Extending AppArmor to the Userspace

AppArmor Documentation

- AppArmor Core Policy Reference Manual
- Understanding AppArmor Policy
- AppArmor Developer Documentation
  - Kernel Notes
  - Policy Layout and Encoding
  - HFA
  - Policy Compilation
  - Extending AppArmor into userspace (this document)
  - AppArmor APIs and Interfaces
- Tech Docs
  - HFA variables

Table of Contents

Preface

This document is intended for developers who wish to leverage apparmor to improve the security of their application or services using the additional semantic information available in the userspace. This document covers the general concepts while the AppArmor APIs and Interfaces document covers the APIs that the developer will need.

Introduction

The AppArmor kernel module can only mediate application interaction at the kernel object level. The kernel object level is good at controlling low level (file, network) accesses but it is problematic for mediating many higher level (address book, system settings) permission requests because the full semantics are lost when the access request is mapped into kernel syscalls; eg. a request to update a single address book entry is seen by the kernel as a set of requests to send messages to a service or update a file.

If an application has direct access to a resource (eg. file) then mediating at the kernel object level is the best that can be done. However if the resource is accessed via a service that is run as a separate process, that process can mediate access to the resource and it can be extended to query and enforce apparmor policy. In AppArmor a service that has been extended to support apparmor policy mediation in userspace is known as a trusted helper.

For a trusted helper to be useful and secure it must, reside in a separate process from the applications it provides resource mediation for, use a secure form of communication, provide data to querying applications in individual units of mediation, policy must be able to be extended to support new the new type of mediation being done by the helper, and communicate with the apparmor kernel module so that it can enforce current policy.

The service is required to be in a separate process so that its memory and communications can be secured and mediated by apparmor at the kernel level. Mediation within an application can not be trusted (guarenteed to be secure), even if the application code is known to behave correctly and respect the mediation returned. This is because the application process must have access to the data at some level and if the application has a flaw that allows for an exploit, new code could be injected that can access the data without proper mediation.

The communication of the trusted helper must be secured so that none authorized applications can not access the service, and the service can trust the applications requesting permissions from it are who they claim to be. This provides the trusted helper the ability to enforce mediation based on the security context of those communicating with it.

The resource access must be in individual mediation units otherwise there is no

point in providing more detailed mediation than the kernel can provide; if the access is for a single key in a file and the service returns access to the file with all the keys the mediation can not be enforced, as the trusted helper is trusting the application that requested the data to only access the requested key. Similarily the apparmor policy should usually be extended to support the new data unit that the trusted helper is enforcing other wise the policy will not be able to express access rights at a granularity finer that what is already enforced by the kernel; the exception is when the decision does not come from policy but from the user, in which case the trusted helper can use the security context and the users response to enforce mediation without extending the policy language.

What a trusted helper needs to understand about AppArmor

To effectively leverage apparmor in a trusted helper there are several things that need to be understood about apparmor. There are several additional documents (Understanding AppArmor Policy, AppArmor API, ...) that cover these topics in more detail but a quick overview is provided here.

Each trusted helper is unique and has its own set of requirements. The author of the trusted helper should way the requirements and different options when deciding on what to do. This document is only a guide to what is possible and can not cover all possibilities.

AppArmor is a MAC based security system that uses a centrally managed policy that is enforced on processes (applications), on a per process bases. This means individual user applications can have different security applied to them and resources that would be available in standard unix DAC may not be available to an application.

AppArmor applies a security context to every process in the system, and may or may not label system objects. If AppArmor does not explicitly label a system object then an implicit labeling is available within the policy rules. Authors of trusted helpers will need to choose how best to extend the apparmor security information to the trusted helper, whether that is just leveraging the existing information apparmor tracks or extending this information tracking to objects that the trusted helper is responsible for.

AppArmor rules and policy can be very fine grained so it is usually best when querying apparmor to use the subject or object as part of the query instead of just the security context. This is because some apparmor rules may be conditional up certain subject/object properties. And this information is not available from a generic security context.

AppArmor is extensible so that if a trusted helper needs to extend apparmor policy to provide finer grained mediation rules, it is possible to do this within policy, with the trusted helper being responsible for enforcing those rules. Policy

extensions will should go through the standard policy compiler and must conform to the syntax supported by the compiler. The trusted helper we be responsible for understand how to form queries against the compiled form.

It is best if AppArmor policy can be managed centrally so there is only one place to look to discover restrictions that apparmor is enforcing. There are exceptions to this rule but a trusted helper author is encouraged to follow this guideline if at all possible.

The manipulation and loading of AppArmor policy is a privileged operation that may not even be available to all administrators/root processes. Trusted helpers that wish to change and reload policy dynamically will need to have sufficient privileges to do so.

AppArmor only considers a trusted helper privileged within its domain, and as such AppArmor may be enforcing policy restrictions on the trusted helper even as it relies on the trusted helper to extend its mediation to new types. For example the dbus daemon can be a trusted helper and it is trusted with respect to mediation of messages on the dbus message bus, but the daemon it self may still be restricted from accessing certain files, network communications or even from communicating with certain applications.

For discussion purposes we will break trusted helpers into two broad categories trusted backend (or services) and trusted intermediaries. While both catagories have much in common trusted intermediaries introduce an extra layer and often have a different purpose than trusted backends so their design requirements are different.

Understanding apparmor policy
Label on tasks
Labeling of sockets communication.
Objects – helper has to label or track its own objects

peers
appamor api to query perms
extending policy
storing decisions
- local acl
- updated policy

policy load is privileged

-try to keep policy in one place instead of scattering it in different locations (polkit, secmark, bad bad bad)

- label on task
  - pid race/life time
- label on socket

- querying policy
- extending policy
- storing decisions
  - local vs extending and reloading profile

Each trusted helper is unique in its requirements and how it extends apparmor, but there are a few

Communicating with the
???? Getting the security context
???? Using just the security context and not using apparmor policy to make the decision
???? storing in application data vs in apparmor policy/or centralized db
- apparmor policy is searchable
- centralizing all policy decisions
  - known where to search (even in a per user, per app type situation)
   - easy to cleanout if profile is removed (don't leave bread crumbs)
   - extensions can be stored in dir or other place that is not part of main profile
- extension to kernel mediated features need to go into policy (add file access to sandbox)
vs. in backend
- in backend app/db/location, could be more convient easier to implement
- have more context when selecting what to remove privilige from
- may need encrypt that is not possible in policy
- need away to clean out when client app is removed so it doesn't leave bread crumbs
vs. in app
- hard to find search
- app can't have direct access to information
- app can clean out the information easily if stored in place only belonging to application
- data needs to be encrypted by backend private key, and then sent to application for storage, application then send back encryped data, and backend decrypts
  - this works only for a limited set of data like geoip
- doesn't work for online accounts?

Delegation vs. extending the profile vs. extending the task

Trusted Backends (2 Layer model)

In the trusted backend model the “Backend” is a process that provides a service to the Application. The service could be anything from mounting new disks, to looking up a contact in an address book. The important point is the Application does not have direct access to the underlying function or data that the service provides.

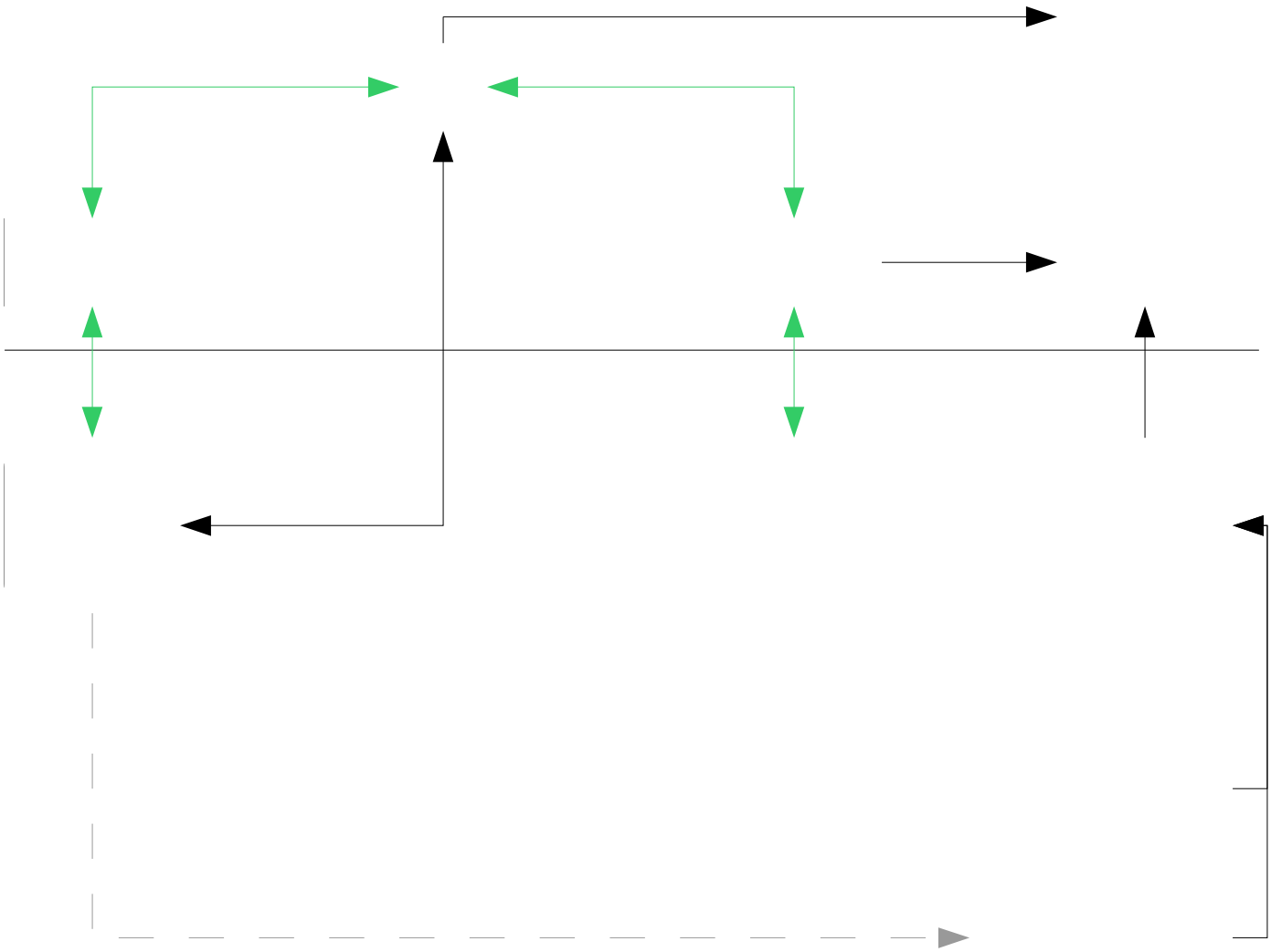

Figure 1: Application and Backend communicating via a kernel mediated channel

Shows how the relationships in a simple (direct) trusted helper model. In this model the application communicates directly with the trusted helper, which in this case is known as a trusted backend. The kernel uses the apparmor policy to provide a security context for both the application and the backend. The trusted backend can then query the apparmor policy to determine the permissions for the applications request, and then it can reply back to the application.

- can only extend rule set as direct communication is already being mediated
- apparmor is controlling the messages between the applications

- get security context of Application.
- kernel provides security context
- trust the security context

- make decisions based off of it (query via api)

- save acls locally
- extend policy

An example of this type of Trusted helper would be ????.

bar

Trusted Intermediaries (3 Layer model)

While most services fit into the trusted backend where the application directly connects and talks to the service, there are several cases where an extra application/service, the intermediary, is involved. The intermediary acts as a third party in an exchange, message relay, bus sending the message on to the target, after having done its work, whether its just looking up the targets address or reformating the message. An example of a trusted intermediary is the D-Bus message bus when it lives in userspace. An application sends messages (via the kernel) to the dbus daemon which then relays the message on to the correct backend. Similary any replies from the backend are sent to the D-Bus daemon before being relayed on to the Application.

The apparmor kernel module mediates communication between the application and the intermediary, and between the intermediary and the backend service. However it can not mediate the contents of the message nor which applications or backends the intermediary chooses to relay messages on to. As shown in Figure 2, once the intermediary receives a message from application A it could choose to send it on to application B, or C or D, as unextended apparmor policy can only control the direct communication between processes and the intermediary is allowed to communicate with each of the processes A, B, C, and D.

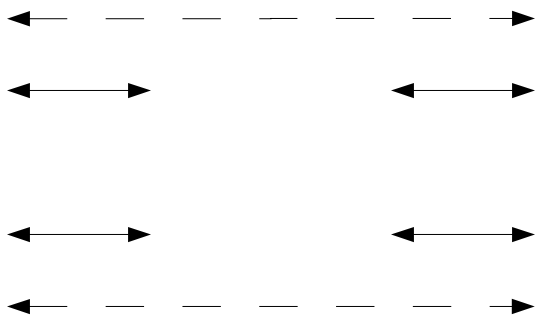

Figure 2: Process communicating via Trusted Intermediary

When an intermediary is involved a few conditions must be met for it to be considered a 3 layer model. The intermediary must be “trusted”, otherwise there is no value in extending apparmor support to it as no additional enforcement can be done and the intermediary just becomes a variation on the two layer model. The application also must be in a different profile (domain) from the intermediary and the backend, otherwise apparmor can not be guarenteed to enforce restricting communications between the processes.

3 Layer Trusted Helper - Trusted intermediary only

To improve mediation when communication goes through a userspace intermediary, the intermediary can become a trusted helper, extending apparmor policy and enforcing the extended rule set. The intermediary becomes responsible for determining the type and extent of enforcement being done. AppArmor “trusts” it only within the domain its enforcing, that is apparmor still enforces the mediation that is done between the processes as if the intermediary isn't trusted. Depending on how the intermediary is extended to support apparmor, it can expand the types and granularity of mediation being done or it can just enforce policy as if the application is directly communicating with the backend service.

Trusted intermediary without extending policy

The simplest trusted intermediary enforces mediation as if the application and backend are directly communicating with each other. The intermediary performs the permission cross check between the application and backend security contexts just as the kernel would for direct communication.

AppArmor policy is still required to allow this communication between the various processes. Instead of using an extension the policy is written so that each profile involved can communicate with the the other processes involved as shown in Figure 3.

The specifics of whether generic ipc rules would be sufficient are up to the Intermediary extension that enforces the apparmor policy restrictions. If for example both the Application and the Backend are communication with the intermediary over unix domain sockets. The Intermediary might make the permission query as if the Application is talking to the Backend directly over a unix domain socket. But if the Application and the Backend are using different types of communication channels then it might be better if the intermediary used a more generic permission query.

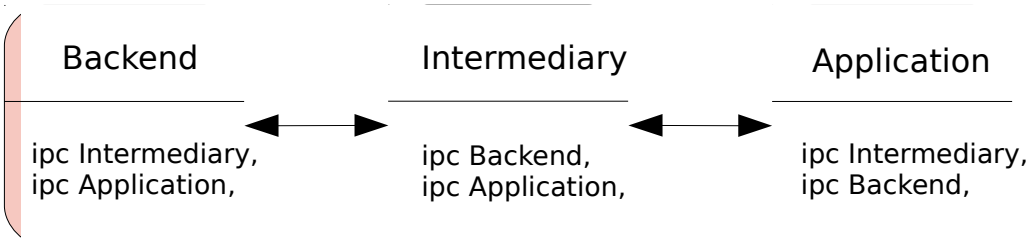

Figure 3: Intermediary enforcing communication restrictions without extending policy

Trusted intermediaries with extended policy

For a finer grained control of the services the Intermediary provides it is possible to extend policy in a manner similar to that used for trusted Backends.

dbus
policy kit

- Backend fine grain not part of the Intermediary
- dbus example

Extending the apparmor policy for Intermediaries is similar to extending

- can be done by intermediary storing acls instead of updating policy, just as in 2 layer

Simply extending the intermediary to enforce existing apparmor policy often does not provide as fine grained mediation as could be possible as the existing apparmor policy can not express the full semantics of what the intermediary is doing. In these cases apparmor policy can be extended (see ???) to add a new class and the intermediary then enforces this policy extension.

- doesn't require app communicating with backend over socket rule
- extended type
- abstracting new class to include the class and the socket communication

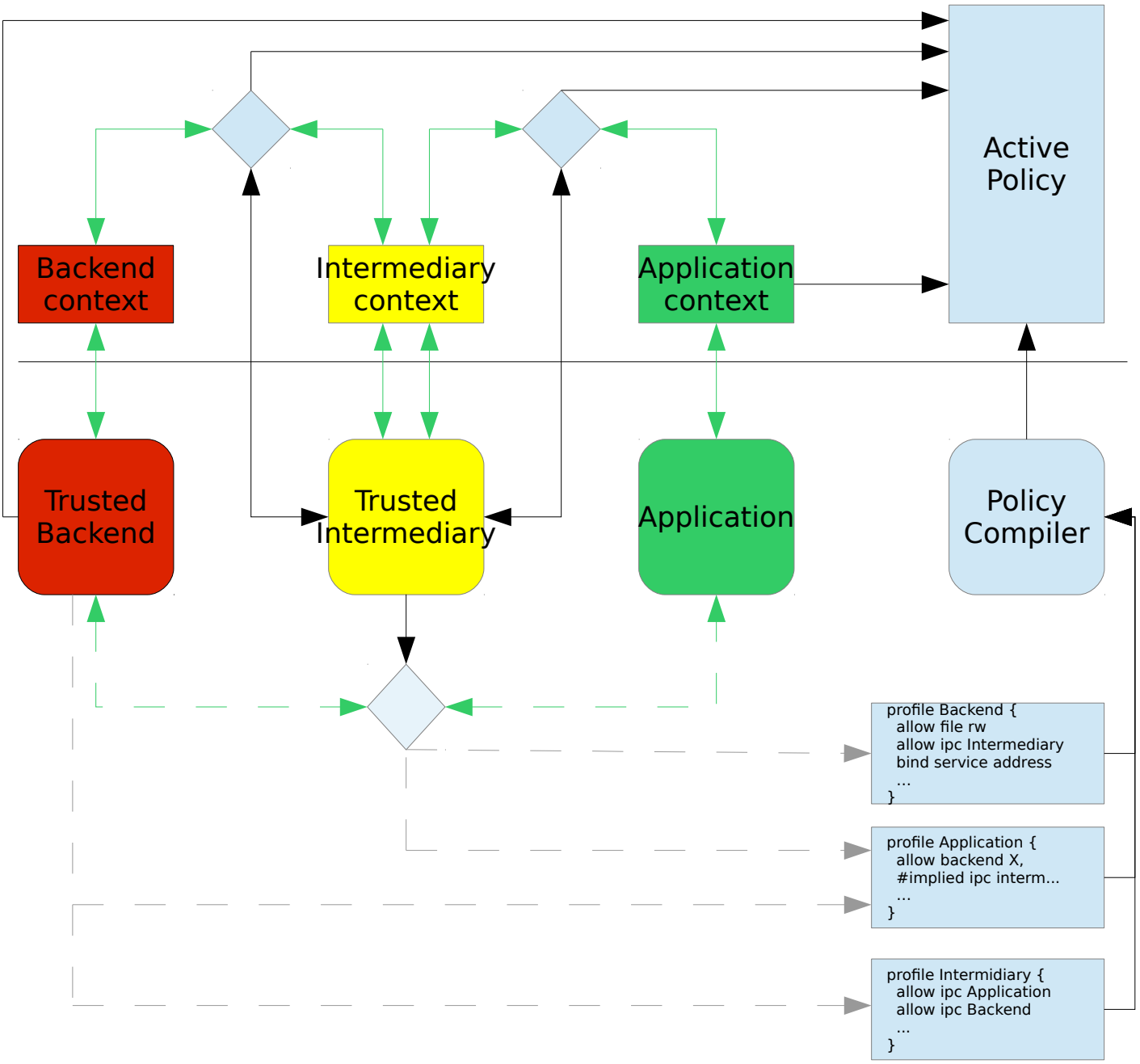

Illustration 1: 3 Layer model: Trusted Intermediary

bar

3 Layer Trusted Model - Trusted Intermediary and Trusted Backend

fooo

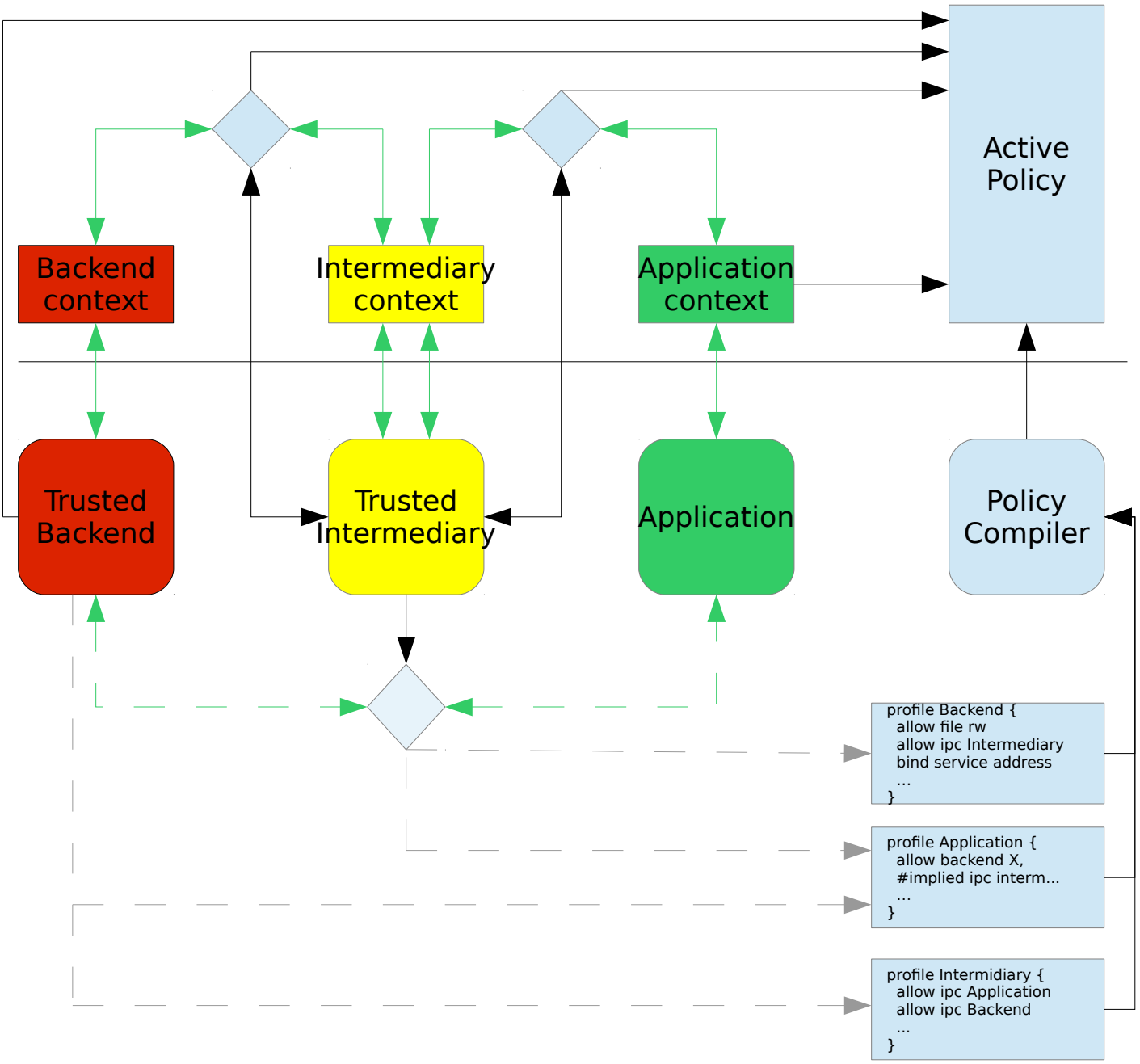

Illustration 2: 3 Layer Model: with Trusted Intermediary and Backend

bar

policy kit example – extend implicit perms or choose when to prompt
have helper, policy daemon, service and app in different profiles. Helper uses apps security context, passes that to policy daemon, daemon uses that to check perms instead of uid auth. Service checks apps sec context passes it to auth daemon to verify permissions ...

Extending Policy to support Trusted Helpers

- how to update policy blah, blah, blah

- syntax
- underneath
  - type, ordering of query

- policy enforced by trusted app does not have to be done by apparmor
  - however then some communication enforcement can not be enforced
  - policy is split between different systems
    - better to keep policy in unified place
    - also this allows policy to leverage apparmor security labeling, so not just can X access Y, X can access Y if in state Z

- trust app needs to track and deal with handle update/mediation.
- provide api similar to what kernel uses to build labels and check them
- api can cache perms or send the request directly to the kernel
- don't have to necessarily directly pull the matching engine from the kernel, could just cache decisions, dependent on needs.

Need
- label_insert(X) – look up or insert new label
- label_insert_merge(X, Y) – lookup/insert new label that is merge of two

oh, ouch ordering problem for lookup – what to do when replace renames the profile?
- regular replacement, removal and even revocation are not a problematic
- for renaming replacement, have to build a new sorted label

Eg Multiple settings stored in a single db file

Mechanism, not policy

- backend
- intermediary
- intermediary + backend
This loss of information means that mediation of thes

trusted helpers and namespaces
- trusted helpers don't see or know they are in a different namespace
- they treat all their requests as if every request is in the same namespace/profile. eg. Even if the helper and app are in the same username space with different profiles, and their system namespace profiles are different, the helper only makes it decision based on the usernamespace profile (it can't see the system)
- if the trusted helper is in a system namespace it can see the request is coming from something from from a usernamespace and take that information into account
instead of

  ipc socket,
  dbus foo,
  allow backend X
we just have
  allow backend X
allow gsetting foo write,
- dbus interface=gsettings method=set field=foo,
- @{dbus_session} rw,

or

- gsettings field=foo w,
- dbus interface=gsettings method=set field=foo,
- @{dbus_session} rw,

- is there a case where we system namespace wants to ignore the usernamespace labeling?
  - almost certainly, infact probably most of the time.
  - in fact unless other wise specified all subnamespaces should probably be assumed to be allowed, that is for most rules, its only the labeling within the namespace that counts

- trusted apps and trusted intermediaries
- policy extensions that are at the intermediary level not the system object level
  - intermediary rules can imply other rules, the set of rules that are implied can be abstracted
    - this means policy isn't hard coding the how (separate policy and mechanism)
    - this makes for smaller policy
- policy needs to be able to be flexibly extended

Hrmm, stack needs to be a different permission than change_profile other wise, when setting up a confined parent.

Hrmm should domain transitions only affect the bottom layer of the stack?
Or perhaps those in the same namespace.

From a system, user profile pov. You want each applying their profiles.
- Unless you are only going to treat the user in a broad stroke. Ie stick the user in a specific profile and not change it based on what the user runs.
  - What of suid programs then?

From a container/chroot pov the transition does not change what is being done.
- this can be done with a custom profile that traps domain transitions (ix)

- we would need ix for change_hat, change_profile, stack?

Network Iptables as an example of a trusted helper