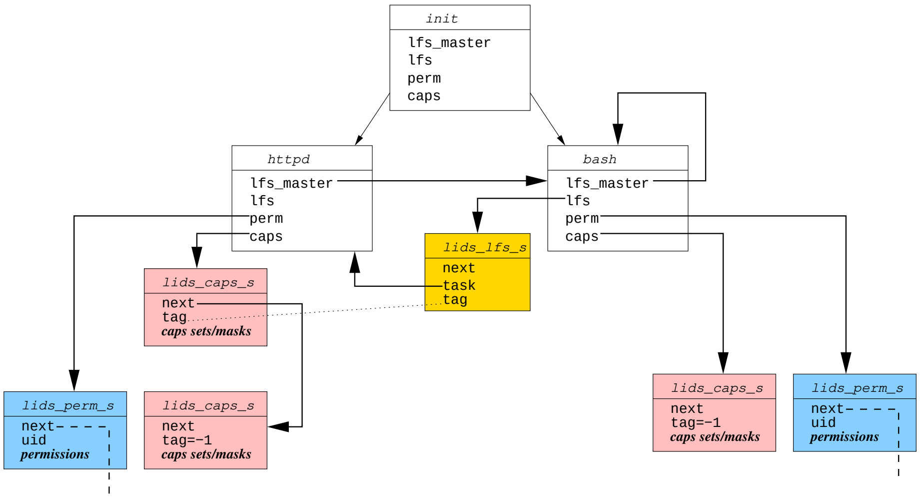

Figure 14.5: LFS step 4 : overload of the capabilities

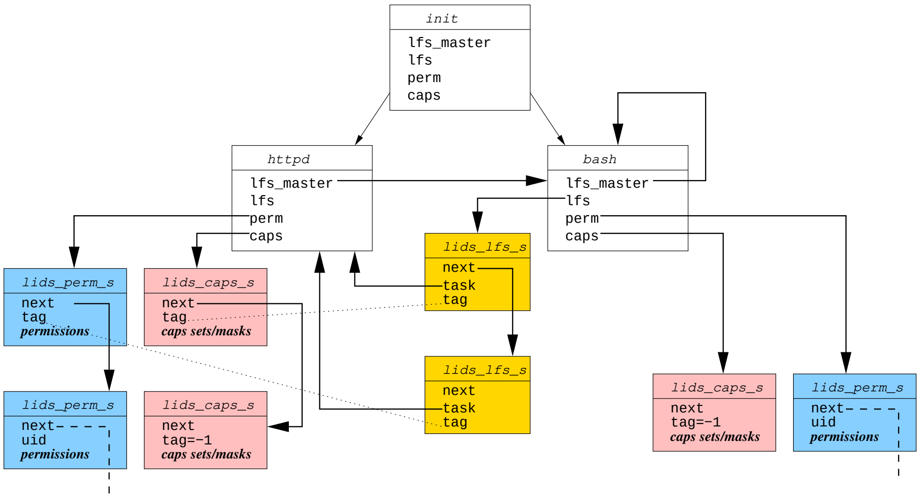

Figure 14.6: LFS step 5 : overload of the permissions

Chapter 15

Communication between `lidsadm` and LIDS

The communication will be done with a sysctl file in `/proc/sys/dev/lids`.

The protocol is not defined yet.

Chapter 16

Algorithms

16.1 Config file reading algorithm

The file must be run through twice.

1. Allocate memory for the hash table. The hash function is the lower part of the inode number.
2. First reading : each time a `NODE=...` is read :
   (a) Allocate a `lids_prog_s`
   (b) Fill fields `next`, `perm` and `caps` with `NULL`
   (c) Fill fields `dev` and `inode` with the read values.
   (d) Place the structure in the hash tree.
   (e) Each time a `CAPS=...` is read :
      i. Allocate a `lids_caps_s`
      ii. Fill field `next` with `NULL`
      iii. Fill fields `uid`, `caps`, `mask`, `lids`, `lids_mask` with the read values
      iv. Make the structure be pointed at by the end of the current caps list.
   (f) Each time a `PERM=...` is read :
      i. Allocate a `lids_perm_s`
      ii. Fill field `next` with `NULL`
      iii. Fill fields `dev`, `ino`, `perm` with the read values.
      iv. Make the structure be pointed at by the end of the current perm list.
3. Second reading : each time a `NODE=...` is read :
   (a) remind the device and inode numbers
   (b) at the first `CAPS_INHERIT=...`, if it is not `NONE`, make the last `next` of the reminded program's `caps`'s list point at the begining of the `caps` list of the program which is indicated in the `CAPS_INHERIT` field.
   (c) at the first `PERM_INHERIT=...`, if it is not `NONE`, make the last `next` of the reminded program's `perm`'s list point at the begining of the `perm` list of the program which is indicated in the `PERM_INHERIT` field.

16.2 `fork` system call

When the process has forked, we must duplicate the whole capabilities chained list, and the overloads of the permissions chained list. We must also ensure that the new process inherit the LFS master of its parent, that the tags inherited from tags referenced by the LFS master are added to the references (`lfs` chained list) of the LFS master. If the LFS master forks, the child doesn't inherit the `lfs` chained list.

16.3 `exec` system call

There will be two new fields in the `task_struct` : `lids_perm_s *perm` and `lids_caps_s *caps`.

Each time the `execve` system call is invoked,

1. we check that `parent`[1] have the permission to read and execute the program.
2. `child` inherit of `parent->caps` chained list. (Duplicated if `fork`ed)
3. `child` inherit of `parent->lfs_master`
4. `parent->lfs_master->lfs` is updated with the inherited overloaded capabilites of `child`
5. we search for rules in the configuration about this program, or a parent directory
   (a) we search for the `lids_prog_s` structure of the program
   (b) if none are found, we search the one of its parent directory, and so on until one is found.

   If no `lids_prog_s` is found, `task_struct.perm` will be `NULL` and the caps/masks will be taken empty. Else, `task_struct.perm` will point on `lids_prog_s.perm` and caps/masks will be taken in the `lids_caps_s` whose `uid` field is `-1` or equal to `parent->uid`. If both are found, the second will be chosen. If none are found, we'll take empty sets.
6. Now for each element of the `child->caps` chained list, we do

   ```
   elt->caps = found_caps | (elt->caps & elt->mask)
   elt->mask |= found_mask
   ```

   This is the inheritance rule for capabilities.

This suppose that the very first process is born with its `caps` field pointing to a `lids_caps_s` whose caps/mask are empty and whose `tag` field is `-1`. Its `perm` field should point to a rule that allow it to read and execute what will be the `init` process. So that it will allow it to read and execute everything.

[1] `parent` is `current`

16.4 Overload

An overload request is the execution of `lidsadm` with the `-O` parameter. For example :

```
lidsadm -O -t 1 --caps ULL_SET -CAP_SYS_ADMIN -LIDS_UMOUNT --mask =EMPTY_SET
lidsadm -O -p 123 45 385 -t 2 --caps =SET_ALL_CAPS -CAP_SYS_ADMIN +LIDS_UMOUNT
        --mask =EMPTY_SET
lidsadm -O -t 2 --caps +LIDS_UMOUNT
lidsadm -O -p 45 -t 2 -r
lidsadm -O -t 2 -r
lidsadm -O -r
```

When the admin is requesting an overload, for each process concerned with this request :

- If the process does not have any overload tags[2] or have some but none with the specified tag :
  1. a `lids_caps_s` is allocated
  2. the `tag` field is set to the given value, or the smallest that is greater than the existant ones if none is provided
  3. the caps/masks sets are filled with the provided values
  4. the structure is inserted in the chained list, the `tag` field value decreasing.
- If a tag was provided and the process has a corresponding overload tag, the following operations are done on this overload tag. If no tags were provided, the following operations are done on each overload.
  1. applying the provided update mask (`caps &= update_mask`) to the caps/masks
  2. applying the provided updated caps (`caps |= updated_caps`) to the caps/masks

16.5 Hash table

The used hash function will be the lower byte of the inode number of the executable.

16.6 Permission check

When a process want to access a file and doesn't have the `LIDS_OUTLAW`[3] we must follow the permissions chained list whose head is given by `current->perm`.

One by one, the rules are tested, in the order they are linked. If the `uid` field is negative, it's either the tag of an overload rule (`uid < -1`) or the default uid (`-1`), so that the rule is effective. If the `uid` field is positive or nul, it's an uid and must match `current->uid` to be effective.

[2] i.e. `lids_caps_s` with a positive `tag` field

[3] the access is granted regardless of the permissions. See section 2.2

If the rule is effective, it must match the device and inode numbers of the accessed file to be used. If it is an overload, or an uid match, the rule is used. If it is the default tag, we must search for an uid match, and we will use the default one only if no uid matches are found.

If used, it gives the authorized rights of the process. Access is granted only if it has sufficient rights.

If no rules are found for the given inode and device, the same test is done for the parent directory, until a rule is found. If none are found, the access is denied.

16.7 Capabilities check

When a process want to do something that require one or more specific capabilities, we take the head of the chained list, ie the structure pointed by `current->caps` and check if the needed capabilities are present.