Understanding AppArmor Policy

Table of Contents

AppArmor Documentation

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

Preface

This document is intended for policy authors who want to have a better understanding of AppArmor policy. It is a mid- level description of the concepts in AppArmor policy as enforced on a system. While it uses example rules from the AppArmor reference language[1] it does not describe the language in any detail and alternate languages could be created to express policy. Similarily while it may mention some of the implementation details of the enforcement engine in passing, such mentions are intended as examples only and actual detail should be obtained from developer documents[2] and source code.

1 See the AppArmor Core Policy Reference Manual
2 See the AppArmor Developer Manuals

Overview (TLDR version)

- AppArmor is a hybrid access control system that provides enforcement via rules being enforced on a subject as it accesses objects
  - rules are based on access path, object properties, and requested permissions.
  - apparmor is not based on type enforcement, though the label can be used similar to a type
    - the apparmor label does not make a distinction between type and domain (extended type on subject)
- Subject (a task/process or a proxy for a task/process)
  - subjects are labeled
  - the labeling on a subject determines the set of rules being enforced
- Proxy (a task/process or object acting on the behalf of another Subject using that subjects security context.
- Object (files, records, messages, …)
  - are labeled either explicitly, or implicitly
  -
- A label is a unique name
  - an explicit label is a label directly stored on an object (eg. inode)
    - identies it as an exception from the implicit labeling
  - an implicit label is the access label for an object derived from the set of access rules being enforced
- A profile is a label with rules and mode
  - the profile mode is just a modifier of the rules in the profile (eg complain or enforce mode).
  - rules are applied from the subject (task/process or proxy for a task/processes))
  - rules that don't specify a label don't apply to explicitly labeled objects
  - rules can specify a label to determine permissions when an object has that label
- All labels are profiles, however
  - the word profile is used to refer to labels that have had a set of rules/modes explicitly defined
  - A label that has not been explicitly defined as a profile is treated as being in the unconfined, noexec mode
  - Two profiles are defined by default for every namespace
    - the unconfined profile – which has a mode of unconfined and is immutable.
    - the default profile – which has a mode of unconfined and can be replaced or removed.

  jdstrand> what about the default profile?
- labels can be simple or compound
  - simple == a single name
    - profile

        - label
            - profilewithout explicitly defined rules, these can be reloaded to be a profile
            - a labels mode is unconfined, noexec
            - a label attached to a subject enforces its mode of unconfined, noexec
            - if used
        - a compound label is a set of multiple profiles/labels
        - name is the profile/label names joined with //&
        - rule permissions are the intersection of all profiles/labels
-
-
- When two subjects communicate the permissions the two access are cross checked. That is “can A talk to B?” and “can B talk to A?”.
    - neither A or B dominates, the permissions granted are the intersection of what is allowed
- Profiles and labels exist within a Policy namespace (jdstrand> is the namespace name part of the label?)
    - Policy namespaces have a unique name
    - Are hierarchical
    - the namespace name is prepended to the label name when viewed from outside the namespace (from a parent namespace)
    - within a namespace the namespace name is not visible and is not part of the label name
    -

jj> this is too long we need to trim it. This is ideally a quick summary and some one who needs more can look at the references

Where to put this

Security goals
- reduce attack surface area
  - remove interfaces/syscalls that aren't in use
- identify unexpected behavior
  - further granularity file name/permission level of interfaces that are in use
  - patterns of accesses (ie read then write not supported, write once – maybe supported with vars)
- reduce shared mutable state
 - allow finer grained security based labeling of resources (not restricted to user id)
- reduce use of identity for authority
 - instead of asking for roots permission to do something (prompting for password), ask if has authority to do X, and allow delegating that authority.

- Not a static MAC system but a dynamic MAC system
- use same abstractions at all levels
- decompose, identity (security id) and authority (ability to do this). Capability is the token carried to indicate whether has the authority. Delegated tokens carry the information about what authority is granted.

Make apparmor less of an ambient authority system, shoot for Priciple of Least Authority

sparse capabilities – long unguessable keys giving access to something. Eg google random urls to access an file, blog post etc.
object capabilities – token to access object.

Persistent storage of delegation vs. temporal storage of delegation.
- store in authority rule in the profile (maybe special section) – persistent
- store on file handle
- there is another level of temporal, between permanent and object only, at the session level. Possible to store a set of rules at this level that then get cleared.

Static ambient security – apparmor is not
selinux provides an easy way to analyse information flow from the global perspective.
To do this analysis in apparmor you need to add in the potential dynamic pieces of information
So it comes down to which is more important global information flow or generic containment.
Note: that both can do the same thing its just a matter of ease / management / analysis (ie where they focus their strength).

Hrmm labeling to handle authority, so how do we deal with identy vs label questions vs delegated authority in apparmor

that is – simple authority check (is it labeled X), this can also be taken as the "Identity" ie who is on the other end. How ever we could in apparmor ask instead does X has the authority

authority is carried by the set of rules, + the delegation info

What apparmor isn't – static ambient authority system that makes it easy to do static analyse on information flow. The implicit labeling allows for a more dynamic system that requires mounts and other information to be part of the

analysis, but arguably helps in setting up a more dynamic system. So its focus is not on controlling who can leak your info.

For extending apparmor to the userspace discussion

Authority vs. Identify
When extending apparmor with a trusted helper, a decision has to be made about how the trusted helper will interact with the apparmor policy and kernel module. At a broadest sense this is decision between authority and identity.

The trusted helper can choose to just leverage the security labeling (Identity) provide by apparmor when making its decisions, or it can use/extended apparmor policy and query the policy when a decision is to be made.

- Using identity directly, dealing with delegation

Using identity through policy
- label rule of a given type
- query based on that type
Extending a new type and permissions.
- can use existing type/permission set
- define a new set
- discuss type splitting

To often applications have been written using Identity as the determinant for permission
- if (uid=root) does not allow distinguishing between different apps run by root
- generic permission prompt for user vs application X wants to access

Use of the apparmor security label as identity

- Indentity asking if the other end is I
- vs Authority asking if the other end can do A

The security label can be used as either

If there is only 1 candidate for that can have the permission or the distinction between who at the other end has that permission is not important, then the security label can be used directly to handle the permission.
- just query label, is it X grant permission.
- In the case that the cadidate is extended to more than 1 in the future. Keep the label, have the different candidates set a label on the socket resource, so that communication appears to come from X
** this fails if there is no object/socket proxy (as might be the case in dbus or some other systems, might be able to label the socket that communicates with dbus so we might get away this)
** does not work if we would like to identify peer different from permission. Ie in

a requester we would like to say firefox would like to have access to X, not that label Y (which firefox has access to) would like access to something.

In cases where a distinction between multiple peers is important use an extended permission instead of a label.

Extended permission vs. label

One of the choices that needs to be made for a trusted helper is what is how the permission accesses that the helper is going to check are encoded in policy. ???

so permission needed if different subject need to access object with label X at the same time and have different permissions.

there is no clear distinction of when a permission should exist vs using a type/label
all permissions could be represented by type splitting and a single binary permission (true/false)

eg. permission for Fred to read/write a file of type foo could be represented by the tuple

Subject	Object	Permission
Fred	foo	rw

However type splitting could also be used to come up with 3 different types to represent r, w, rw (foo.r, foo.w, foo.rw). The tuple then becomes

Fred	foo.rw	true

However this requires more label splitting than is needed, nor does it have the benefit of carrying common semantic information, unless the label/type format is specified to carry it. That is read and write are common across multiple types and their combination (rw) is also common and specifying them as permissions allows for policy to be smaller.

Oh and splitting out the permission allows for different subjects to have different access rights to a label.

Having trusted helper store “intent/permission” based using identity vs. doing it in apparmor policy

Having trusted helper store “intent”
Advantages
- only requires being able to query security label

  - possible to take advantage of extended instantiation
- localizes it per user (can be finer if it leverages instantiation)
- is doable without apparmor extension to policy
- does not require helper to be able to load policy - is doable without apparmor extension around policy loading

Disadvantages
- splits policy so that what is allowed can not be introspected in one place
  - depending on how intent storage is done may require additional tools/per trusted helper plugins be created to introspect policy
- can not preseed any perm as part of policy, could be done via preseeding each helpers “db”
- can not take advantage of fine grained policy controls that are built in, like per rule audit controls, prompt hints etc.
  - can do course mode based stuff
- ties decision to a policy ID instead of a permission
  - this is some what neutral in that the local storage can come up with its own set of permissions for the ID, but the permission is not policy based it is only known about in the helper code, again this will require special tooling to introspect
- won't be able to integrate delegation, at least not in the same way as the rest of policy.

Neutral
- speed (this could go either way)
  could be faster to query and determine perms locally, but likely needs a trip to the kernel to get the security id anyways
  likely won't have precomuted matching engine to query permissions against, so could be slower

Storing “Intent” in Policy with extended types and permissions.
- keeps policy in one place (introspection)
- single tool for revocation
- can leverage fine grained policy features
- can levage policy based delegation

Requires
- way to extend policy with new types/perms without updating the compiler
- Helper to be able to load policy
  - needs extensions to control what policy/bits of policy the helper can load
  - needs a way to do partial policy compiles and loads so that “helper” policy modifications don't affect other policy.
    - this means helper policy is “referenced” but loaded separately, it can be, updated, compiled, shared, and replaced separately from the rest of policy
  - needs a location to store “intents” without modifying base policy (could be a dir)
  - needs a way to store “intents” per user or finer

- could be different conditional blocks or files/directories
  for user based trusted helpers likely different directories
  for policy based trusted helpers a dir for each policy, which could be divided further by dirs for users, etc.

Instantiations
- are different versions of a profile
  - eg same profile but different user, and the profile has rules conditional on the user
- different instantiation will result in different permissions so they can not be treated the same for short circuiting purposes
  - replacements need to create replacement instantiations
  - need to identify which run time variables a profile is dependent on, and create instantiations if this any variable changes.
  - owner where uid == ouid is special and doesn't need to cause an instantiation but other uses of uid (not ouid) do need to cause an instantiation.
  - run time variables that are supposed to affect every task using the variable do not affect instantiations, only per task/user variables need instantiations

Introduction

AppArmor is a flexible hybrid (jdstrand> hybrid-enforcement or HE (maybe?) maybe? security system that can be used to provide application sandboxes, resource usage limits, MAC (mandatory access control) policy, and even an MLS style policy. It consists of a number of userspace tools and libraries and a kernel based enforcement engine.

Jdstrand> maybe add background on TE and DTE. Eg: “Other existing security systems exist, such as Type Enforcement (TE) which defines types for each object and the applies a global policy to these defined types and the more flexible Domain Type Enforcement (DTE) which defines types for each object and different policy domains that apply to these defined types.

jj> Not here. Maybe add a reference to them, or a reference and an appendix, I'd rather keep this focused as a mid level view for people interested in authoring policy, than deep dive into differences.

Where TE and DTE can be thought of as taking an object-centric view, AppArmor takes a subject-centric view and uses Hybrid Enforcement (HE) to focus on the application. In the HE model, the concepts of domain rules and typed objects are combined for each subject such that policy can more flexibly describe interactions between subjects as well as accesses to objects.” - reword

jj> hrmmm not quite
TE and DTE (DTE is really just a variant of TE. Take a tuple view. Rules are based on
  subject object permissions
TE sets the rules up as a global set of rules and some TE systems (selinux) don't distinguish between subject domain types and object types. They are the same. DTE explicitly splits them into a domain type that can exist on subjects as it contains the rules and object types.

Apparmor takes the view that the “types” are the same (similar to TE), but allows/has (the implicit mode can be seen as defining rules) rules on all types, just that the rules are not enforced when the “type” is on an object. I quote “type” because we are not exactly doing a type but a label that could be a the equiv of a type if partitioned correctly (or on a subject) but is often not a type because we don't force the rule partioning on the object labels, and handle it lazily. This allows us to dynamically change what type is possible with delegation.

We do need to explain this but it will need to be more gradual through out the document and in the appendix

The level of security provided is dependent on the policy, which is the set of rules that define what permissions are allowed. Policy is largely written from the point of view of an application (subject), but mechanisms are provided for specifying

some policy from a global point of view.

The AppArmor system is flexible in its enforcement requirements and providesing the ability to define restriction based on both access paths and labeling.

Policy

AppArmor policy is the full set of rules that are enforced on a system. It is defined by specifying the behavior of several interacting components (labels, profiles, namespaces, variables).

The AppArmor project provides a reference language[1] to define system policy and a compiler to build and load it into the kernel for enforcement.

Labels

Labels are the base unit of mediation in AppArmor even though they are the direct use of a label is a special situation in reference policy (jdstrand> reword). They are the mediation identifier attached to an object, which is used in determining what permissions a subject (task) has when accessing the object. In AppArmor, labels are applied to all objects but for most objects the label is implied and not permanently stored and, they are lazily generated as needed at run time from the set of rules being enforced. Labels that are lazily generated from rules are known as implicit labels while labels that are stored on an object (eg, in the inode) are known as explicit labels.

All labels share a few basic properties, a name which can be used to specify interactions with the label, a modeed), and, if the label is a profile, a set of rules. The label name is must conform to a defined format and of what is allowed and is unique within a policy namespace.

The label name is unique only within its policy namespace. Labels from different namespace with the same name are unique when the policy namespace name is prepended, this is known as a fully qualified name (fq name).

Name restrictions.
Namespace names
simple names compound labels/names
fqnames

A label can be anything from an individual unit (ie, an explicit label or a profile), a representation of a set of units (ie an implicit label or a profile stack, or a simple label (one that is just a name).

Profiles (jdstrand> is heading 5 correct here? jj>Maybe the intent was to be a sub of Labels as they are just labels with explicit rules, but maybe it would be better if it was its own section instead of a subsection)

Profiles are labels with rules, and are how task confinement is specified in AppArmor. Each profile has a unique name, some control flags, and a set of rules defining access rights, resources, and privileged operations that an application (subject) will be restricted to when the profile is applied. If an application (subject) is labeled with a profile and the profile is not set to the unconfined mode it is said to be confined by the profile. (jdstrand> what about the mode?)

Profile names

In addition to being a label name, the profile name may also specify the profile's attachment specification, and as such has fewer restrictions placed on it than other types of label names. More specifically if the profile name is rooted (begins with the / character) it will provide the profile's attachment specification if the profile does not specify an alternate attachment specification rule. The attachment specification of a profile when combined with execute rules is used to determine when a profile is attached to a task. (jdstrand> should we mention path-based attachment more clearly here? No below in the attachment specification section)

Hierarchical ???

Attachment specification

Foo

Profile Alternates (Conditional Profiles) Instantiation?

Profile alternates provide a convenient way to specify multiple versions of a profile which are used under different conditions. That is to say the same profile can be specified multiple times each with a different set of conditions, and the appropriate variant of the profile is chosen at runtime for rule enforcement.

```
profile example  uid=fred { }
profile example  uid=george { }
```

Profile alternates are not actually separate profiles.Each alternate shares the same unique label name and exists together with the other alternates. They are in fact just an anotherlternate way of specifying a single profile with different conditional rule blocks, except that the rule blocks can be loaded and replaced independently from each other.

The ability to add, replace, and remove alternates independently of replacing a single profile is provided for a few reasons. ?????

??? how to specify removal of an alternate

Profile modes

Profile modes are used to define how a profile behaves and can be viewed as a way to modify all the rules in the profile. The different profile modes are:
enforce (default mode, unspecified access is denied and logged)
kill (unspecified access is denied, logged and the application is terminated)
audit (all accesses are logged, unspecified access is denied and logged)
complain (unspecified access is allowed, but logged)
unconfined (all access is allowed)
noexec (Execute and profile transition permissions are denied but all other access are treated as unconfined.)

why label needs rule context – not unique to lookup perms, unless it is a name only label
Label and modes
Simple labels default to the noexec mode.

Reporting of Modes
One interfaces that report the mode of a label/profile the mode may be reported as either it name or the first letter of its name. Compound labels always report a set of modes using the first letter of the modes of the labels that the compound label is made up of. The order of the modes reported are the same as those of the simple labels in the compound label. For example A//&B (EU) means A is in the enforcing mode and B is in the unconfined mode.

Attachment specification stuff ...
 The profile name may be used to provide the attachment specification. This is optional and used to determine when the profile attaches to theagainst an executables name. The profile name and attachment specification are often the same but there is no requirement that they have any relation.

Profiles names have several limitations, hierarchical (jdstrand> all these could be more clear)
: - namespace
& - stacked
+ - delegation
# - explicit label – property of object/handle not the label or profile

// - path separator
other leading symbols are reserved

/ not allowed as a trailing character

jdstreand> we were talking about profiles, but now talking about labels. This should probably move?
label/profile states - unconfined

domain transitions
exec - px
noexec - change_hat

whether object or handle is labeled depends on the type of object and its lifetime

not really the document to talk about syntax and sibling transitions, stacking transitions, multiple targets etc.

for conditional label rules - must be able to communicate with every entry (profile or label) in the label, not just a subset, if passes test for every one then it updates the label with its label, label is updated via rcu
use opportunistic locking

when to use labels, vs. implicit labeling, overlay to provide writable location

need to work on run time aliases

Implicit labeling consistency in move/rename
- labeling needs to be checked/reevaluated every move, rename, link, mount (move), unmount, profile manipulation. Basically check if any profile is invalid if so don't use it/remove it

Delegation and no-newprivs. Delegation from outside allowed? Need to track no newprivs to where it happened in stack.

Implementing implicit labeling (jdstreand> not this document? Hrmm a little bit)

global policy – single dfa: state provides label
- problematic in dynamic system, changes in namespaces etc (have to recompile dfa etc)

- Solutions, multiple dfas and composition, options
  - list of profiles, can't check which path
  - list of profile states – could have multiple states for a give profile, can reconstruct which state, may represent multiple profiles, can not be shared across types (different states), work for object label if you already have task profiles, for tasks need reverse mapping to find profiles from labels so you can ask new questions
  - list of accept states – per profile accept info

Storing on object vs Handle
Object
- need to update object with policy changes
- need to update object for all policy on each mv/rename etc.

Handle
- need to update as handle is passed around
- Fall back to revalidation
- can fail for disconnected paths

Introduction

Profile
- attachments
- domain transitions

Label
- task labeling
- object labeling

Label modes

Policy Namespaces

A policy namespace contains the set of labels including the set of profiles that are visible and available to attach to a given task. They allow separating labels into separate functional groups that can be managed independently, allowing for different policy to be applied to different groups within the system.

The labels with in a namespace are independent from the labels in another namespace. Thus while a label name must be unique within a policy namespace, each namespace can have its own unique version of a label that is tailored for the task it is confining. Just as importantly a namespace does not need to define labels (or profiles) for the same set of executables as other namespaces. The only label that a policy namespace is guaranteed to have is the unconfined profile which is unique for each namespace.

Policy namespaces are hierarchical, with a default root namespace defined on system setup. New namespaces are created as children of the root namespace, and those namespaces can define their own children.

Within the hierarchy the parent namespace has control over its children, that is it can see, define and manipulate the policy of its children, but its children cannot see[3] or manipulate the parent namespace. From the point of view of any child namespace it is the root namespace of the system.

When a task manipulates policy it does so against the current namespace, which is the namespace at the “top” of the label stack. If the current namespace is changed then said task will not be able to manipulate the original namespace, all manipulation will be applied to the new namespace.

Tasks inherit the current namespace from their parent. They can switch to a new policy namespace if a the namespace has been defined and the tasks has sufficient permissions to perform the switch. After the switch is made the new namespace is what will be inherited by its children as their current namespace.

It is important to note that a switch to a new namespace can not be forced on a task (that is only a task can switch its own namespace), and that once the switch has been made the task can not switch back to the original namespace; it can not even see the original namespace as the new namespace will be a child of the original namespace. A change in policy namespace also means a change in profile confinement which may be a profile of the same name or could be a different profile entirely.

User policy namespaces

User policy namespaces are a special namespace that allow a user to define policy for their own tasks. They allow for users to be able to define and manipulate MAC policy for their own tasks independent of what system policy

3 The visibility of the parent namespace can leak into the children via audit logs, etc. If they don't have system namespacing applied

defines.

User policy namespaces can have limits enforced on them by the system, and they may not be available on a given system.

Namespace names

begin with colon
end with colon
// after is optional

hierarchical // separating children

Namespaces and labels names

Namespace names
When an object has multiple labels that are from different namespaces the labels are stored in separate sets, one for each namespace.

When multiple labels are stored on an object a compound label is used to represent that combination of labels.

Visibility????

Policy namespaces interaction with system namespaces

AppArmor policy namespaces are independent of the other linux system namespace, and can be used in conjunction with or independently of the other namespace types. It is important to note that the transition is done using a different api, and at no time does the new namespace and old namespace contain the same set of profiles, that is to say even if they are exactly the same they are considered as different profiles for the purposes of mediation (this is important for labeling and revalidation).

Explicit Labeling

Foo, can be simple or compound, per NS, profiles must agree on each label being set

Stacking - dynamic composition of profiles

AppArmor allows for a task to be confined by multiple profiles, known as a stack, allowing for a limited dynamic composition. For mediation purposes a profile stack is a compound label, the only differences are that a task can explicitly request to modify its labeling and that the stack is responsible for tracking the tasks current namespace. As with compound labels there is a single set of profile for each namespace involved in the confinement.

Mediation of permission requests is based off of the intersection of all profiles, that is a permission request is only granted if it is allowed by all profiles that are labeling the task.

For domain transitions if the transition is allowed, then each profile determines its own transition independently and the results are combined to achieve the new label (stack). If multiple profiles make a transition to the same profile during the domain transition then size of the stack will be reduced as the labeling is a set of profiles.

It is not possible for the stack to have a profile used multiple times within the stack. If this occurs the entries within the stack will merge as all subsequent domain transitions and mediation requests will be the same. Profiles with the same name but in different namespaces are not considered to be the same, so a task could well be confined by a stack of two or more unconfined profiles as long as they where from different namespaces.

Once a task has created a stack there is no way for it to request that it be unstacked. Policy transitions may result in a stack being merged down, but this is an unlikely situation and is controlled by the policy author.

The current namespace will always be set to the last namespace added to the stack. This is used to determine which policy the tasks confined by the stack can manage. The current namespace can only be changed by new stacking calls or transition from the current namespace. Transitions from other namespaces in the stack do not change the current stack.

If there are other namespaces referenced within the stack, the tasks within can not manipulate or manage them but other tasks on the system with different confinement might be able to. That is not to say that other tasks can direcly change a tasks stack but that they may be able to load or unload policy to a namespace that the task can not because that namespace is not considered its current namespace.

When a task requests access to an unlabeled object the stack label is used as the basis for the initial implicit labeling. The exact labeling will depend on the policy and how it is loaded, as it may provide additional labeling information that can be extended beyond what the stack provides.

System Namespaces

??? Determining permissions, access paths, …
aliases if policy ns is shared,
should switch to separate policy ns, or use explicit labeling

Conditionals

AppArmor defines three types of conditionals that can be used from the kernel:

kernel object, policy, and match; that can be used to dynamically determine permissions.

The kernel object conditional is based on values that are stored as part of the mediated object and is used during permission evaluation. Object uid and file system type are examples of object conditionals.

Policy conditionals are conditionals that can be toggled at run time to change what permissions a match can select. They can be used to support a wide variety of different policy model such as time based access restrictions.

Match conditionals are variables that have been embedded into the path matching engine and can be used to change which access paths result in a valid match. Match conditionals are an extension of policy conditionals, and rely on the matching engine supporting variables.

Permission Check

When a task wishes to access an object a permission check is done to determine whether the task has sufficient permissions under its current confinement. The permission check is done in multiple steps and can succeed or fail at any point.

The permission check is done for each namespace in the tasks current labeling.

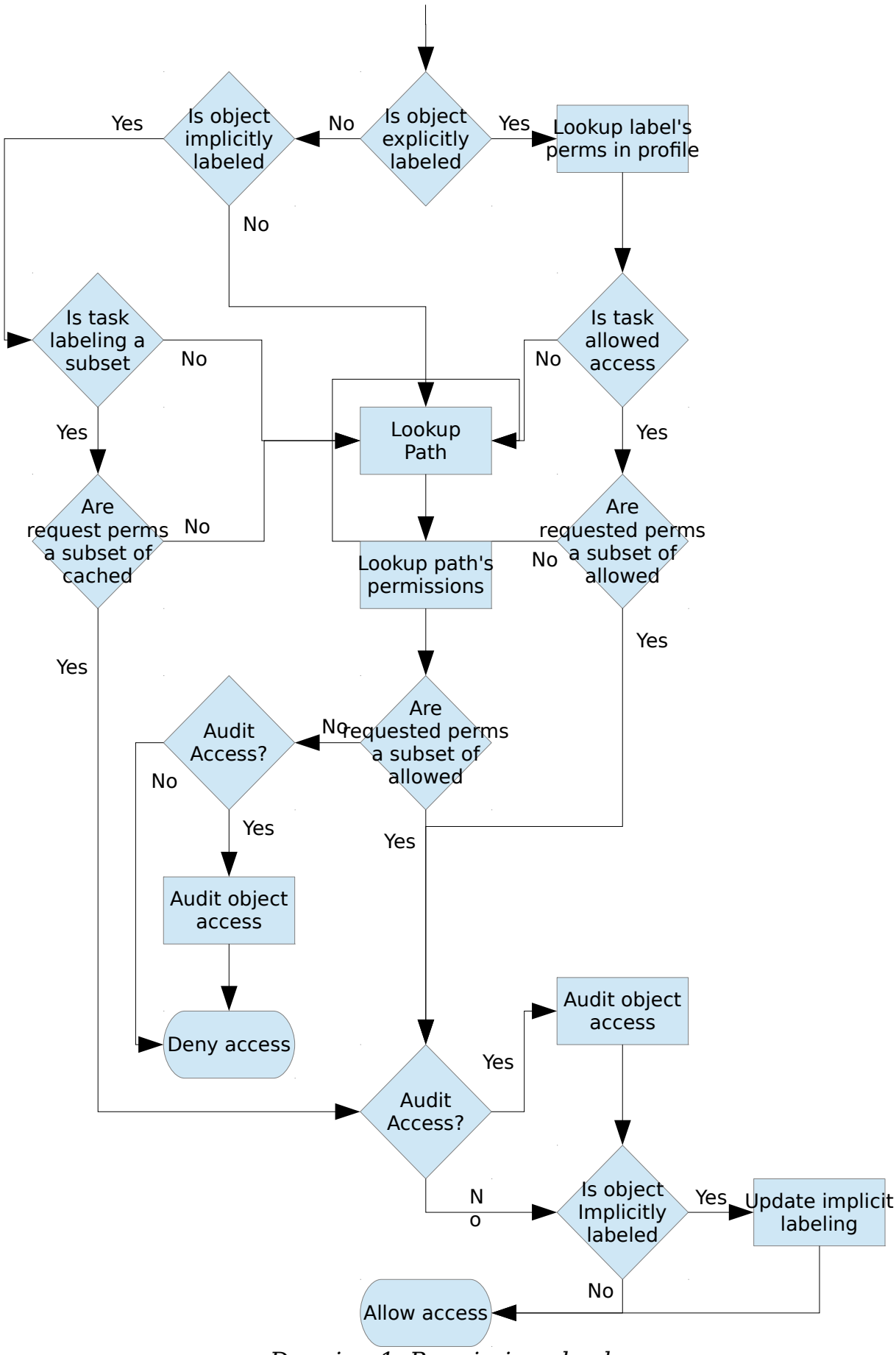
First the labeling on the object is checked. If the object is labeled with an implicit label that matches to the task's current labeling, or the task's labeling is a subset of the objects labeling then access is allowed because at some point a full permission check was already done. Note that the equivalence and subset tests are carried out against the labels within a namespace, not across namespace.

If the object has an explicit label it is checked against, the generic label permissions, and if access is allowed the permission check for the namespace is finished. This will only succeed if the label is granted for all connected and disconnected path checks.

If permission was not granted in the previous step, a path lookup for the object is done.

If the path is connected the path is matched against the match db for each profile confining the task (the stack). If the match results in an unconditional permission, it is used as the result. If however the match results in a conditional permission, the object and conditional index for the match are evaluated to determine if permission is granted. If the object is labeled with an explicit permission it may be evaluated here.

If the path is disconnected the disconnected path check is done. It may use the partial path and any object permissions. ???? conditional proc – get rid of nasty sysctl



Drawing 1: Permission check

Delegation

foo
Implicit Delegation
Implicit delegation allows a profile to specify out of the set of objects that that the profile allows access to, a subset of objects that can be passed to another profile. Implicit allows for a basic form of delegation and dynamic profiles without modifying an application. However it will not work for applications that pass information via access path.

Implicit delegation is done at the open object level so that a task must have an opened object (eg. file) before it can be passed to another task/profile. The receiving task then has access to that object but does not have permission to open a new version of the object.

The passing of an object can be done at exec time through inheritance, or passing the object over an ipc communication channel (eg. unix domain socket).

Eg. If the file at /etc/example is passed a receiving task, that task has the rights to use the file via the handle that was passed but it can not open /etc/example, and it will lose access to the file if it closes the handle it was passed.

It is important to not that implicit delegation is done at the profile level with implicit labeling. When a access path is marked as being delegated, the permission set for that path is marked as having an implicit label of the profile + the target profile. If a particular path can be delegated to more than one target then it will be marked as being labeled with the full set of profiles. When an object is opened it is labeled with the implicit label (either the opening profile, or the set of profiles specified for that path). Subsequent accesses are done via the label check.

This means that the recipient task can redelegate the file but only to task that are allowed by the labeling. This means the initial task that delegates its authority must list the full set of profiles for all tasks that need to access the object.

The labeling implicit labeling that delegation places on an object is no different than standard implicit labels and may be updated by revalidation checks.

??? In some ways it would be nice to allow the inheritance tree, this would be a loosening of perms so could come later if such is needed

??? It would be nice to be able to specify delegation at the exec level instead of just profile because, the profile may depend on what is attached (eg. px may not find the expected profile).

Explicit Delegation

Explicit Delegation allows a task to indicate an access path that can be passed to another task. This requires the application to have delegation logic.

Details pending.

- good for power box, file dialogs etc.
- doesn't need to be as broad as implicit
  - eg. can dynamically control the profile labeling set.
- could be implemented by reloading profiles, or allowing addition exact match rules.
- Could be limited to open files like implicit, but allowing tighter control of the labeling.
-

delegation is done at the file descriptor level, and is controlled at the execution and file descriptor passing boundaries.

Add note: on how delegation plays into revalidation check

Implicit labels – matching permission check vs label check

- only need to check if profiles in set and then if not add profile
- vs. label check. Eg. Ipc profile=A, the implicit label could expand and cause problems
-

Delegation and domain transitions

- does the delegated object go through verification like other object? With the current simple labeling plan it does, as there is no way to distinguish which profile was added via delegation.

Explanation of Profile attachment model – more indepth

Stack order does matter for the purposes of ipc

- foo

- that is system profile should care what usernamespace is labeling the object. Hrmm maybe not stack level but only labeling within the namespace

Profiles, Labels and Stacking

A label is – base label, profile, set of profiles

A profile is a label with rules associated with it

A stack is a special label where ordering of the subcomponents matter for policy administration

A secid is a unique number that maps to a label

A compound name a text representation of

Context
- set of labels and other necessary information for the object

Unconfined State/profile

no mediation except exec transitions, governed by /** px,

over lapping label and implicit label rules, which gets applied
ie.
  label=foo /** w,
  /** rw,

well obviously r is applied as implicit, but should w result in label or implicit labeling?

- seems like explicit should win the overlap.

Need a section on the interaction of linux namespaces with apparmor policy

- Do we want the parent namespace to be able to mark child namespace profiles as immutable, yes
- How do we handle ipc, application has files labeled with both namespaces

- How do policy namespaces interact with other system namespaces
-

used in conjunction with stacking, chroots, containers

Uses for policy namespaces

- containers/chroots
- user namespaces

uses for stacking

- containers/chroots
- user/system confinement
- policy composition

A Note about implicit labeling

Implicit labeling is largely an implementation detail and theoretical construct, that is used to provide a more functional implementation in an architecture (LSM) largely designed around labeling, to explain profile path based rule interaction with explicit labeling, and as a theory to allow a more detailed

comparison with labeling based security systems. Despite its extensive use as a core part of AppArmor's policy the average user/administrator does not need to know what implicit labeling is.

labeling, namespaces and revalidation. What if a change in namespace is encountered

labeling and stacking (how does this work in with namespaces)

implicit labeling and stacks. - how the stack label is expanded when a path specifies a compound label

implicit labeling is handled at the instance level instead of the permanent object level so, that a clean label is started with every time. This is done becomes policy reloads may change the labeling and currently policy loading can be done incrementally meaning that the full label can not be computed in advance.

What of label comparison between a stacked and unstacked

1. If they have the same namespaces then the comparison is as normal
2. If one has more namespaces, a deeper stack
    1. If the actor has the smaller stack, it is not confined by the deeper stack and so comparison is at the same level of their stack
    2. If the actor has the deeper stack, it really shouldn't be able to interact with the other namespace?
3. What if they have the same depth but different namespaces
4.

When is the label subset test used vs. when a label equivalence test used.

Ie communicate with A != communcicate with A&B, but for revalidation purposes A is a subset of A&B is valid

Labeling can be done on File object or inode

- delegation needs to be done on the file object as it should not be visible globally?
    - Well if delegating to profile it could be visible globally
    - if delegating to task, the it must be on the file object
- Labeling or the inode could be done incrementally or all at once
    - if incrementally labeling rules have issues
    - if done all at once, its only done at the fs namespace level because the name can only be looked up against that namespace.
        - Lookup name, iterate over all profiles, build label.

    - If profiles are replaced or added the label still needs to be incrementally updated
    - Does a change in labeling lead to revocation? (separate but related issue)
- What of filesystem namespaces, do we force a new profile namespace when a new namespace is created?
    - Do we track old roots?

If labeling is stored on an inode then, a rename needs to invalidate the old implicit labeling.

How this works for fs namespaces

- they are aliases
- two possibilities
    - different fs namespaces are confined by a separate profile namespace
    - different fs namespaces are confined by the same profile namespace

How apparmor policy is static and dynamic.

- Implicit labels static for any given location
- Dynamically composed, so run time situation, mounts, dynamic hardlinks, are important
- how conditionals and explicit labeling change this

how to handle replaced profiles

- forwarding
- ignore old label / update label?
- Revocation
- revalidation
- deny access because of label conflicts
- problem want old labeling to go away, but sticking it on inodes is problematic, as it may not go away where it can on the file. But we are told we can't change the file.

Policydb layout

Task Label – issues

- using the dfa state to represent a label has several issues
  - the dfa must have a state to profile mapping so which profiles are in play can be found
  - the extra level of indirection slows anything requiring profile access down
  - name aliases means that a label might not be unique for the profile list. That is permutations of the profiles are possible because aliases match different states in the dfa
- Advantages
  - faster, can reduce the # of states to walk (one state can represent multiple profiles)
  - can possibly be used to help solve disconnected path problems

Idea for multiple profiles on the same dfa, allocate contiguously, guarantees sorted into a group (if sorting by address)

discuss profile hierarchy

Do an over view of how it fits together with references to more detail sections

understanding apparmor policy

profiles written from subject pov

access path based rules

- give access to files that are not explicitly labeled

- file is implicitly labeled

label rules

- access to files with matching label

- labels always cross check

label + access path based rules

label access is cross checked, profiles don't dominate (cross check)

persistent vs. temporal label

Cover basics, then revist with more detail, so unix/linux specifics don't come in until more later

- file descriptor/handle caching to handle revocation
- rename???? and aliasing????

Understanding AppArmor Labeling

AppArmor policy is defined by the profile, which when attached to a subject (task) is the set of rules used for access mediation when the subject is permforming an action (executing), that is the profile(s) is/defines the subjects domain. We use the term domain for a subjects confinement because while a profile can be a subjects domain, the domain may also be defined by a stack of profiles (compound label). Initially we will discuss apparmor policy where the domain is always a single profile and the expand upon that????reword/fix me???

Many of apparmor's rules are expressed in the form of access paths, however a label can also be specified as a further refinement or alternate form of control. The behavior of labels and label rules is tightly coupled to profiles and profile names (profiles are labels); understanding this coupling is important to be able to author and understand apparmor policy.

Implicit labeling – deriving labels from access paths

Internally apparmor treats all objects as labeled, however the label is often lazily derived from the access path and the domains operating on it. These dervived labels, known as implicit labels, won't often appear directly in policy. The set of implicit labels is dynamic and may change as the set of rules in the system change (profiles are added, removed, or replaced).

In apparmor access path rules are a way of specifying a type/label indirectly. Instead of saying an object has a type of X, and the domain has access to type X. Access path rules allow a domain to say it has access to the object at the access path, and thus can communicate via the object at that access path with other domains that also have access to the object at said access path.

If apparmor only supported access path rules, then no labeling of the object would be necessary but since apparmor combines access path and label rules it must have a way of derving a label from the access path rules in the different profiles in policy.

Deriving the label - the intersection of profiles and access path rules

To understand how to derive label from apparmor's rules we start with a simple example (Figure 1) with three profiles (A, B, C) that each contain a set of path based access rules. Each profile has access to a unique file and some files that are shared.

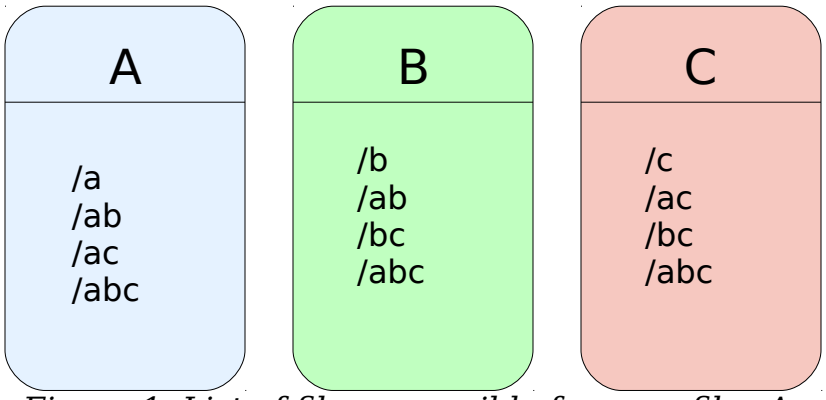


Figure 1: List of files accessible from profiles A, B, and C

The intersection of these profiles is nicely shown with a Venn Diagram. Each section of the Venn Diagram shows which profiles and rules that intersected to create the section. The set of profiles that creates the section is the “derived” label, so for the example as shown in Figure 2 we have the set of derived labels A, B, C, AB, AC, BC, and ABC.

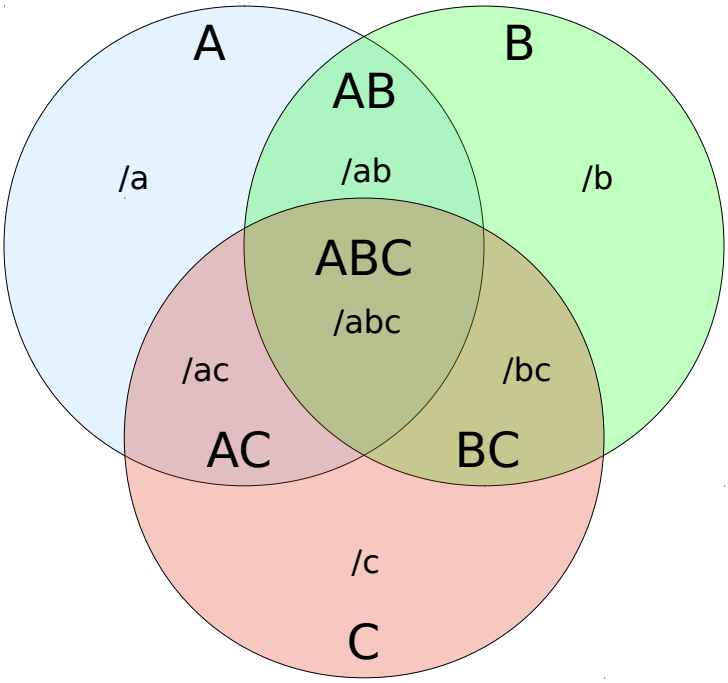


Figure 2: Labels derived from profiles A, B, and C

These derived labels are the basis of AppArmor's implicit labels. The derived (and implicit) label literally represents the set of profiles that can access an object at a given access path. So the derived label for the access path /ac is AC (or A & C), which is means that profile A and profile C have access to the object at access path /ac.

AppArmor's implicit label is based on the derived label, but is more dynamic. It

may start out as a subset of the profiles in the derived label and can grow to be the derived label, or when label based rules are used it may deviate from the derived label to become a super set of what is allowed by the derived label.

The object type and how it is accessed will determine how close to the derived label the implicit label is. When ????.

As long as label rules have not been used in the construction of an implicit label on an object, it is not considered when an access path rule is checked to determine access to the object. The access path rule will be checked and then if the access is allowed the profile will be added to the implicit label. However dependant on how expensive an access check is the implicit label maybe checked as a cache of previous access decisions, because if an implicit label contains the profile in question then access to the object for that profile was already granted.

Using label rules with implicit labels

While the implicit label is not used directly by access path rules it can be used by label rules to reason about access to an object.

Label rule is checked against each profile in the implicit label

When label based rules are used they can modify the implicit label in such a way that further access

That is as long as access path rules don't intersect with label based rules in the policy then access to an object is determed solely via acces path rules.

n't considered when a profile's access path rule is checked to determine access to an object. When a new
but it can be used by label based rules to mediate access to objects.

??? when does an implicit label grow, vs. be used in its entirety ???
??? Move this else where
the difference being that the implicit label is dynamic and may start out as a single base profile, and tighten as the intersections are discovered.

More on this dynamism???

Explicit labeling ???

Access path rules and their dervied labels provide a default security view of the system. However this default labeling is insufficient to maintain isolation between security domain when there are overlapping read and write permissions

between. While some times the overlap is appropriate, there are other times when this overlap allows for sharing that should not be allowed.

For example /tmp is often used by multiple applications and most policy requires both read and write privileges to /tmp. But if a user runs applications in different security domains it is likely that they should not be able to access each others /tmp files. To achieve this explicit labeling is used to set a label on an object that is different from the default implicit labeling.

The explicit label is stored on the object and it is used instead of the implicit label for access checks. Further more it prevents plain access path rules from accessing the object as plain access path rules are conditional on the object being implicitly labeled.

The explicit label behaves in a similar maner to an implicit label that has been extended by a label rule. That is the explicit label forces a check????

as there are times when object needs to have a different labeling than what

the derived label can expresses the view only via access path
sometimes we want to be able to express an object is outside the pattern provided by implicit labeling, in this case the object can be explicitly labeled.

Explicit label says regular label derivation does not apply.

Need label rules to access object

label rules can combine both label and access path

Combining label rules and implicit labels???

label rule can be used to access an object that is implicitly label

If access is granted it expands the objects implicit label
label access rules can be applied to each profile in a compound label (implicit label)

New accesses to the object by path or label, are checked against the profile(s) that where added via label based rule. As that profile is not allowing access to anyone on path X, but only access to domains A, B, C.

????
When a label is on an object its rules are not applied unless the object is acting as a proxy for another subject so that the cross check is applied.

??? shoul cross check be applied to files??? It is possible!
- if applied eagerly only sane way to block unallowed communications
- Hrmm could have a global type table like DTE too (yuk, no).

- Global rules that get applied to all profiles?

need to bring compound labeling in after basic labeling discussion, maybe even after basics of all
need cap, resource limits sections
need domain transition section
need namespace discussion, and how it applies to each type

socket/fd passing
delegation

peristent storage implicit labels treat different than transient communication channel
- storage full derived label, for label rule(s) to pass
- comm channel, partial implicit label
  - check against current, adds to label and can block new additions because of cross check
    - new path entries could just be added but current labeling may not
    - should have flag to indicate when label isn't pure implicit? It would speed up check but could not be removed

Communication between Subjects

The direct communication between Subjects is a little different than the use of a file. As shown in Figure 3 communication while it has the appearance of being direct, passes through some channel in the kernel, where subject context and apparmor policy are applied before the communication is completed.

In apparmor policy the profile for neither subject is dominant and the policy is applied at both the sending and receiving end of the communication. This means that if Subject A is going to send a message to Subject B, the profile on Subject A must allow sending to Subject B on the chosen type of communication channel, and Subject B's profile must allow receiving a message from Subject A on the chosen type of communication channel. This check is known as the cross check and it is done for all types of inter subject communication.

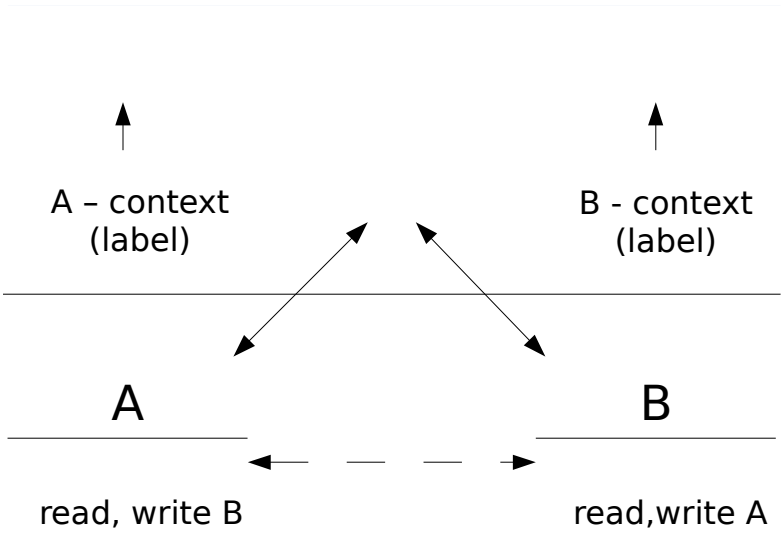


Figure 3: communication between two subjects confined by AppArmor

For some forms of communication (signals, ptrace) this simplistic model is enough but for some other forms of communication this basic model is insufficient. This is because many forms of communication do not go directly to the task but go through an object or objects that can be shared between different subjects, examples of this would be unix pipes, fifos and sockets.

Unix pipes and fifos are examples of communication that share a single intermediary object with a handle in each subject to access the object. In Figure 4 we see that when process A creates a pipe and its two handles (file descriptors fd_r, fd_w), the pipe and handles are labeled with the same label as process A.

Figure 5 show what happens when process A pass one of the file descriptors (fd_r) to process B. Access to the file descriptor and object (pipe) are checked for process B. If process B can communicate over pipes and communicate with the labeling of process A, then process B is allowed to receive the file descriptor. If the labeling is not explicitly set then the labeling of the shared object is updated to reflect that it is shared by both A and B.

??? do we label the handle
??? do we update both the handle and label
??? does pipe labeling step back down in labeling when a handle is closed by a process? Neat but probably not
- would work for files too, files can be assumed to always be the full set due to persistence. Ie tracking lost over time.
This affects how the label is applied for temporary objects the label represents the current can access set. This affects how label rules can interact with a file or ipc object. In the file case the full access set must be in the can comm set.
- how do we handle dir renames? Need to invalidate in some way
- handles allow for access after a file is not accessible.
??? we could delay inode labeling to when a label rule forces it. Or we could do it

eagerly as files are accessed. Both require invalidation for renames unmounts, bind mounts. Eager could cat hard link issues
- files we don't store labeling on inode and use path to derive label, we do this to avoid the dir rename problem, that is a dir move causes a relabeling but we don't propogate the info to the children
- the full label on the file should be the derived label
- we have the dir name problem with fifos
- comm channels need labeling for cross check, but why not files?

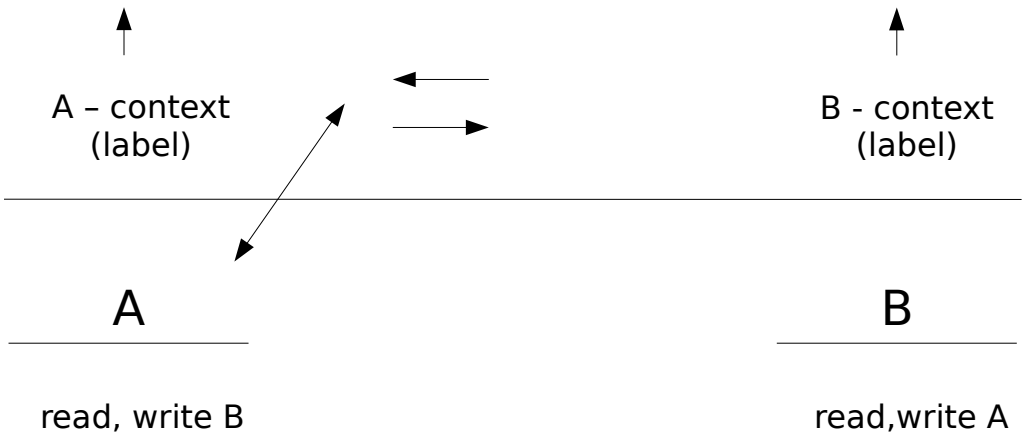


Figure 4: Process A creates a pipe

If the pipe is further shared as in Figure 6 the cross check is done again and the new participant (process C) must be able to communicate via pipes with both process A and process B, and process A and B must be able to communicate with process C via a pipe.

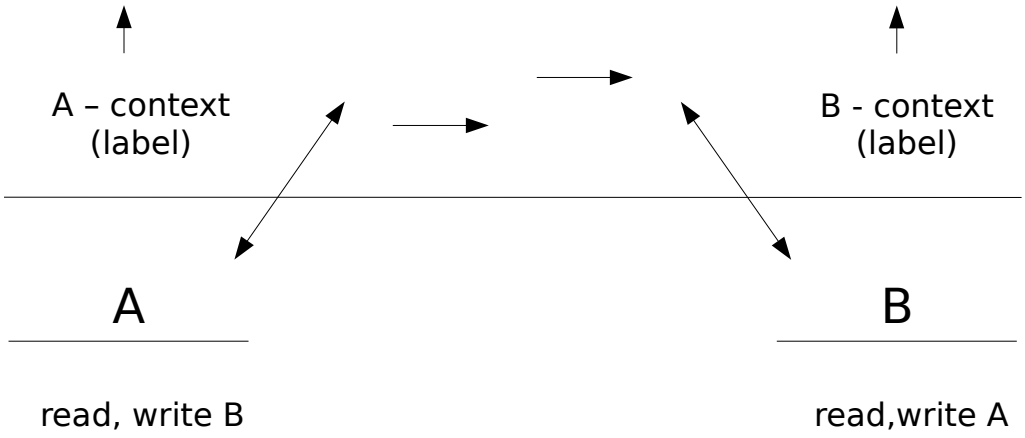


Figure 5: Process A shares pipe handle with B, causing the labeling to be updated

??? pipe – 2 diagrams, A sets up pipe and handles. 2Nd A pass handle to B

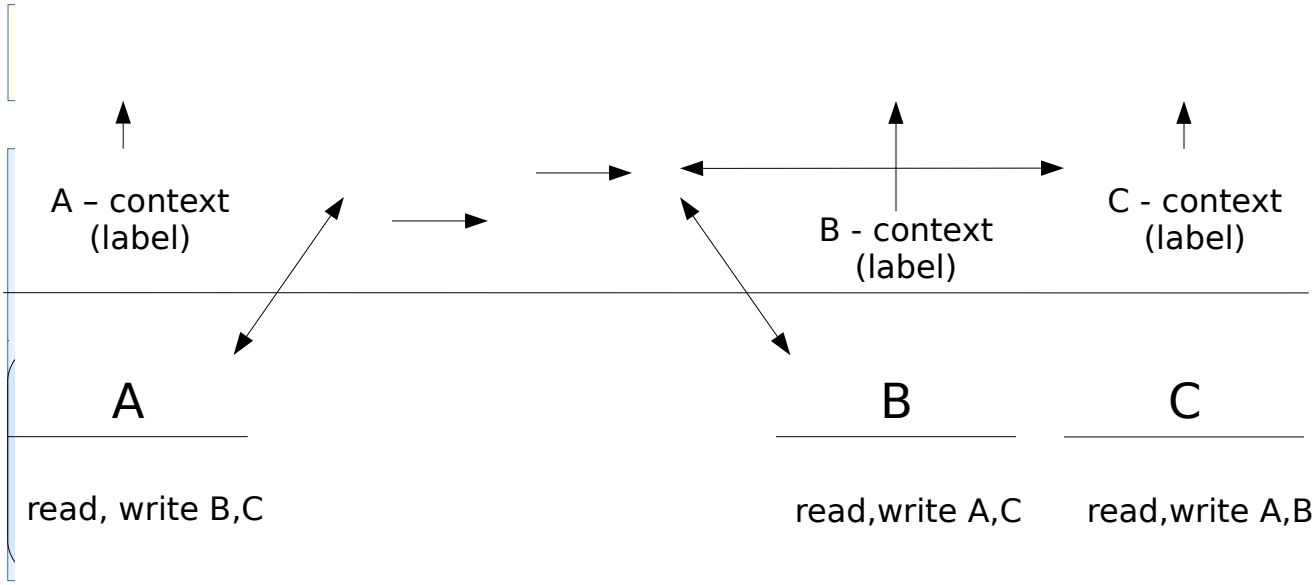


Figure 6: labeling of pipe shared between 3 processes

??? fifo – A sets up fifo and handle, B opens path to object. Instance label to ref count without inserting in tree?

??? shared single object (inode), starts out a single labeling picks up, other when passed.

Fifo?

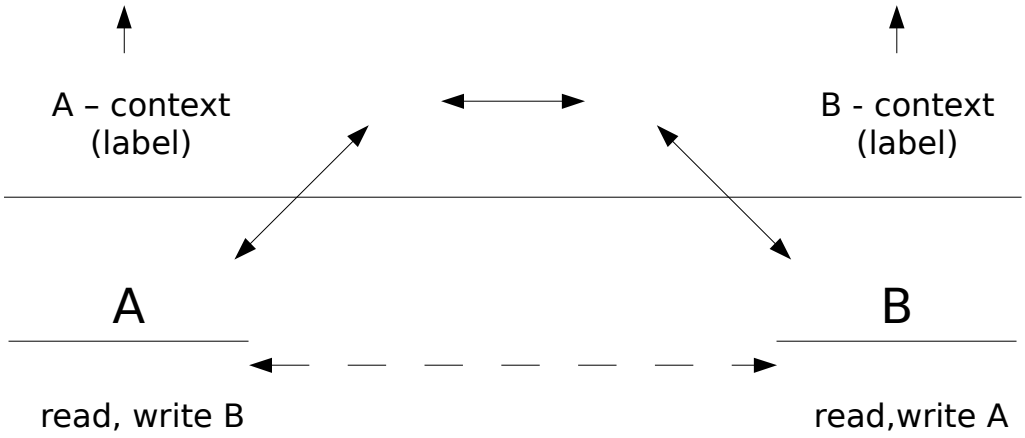


Figure 7: communication between unix file descriptors

Figure 7 Show the basics of how communication across a file descriptor is handled. The descriptor is labeled by default with the Subjects label. The label on the fd is then used in determining access restrictions. ??? What of fds that don't have a peer????
Get labeled then passed

??? diagrams 2 tasks with 2 comm ends

??? 3 tasks with a shared comm end
race to accept

???? comm type depends on whether it is path based or label based (or both) some forms of communication require label check, some can use path check – path == allow comm with anyone at this address.
Label check forces a block on the implicit label as any profile doing a label check must check all new labels being added to the implicit label

Communication over a bus

???? communication over a bus diagram and discussion
similar to shared comm end, except no race delievery to every entity

Communication across a network

???? communication over network discussion
??? labeling on message vs relabeling at the end point

That is labeling of each subject is used as an object label in one of the checks.

- using subject and peer label label
- labeling of the socket
  - cross check (checking at send and receive)
- using “access path”/”address”
  - implicit labeling of the socket
- setting the label

Communication across on Network

comm figure with cloud, and network matching rules.

- like communication between subjects
  - does cross check based off of label
- we don't have direct access to the peer subject, so where does the label come from
  - across the network (cipso, ipsec, …)
  - packet labeling
    - specifying the label
    - implicit label from remote end rules

AppArmor policy in relation to other security models

AppArmor is not a type enforcement system though it can enforce as one depending on how the policy is constructed. To better understand apparmor policy we can examine how to make apparmor policy behave like Type Enforcement (TE) or Domain Type Enforcement (DTE) and analyse the the differences.

The basics of TE are that it is an access control model where types are placed on subjects, and objects. The rules governing access are writen as a tuple of the subject type, object type, and permissions, which are stored in a table. Enforcing policy involves looking up the permission entry in the table using the subject and object type.

DTE is a variation of TE where the subject's type is known as the domain and it contains the table of rules to enforce when a subject is under that domain. For network communications mediation is performed at both the send and receive time, so that the sending domain must be able to write to the receiving domain and the receiving domain must be able to read from the sending domain.

The original DTE prototype[i], stored file typing information in an external structure instead of on the files. The type of the file could be set based off of the location of the file in the filesystem and to avoid directly storing the relationship of every file and type pattern matching could be used, this was called implicit typing. However the detail of how the file object type relationship is stored is mostly irrelavent to the basic DTE model.

AppArmor is very similar to an extended DTE model. Its implicit labels are similar to the implicit typing of the DTE prototype, and for network (inter process) communication the security check at both send and receive is also very similar.

Diffs
rules that create the implicit label are embedded in the domain, this is not enough to create a type in the DTE sense.
Apparmor rules specifying labels could be used to create a typing
lazy construction on handles – expensive cross check if it gets large, but much smaller cross check on expansion.
No revocation of existing handles by default
stacking
cross check – can be more than just at send/receive. (both at send/receive)
label may be treated as a type but may have to follow other info, like label at path
all labels have rules – rule is only applied for subject or proxy
handles – we deal with more than just the object we deal with handle level
derived label – may represent an impossible combination but the implicit label is

dynamic and can reflex actual usage.

start with apparmor as purely labeled system, no implicit labels? - As long as labels are set for the types it is a effectively a dte system.

AppArmor has several similarities, it places labels on both subjects and objects, and like DTE types, labels can be implied through a set of rules. AppArmor profiles are very much like DTEs domains and when on a subject could even be considered a type. Unlike DTE there is no difference between apparmors labels on a subject or an object, however only the subjects rules are applied.

AppArmor's labels are not types because it is not required that they be partitioned fine enough that they can be used independently from the rules that imply the label. In fact apparmor's implicit labels are only valid within the context of the rules in the profile.

??? implicit labels are dynamic and lazy

AppArmor's file rules that the label is also different than DTE rules that imply type because the rules are in the profile instead of global set of rules setting the type. This the type information can not be derived from the rules with out taking all other profiles into account.

Building types from AppArmor policy

Ref back to example about building implicit label

To understand how to derive types from apparmor's rules we will start with a simple example (Figure 8) with three profiles (A, B, C) that each contain a set of path based access rules. Each profile has access to a unique file and some files that are shared.

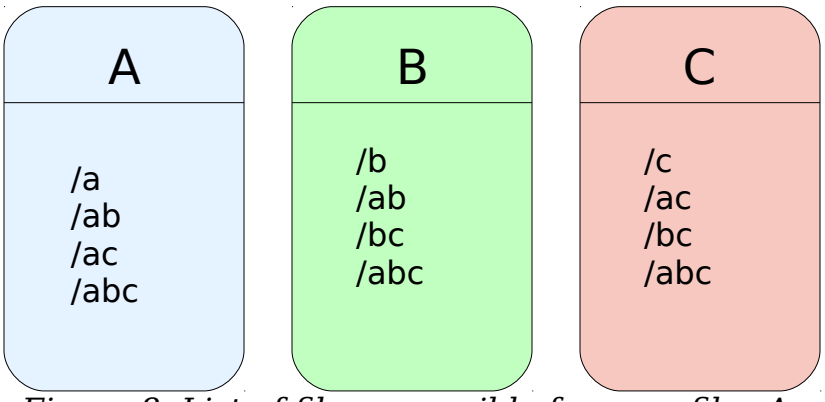


Figure 8: List of files accessible from profiles A, B, and C

The intersection of these profiles is nicely shown with a Venn Diagram. Each section of the Venn Diagram shows which profiles and their rules that intersect to create partitions. The set of profiles that creates a partition is the “derived” label, so for the example as shown in Figure 9 we have the set of derived labels A, B, C, AB, AC, BC, and ABC.

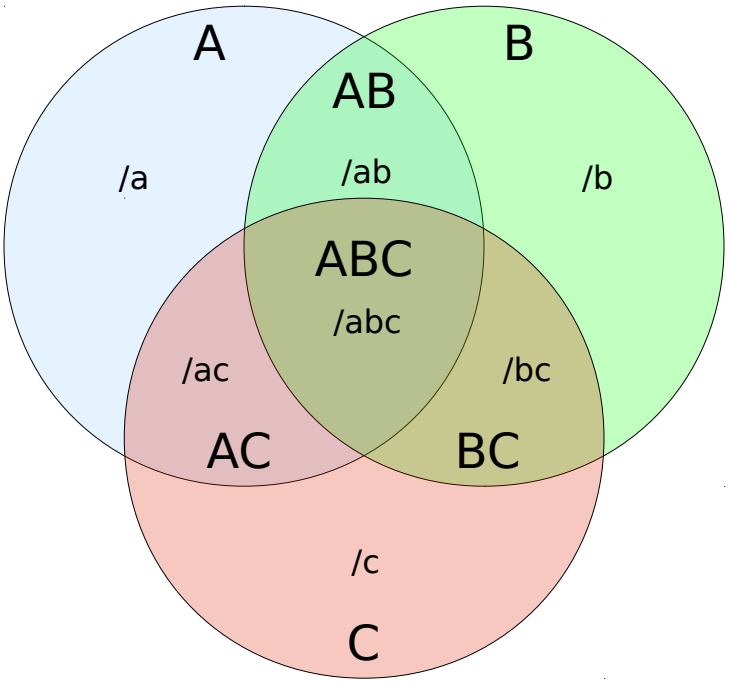


Figure 9: Labels derived from profiles A, B, and C

These derived labels are the base of AppArmor's implicit labels, the difference being that the implicit label is dynamic and may start out as the base profile, and tighten as the intersections are discovered.

???
It is important to note that the derived (or implicit) label is not a type (in the sense of type enforcement) as by itself it does not have enough information to determine access permissions. The derived label however could be further split to arrive at a “type”.

To find the “type”, in a system with a single profile it is sufficient to partition the profile rules into different “types” based on the set of permissions granted. For rules that overlap and have different permissions the overlapping rules are split into new rules and assigned to the appropriate partition. The new rule that represents the intersection has permissions that are the union of the two rules.

A simple example of this partitioning to find the “type” can be seen in Figure 10. The type name in this case is based off the profile name A, and the permissions represented by the partition (A_r, A_w, A_k, A_rk, A_wk, A_rwk). There is no unique partition based on the */abc rk,* rule because this rule intersects with other rules and the permissions of those intersections merge resulting in the A_rwk partition.

The sytem could then be labeled with the partion types using the rules in the partition. Permission lookups then could be performed by directly mapping the type to the allowed access permissions for an object without using the file path

and match rule.

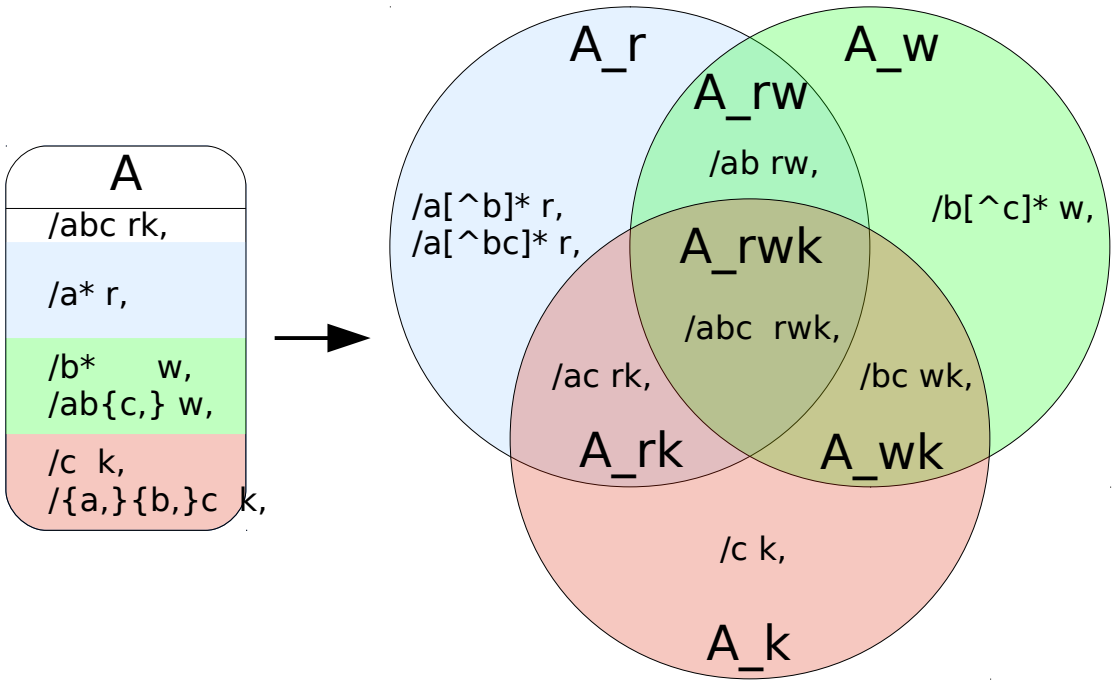


Figure 10: Partitioning of a profiles rules for type

This single profile direct mapping of type to access permissions is not sufficient for systems with more than one profile. The partitioning needs to be done for every profile, and then the intersection of the partitions between all profiles. This will arrive at a unique type for all possible intersections. As shown in Figure 11.

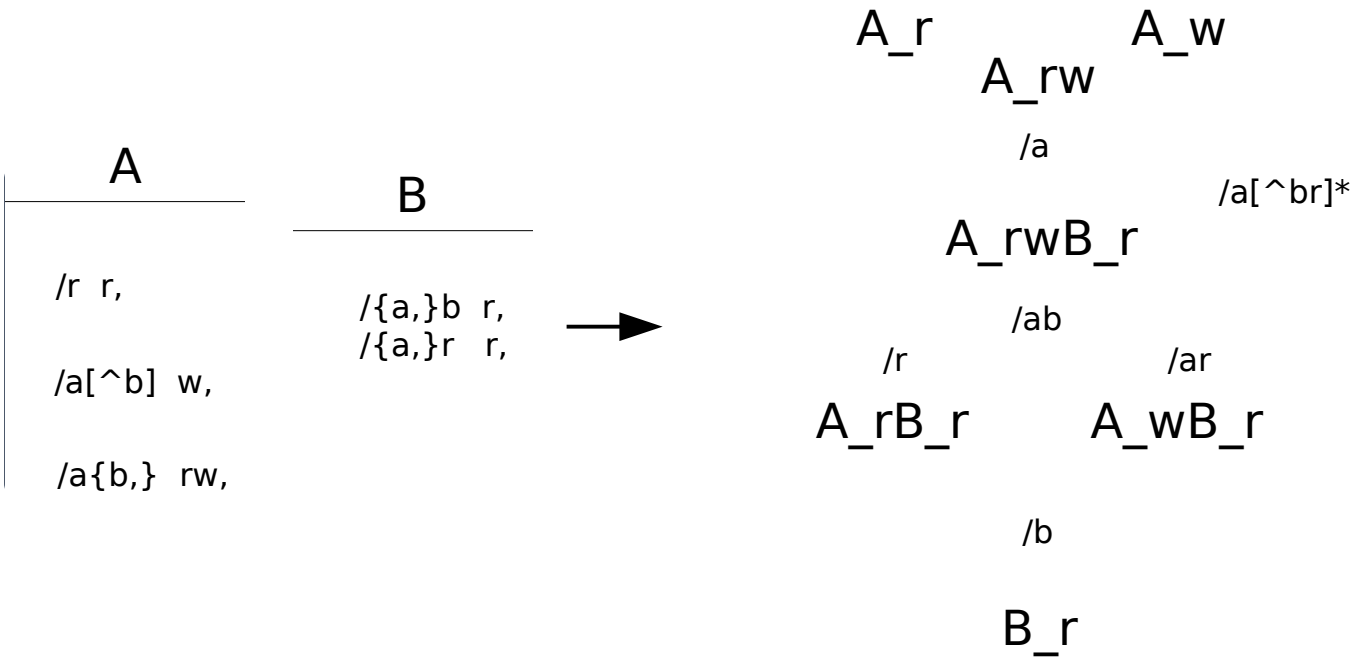


Figure 11: Types derived from the intersection of 2 profiles

For the policy in Figure 11 the ??? no A_r type in final set of types

The basics of splitting rules to achieve a type when regular expressions are involved is covered in ????.

???The derived “type” comes from combining the profile name, rule segmentation and permission sets. For a system with a single profile, the “type” would be derived from the different permisions as shown in ???.

??? shows two profiles with file accesses and the their permissions for each profile.

??? why we are not using type – HUGE, too brittle for apparmor purposes (keeps changing), results in really long type names, doesn't mesh with partial loads, and needs for cross checks.

After computing the set of derived labels we could then label each file with a label that would indicate the set of profiles that have access to it. The set of access permissions could either be found by looking up the intersection of the set of rules in the profiles, or making a set of global rules based on the derived type.

However apparmor does not specify labels on all objects, in many policies having an explicit label is an exception.

global policy – single dfa: state provides label
- problematic in dynamic system, changes in namespaces etc (have to recompile dfa etc)

- Solutions, multiple dfas and composition, options

- stacking
- interaction of tasks

- not type

- cross check (because profiles express independently who can communicate) communicating domains

- live replacement/no revocation

crypto graphic hash and cipso label

- can only be trusted if you trust the network and the machines on the

network
    - provides a uniq name lookup that can be guarenteed to fit in cpiso field instead of variable length apparmor label. Can be pre computed so doesn't have overhead of generating
- might need to be augmented to provide compound label lookup
- on label name
    - only provides a name lookup, no policy check
- on policy
    - provides name lookup + check that machines policy is the same.
        - Assuming you trust the remote and the network
        -

Hrmm tainting for Delegation
- does it only go on the handle, or the inode as well? Should it get stored

- Only on handle
    - handle shares well, other processes can access the file (files can be see as ipc)
    - Does it interfer with sharing of handle
        - Access rights are of the delegator
        - Delegator trusts Delegatee
        - Delegator could potentially pass the delegatee information other ways, so could bypss a locked down tainting check
        - If delegatee label shares in cross check
            - tightens communication and sharing via the handle
            - may require more rights so it can communicate with others wanting to use the file. Order of building the label counts
            - Note label check is a little different than implicit label check
            - more like the cross check fone for other communication
            - Hrmmm should the cross check apply to file labeling?
- On inode
    - same as on handle but not per instance, could prevent other apps for accessing the file while the delegatee is using it
        - no need unless the label participates in a cross check
        - could auto remove the taint on handle closing (would need independent ref count)
        - shutdown
        - could store taint so it carries over until cleared
            - could have permissions that clear it like trunc
    - this would keep tmp files etc from being shared without explicitly labeling them
    - need to think about it more

Deriving types from regular expressions in AppArmor's Policy

The addition of regular expressions to the set of domain rules complicates the analysis of domain intersection, but does not make it impossible. To determine the intersection of the domain rules we turn to analysis using a minimized dfa. The accept states of the dfa will encode where the domain's state belongs, and from that we can derive the labeling.

We will revisit the above example using regular expressions to specify the files that each domain can access. Each domain has a single rule that will match all of the files specified in the previous example and several more. The domains overlap again but in a more general way because of the regular expressions.

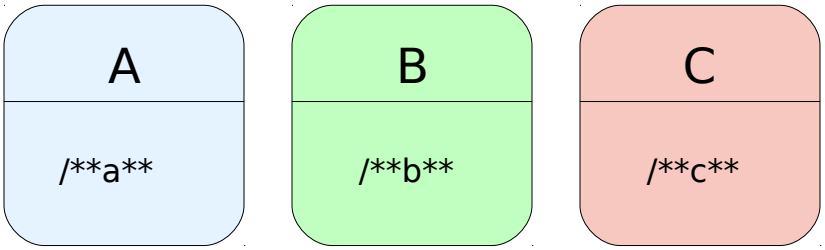


Illustration 1: regular expressions showing what files a domain can access

The minimized DFA for the regular expression for domain A is shown in Illustration 2. The domain labeling **A** is encoded on the accept state. The DFAs generated for the other domains are the same except that the domain label encoded on the accept state and the character being matched is substituted.

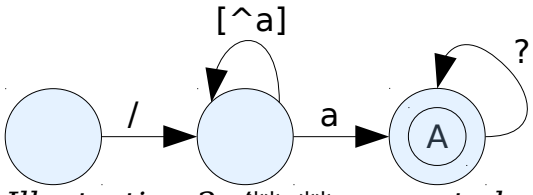


*Illustration 2: /**a** converted into a minimized DFA.*

To find the policies' implicit labeling, the DFAs for the domains are merged together into a single DFA. When an accept state is merged with another state it is given a new label that is the union of the accept state and the state it is merged with (nonaccept states have a null labeling). The new DFA is then minimized to ensure the minimum number of states (a minimum number of states isn't strictly necessary but a minimum number of accept states is). The

result is a DFA as seen in Illustration 3 with labels that match points of domain intersection similar to the Venn diagram.

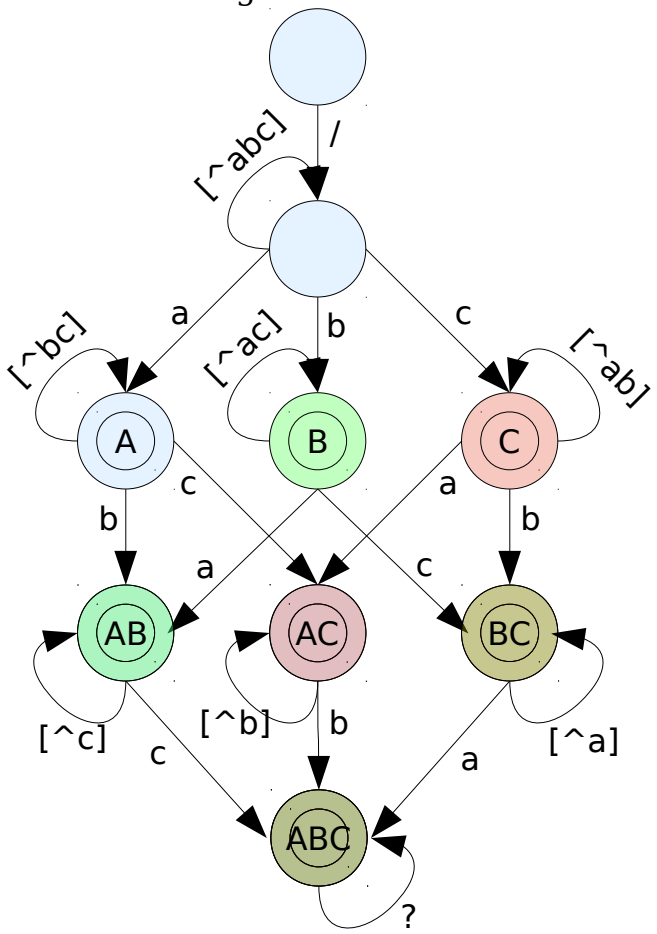


Illustration 3: Merged DFA showing the implicit labels for the policy created from domains A, B, and C

Like in the Venn diagram example the unique accept state labels comprise the base set of labels that are enforced by the current policy, and just like the early example these labels are fragile in that they may change if any rules in a domain is changed.
The situation however isn't quite this simple. The base set of labels is really just the set of all possible combinations of domain names. What is fragile is which of these labels are in use by policy and what a given label represents. The representation of the label changes as it represents a specific set of files that are shared by the policies current domains, at any given time.

Bringing permissions into the analysis

Inter domain labeling is not sufficient for analysis, it needs to be combined with the per domain permissions to arrive at the final label set. To do the analysis a domain needs a single set of permissions per label but for any given inter domain label a domain can have may have more than one rule, with potentially different permissions, that at least partially maps to it. That is a domains permissions may

cause an inter domain label to be split.

To achieve this instead of labeling accept states with just the domain or just the permissions each state gets encoded with permission domain pairs. These pairs are then merged as the label sets just as was done before. The resultant labeling contains is unique for the set of domains and permissions.

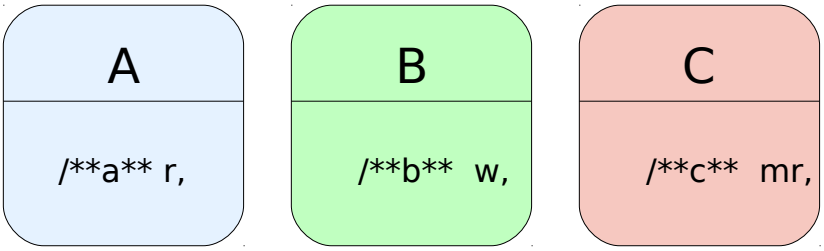


Illustration 4: Domains from Illustration 1 showing rules with permissions

Reworking the domain example with permissions we can see how ???

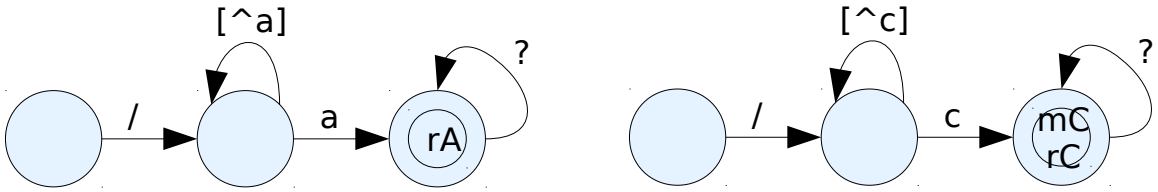
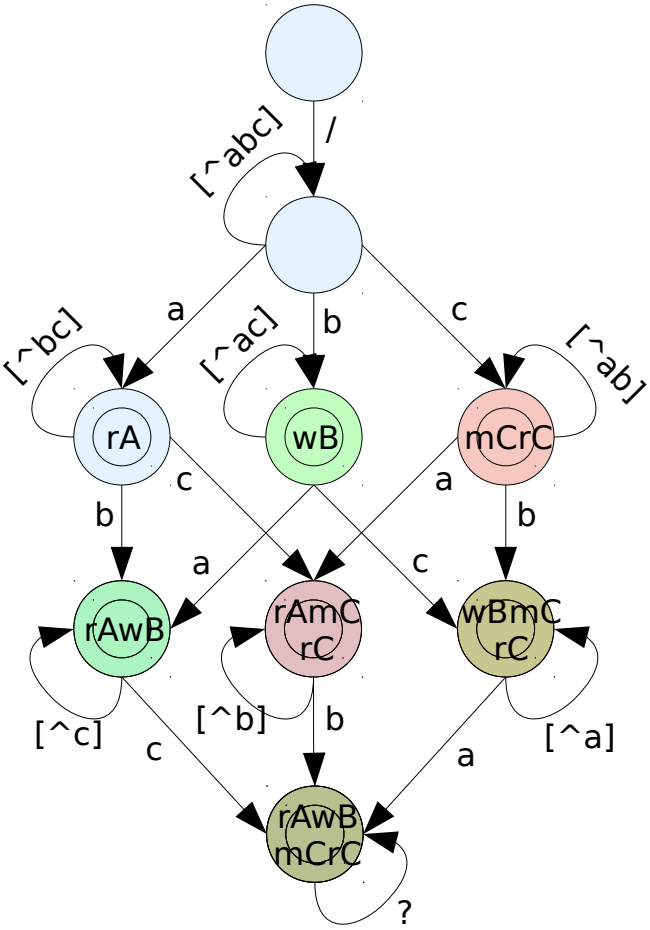


*Illustration 5: /**a** r, and /**c** mr, rules with accept state labeled with permission domain pairs*

foo



The label can be used directly for analysis or the permissions for each domain can the be extracted, along with the set of intersecting domains, and the domain label can be renamed to make it easier to deal with.

?????? revisit previous example with permissions, can we get a split label with it

?????? NOTE: below needs reworked, moved or deleted ?????

Rule	Label
/home/ r,	r
/dev/null w,	w
/tmp/ rw,	rw

Table 1: Example domain requiring 3 unique labels

To understand how rule overlaps are calculated when regular expressions are present in rules, a little formal language and automata theory is needed. For every regular expressions there is a regular language and a set of equivalent finite automata. Using automata theory it is possible to do set operations to determine the intersections of the domain rules. For the purpose of our analysis we will use a minimized deterministic finite automata (DFA).

The regular expressions in the domain can be converted into a DFA using the well known steps of first converting to an NFA and then using the subset construction to generate the DFA. The DFA can then be minimized and its accept states can be encoded with the permissions (labels) that a match belongs to. This conversion can be seen in Illustration 6 where the apparmor rule /**a** r, has been converted into a dfa with its accept state labeled with the r permissions.

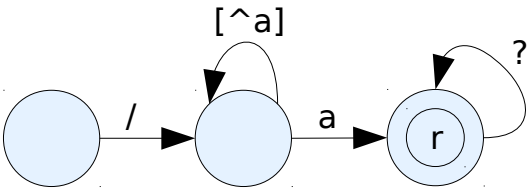


*Illustration 6: AppArmor rule /**a** r, as a minimized dfa with the accept state labeled with the permission granted*

When rules intersect we need to resolve what permissions will be assigned to the portion of the rules that overlap. For AppArmor this usually is done by accumulating permissions as shown in the Venn diagram in Error: Reference source not found.

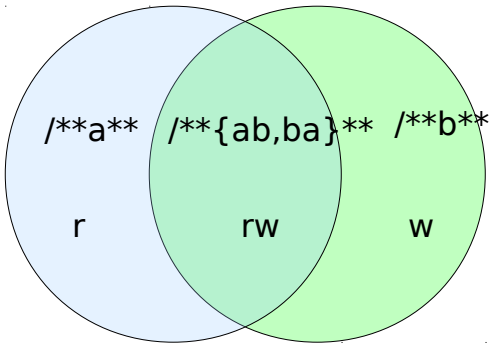


*Illustration 7: Permission for intersecting rules /**a** r, and /**b** w,*

The overlapping of permissions is handle by the dfa construction where accept states get labeled with a set of permissions. When two accept states are merged due to minimization then the new states label is union of sets of permissions of the merged states. The resultant minimized dfa (shown in Error: Reference source not found) will contain M accept states with M unique permission sets (some accept states may have the same permission set).

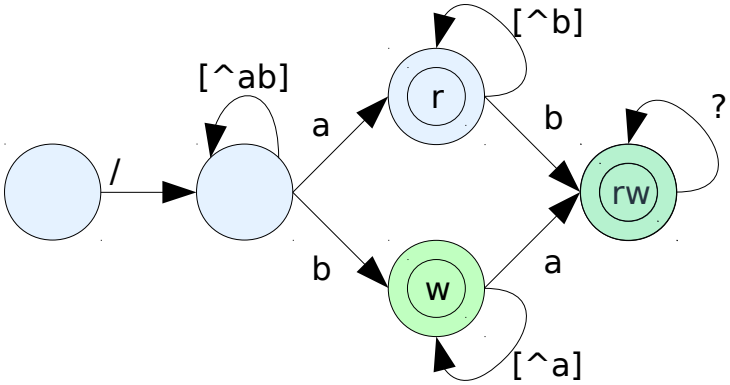


*Illustration 8: Dfa for combination of AppArmor rules /**a**r, and /**b** w,*

Resolving dominance for overlapping rules

Glossary of Terms

Subject

An active entity (task/process) or a proxy for a task/process. The labeling on a subject determines the set of rules being enfoced.

Proxy

A task or object that is acting on the behalf of a subject. It uses the subjects label to determine the set of rules to enforce.

Object

A passive entity (a resource, file, record, message, ..) that a subject interacts with.

Task

A kernel level process or thread of execution

Confinement

A generic term for the set of restrictions that AppArmor is placing on a task.

Confined

A task is said to be confined if AppArmor is mediating its access to any objects or resources.

Unconfined

A special task state where AppArmor is not mediating its behavior. It is also a special profile that AppArmor uses to track which tasks are in the unconfined state.

Profile

The profile is a label with rules and is the base unit of task confinement in AppArmor. Each profile has a unique name, some control flags, and a white list that defines the sets of rules and relations that govern what permissions a task will have.

Profile name

A unique name that is used to distinguish between profiles. It may also be used to determine how a profile attaches if an attachment is not provided.

Attachment

Attachment is how a profile specifies which executables it applies to.

Fully qualified profile name

The name of a profile including its namespace and all path components separated by //. eg. :namespace://parent profile//child//grandchild

Policy

The total set of profiles that determines system confinement.

Subprofile/Child profile

A sub or child profile is a profile that is declared to exist within another profile. It acts similar to a namespace for profiles in that child transition x rules will do attachments against the child profile set.

Sibling profile

A profile that exists at the same level (within the same list), as the profile in question.

Hat profile

A hat profile is a special child profile that has been flagged as being available to change_hat operation.

change_hat

A task directed profile change via an api call. It allows temporarily changing a task's profile to a hat profile, and then return to the original profile via a random token.

Domain

The confinement on a task. In apparmor this is the usually a profile, but may be a compound label.

Domain transition

The process of transition a tasks domain from the current confinement to a new confinement.

change_profile

A task directed profile change via an api call. This is analogous to selinux's setcon. In apparmor terms it is not considered as setting the context as it may or may not operate on the full context (stack of profiles).

Label

A label is the type/mediation information attached to an object, which is used in determining what permissions a task has when accessing an object. Each label has a unique name which can used to reference it. In AppArmor. A label can be anything from an individual unit (explicit label, profile), or a representation of a set of units like an implicit label or profile stack.

Explicit label

Explicit labels are labels that are defined as part of policy and can be permanently applied to a system object. As such they can be stored on disk and remain defined across sessions and machine reboots.

Implicit label

Implicit labels are labels that are derived from profile rules for access paths, and literally represent the set of profiles that have access to a given object on a given path. The simplest implicit label is a profile, with more complex implicit labels being the set of

profiles that can access the object.

The name of an implicit label is based off of the name of the profiles it is derived from with the simplest being the profile name. More complex labels are known as compound labels.

Implicit labels can converted into explicit labels by a rule specifying that the labeling should be stored.

Simple label

A label that consists of just a name that is a single term in length.

Compound label (name)

Is a combined set of namespaces and labels. Usually just referred to as a label.

Term

A subcomponent (grouping) of a compound label that is used to determine domain transitions.

Access path

The access path is the “name” and information used to access/find an object, and it used in computing the implicit label for an object. The access path will vary depending on the type of object an example would be a file path name.

Handle

A handle is an identifier used to reference an object without specifying an “access path” (eg. file descriptor). Objects referenced by handles will have a labeling associated with them, and may grant different access permissions than are available via a lookup of the object done by access path. This is due to permission changes to a live object are not generally revoked, and delegation may grant permissions not granted via access path rules.

Disconnected paths

Disconnected paths are paths that can not be resolved to an expected root for the object type. These can occur in file names when a file system is lazily unmounted our a mount has been removed from the namespace, either via chroots, bind mounts, or pivot roots.

Revalidate/Revalidation

Revalidation is a permission check that is done against an object after it has been opened. It is done when an object fails the labeling test, which forces a path lookup and reevaluation of permissions and conditionals.

If permission access is granted revalidation may update an objects labeling. If it does, further accesses may be subject to revalidation as well.

Revoke/Revocation

Revocation is the process of revoking access to objects that a task may have already been granted access to. Revocation will cause further access to revoked objects to fail with an error. In apparmor revocation is handled at the profile level, when a profile is revoked all live objects that have access granted by that profile will possible lose access

pending a revalidation check.

Taint/tainting

Tainting is the process of placing a special taint mark on and object. In apparmor tainting occurs when an object label is extended through delegation, or revalidation. That is tainting in apparmor is just the dynamic extension of the label set on an object. Tainting when combined with rules can be used to dynamically limit what profiles can interact with the object.

Taint loss

Taint loss is the process of losing a taint. In apparmor this may occur at domain transitions.

Namespace

A namespace is a logical separation of names so that a given name can be used to represent different things in the separate spaces. Linux has several namespaces (fs, net, pid, ..) and they may affect the mean of access paths in profiles. In addition AppArmor defines its own policy namespaces for separating out sets of profiles.

Policy Namespaces

A policy namespace contains the set of profiles that is visible and available to attach to a given task. They allow separating profiles into separate functional groups that can be managed independently, allowing for different policy to be applied to different groups within the system.

User policy namespaces

User policy namespaces are a special namespace that allow a user to define policy for their own tasks. They allow for users to be able to define and manipulate MAC policy for their own tasks independent of what system policy defines.

Profile stack

A profile stack is the set of profiles (labeling) on a task. The labeling is distinct from the labeling of an object in that the last profile added to the stack is tracked as the top of the stack, and determines the current namespace.

Stacking

The processing of adding a new profile and/or namespace to a tasks current labeling (profile stack).

Conditional

A conditional is a rule modifier that modifies the kernel object, policy, and match; that can be used to dynamically determine permissions.

Variable

Any policy variable whether boolean, single or multi-valued, compile time or kernel side.

Static variable (compile time)

A variable whose value is substituted at policy compile time, and can not be changed unless policy is recompiled and reloaded.

dynamic variable (usually a kernel variable)

Any variable that can have its value altered dynamically after the policy has been loaded into the kernel.

Instance variable

A variable whose values is determined by a particular instance or instantiation, eg. UID

System kernel variable

An instance kernel variable whose value can not be set but is determined by system values like PID, UID, etc.

Namespace kernel variable

A kernel variable that is scoped at the policy namespace level.

Profile kernel variable

A kernel variable that is scoped at the profile level.

Instance kernel variable

A kernel variable that is scoped at the task/cred level.

Conditional variable

A variable that can be used in a conditional expression.

Matching variable

A variable that can be used in a pattern match.

Policy Compilation

The process of converting text based profiles into the binary state machines and permissions used to enforce policy.

Policy Cache

A cache of precompiled profiles that are used to speed up loading of policy on boot or policy reloads as long as there are no changes to the cached profiles. The policy cache is dependent on policy, compiler and kernel versions.

Permission Check

When a task wishes to access an object a permission check is done to determine whether the task has sufficient permissions under its current confinement. The permission check is done in multiple steps and can succeed or fail at any point.

Delegation

Delegation is the ability of a task to delegate some of its authority to another task confined by another profile. This allows a task to effectively expand the permissions of another task beyond what is allowed within its statically defined profile.

Implicit Delegation

Implicit delegation allows for a basic form of delegation in which a set of objects can be automatically marked for delegation when accessed without modifying an application.

However it will not work for applications that pass information via access path.

Explicit Delegation

Explicit delegation allows a task to specify which objects it wants pass to another task, out of the objects it has access to (within the constraints of the confining policy). This requires a task be modified to so that it can "identify" the objects that should be passed.

Explicit Object Delegation

Explicit object delegation allows a task to specify which open objects can be passed to another task/inherited.

Explicit Rule Delegation

Explicit rule delegation allows a task to indicate an access rule that can be passed to another task. This allows extending the target policy so that it can access object that the delegator may not have been able to access (ie allows passing a path instead of an object).

Profile instance

A specific version of a profile, as profile can be changed by profile replacement, etc.

Derived profile

A profile that is derived from a base profile. This is usually used to refer to profile instances modified by delegation of stacking, but could also be applied to a modified profile loaded through profile replacement.

Context

A context is the full set of information used to mediate a given object. The context consists of a label and some other information depending on the object in consideration, there are different contexts for different types of objects (task context, file contexts, …). The context is a kernel based view of mediation, and is only partially exposed in userspace through the api.

Security Identifier (secid)

The secid is a unique number that maps to a label. For explict labels the secid is a predefined value that is part of the label definition. For implicit labels the secid is dynamically defined when the label is encountered for the first time.

Authority

Authority is the right/privilege to perform a given action. Authority, permission, and capabilities are often used interchangeably but can have different meanings depending on the definition that is used. Authority is often used as permission and capability have multiple definitions.

Permissions

1. The authority/right/privilege/capability to perform a given action.
2. The per Subject fine grained set of abilities that can be associated with a label. In apparmor this set of permissions can be unique on a per mediation type basis.

Identity

Who a subject or object is. For apparmor policy the Identity could be the security label, or an extended Identity that takes into account the user id as well as the security label.

Capability

1. The authority/right/privilege/capability to perform a given action.
2. The set of Posix capabilities available on the system that can be mediated with capability rules in policy.

Ambient Capability

A capability that is accessible from properties of the global system, it is not stored or carried on an object. Eg google document urls...

Object Capability

A capability/access right that is carried on/with an object.

Extending AppArmor with with Trusted processes

- X, Dbus

Extending Dbus to support AppArmor

Dbus is a userspace message bus that pretends to be object oriented, but in reality is just a structured message bus, with the bindings providing the object orientation.

Several fields as part of its addressing
policy in userspace vs kernel
messages go in and out of the kernel
apparmor reletionship diagrams

dbus provides the id of the task
- look up the task confinement
  - if task goes a way get peer sock labeling (why use task first? Socket could have tainting on it, maybe we should use only)
- do a permission check
    - policy loaded into the kernel
        - policy doesn't need to be in kernel but is there to provde a single point for policy lookup, and support AF_DBUS (kernel debus) if it happens
    - userspace could send the check to the kernel, or sync against kernel policy and do check in userspace
        - userspace check is more efficient, cache policy decisions in userspace
        - matching message contents vs. address probably won't be done in kernel
- do the operation

- dbus rules combine path access and label rules
    - don't need to test label at other end if only one task can bind to a give address
    - policy can be written as if the address is trusted, or untrusted by inserting a conditional label rule

dbus needs to be setup as a “trusted” app with a profile
- apps are then given permission to talk to dbus in their profile, this does not grant them privilege to talk over the dbus
    - the system bus, and session dbus can use different profiles (names), with the same basic contents, or the same profile (same profile is easiest)
- app profiles need dbus rules to indicate who they can talk to, what interfaces they can use, what interfaces they can bind to
    - these rules govern who/what can be comunicated with over the dbus
- policy is not configured as part of the dbus config file, it is included as part of the profile
- if a patched dbus is not used, dbus policy is ignored

- logging done to syslog

- delegation and dbus ?

```
A dbus address consists of
- socket/port that the task communicates with dbus over.   We ignore this in the
  dbus portion of the policy, but it is mediated by the file rules.
- connection/bus name.  Basically each new connection get a unique name like
   :34-907
  but a connection can acquire (bind) to a well known name (you only do this if
  you are providing a service)
   com.acme.Foo
- message/operation type.  method_call, signal, aquire, method_return, error
- object/path.  Yet another name for the "object" being communicated with.  They
  look like file paths.
   /com/acme/foo
- interface.  An object can support multiple interfaces, and message can specify
  which interface its intended for (this is only needed if two interface overlap).
  They look like connection/bus names
   com.acme.Foo
- method.  The name of method being called.  They look like
  ListName
- actual message pay load (we don't do anything with it currently)

And of course there are multiple buses (system and session).
```

X mediation
- uses Xace hook infrastructure
- uses labeling of structures based on task confinement
- when a task sets up window structure, Xace grabs labeling of tasks
    - when task labeling is unreliable, or unaviable use labeling on object
    - use sock getpeersec when necessary

- policy is based on label and capability
    - check label is correct
- check that capability (screen grab …) is in label capability

- logging done to syslog
- setting window colors via tag that the WM can interpret and assign color/name info to
- X should have its own loose profile so who can communicate with X separate from unconfined tasks can be controlled.
- X rules are then used to mediate what can be done in X, and who can be communicated with over X ipc

- delegation and X objects

Notes/Outline

explicit label
- check label rule
- check path rule if more specific
implicit label
- check label rule
- check path rule if more specific, augment label if access granted
invalid label
- task: try to update, if can't use stale label. For each confined profile
- subset test: use most recent version (name cmps and rcu off of replacedby if needed)
invalid profile in label, not yet invalid itself
- trigger subset matching similar to invalid label
delegation in label
- not stored on task context but object context
-
- doesn't actually change label, delegating a static label doesn't change it, neither should it for implicit
- patch order, don't store perm indexes, just for relookup, storing indexes is a later patch

invalid label with name change
- set flag indicating invalid and needs reordering
revoked label
- set flag indicating some perms have been revoked

static explicit labels – are just profiles, when encountered they are looked up and linked to a profile, if no profile is found an auto-remove profile is created and put in the unconfined state.
- static labels without rules == unconfined
- no special namespace
- any profile name can be written as a static label

How does stacking interact with explicit labels?
- can't set unless all profiles in the stack (within a namespace) agree
- can be compound but only if all profiles agree to write a label
  - if they don't it would change access perms
- profiles in and of themselves aren't static so how do we mark a label as static and not a profile?
  - can have a single/multiple entry label, pointing to a profile

- explicit labels take precedence
  - are the only part of a label considered for label rules, if no static full implicit is considered
  - can still cache profile accesses and delegation
  - double walk on explicit label
    - check if label is subset, on explicit and the implicit
    - if that doesn't pass check label rules
      - label without path (so only disconnected) ??? Do we want these
      - label at path
      - any label, want these as often the rule would be
        label=foo rw,
        this also lets us short circuit and avoid a path lookup
- need flag to say its an explicit label
   - should not be set on the “profile” label so it can be replaced, and the labels referencing it can still behave
     correctly
   -

- how does explicit label affect attachment?
  - specific label overrides
  - this affects which labels are checked in task checks

- overlapping path perm bit to let label rules know there is a more specific path rule that overrides, can drop for overlapping reads, but could provide deny, audit, more specific write

separate out design consideratio exposition (different choices etc to different section of document)

Multiple documents
- Over view document discussion policy, labeling etc
- technical documents discussing details
  - document on deriving labels from paths
  - aliases
  - access path and permission layouts
  -

Hrmm dealing with which profiles could apply to a file based on namespacing could be an issue. It will require at least a partially incremental build

Hrmm this needs to be presented at different level
- A global overview, profile, namespace, object, task, context, stack, delegation, permission check

A more specific view of the profile and attachment, delegation etc

Introduction
unconfined/confined
Profile
- Simple example

globbing/re
attachment
- globbing in profiles
Default profile
labeling
Namespaces

Policy composition
- stacking
- delegation
  - explicit delegation
  - implicit
  - object /rule
- Interaction of Stacking and Delegation

Under the hood – labeling
- explicit labeling
- implicit labeling
  - lazy evaluation
  - profile not enough, dfa state, permission
- object vs. handle labeling
- namespaces
- stacking
- delegation
- putting it all together

Aliasing and Namespaces

Misc Discussions, where to put them?

Alternative strategies to explicit labeling

update profiles with stacking like information, could be single global list with conditional rules, save off and update at next load. Maybe write it from an exploring options pov instead of alternative stategies

Explicit labeling has its disadvantages in that information must be stored on disk, which might not always be possible because of the filesystem or tools in use. There are two strategies that can be used to avoid the use
of explicit labels.

Segregation: force apps to only write to an exclusive location or at least a location that can be treated the same by implicit labeling.

Overlay: create a filesystem overlay that can store an application writes to a separate location but that appears to the application to have standard filesystem access. How does this work in with domain transitions??? Generally you don't want to transition domains unless you can also transition overlay.

On changing Implicit labels and orthogonality to explicit labeling

When an object (file, …) is moved/renamed, given an alias (hard link, bind mount, ..) this may change the implicit label on the object. This happens regardless of whether an object is just renamed within a file system, or the rename is a copy across to different filesystems. The labeling of a file will be consistent regardless of whether it has been renamed or the contents have been read, and then written to a new location.

This is in contrast to an explicit labeling system may keep the label on a file the same for a rename, but if its contents are read and written to the same location as what the rename would have been a different label if chosen based of the domain of the task doing the copying.

These properties are orthogonal, for implicit labeling to keep the label in the case of the rename, it needs to recognize a rename is happening, and modify the policy rules that provide the resultant label. For an explicit labeling system to ensure that the labeling at the target is always consistent it needs to have rule to recognize the change and adjust it.

In practice both properties have there uses dependent on the resource being accessed, and the type of policy being enforced. To achieve this AppArmor uses a hybrid system with both implicit and explicit labeling, which is more efficient than updating the policy dynamically. The type of policy will determine what type of labeling is dominant. While the AppArmor profile language favors implicit labeling, it would be possible to use an alternate language and compile it into an apparmor policy usable by the enforcement engine.

Shared write locations when data labeling should be unique

Implicit labeling provides a default labeling for all the files in the system. It works well when the set of files being mediated has known properties and locations, however it is less effective when a files labeling needs to be modified in response to an confined applications actions.

For security purposes file labeling may need to be modified beyond the default labeling when a confined application writes a file to a shared location. If the location is not shared then no special labeling is needed because a single profile encompasses all possible interactions, and if a file is not written/updated by application it can not change the file in a way that the labeling would need to be changed.

To handle the cases where there is a shared location (/tmp, users home) where the data written by one application but should not be able to be read/written by another application apparmor uses explicit labeling. An explicit label is a named label written to disk as part of the file object (using the security xattr field).

An explicit label on a file takes precedence over implicit labels, so that even if another profile has rules stating it can access a file at that location it can not unless it can access a file with that label.

Accessing labeled objects

setting labels
file move/rename – label changes done by kernel (keeping label, adding label, dropping label, changing label)

How to provide global rules
- write them in an include.
- need to discuss default profile, with unconfined
- hrmm need a separate doc discussing how to best author policy

Docs needed
- policy author guide – guide to writing policy, things to do and watch for
- core policy reference manual
- apparmor policy internals (this doc)
- techdoc – variables
- techdoc – policy layout
- techdoc – extending apparmor for trusted userspace daemons (dbus, Xace) mediation

implicit (implied, derived) label – a label that is implicit in the set of rules, and thus does not need to be written out. It is more dynamic and changeable than an explicit label. There are three ways to do implicit labels
- eager static: compute as profiles are loaded.
  - Requires updating existing objects
  - Requires mass relabel on dir move
- eager dynamic (compromise between eager and lazy): compute complete label at access
  - don't relabel objects on new profile loads
  - don't do mass relabel on dir, … moves
  - use revalidation
- lazy: compute label as profile accesses object.
  - More dynamic can result in partial implicit labels
  - use revalidation (relookup path, and find access).

This means all objects can be considered labeled, and label rules should work with them.
- a missing label rule means use the implicit label

An implicit label is the intersection of profiles and permissions. We do not need to store the permissions because they are implicit in the rule. So only the set of profiles is needed. Permissions can be “cached” to avoid having to relookup the access path (avoid disconnected paths), but should not be written to disk.

Implicit labels start based on the task “labeling” the handle, and maybe others from a set of labels the initial lookup is applied against.

There needs to be an explicit labeling that is equal to the implicit label.
  label=implied

There needs to be an explicit labeling that is equal to the task
  label=task
  label=inherit

label

There needs to be a generic label rule that covers implicit and explicit
label=**

How do label rules play into matching implicit labels, since implicit labels maybe stale or partial

How does handle labeling differ from object labeling?
- the handle label may be stale this is how we localize stale labels
- the handle provides access to objects that are no longer accessible via natural lookup unless revocation is used

Explicit labels are only created if a special label permission is used. What should that permission be?
- should be able to specify a label or take that tasks label, use a variable, ..

Information flow and labels?
- write end of a socket “labels” the packet
- read end, uses supplied label, get_peersec or applies rules to label a packet
- socket label becomes proxy for task. How does this work for explicit label?
  - get implicit and explit label?

Do we ever store permission as part of explict label derived from task type? No setting a label makes it no longer implicit.

The move/rename of a file does not result in loosing access to an object (there is no revocation of existing handles) but it is sufficient to stop new lookups. If revocation of object access is needed then profile revocation needs to be done.

i ???clean this reference up??? A Domain and Type Enforcement UNIX Prototype

Lee Badger
Daniel F. Sterne
David L. Sherman
Kenneth M. Walker
Sheila A. Haghighat
Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, Maryland 21738