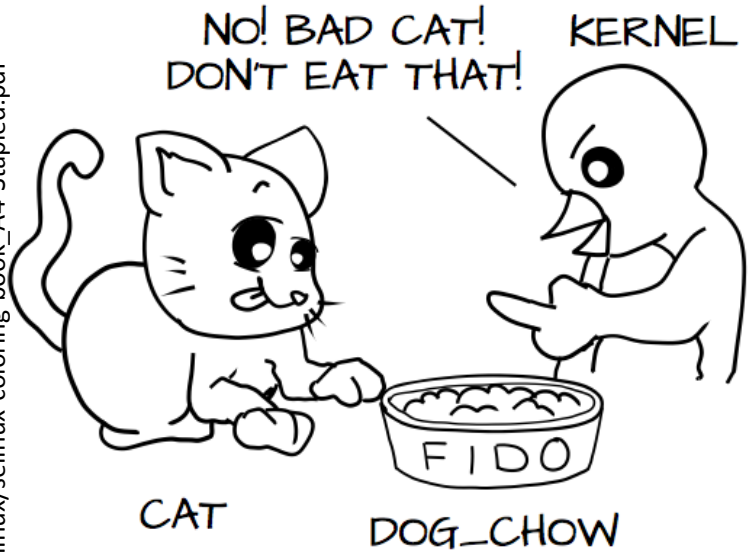
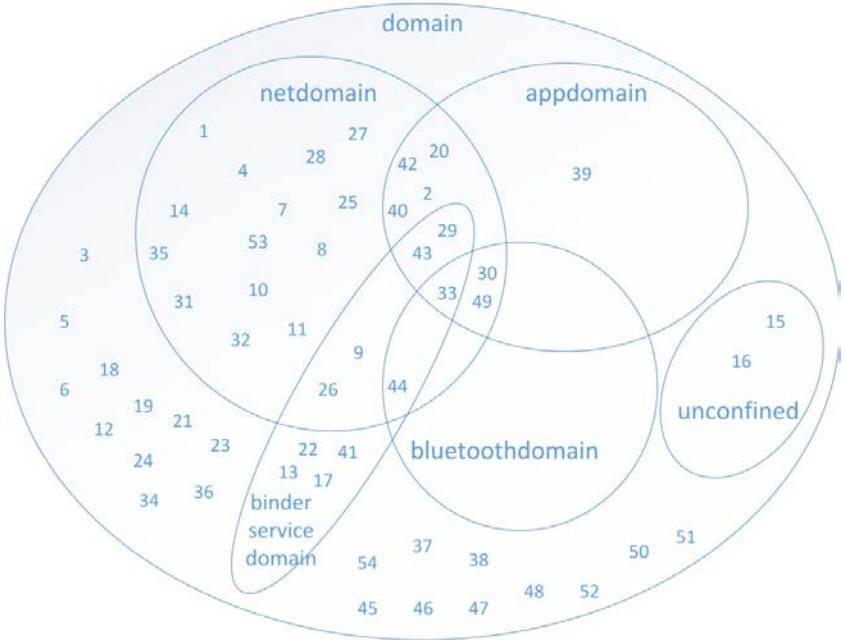
SEAndroid – in 90 minutes



Pic: http://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf

allow dog fido : dog_chow eat
neverallow {domain -dog} fido : * *



pic: Elena Reshtova

Jan-Erik Ekberg

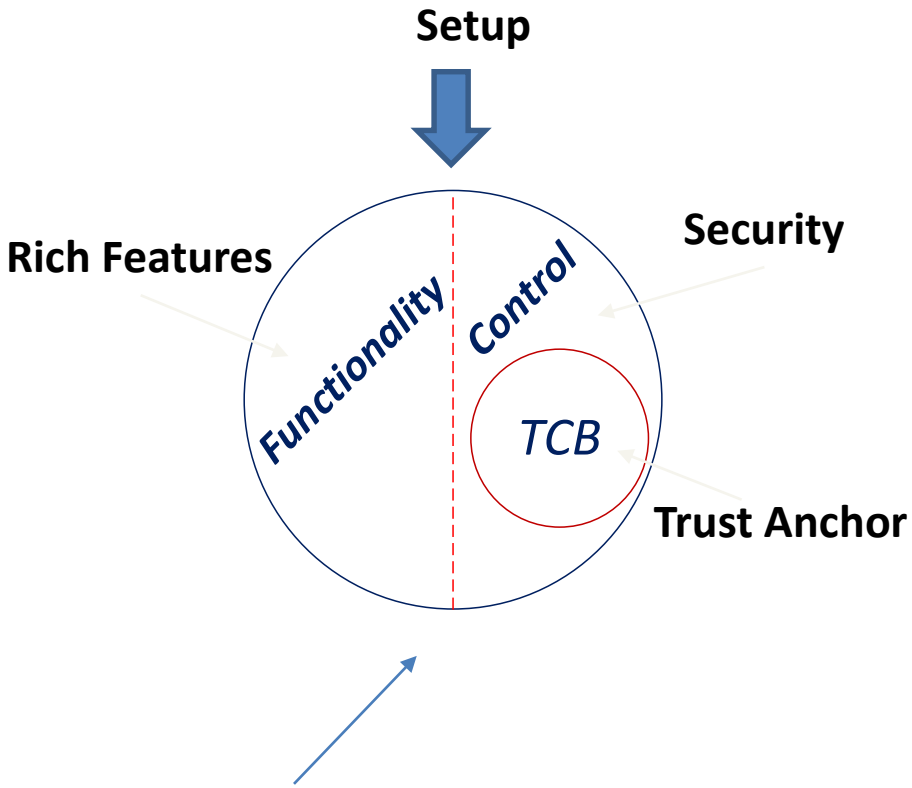
Huawei, CTO, Mobile Security

26.2.2019

Advert (for Thursday)

What is platform security?

1. Isolation of apps, actors, containers, sandboxes
2. A way to set up and configure the isolation (secure boot)
3. Some keys and trust roots



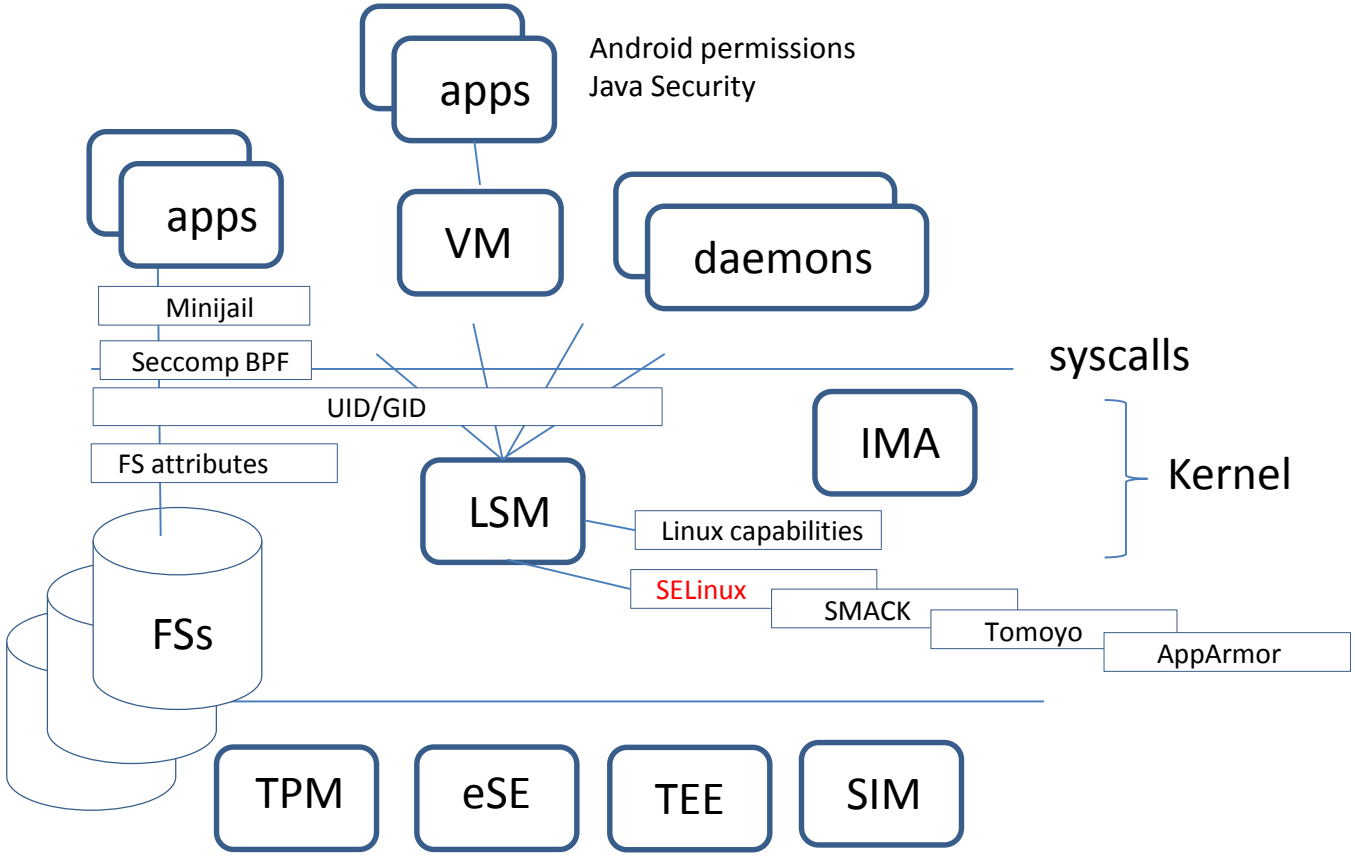
Today we look at one (abstract) access control paradigm

On Thursday we look at practical problems with all of this

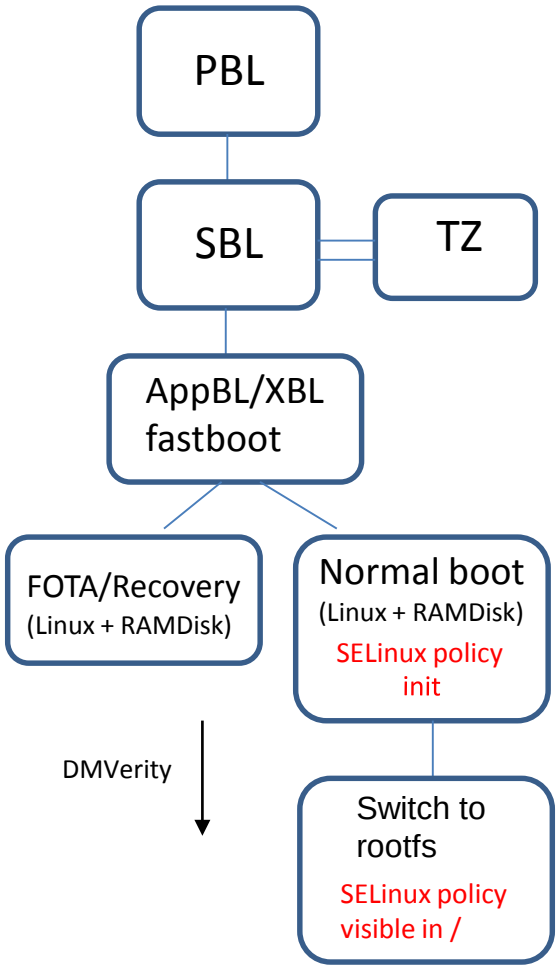
We are also at a crossroads where Spectre, ROCA, PQC(?) are in a sense deteriorating the fundaments of PlatSec

Platform Access Control (Linux)

1. Constrain access to least-privilege
2. Protect against infection
3. **Multi-user isolation**
4. **Multi-app isolation**



Secure Boot (~Android)



1. Chain validated based on PK Hash in fuses
2. Many bootloaders from different stakeholders
3. Integrity guarantees (w. rollback protection)

1. Policy in RAMdisk
2. --> integrity “guaranteed”
   - Activation in init binary
   - --> Time of activation before full filesystem in use

Some milestones

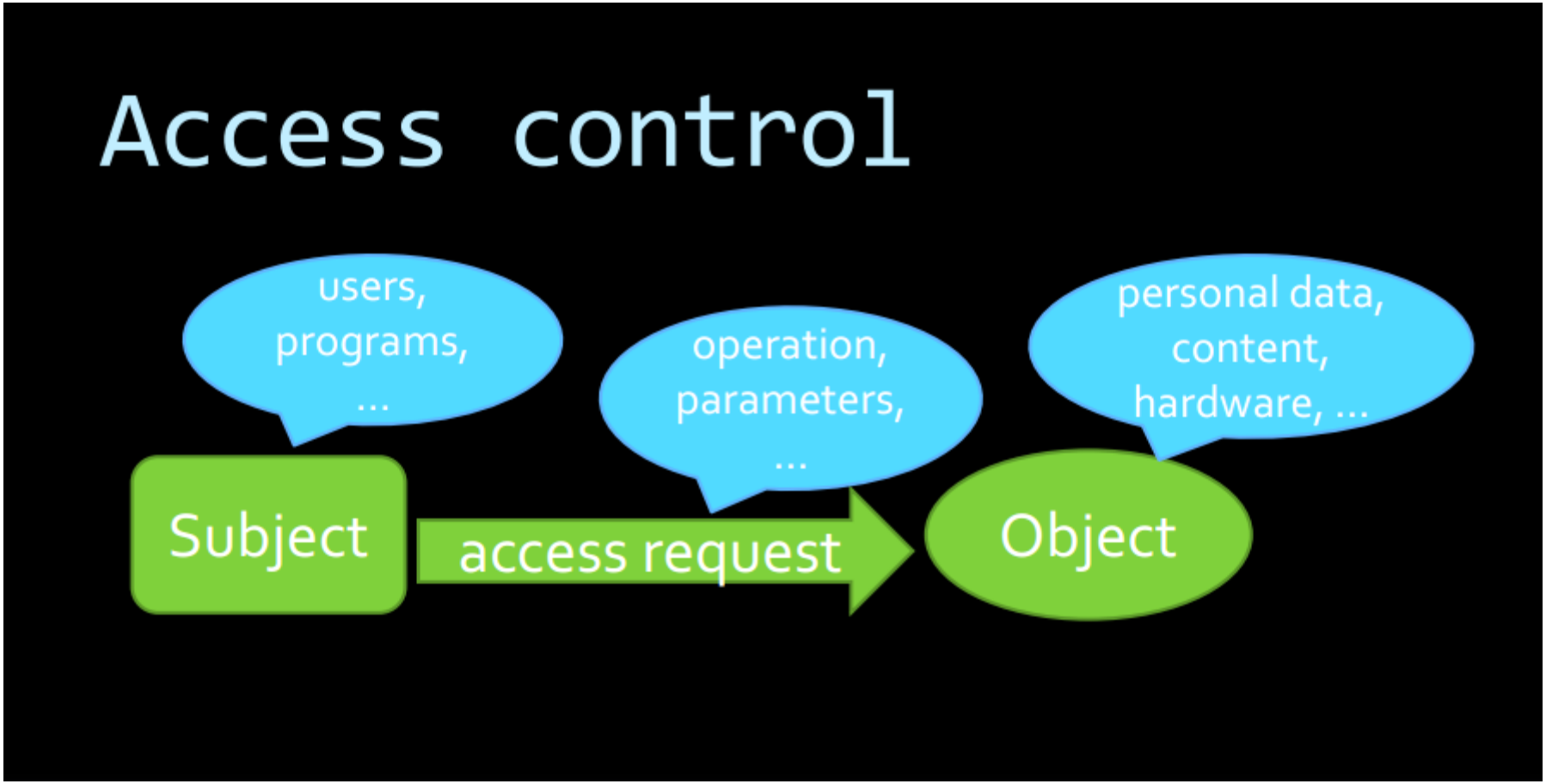
- (Tech rep 1973) Multics Security Enhancements

System (kernel) access control

- (Usenix 99): The Flask Security Architecture
- Access control decision and enforcement separation
- (Usenix 01): Meeting Critical Security Objectives with
- Security-Enhanced Linux (Loscocco / Smalley)
- Domain-type enforcement for Linux (SELinux)

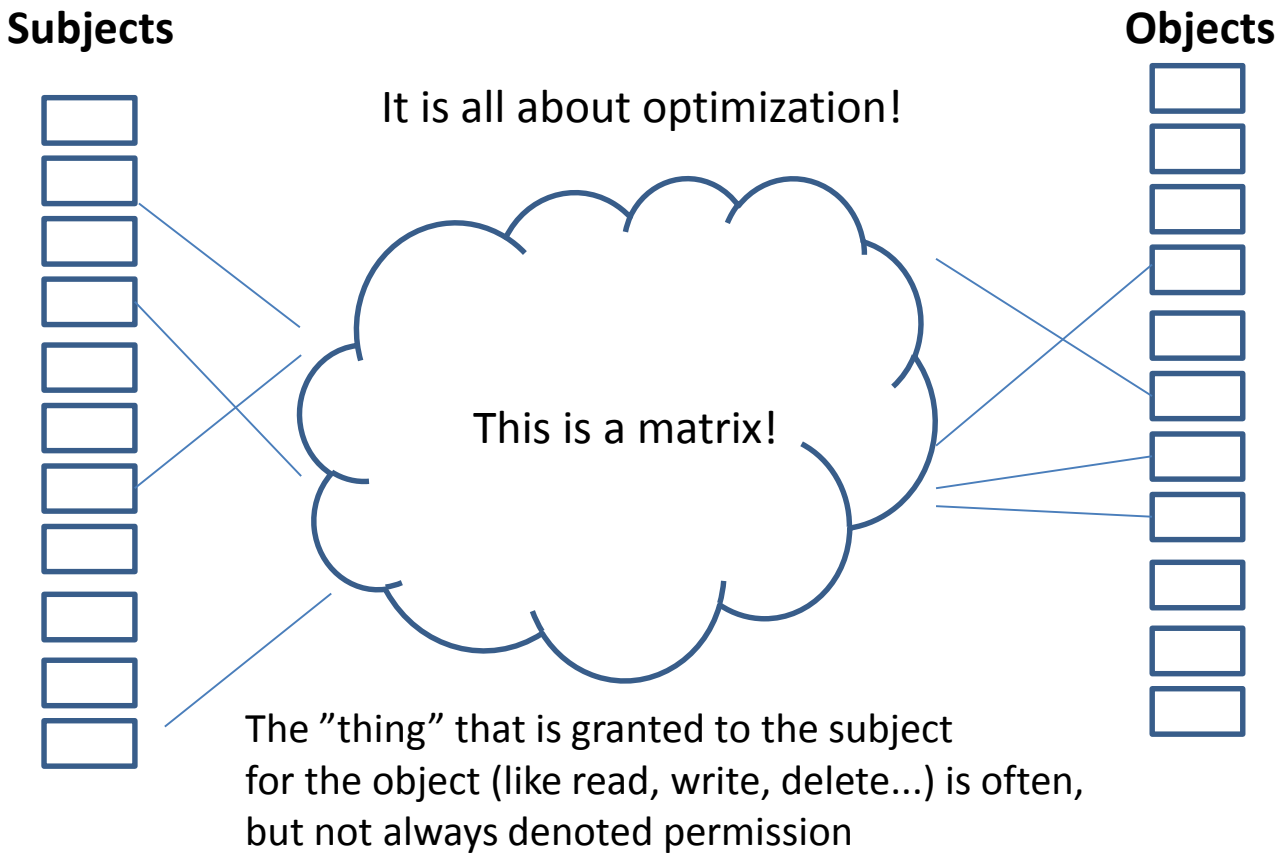
After 3 years of innovation and 15 years of implementation work →SEAndroid

This you already learned ...



Access control mechanisms

1. Access control lists
2. Permissions / capabilites
3. **Domain-type**



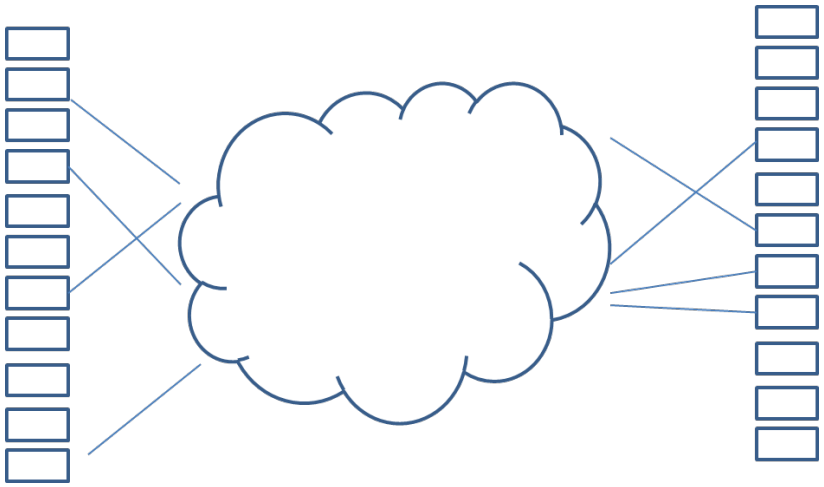
Other ”interesting avenues”: Role-base access control (RBAC), Low-watermark (security levels), Lattice-based, Discretionary Access Control (DAC) = user access, information-flow based

A type is a domain when it is not a type

Not like this: “There is no practical difference between a type and a domain. The policy rules gives them significance. In particular, if an object has the same type as a process' domain, this means something only if the policy explicitly says so (it usually does). All types can be applied to any object since they are just names”

Subject types are domains **Object types are types**

A subject operates on an object, and as an optimization a domain operates on a type



SEAndroid types

Computing system objects

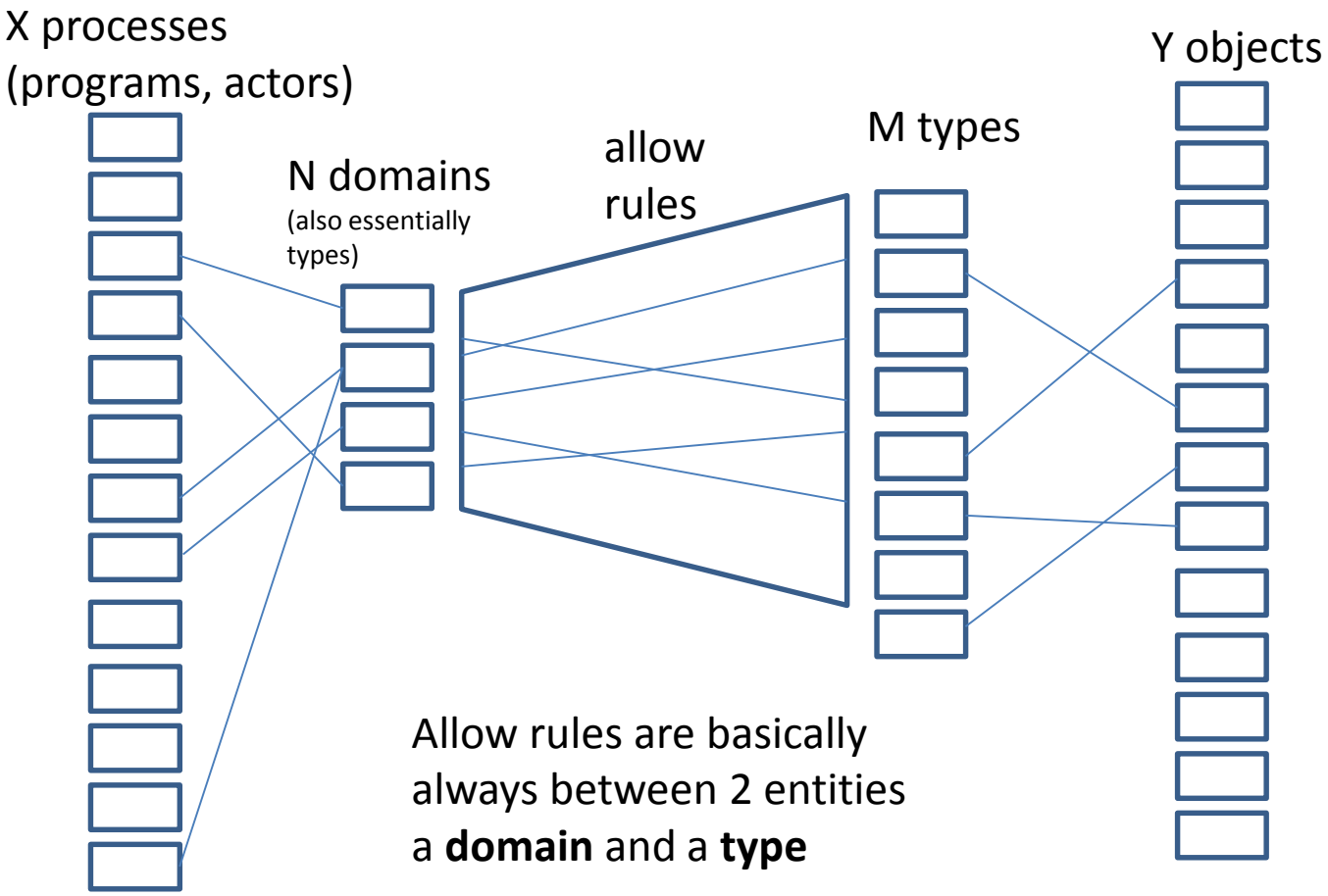
SEAndroid types

Computing system objects

Basic principle (dimension 1)

Subjects are primarily processes. Processes that participate in the access control are assigned a **domain**

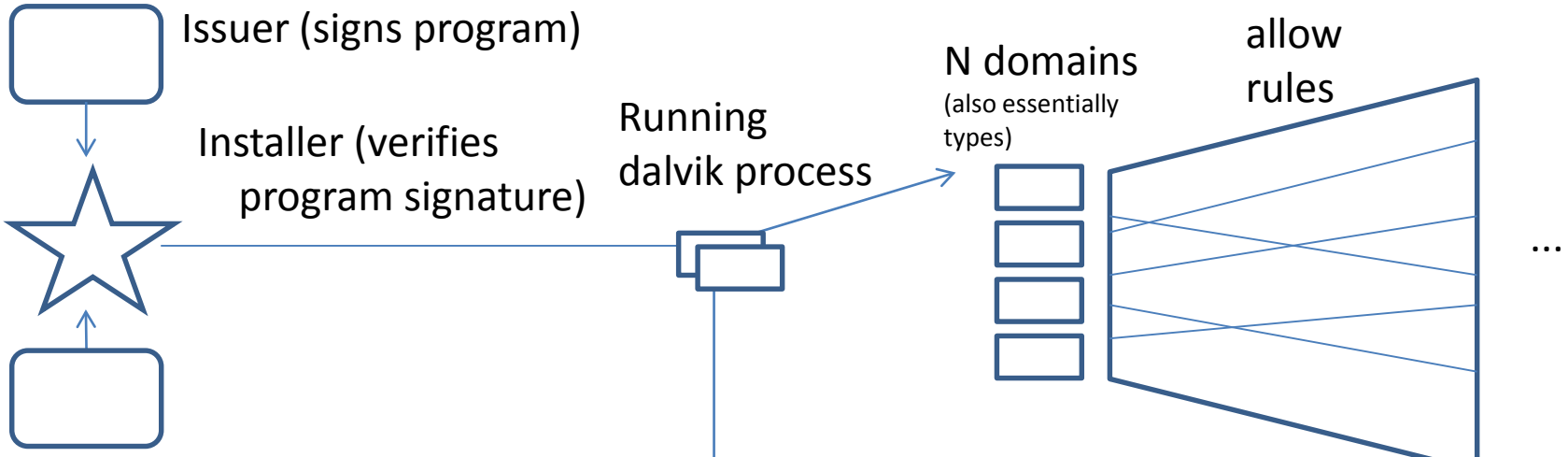
Objects are e.g. files, sockets.. Objects that participate in the access control are assigned a **type**



Domain and types are defined by each policy, and vary. Domain and type names are NOT FIXED by any agreement, i.e. **domains and types cannot be counted on to remain consistent in name or meaning across policy generations**.

Basic principle (dimension 1B)

In Android, the processes are to a large extent **middleware** processes running In the Dalvik virtual machine, and installed by the Android installer. The Middleware MAC (MMAC) addresses this detail



SEAndroid MMAC policy
(**key:** public keys for programs
values: 1) seinfo→domain,
2) android permission list)

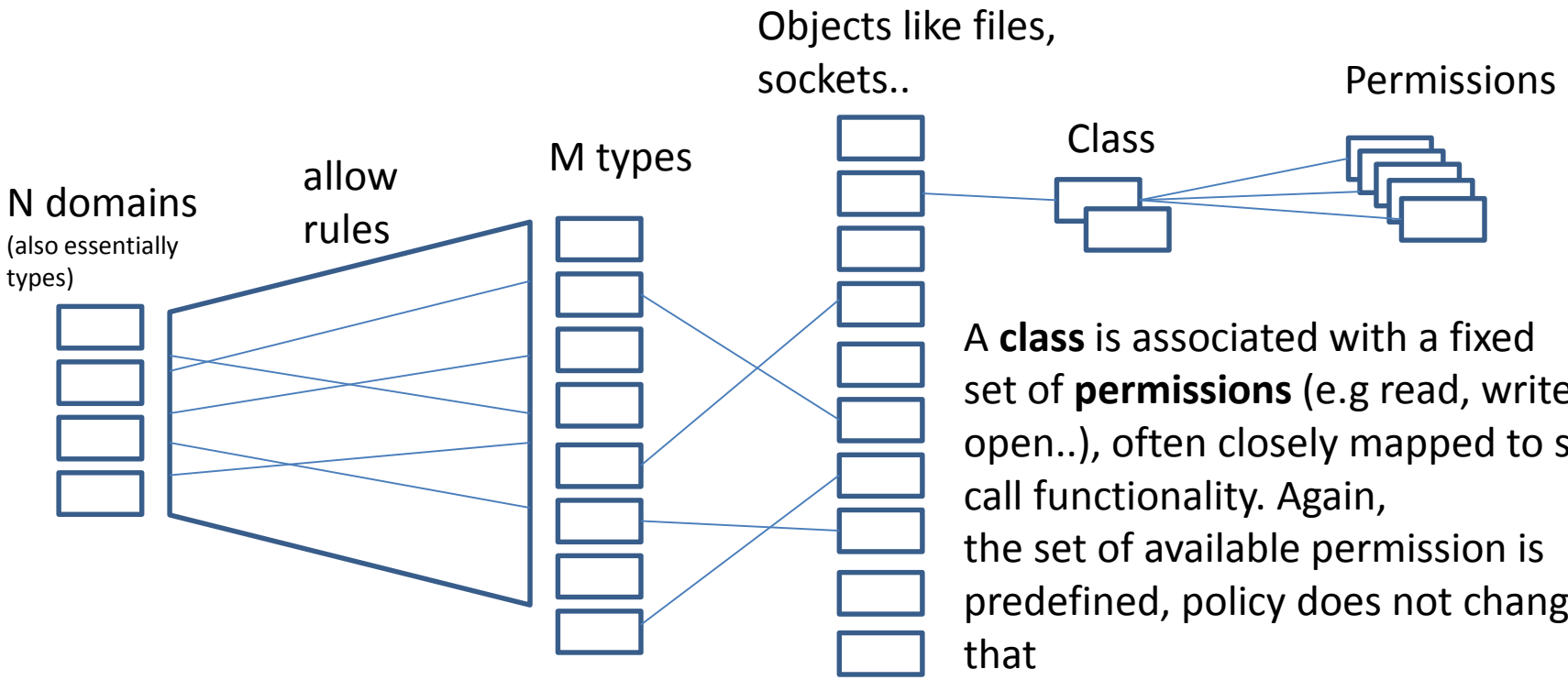
Android permissions
(e.g. access to camera, net, ...)

Android subject identity:

- **Origin (RSA key)**
- **Package name**
- **UID (local)**
- **Filename (system services)**

Basic principle (dimension 2)

An object is **always** an OS primitive of some sort. This is not a policy issue, this is reality. An object therefore belongs to one or more **classes** which are predefined by the system proper. A class can be e.g. **file, socket, character device, ...**



A **class** is associated with a fixed set of **permissions** (e.g read, write, open..), often closely mapped to system call functionality. Again, the set of available permission is predefined, policy does not change that

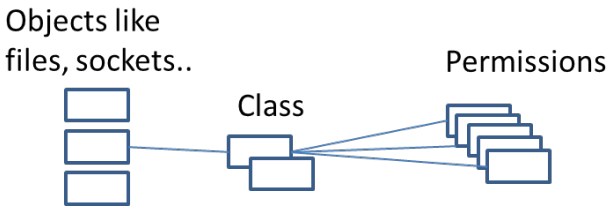
But! An allow rule includes also the class, and within that the permissions allowed by the rule. Class and permission names are ”well known” as defined by NSA / SELinux

Classes and permissions (examples)

These are more or less defined by

http://selinuxproject.org/page/ObjectClassesPerms

(or in Android source code:
android / platform/external/sepolicy / master / . / access_vectors



A daemon or service provides Unix domain socket access for clients

unix_stream_socket

append, **bind, connect, create**, **write**, relabelfrom
ioctl, name_bind, sendto, recv_msg, send_msg
getattr, setattr, accept, getopt, **read**, setopt, shutdown, recvfrom
lock, relabelto, **listen**, **acceptfrom, connectto**, newconn

A device driver (say a serial port) provides device access

chr_file

append, create, execute, **write**, relabelfrom, link, unlink
ioctl, getattr, setattr, **read,** rename, lock, relabelto, mounton
quotaon, swapon, audit_access, entrypoint, execmod,
execute_no_trans, **open**

Class/ Permission illustration

Inside fs/open.c (sys_open syscall)

```
static int do_dentry_open(struct file *f, int (*open)(struct inode *, struct file *),
                          const struct cred *cred)
 ...
  error = security_file_open(f, cred);
  if (error) goto cleanup_all;
```

← class 'file', permission 'open'

Inside security.c (LSM)

```
  int security_file_open(struct file *file, const struct cred *cred) {
      int ret;
      ret = security_ops->file_open(file, cred);
      if (ret) return ret;
      return fsnotify_perm(file, MAY_OPEN);
```

Inside selinux/hooks.c

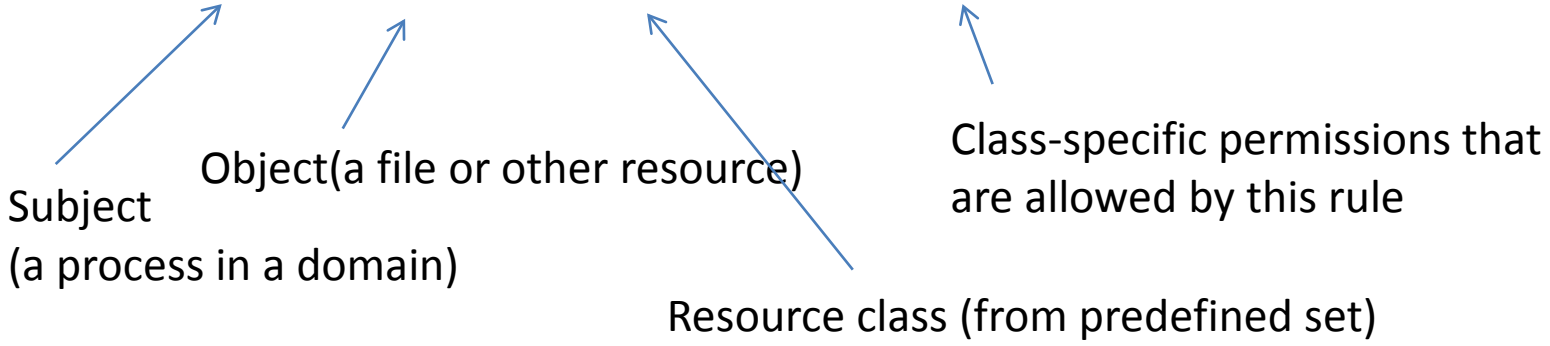
```
  .file_open =  selinux_file_open,
```

SEAndroid policy allow rules

Since SEAndroid essentially is a single-user system, much of the complexity of users and roles is collapsed (one user, one role only)

A simple rule (there is in fact more to it) consists of:

ALLOW [domain] [type] : [class] {[allowed permissions]}



The [type] can be **self**. If applied to an attribute (explained later), it allows only access within each atomic domain, not among all pairs of domains covered by the attribute.

Allow rule examples (from a commercial Android phone)

1) allow **untrusteddomain** **hci_attach_dev** : **chr_file** { ioctl read getattr lock open } ;

2) allow **system_app** **dhcp_data_file** : **lnk_file** { ioctl read getattr lock open } ;

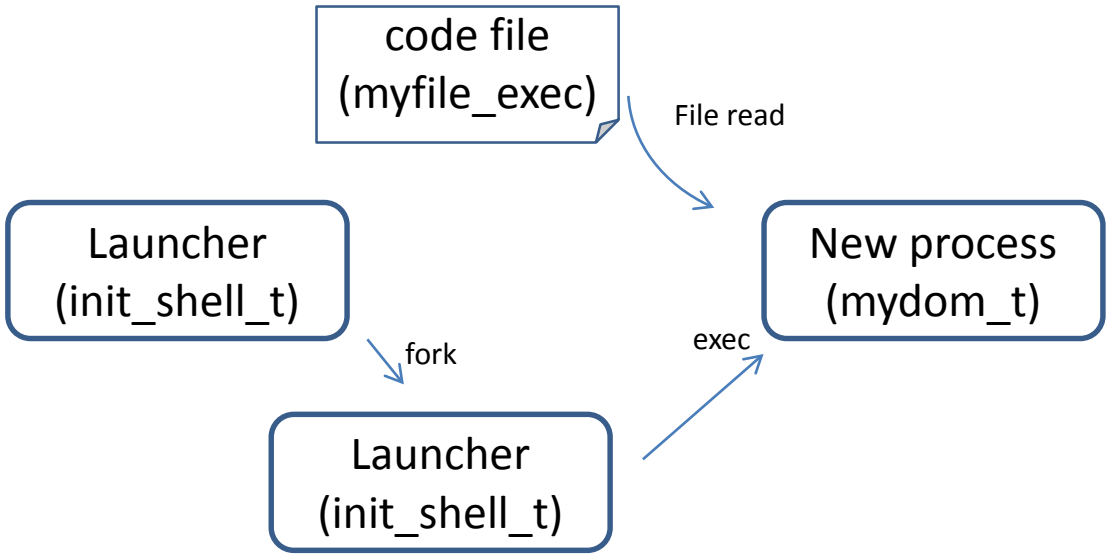
3) allow **untrusteddomain** **system_app** : **binder** { call transfer } ;

4) allow **logwrapper** **dhcp_system_file** : **dir** { ioctl read getattr search open } ;

5) allow **healthd** **healthd_exec** : **file** { read execute entrypoint } ;

Transition rule(sets)

Subjects and objects rules operate according to their domains/types : But how does a new process get into its domain? (http://selinuxproject.org/page/TypeRules#type_transition_Rule)



Intent ("what we want to happen")

type_transition init_shell_t myfile_exec: process mydom_t;

File execution right

allow init_shell_t myfile_exec: file execute;

File type is an entrypoint into a domain ("object firewall")

allow mydom_t myfile_exec: file entrypoint;

Process type needs transition right into a domain ("subject firewall")

allow init_shell_t mydomain_t: process transition;

File contexts / labeling & transition rule for files

The file contexts file is (in SEAndroid) the source of filesystem labeling

Example lines:

/var/log u:object_r:**var_log_t**:s0
/dev/block/ram[0-9]* u:object_r:ram_device:s0
/dev(/.*)? u:object_r:device:s0
/system/bin/sh u:object_r:shell_exec:s0

When we want to control the type of files being written (in a shared dir..)

Intent ("what we want to happen")

type_transition mydom_t var_log_t : file tmp_t;

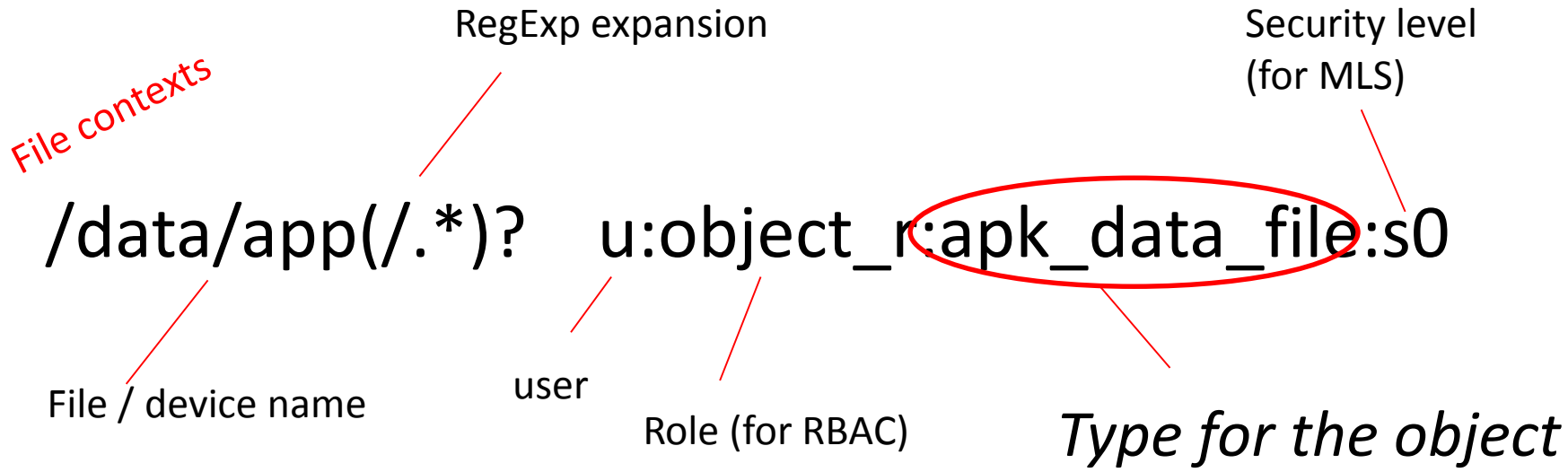
Right to write to the directory (with type var_log_t)

allow mydom_t var_log_t: dir { add_name write search } ;

Right to write files of type tmp_t

allow mydom_t tmp_t: file { create write };

Context files have more attributes than allow rules



SEApp contexts

(different parser, obviously..)

user=_app seinfo=myapp domain=myapp_app type=app_data_file

View of one SEAndroid policy

Different kinds of ”targets”

”target” permissions

rules

Domains and types

”domain entry rules”

```
Statistics for policy file: sepolicy
Policy Version & Type: v.26 (binary, mls)

   Classes:              84    Permissions:        249
   Sensitivities:         1    Categories:        1024
   Types:               646    Attributes:          44
   Users:                 1    Roles:                2
   Booleans:              9    Cond. Expr.:          9
   Allow:            112271    Neverallow:           0
   Auditallow:            0    Dontaudit:          173
   Type_trans:          227    Type_change:          0
   Type_member:           0    Role allow:           0
   Role_trans:            0    Range_trans:          0
   Constraints:          63    Validatetrans:        0
   Initial SIDs:         27    Fs_use:              16
   Genfscon:             18    Portcon:              0
   Netifcon:              0    Nodecon:              0
   Permissives:           0    Polcap:               2
```

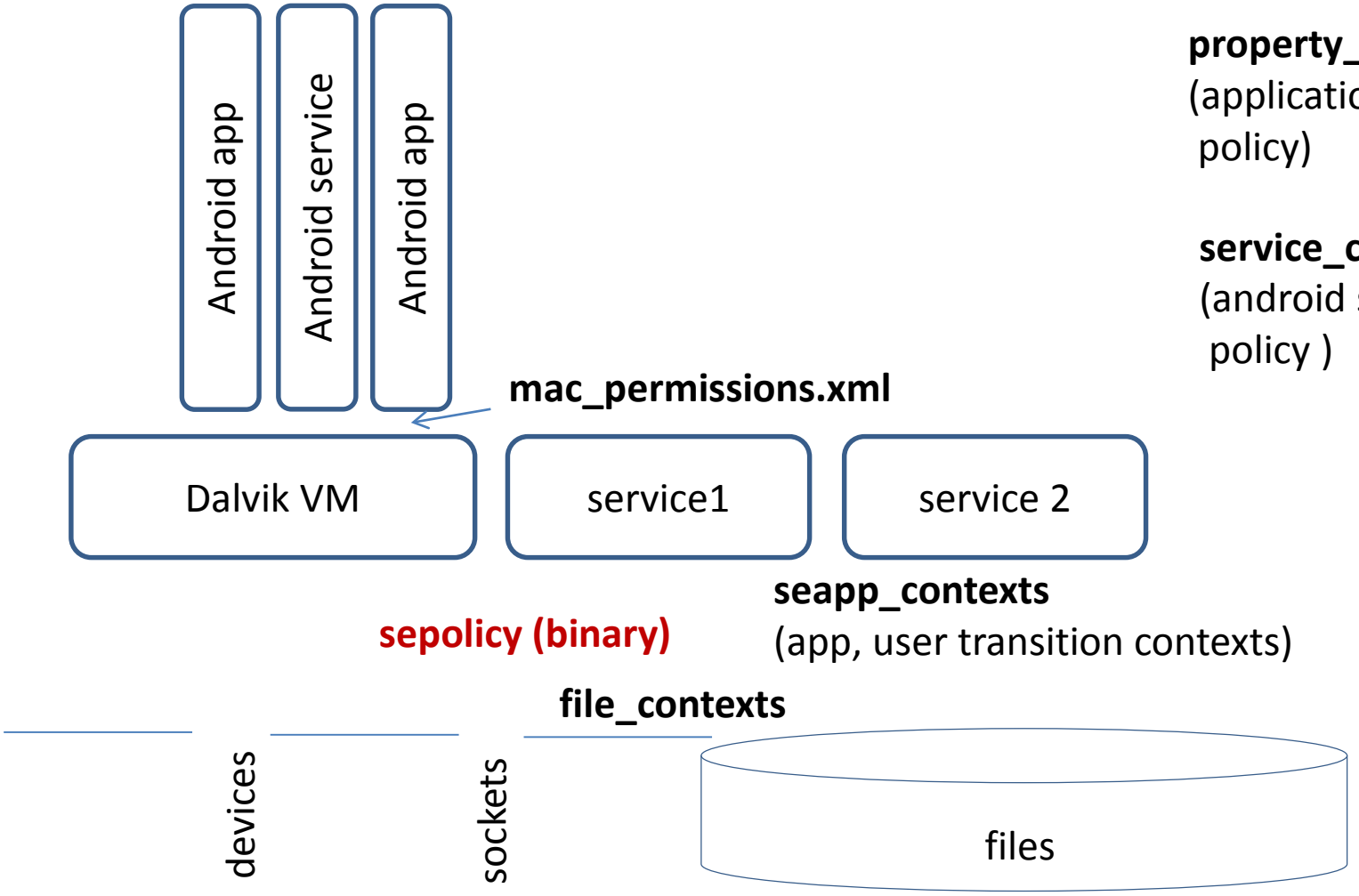
The complexity of the listed policy is close to that of Fedora

Smalley & al (SEAndroid)

	SE Android	Fedora
Size	71K	4828K
Domains	39	702
Types	182	3197
Allows	1251	96010
Transitions	65	14963
Unconfined	3	61

Table 3. Policy size and complexity.

System configuration view



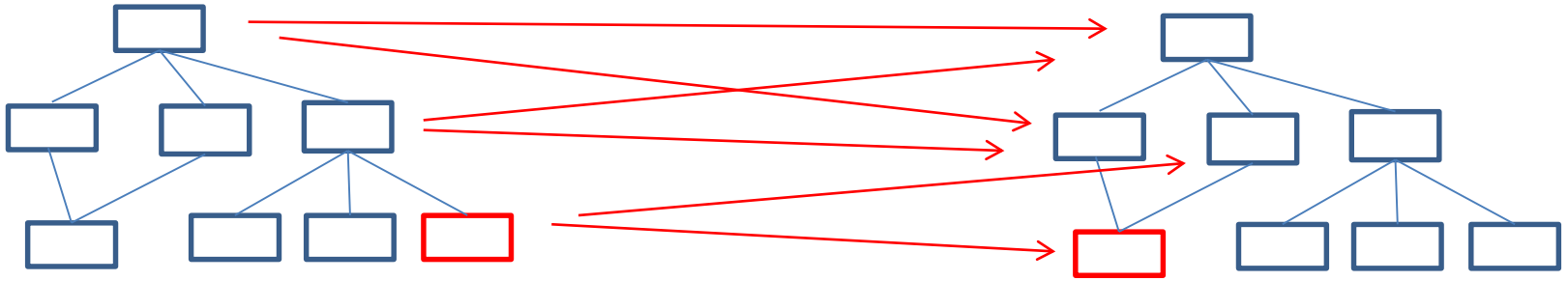
Attributes adds complexity to analysis

Attributes are groups of types (or domains). Rules can also be defined using attributes.

N domains in attributes
(also essentially types)

Examples of applicable allow rules

M types with attributes



It is typical that class attributes for overall access come from **different rules**. Say between a process and a resource, the read rule can come from attributes higher in the hierarchy, whereas the write access may be specific to individual domains and types.

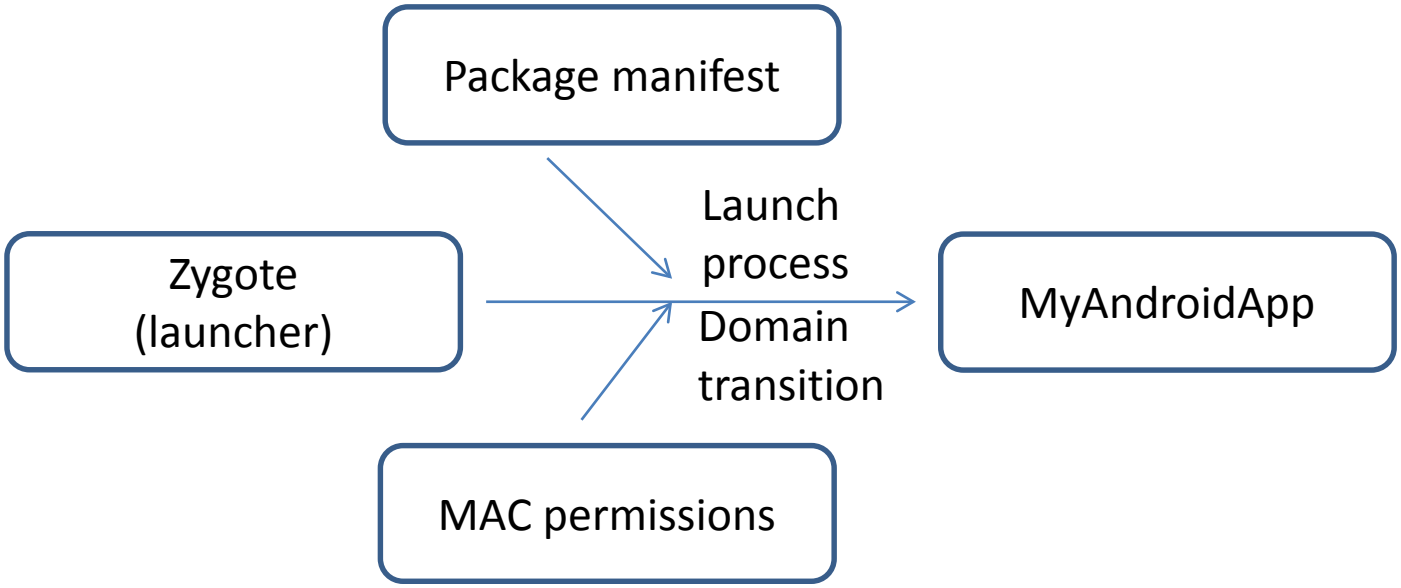
mac_permissions

(From comments in the XML file)

- A signature is a hex encoded X.509 certificate or a tag defined in keys.conf and is required for each signer tag.
- A signer tag may contain a seinfo tag and multiple package stanzas.
- A **default tag** is allowed that can contain policy for all apps not signed with a previously listed cert. It may not contain any inner package stanzas.
- Each signer/default/package tag is allowed to contain one seinfo tag. This tag represents additional info that each app can use in setting a SELinux security context **on the eventual process**.

```
<!-- Platform dev key in AOSP -->
<signer signature="@PLATFORM" >
  <seinfo value="platform" />
</signer>
<!-- All other keys -->
<default>
   <seinfo value="default" />
</default>
```

Launching with MAC



In Android, the PackageManager is a front-end to Installer data

```
String[] packageNames =
        getPackageManager().getPackagesForUid(uid);
try{
   PackageInfo pkgInfo = getPackageManager()
                        .getPackageInfo( packageNames[0],
                         PackageManager.GET_SIGNATURES);
 android.content.pm.Signature[] sigs = pkgInfo.signatures;
 Log.i("Signature", sigs[0].toCharsString());
...
```

”Example” mac_permissions.xml entry

<**signer** signature=“… 16ef8108a353a9f7300d06092a864886f70d01010b0500038
20101002ae36b53bd209841e4"><allow-all/><**seinfo** value=“**myprog**"/></**signer**>

For the MMAC part of the policy, the applications are bound by origin. I.e. the mac_permissions.xml contains a hundred or so public keys in x509/DER format encoded as XML as the example above. The fields are

-- **signature:** The x509 certificate containing the public part of the application signing key
-- **<allow-all/>:** The MMAC puts no extra constraints on the android permissions
-- **seinfo:** **A mapping to the domain of the policy**

As it happens we find the following line in the **seapp_contexts** policy file:

user=_app seinfo=**myprog** domain=**myprog_app** type=app_data_file

I.e. any application signed with a private RSA key corresponding to the public key mentioned in a certificate in the mac_permissions.xml that follows the template above, will be mapped to the **trustonicpartner_app** domain and be accessible (as an object) in accordance with the type **app_data_file**

Android Properties

A policy name resolver for property contexts

A database of configurations and status (like windows registry)

Listed in property_contexts file, e.g.

net.gprs u:object_r:net_radio_prop:s0 *)

selinux. u:object_r:**security_prop**:s0

Associated with an allow rule, e.g.

allow system **security_prop** : property_service set

Fixed

1) Property service calls **selabel_lookup (.., .., ”selinux.reload”, ..)**
2) Lookup finds, and returns *) → **u:object_r:security_prop:s0**
3) Service asks policy using **selinux_check_access()** for source request

SEAndroid protection is a moving target (1)

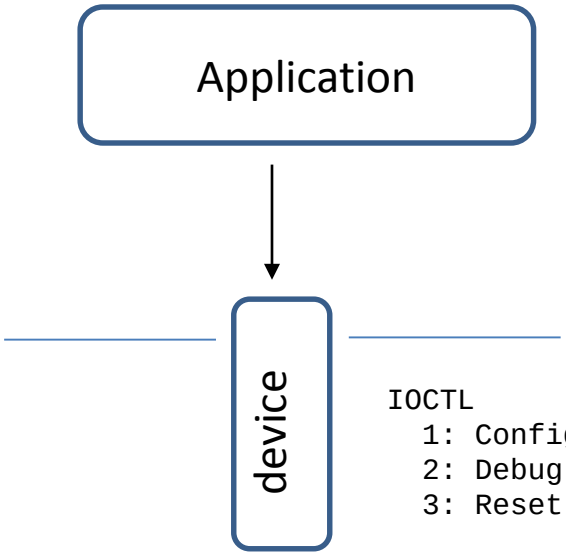
Addition (v30): “Extended permissions” == IOCTL protection

Available rules: allowxperm, dontauditxperm auditallowxperm and neverallowxperm

Examples:

```
allow src_t tgt_t : tcp_socket ioctl;
allowxperm src_t tgt_t : tcp_socket ioctl ~0x8927;

allow tee tee_device : chr_file open read write ioctl
allowxperm tee tee_device : chr_file ioctl 0x917
```



The ioctl command is a 32 bit number comprised of four fields,
number - sequence number of the command. 8 bits
type - magic number assigned to the driver. 8 bits
size - size of the user data involved. typically 14 bits (arch dep.)
direction - The direction of data transfer. typically 2 bits (arch dep.)

IOCTL
1: Configure
2: Debug (expose internal memory)
3: Reset in case of error

https://selinuxproject.org/page/XpermRules

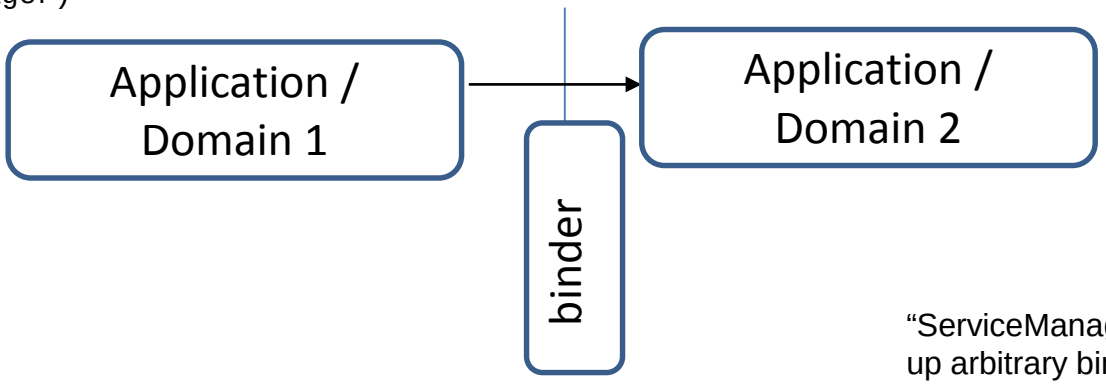
SEAndroid protection is a moving target (2)

Binder protection: "RPC" for Android applications

Has been available since day one, increasingly taken into use
If carefully used, can limit some obvious data flow attacks

Permissions for Binder class:

```
call                    Perform a binder IPC to a given target process domain (can A
call B?).
impersonate             (Not currently used)
set_context_mgr         Register self as the Binder Context Manager (aka service-manager)
transfer                Transfer a binder reference to another process (used by service-
manager)
```



"ServiceManager prevents apps from looking up arbitrary binder services."

Permissions for Servicemanager class (userspace object):

```
add                     Add a service
list                    List services
find                    Find services
```

http://kernsec.org/files/lss2015/lss2015_selinuxinandroidlollipopandm_smalley.pdf

http://selinuxproject.org/page/NB_SEforAndroid_1

SEAndroid protection is a moving target (3)

Multi-Level Security (used since v5 or thereabouts)**:**

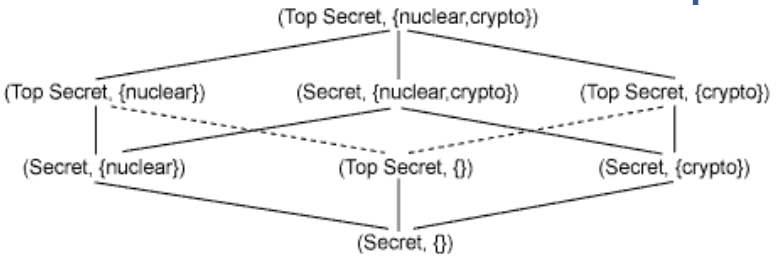
Separation between apps and users on a policy level.
In general, MLS allows domains to access types based on **level** (ordered) and **category**(unordered)
What we see in SEAndroid are really categories

Example avc error (adb logcat):

type=1400 audit(0.0:7): avc: denied { search } for name="com.android.providers.downloads" dev="mmcblk0p23" ino=81932
scontext=u:r:system_app:s0 tcontext=u:object_r:app_data_file:s0:c512,c768 tclass=dir permissive=0

Google m4 macros (later) has support:

mlstrustedsubject
levelFrom=user, app, all



http://www.cs.cornell.edu/courses/cs5430/2012sp/mls.html

Process (domain) assignment:

```
            if (cur->levelFrom != LEVELFROM_NONE) {
char level[255];
switch (cur->levelFrom) {
case LEVELFROM_APP:
        snprintf(level, sizeof level, "s0:c%u,c%u",
                 appid & 0xff,
                 256 + (appid>>8 & 0xff));
        break;
case LEVELFROM_USER:
        snprintf(level, sizeof level, "s0:c%u,c%u",
                 512 + (userid & 0xff),
                 768 + (userid>>8 & 0xff));
        break;
case LEVELFROM_ALL:
        snprintf(level, sizeof level, "s0:c%u,c%u,c%u,c%u",
                 appid & 0xff,
                 256 + (appid>>8 & 0xff),
                 512 + (userid & 0xff),
                 768 + (userid>>8 & 0xff));
```

http://marc.info/?l=seandroid-

Miscellaneous

neverallow rules (assertions)

- a way to shield off unwanted patterns

neverallow { domain -debuggerd -vold -dumpstate -system_server } self:capability sys_ptrace;

dac_override (and other capabilities)

- does show up in policies – one of the 32 linux capabilities
- overrides all ”standard” file permission checks

allow installd installd : capability { dac_override, sys_nice}

unconfined (macro expansion) – not available any more

- the macros are discussed later. A domain in the attribute unconfined, will be allowed all (class/permission) access to any type. Shows up as a domain in the final policy
- a testing tool, e.g. to determine interaction with DAC

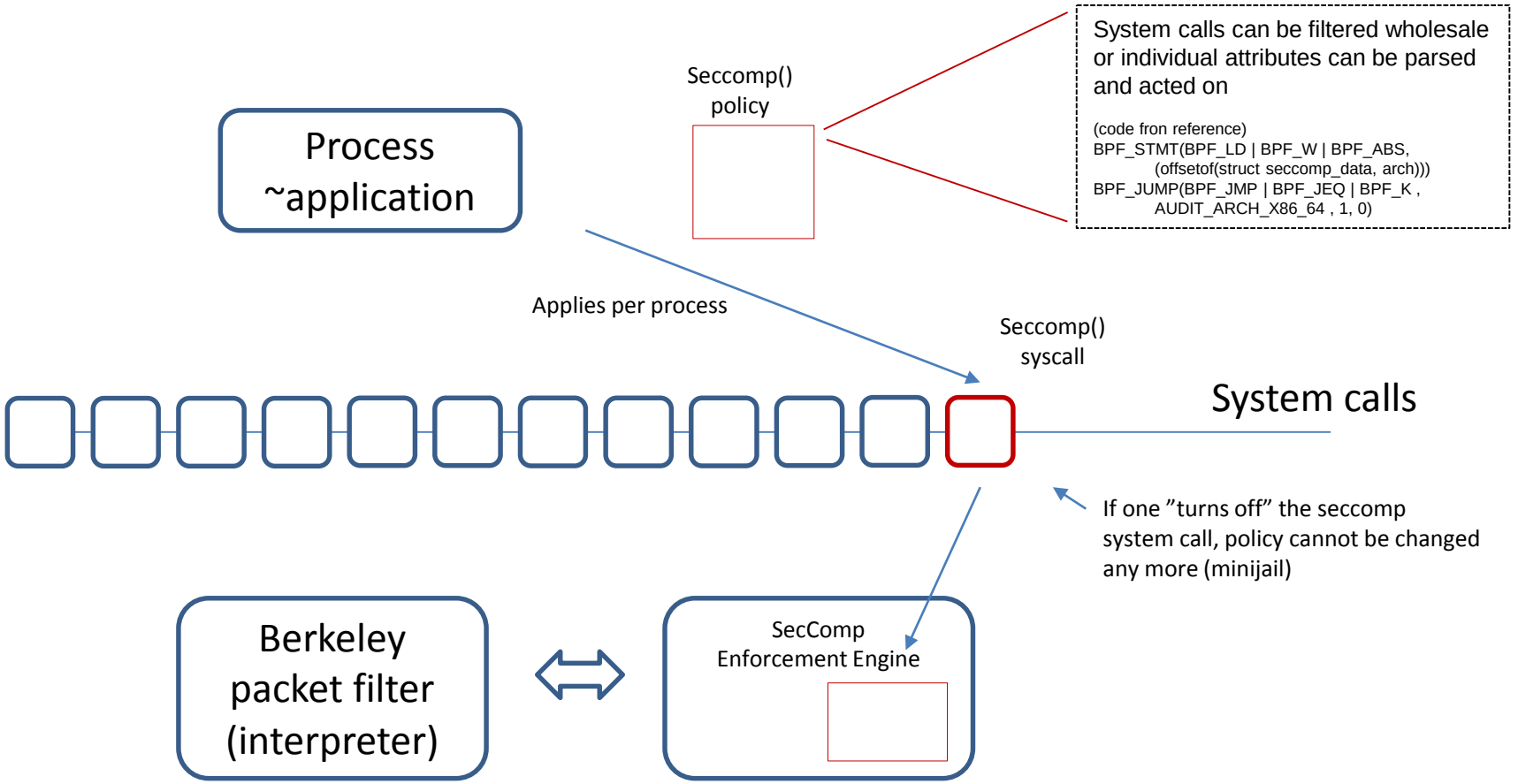
self (macro expansion)

allow netd self: { tcp_socket udp_socket} *

- As a target, denotes the domain itself, but not any parent attributes

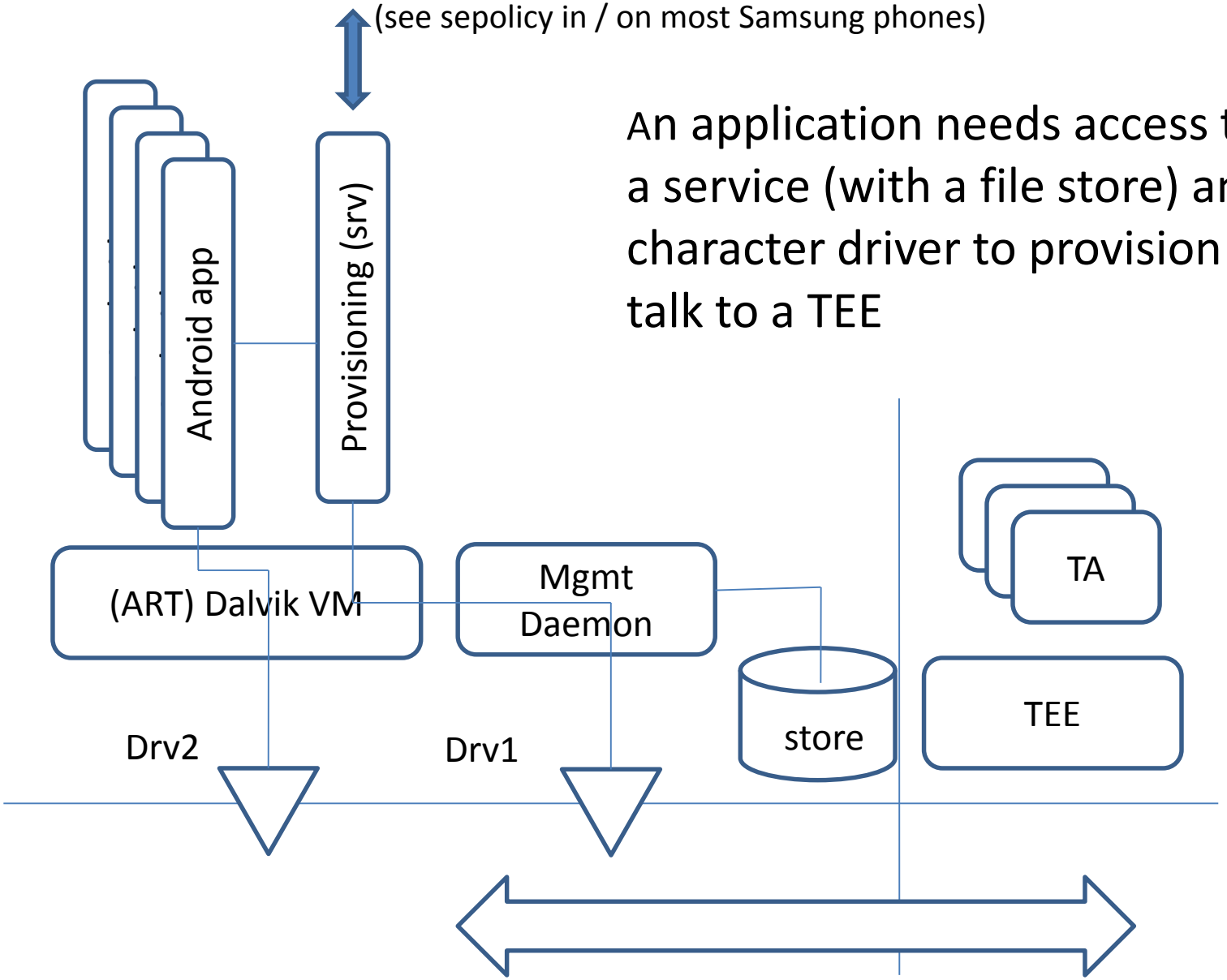
SecComp as the ”next level of access control”
-- Android O forward – all apps have a seccomp filter

- ”Secure Computing” filter == SecComp, first version 2005
- ”Progammable policy for application on system call layer – very fine grained
- Depending on policy, may cause double-digit performance overhead



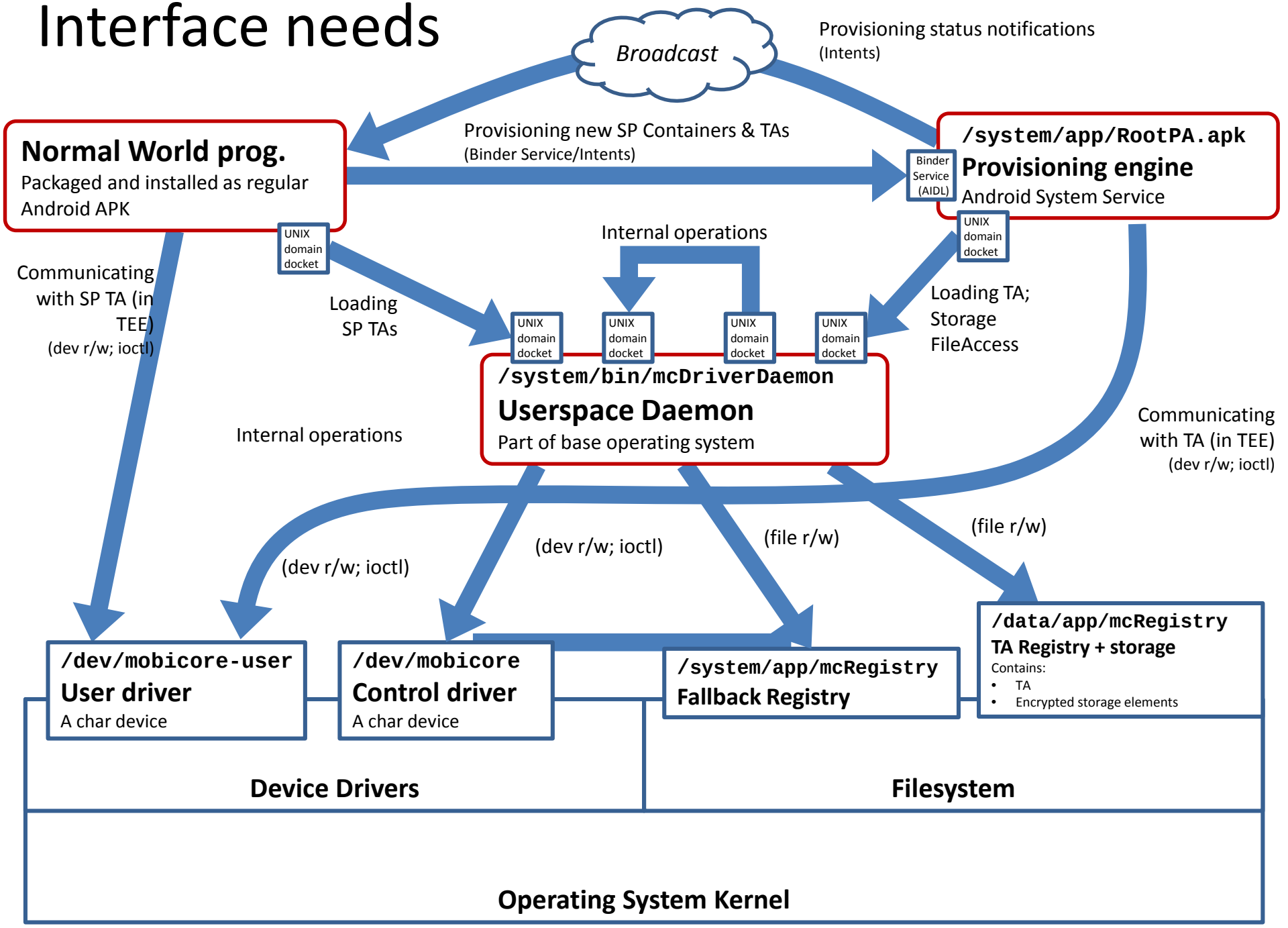
Introduction: https://lwn.net/Articles/656307/

”My” example use case (TEE in Samsung)



An application needs access to a service (with a file store) and a character driver to provision and talk to a TEE

Interface needs



Interfaces in practice

Userspace daemon

```
mcDriverD  2399  root  exe  ???  ???  ???  ??? /data/app/mcDriverDaemon
mcDriverD  2399  root    0  ???  ???  ???  ??? /dev/null
mcDriverD  2399  root    1  ???  ???  ???  ??? /dev/null
mcDriverD  2399  root    2  ???  ???  ???  ??? /dev/null
mcDriverD  2399  root    3  ???  ???  ???  ??? /dev/log/main
mcDriverD  2399  root    4  ???  ???  ???  ??? /dev/log/radio
mcDriverD  2399  root    5  ???  ???  ???  ??? /dev/console
mcDriverD  2399  root    6  ???  ???  ???  ??? anon_inode:dmabuf
mcDriverD  2399  root    7  ???  ???  ???  ??? anon_inode:dmabuf
mcDriverD  2399  root    8  ???  ???  ???  ??? /dev/log/events
mcDriverD  2399  root    9  ???  ???  ???  ??? /dev/log/system
mcDriverD  2399  root   10  ???  ???  ???  ??? /dev/mobicore
mcDriverD  2399  root   11  ???  ???  ???  ??? /dev/__properties__ (deleted)
mcDriverD  2399  root   12  ???  ???  ???  ??? socket:[4949]
mcDriverD  2399  root   13  ???  ???  ???  ??? socket:[5325]
mcDriverD  2399  root   14  ???  ???  ???  ??? socket:[5326]
mcDriverD  2399  root   15  ???  ???  ???  ??? /dev/mobicore-user
mcDriverD  2399  root   16  ???  ???  ???  ??? socket:[5328]
mcDriverD  2399  root   17  ???  ???  ???  ??? socket:[5329]
mcDriverD  2399  root   18  ???  ???  ???  ??? socket:[4958]
mcDriverD  2399  root   19  ???  ???  ???  ??? socket:[4960]
```

Control driver
/dev/mobicore

User driver
/dev/mobicore-user

Issued (signed) program

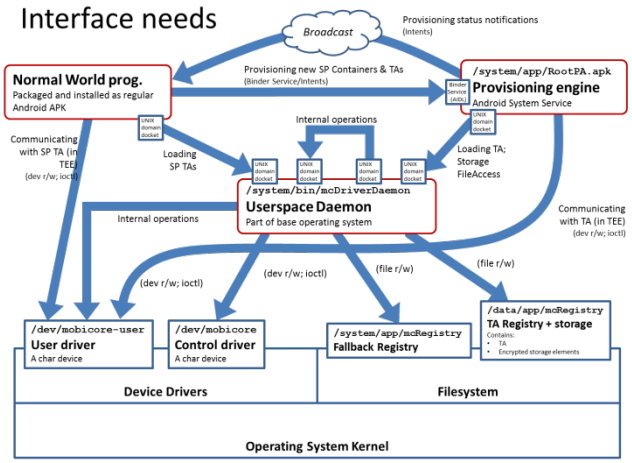
```
someca  2424  root  exe  ???  ???  ???  ??? /data/app/lta
someca  2424  root    0  ???  ???  ???  ??? /dev/ttySAC2
someca  2424  root    1  ???  ???  ???  ??? /dev/ttySAC2
someca  2424  root    2  ???  ???  ???  ??? /dev/ttySAC2
someca  2424  root    3  ???  ???  ???  ??? /dev/log/main
someca  2424  root    4  ???  ???  ???  ??? /dev/log/radio
someca  2424  root    5  ???  ???  ???  ??? /dev/console
someca  2424  root    6  ???  ???  ???  ??? anon_inode:dmabuf
someca  2424  root    7  ???  ???  ???  ??? anon_inode:dmabuf
someca  2424  root    8  ???  ???  ???  ??? /dev/log/events
someca  2424  root    9  ???  ???  ???  ??? /dev/log/system
someca  2424  root   10  ???  ???  ???  ??? socket:[4957]
someca  2424  root   11  ???  ???  ???  ??? /dev/__properties__ (deleted)
someca  2424  root   12  ???  ???  ???  ??? /dev/mobicore-user
someca  2424  root   13  ???  ???  ???  ??? socket:[4959]
someca  2424  root  mem  ???  b3:03     0  26948 /data/app/lta
someca  2424  root  mem  ???  b3:03  16384  26948 /data/app/lta
someca  2424  root  mem  ???  b3:03  20480  26948 /data/app/lta
```

Actual policy left as an exercise

One real-world policy for this setup, with support for virus checkers, file management, backups etc. has

6 types
852 allow rules

In the exercises you will get, one task is to define a policy for a set-up close to the one above. Simpler, of course...



As the multitude of classes and permissions makes policy writing ”by hand” tedious and error-prone, Google/Android has introduced a macro expansion tool, described next...

Google Policy Macros

An m4. macro set for representing common rule patterns in a a more readable format

tee.te

```
##
# trusted execution environment (tee) daemon
#
type tee, domain;
type tee_exec, exec_type, file_type;
type tee_device, dev_type;
type tee_data_file, file_type, data_file_type;
init_daemon_domain(tee)
allow tee self:capability { dac_override };
allow tee tee_device:chr_file rw_file_perms;
allow tee tee_data_file:dir rw_dir_perms;
allow tee tee_data_file:file create_file_perms;
allow tee self:netlink_socket create_socket_perms;
```

file_contexts

```
/dev/tf_driver u:object_r:tee_device:s0
/system/bin/tf_daemon u:object_r:tee_exec:s0
```

Recursion and relations between policy files

tee.te

```
allow tee tee_data_file : dir rw_dir_perms;
allow tee tee_data_file : file create_file_perms;
```

global_macros

```
define(`rw_file_perms', `{ r_file_perms w_file_perms }')
define(`create_file_perms', `{ create setattr rw_file_perms link_file_perms }')

define(`r_file_perms', `{ getattr open read ioctl lock }')
define(`w_file_perms', `{ open append write }')
define(`link_file_perms', `{ getattr link unlink rename }')
```

access_vectors

```
// contains a meta-definition of classes and their relations
// for validating the macro expansion?
```

te_macros

Contains template rules for some 'special' operations

```
define(`domain_trans', `
# Old domain may exec the file and transition tonew domain.
allow $1 $2:file { getattr open read execute };
allow $1 $3:process transition;
# New domain is entered by executing the file.
allow $3 $2:file { entrypoint open read execute getattr };
# New domain can send SIGCHLD to its caller.
allow $3 $1:process sigchld;
# Enable AT_SECURE, i.e. libc secure mode.
dontaudit $1 $3:process noatsecure;
allow $1 $3:process { siginh rlimitinh };
')
```

```
define(`domain_auto_trans', `
# Allow the necessary permissions.
domain_trans($1,$2,$3)
# Make the transition occur by default.
type_transition $1 $2:process $3;
')
```

Similar macros exit for file type transition (based on e.g. dir), binder use (NEW), and special cases like DRM and debugging

Looking at things

The ”master” SEAndroid branch

https://android.googlesource.com/platform/external/sepolicy/+/master

/sepolicy , /file_contexts ... on most Android 4.4 or 5 phones

Google strongly recommends that the policy files are kept in the root directory of the phone. However alternative locations sometimes apply, especially if the policy is dynamically updatable (http://seandroid.bitbucket.org/PolicyUpdates.html)

Tools for parsing the binary ’sepolicy’

The **apol** GUI (https://github.com/TresysTechnology/setools3/wiki)
seinfo / sesearch tools in setools (Ubuntu) package

E.g. **sesearch –allow** sepolicy gives a nice textual dump of all the allow rules in the provided policy. **seinfo –x –t sepolicy** gives a list of types and parent attributes

SELinux Notebook

http://www.freetechbooks.com/efiles/selinuxnotebook/The_SELinux_Notebook_The_Foundations_3rd_Edition.pdf

Final words (if any)

1) SEAndroid is a ”tweaked” SELinux, with increased functionality especially for middleware (like the VM)

2) Earlier MAC systems in user devices include Symbian capabilities and CentOS SELinux. Android 4.4-> SEAndroid is destined to **become the most widespread and complex MAC** ever deployed on consumer devices.

3) SEAndroid policies are inevitably complex and writing them requires an understanding of both the target environment AND the policy framework simultaneously.

Executive reference

http://events.linuxfoundation.org/images/stories/pdf/lcna_co2012_smalley.pdf