AppArmor Developer Documentation

Kernel Notes

Table of Contents

AppArmor Documentation

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

Introduction

This is a collection of overviews notes, and explanations intended to help developers understand what is happening in the code. Higher level descriptions of AppArmor policy and use are available in other documents.

Temp place for think on this - Okay Conditional types and how they relate to permission queries
Compile time (none have dynamic effect at run time)
- access path
- profile
- namespace based
- global

Run time
- instance

  - subject based (task pid, uid, fsuid, gid, ...)
  - object based (file uid, security label, ...)
  - policy defined instance (maybe env var that is locked in place)
- parameter based (parameters in access path, not necessarily stored in subject or object)
- custom defined (external variable)

- profile scoped – policy defined var - like compile time variable but can be replaced at runtime independent of profile – equiv to reloading profile with different values for the variable. May not ever do, does not affect permision query as can be resolved at profile level

- namespace scoped – like profile scoped, except at namespace level. Updating would be equiv to replacing all profiles in namespace. Not now. Does not affect permission query as can be resolved at the profile level

- global scoped – like profile scoped, except over total policy. Updating would be equiv to replacing all profiles in all namespaces. Not likely to do. Does not affect permission query as can be resolved at the profile level

- access path variables – any of the above type encoded as part of the access path. Eg. pid in /proc/<pid>/
Resolution depends on the type it is from above.

Permission Query string
Permission query encoding requires feeding in a string that would be queried against the rules. When the kernel walks a query it may fill in certain parameters implicitly based off of subject or object variable values. Requests from userspace must provide these values instead.

How can userspace provide the necessary values, as part of the query.

Instantiation of profile?
The profile does not stand alone it is a set of rules and the full set of permissions granted for an access request may not be known without full context.

To instantiate profiles to provide this means we would need to instantiate both subject and object profiles but this is not enough as some permissions may be dependent on both subject and object variables (note subjects may also be objects in some cases but the set of variables does not change, a task is still a task even if its the object of a request). To solve this with instantiation the cross subject, object instantiations would have to be provided to a query.

What of partial instantiation (only instantiating the subject)?
This reduces the set that needs to be instantiated, but the instantiation must be per thread because of the @{pid} variable. And we must still provide the object for any query that could be conditional on the object

Major versions of AppArmor

Architecturally there have been four major revisions since Novell acquired and released AppArmor to the open source community. The early versions of AppArmor where only released as part of the Immunix Linux distro and not fully open sourced.

AppArmor 2.0.x – 2.0.1 (Linux kernels ???)

The apparmor 2.0 release was a minor cleanup and opensourcing of the Immunix version of AppArmor. It will not be discussed in detail, but a quick overview follows.

While Wirex/Immunix helped to create the Linux Security Module (LSM) infrastructure, it did not want its security module upstream for fear of its IP being stolen, and as such the LSM did not contain the hooks necessary for AppArmor's path based mediation. Instead Immunix used a minimum set of LSM hooks and patched the kernel to export d_path and the locking necessary to do some nasty vfsmount discovery.

However the vfsmount was not available in several LSM hooks so, a search of the namespace was done to find the vfsmount of the dentry. This wasn't racy due to the ns locking but could not guarantee a single correct vfsmount. It was a direct result of the above mentioned decisions about upstreaming.

Profiles where loaded into the kernel through an earlier version of the apparmorfs.

Profiles where attached to the task via the tasks security structure as with other LSMs.

Files where not labeled, every access to a file resulted in a new lookup, and permission check against the profiles access rules.

The rules of a profile where stored in a linked list. When a permission lookup was done, all rules in the list where iterated over and if conflicting permissions where found the match was aborted early and an error reported. A successful lookup always had to iterate all rules in the profile list.

There where three types of rules, exact match, tail glob, and pattern. This was an “optimization” done to avoid invoking the matching engine for each pathname to be matched. Exact match rules where matched using strcmp, tail glob rules with strncmp so that only the head of the patch had to match and pattern rules and to invoke the pattern matching engine.

The pattern matching engine in this version of was based on the pcre matching engine, though only a limited subset of it actual abilities where used. The matching engine was invoked for each rule in the text policy that had a pattern

glob in it. There was no rule or permission combining and it was very inefficient.

There was minimal compilation of profiles in this version of AppArmor. Permissions where converted into an integer mask, rules where copied directly, unless there was a pattern in which case the pcre engine was used to compile it and the pcre blob was added to the rule.

AppArmor 2.1 – 2.3 (Linux kernel 2.6.23+)

This revision of the AppArmor code saw two major changes, a full integration with the LSM and a new matching engine and policy compilation to go with it.

The vfs, and LSM where patched with a large series of patches to pass the vfsmount through to the various LSM hooks, thus getting rid of the need for the nasty vfsmount lookup in the 2.0 version. The patches where rejected multiple times upstream and this approach was eventually abandoned.

The policy matching and compiler saw the other large change. The pcre engine was replaced by a custom dfa engine and compiler. This was a simplified version of the current version so just an overview is provided here.

Permissions where still converted into an integer bitmask, but all rules where compiled into a single dfa. Permission collisions could be detected at compilation time and matches where now linear to the pathname (that is only one match needed to be done, and only against the characters in the looked up path).

File caching of profile

Encoding of perms

AppArmor 2.4 – 2.8 (Linux kernels 2.6.31+)

This version of AppArmor saw a major kernel rework to move the code base from the vfs patches to supporting creds and the new path_security hooks in the LSM.

Path hooks, creds

free_profile, kref_put

DFA
kzalloc

Locking

access path conditional encoding

free_profile, kref_put – recursion bug

AppArmor 3.0 – (Linux kernels 3.??+)

AppArmor 3 is a major extension of AppArmor's capability. It is built on top of the work done for AppArmor 2.4 – 2.8. Many of the basic design elements are carried forward, just being extended/modified where necessary.

Motivation/Use Cases for 3.0

1. The kernel structures, and LSM in areas map poorly to the task centric model, being set up more for label based approaches.
    - Networking
    - IPC (signals, semaphores, pipes, and other objects without names)
    - secids
2. Fix the overlapping write problem. When multiple profiles overlap their writes to a single location, apparmor has no way to distinguish and separate out ownership, this means overlap locations like /tmp/ and the users home directory can not be allowed when information must not be leaked between profiles.
3. Allow for containers and chroots to have their own policy separate from the system policy.
4. Allow users to have their own policy separate from the system policy.
5. Allow for policy to be more flexible, and tighter especially for desktop applications.
    - External file pickers
    - Tighter subprofiles
    - Better reuse of profile rules
    - Dynamic composition of profiles
6. Extend apparmor to support to trusted user services
    - dbus
    - X windows.
7. Fix the disconnected paths problem. Where objects are no longer visible because their namespace has gone out of scope.
8. Improve speed. Performing a path lookup for every mediation point can become quite expensive.

Design Requirements

- maintain as much backward compatibility as possible, recognizing that new mediation results in some semantic changes
    - requires profile name compatibility
    - general interface compatibility
        - single profile load
- isolate changes in one namespace from another as much as possible, so that policy changes in one namespace do not affect other namespaces
    - policy changes should not affect other namespaces

- improve performance
  - improve dfa matching speed
  - reduce name lookups and allocations
  - cache previous decisions where possible
  - reduce revalidations/new profile (label) updates on tasks and objects
    - minimize allocations in most mediation hooks
  - minimize locking contention
    - blocking many hooks is problematic
- reduce policy size
  - better dfa structure
  - shared dfa/policy
  - extend dfa capability
- extend mediation ability, make it more flexible
  - extend permission encoding
  - conditionals
  - variables
- allow for atomic policy (set) replacement
- support delegation of authority
- support stacking of policy
- support profile/policy operation
  - load, replace, renaming replace, removal, set profile on task, revocation
  - change_hat, change_profile, stack
- support profile invalidation
- support explicit object labeling
- improve interface
- extend policy to userspace helpers
- fix replaced-by chaining resource pinning
- fix null profile leak, allow auto cleanup of profiles and namespaces
- unified model (paths and explicit labeling)

Overview of techniques

- Namespaces – allows for having multiple sets of profiles that are independent from each other, and can provide a solution to the disconnected path problem.
- Labeling – changes how AppArmor interacts with the kernel. Providing the ability to cache profile information on objects, and even store information on disk. It helps address the kernel and LSM mismatch to the AppArmor model which enables extended mediation of networking, ipc, etc; can fix the overlapping write problem, provide a fix for the disconnected path problem. And improve performance.
- Stacking – an extension of labeling to the task that allows for multiple profiles to applied to a task in a more restricting fashion. It can be used to provide container support and improve policy reuse.
- Delegation – a technique to dynamically extend what is allowed by

confining policy.
- Extended conditionals – can be used in conjunction with labels to make rules conditional on labels.
- Instancing – creates different related labels based on different conditional values

Ref counting

In several places rcu and refcounting are combined, to provide lockless lookup of an object but also allow an object to exist beyond the rcu critical section. To get a refcount on object references that are rcu based aa_get_X_not0 needs to be used. This will return a null reference if the object has been put into an rcu callback for cleanup since the uncounted reference was obtained.

This should only be used when an rcu_read_lock is not sufficient.

Locking

Different locking strategies are used for each different list/tree. In general (except for secids) there is no global lock and locking for lists and trees are handled at the policy namespace level. Other objects may have additional locks beyond that.

The major locks are the namespace policy mutex locks for loading and replacing policy (profiles), the namespace based label tree locks, and the global sid table lock.

See the specific sections for more detail.

3. Policy struct

- parent structure of both profiles and policy namespaces (ns)
- mostly done to share name based lookup routines

4. Namespaces

- are protected by NS mutex_lock on each NS (used to be rwlock before list access went RCU)
- NS mutex_lock protects RCU profile list
- namespaces are ref counted
  - rcu ???
- NS names are similar to profile names
  - unique within a NS
  - NS can be hierarchical separated by //
  - always begin and end with ':'
- NS name is never shown unless the NS is a child of the current NS

- tasks always consider their NS to be the root and can not see NS's above their NS
- there is a true root NS that is setup during apparmor setup
- NS has a depth # that is used in ordering labels
- Auto Removal of Namespaces
  - most namespaces are pinned by a list refcount
    - if list refcount is not used namespace will auto remove itself when its refcount drops to 0
    - namespaces that are still in their list ?????
  - namespace is as pinned by profile back pointer that is refcounted

5. Profiles

Names

- // is the hierachy/dir path separator for profile names
- profile names can not end in /
- hname – hierarchical name. Profile name within a namespace including its parent.
  - eg. a child named bar of the profile foo has an hname of foo//bar
- name/base name – the profiles name without any of its parents
  - using the above example the base name of foo//bar is bar
- fqname – fully qualified name == namespace name + hname.
  - Note profiles in the root namespace don't specify the namespace name
  - eg. :ns://foo//bar

struct

- profile is a ref counted struct, there are cases when being accessed from an RCU list that a ref count need not be taken but those are special cases
  - the refcount is taken with a special version of aa_get_profile (aa_get_profile_not0) for profile pointers from the lockless read case
- profile is a label
  - label is embedded within the profile
  - label has a single vector entry pointing back to the profile
  - shares its refcount with label
- profiles are always stored as part of a NS
  - profiles may have their own children profiles that are part of the same namespace
- each profile in the NS has a unique name
  - children profiles can have the same name as a parent profile, because the parent is treated as a directory

    - if a profile exists in a sub-NS its name starts with the NS
        - NS names always start and end with ' :'
        - NS names are hierarchical like profiles
        - :ns//ns_child://profile//child//grand_child
- stored in RCU list protected by per NS lock
    - Each NS has its own list of profiles, and each profile has its own list for its children
    - all profiles and children are protected by the same NS lock. This is not as fined grained as could be but since locking is only used on updates of the policy it does not need to be.
    - RCU is used so replacement doesn't block exec, which could cause kernel dead lock
        - reader/writer locks used to be used but the updated aafs interface requires GFP_KERNEL allocation, which lead to sleeping in the rw lock.
        - RCU is better for our uses anyways as it is faster because hook read access is >> than writer access (profile replacement)
- most of the profile is static (RCU requires this), but there are a few pointers that point to objects that are updated in a RCU fashion (parent, ...), this is required to be able to profile updates

unconfined state

- flag carried on both profile and label
- used to short circuit rule evaluation (the label/profile has no rules and everything is allowed)
- is only set on a lable
    - if all its constituent profiles are also unconfined
    - or its an instantiation of a label, and has had the flag explicitly set.
-

unconfined profile

- Provided as the traditional default profile
- when a NS is created, a special unconfined profile is also created
- the unconfined profile does NOT enforce any rules (in unconfined state)
- can not be replaced, is not in the profile list
- the unconfined profile is used to track the current NS for the task.

default profile

- created as part of namespace
- Has name default
- Is replaceable (on profile list)
    - can not be an instance of unconfined so that it can be replaced
- Setup in unconfined state, but replacement does not need to be

Replacement/Loading

- profiles are always loaded by their fq name relative to the current namespace for the task doing the loading
  - profiles via their fq name are looked up and inserted into the current NS/child NS etc
  - this is an artifact of profiles always being loaded separately in the past
  - profiles specifying a NS that does NOT exist will result in the NS being created
  - profiles specifying a parent profile that does NOT exist will fail
  - profiles that are loaded together atomically succeed or fail, in the replacement/update
- profile replacement is two stage.
  - The new profile is loaded and added to the system
    - This is done in 3 phases after unpack to support atomic replacement of a set of profiles (ie. They all successfully load/replace or non of them do).
      - Unpack
      - Lock
        - check perm/set parent and base information
        - create files in fs etc (any allocations that could fail)
        - do actual replacement of profiles
          - this includes setting up the replacedby struct
- the old profile has its replaced-by updated, then tasks must update their own confinement information (this is checked on hook entry when getting the profile/confinement info)
  - this is necessary because creds make it so that a task is the only entity that can update the task cred (this avoids the need for any locking).
  - this means that old profiles may retain references to the old profile for a long time
  - only some hooks actually try to update the cred, other hooks (all that may be called from atomic context) merely find the replacement profile and use that without updating the cred (this avoids using GFP_ATOMIC and potential failures)
    - prepare_creds() uses GFP_KERNEL and does not provide the ability to specify the gfp
  - old profiles may also be referenced by file/object labels and so may exist as long as a file handle is held open
  - profile replacement does NOT revoke access to objects that has already been granted.
  - Profile replacement sets up a new “instance” of a profile
    - profile instances are can be compared against, or the whole class of profiles (see ???)
-

Renaming Replacement

- Replacement + a field set in the profile

- Merges replaced profile into the renamed profile + replaces the renamed profile if it exists
- Children conflicts
  - children in the new profile take precedence over inheriting children in the renamed profile
- replaceby struct of the renamed profile points to the new profile but the replacedby struct is not shared

Revocation (unimplemented)

- profile revocation is handled basically the same as profile replacement
  - profile is replaced with a new profile + a revoke flag
  - old profile is invalidated, and all labels using it are invalidated, and the revoke flag is also set
  - hooks that normally skip revalidation due to the profile being present in the label are forced to undergo revalidation, if revalidated the new profile from the replacement is added otherwise the revoked profile is dropped.
  - It can not revoke all access
  - ?????

Set_profile (unimplemented)

- Two possible implementations
  - via replacement
    - extend profile replacement so that it can be conditional upon a task or set of tasks
    - needs a list of possible outstanding replacements to search
    - problem: every task with the profile must check the conditional on every hook that can result in replacement
  - Task field
    - Needs field in task cxt to indicate replacement is needed on the task (rcu?)
    - needs a lock to update the field
    - task does its own replacement when it notices the field is set

6. Replacedby

- The replacedby struct is how profiles and labels find replacement versions of themselvesitself.
  - It makes finding a replacement constant time
  - does not have the memory problems of replacement chaining
  - All versions of a profile/label with active references share the same replacedby struct
  - replacedby is owned by the profile/label it points to
    - it will be freed by that profile when it is freed

    - replacedby holds a refcount to the profile/label that owns it, only when it is refcounted itself
    - the owning profile/label never refcounts the replacedby struct but does point to it
    - once the profile/label goes live is pointer to replacedby is never changed
    - when replacedby is updated the profile/label will gain a ref count to the replacedby that it no longer owns.
        - Note the only way that a replacedby can be freed is there is no refcount on the profile/label and no refcount on the replacedby struct.
- The replacedby struct is used instead of directly linking to the replacement profile to avoid profile chaining
    - profile chaining was used in v2 apparmor
    - with direct references an old version of a profile that still had a valid reference could result in multiple profile versions being held in memory (a chain from oldest to newest)
    - profile chains would get long enough to consume significant amounts of memory (1000+ long was possible)
    - long profile chains slowed down finding the newest version
    - profile chains required special handling when freeing a profile, because freeing a profile would result in the replacement profiles reference being put, which could in turn cause that profile to be freed
        - this could result in hundreds/thousands of profiles being freed at once
        - could blow the stack when put_profile was used from free_profile (kref resulted in all those free_profiles being recursive), hence the special loop handling.
- Profile lookup was NOT used when getting rid of direct replacement profile referencing (and hence replacement chaining) because profile lookup could not correctly handle renaming replacement, and removal (which is usually a special case of rename)
    - renaming replacement would result in a profile of a different name being inserted. This means the profile needed to know the new name to do a lookup
    - multiple renaming replacements would remove the intermediate replacement profile from the list so an old but valid reference could not find the new replacement unless
        - each profile in the replacement chain would need to know the current profile name. Either:
            - need a shared rename struct to get the newest name
            - each version of a profile still in memory would need a replacement name that would get updated by replacement.
                - This means updating multiple profiles on renaming replacement
                    - this is similar to what replacedby does except replacedby keeps a profile ref thus avoiding having to do a list lookup, that tracking the name this way would have to do

        - tracking the set of previous profiles that need updating
- currently there is a form of replacement chaining that is possible when renaming-replacement is used.
    - If renaming is used to rename an a profile to the name of an existing profile. This results in the two profiles being merged under the new name.
    - This means there are two separate replacedby structs
        - one on profile being renamed
        - one on existing profile with the specified name
    - the renamed profile replacedby struct is pointed to the profile with the new name
    - the existing profiles replacedby struct is updated to the new profile
    - If the new profile is replaced, then its a two step chain to update old profiles that where renamed.
        - They first migrate to the original renaming replacement profile
        - Then migrate to the current profile
    - If multiple renaming-replacements are done merging profiles into existing profiles a form of chaining occurs that can hold multiple profiles in memory
        - generally this situation should not occur
        - it will usually hold fewer profiles than old replacement chaining as intermediate replacement profiles can be freed, only the original renaming-replacement profile and the current version are pinned.
        - It does not suffer from blowing the stack problems, that replacement chains did

7. Rules

The rules being split from the profile is not implemented at this time

8. DFA/HFA

???

9. Labels

Locking pattern

```
Lock Read path
seqcount_begin {
  .. do work
} if (seqcount_retry()) {
  read_lock {
     .. do work
  } read_unlock
```

}

Lock Write path

```
write_lock {
  write_seqcount_begin {
    .. do work
  } write_seqcount_end
} write_unlock
```

Quick overview of rcu

- lockless readers
- readers must be able to handle “stale” data
- no in place updates, must copy struct and update pointer to new copy
  - in place atomic updates can be done, like update pointer with proper memory barrier
- pointer and data referenced is only good while in rcu_reader “lock” as the locks guarentee the quiescent period
- writer always need proper exclusion lock

Quick overview of seqlocks, seqcounts,

- seq locks are a lockless reader that prioritizes writes
- writers can live lock readers (cause them to spin forever)
- seqlocks are seqcounts with a spinlock embedded
- writer always takes a real lock, which increments the count as well
  - seqlock encapsulates this, seqcount you need write_seqcount + your lock
- seqlocks can NOT normally be used with pointers, because of the lockless nature, the pointer may change and the object it points to go away
  - if the pointers/objects are protected by rcu then they can be used safely within a seqlock for the rcu period
- seqlocks and seqcounts read pattern is usually

```
do {
  seqlock_begin
  …
} while (seqlock_retry);
```

- this actually spins in two places (seqlock_begin – will spin if a writer is active, do/while will spin if a writer fires while the reader was active

Labels provide the base mediation struct referenced by all subjects, and objects

- labels are dynamic and derived from profiles
  - labels consist of a vector (set) of profiles
  - TODO: (unimplemented) explicit labels are handled as a profile without rules
- labels need a text form compatible with profile format, so they can be reported on tasks via getprocattr
  - use //& as separator as it is not a valid profile name sequence

    - this is similar to the hat separator, so hat routines will need to verify // and not //&
- labels need a canonical form
    - so that they can be efficiently compared
    - so that label read from userspace can be easily converted to to internal form
    - sort labels via profile's namespace level, namespace name, profile name
        - namespaces are always hierarchical,
            - this allows easily excluding parts of label that are not visible from a subnamespace
        - groups profiles in the same namespace
        - groups namespaces
        - Note while namespaces must be hierarchical, labels do not need to be as objects can be shared across multiple namespaces. Eg parent shares file instance with separate tasks each in different children namespaces
            - when this happens the label will exist only a a single namespace tree. The tree the label exists in will depend on the canonical ordering of the profiles in the label.
            - the last profile in the sorted array of profiles (again canonical ordering) determine the ns and set the label is in
    - the root ns name can be excluded from the text label
        - from a users perspective root is the currently visible ns
        - from system perspective root is always the root ns, but internal system does NOT use text labels
    - eg. label /root/profile//&:subns://profile2//hat//&:subns://profile3//&:subns2://profile4
- labels are stored in per namespace trees. This prevents a child namespace from blocking its parent or siblings
- most objects hold valid ref counts to labels
    - labels where kept stable to minimize need for tree access, but invalidation does mean any path may do lookup
    - label lookups are primarily done by inserting new labels in tree via
        - an actual new label
        - merge of two labels which as a good chance of finding the merged label already existing
        - once the tree has stabilized from new additions access to it is dominated by lookups mostly from merge requests
    - merge is optimized to do a read path lookup before falling back to a write path lookup and insertion
- locking
    - label access from a valid refcount is comple lockless – no tree lookup required (labels are not updated except to set invalid flag)
        - if label is marked invalid this forces a lookup on the label tree for the namespace

- the locking of labels is complicated because of how import fast read and new label insertion (progress) are.
  - labels are refcounted because they must exist long term
    - refcount when obtained from refcounted pointer can use regular kref_get style refcount
    - refcount when obtained from lookup can not as, the tree does not refcount its objects
      - this is so that labels will auto delete from the tree on their last put
      - (not currently implemented) labels may be preseeded to the tree without a reference count, even from code or object. When this is done the refcount is -1 and the special get routine increments by 2 and then conditionally decs based on the original value == -1. This skips 0, which is not allowed by the conditional inc of the special get.
      - the rcu based pointer reference is incremented only if the count is not 0
        - if not 0 refcount proceeds as normal
        - if 0 the object has been removed from tree since rcu dereference and before refcount can be incremented, so reference is not returned
  - labels are stored in an rb_tree for fast lookup, using the canonical ordering of the label
    - profile pointers are stored in canonical order, this is leverage to speedup lookup
    - pointer comparisons are used to short circuit equivalence of label and profile before doing text comparisons
  - label updates come in three forms
    - insert new labels because of policy
    - inserting new labels into the tree from merges (this can happen on fd, inode etc access)
      - insertion is biased to look for an existing label first (locklessly), and then if needed do actual insertion.
    - Insert new labels because of invalidation/removal/renaming replaced-by
      - these types cause the label's canonical name to be changed so the canonical ordering needs to be reworked
      - this causes the creation of a new label with the invalidated profiles replaced/removed and resorted
      - invalidation may result in long lock times as multiple labels may need to be touched/marked invalid. Every entry of the label tree for the profile must be walked and all subns label trees.
      - the need for invalidation was reduced by making plain replacement not cause invalidation. This was done by splitting the profile into a name and rules. This also allows for the rule set to be shared by multiple profiles

- the label rb_tree is protected by a seqcount, rcu, and rwlocks to get mostly lockless lookups
  - seqcounts used instead of seq_locks as we already have an rwlock so we don't need the spinlock in the seq_lock
  - rb_trees can not be lockless based on rcu due to the type of update needed for rebalancing
  - rb_trees can not be lockless based on seqlocks/seqcounts due to use of pointers, as the object could disappear after dereferencing it.
  - rb_trees can be lockless based on a combination of seqlocks/seqcounts and rcu where rcu protects the read path object from disappearing and seqlocks/seqcounts handle the lockless state for the rb_tree update
  - A combination of seqcount + rwlock was choosen to prioritize readers but also allow writers to progress consistently. Readers get batched into a group and then a single writer gets to execute while the next reader group is batched. Readers can not starve writers and writers can not starve readers.
    - the seqlock gives priority to writers, but allows readers to progress locklessly as long as there are no writers.
    - a writer blocks readers (spin in seqcount or read_lock) and other writers.
    - readers and writers queue up on rwlock while writer works
    - as soon as writer release rwlock, queued up readers are given priority. While readers queued up on the rwlock are working any new readers, access via lockless seqlock (they don't queue up on the rwlock as any writers are blocked by readers working in rwlock, so seq_lock does not see a writer)
    - as soon as readers queued up on rwlock clear, the next writer gets to go as, no new readers where queued up on the rwlock. The process repeats as long as there are writers are queued up.
  - A spinlock could be used in place of the rwlock, this would serialize both readers and writers. This might be faster for a low number of cpus as only a couple threads could queue spinning anyways. But when you get two or three readers queuing up to wait, the rwlock overhead is not really anymore costly than spinning and then multiple readers can progress at once. Note: overall the extra cost of rwlock is amortized on the read path as most reads should be using the lockless path.
  - raw_seqcounts are used to avoid spinning on seqlock entry, if we are going to spin anywhere it might as well be the spin/rwlock. Though it doesn't really make much of a difference which is spin on.

Label instantiations (not implemented)

- allows a label to act as another label/profile
- allows the instantiated label to have a different name
- allows the label/profile to have its own name
- allows the instantiated label to have its own flags
    - used to carry explicit label flag, for a profile

Explicit Labeling (not implemented)

- stores a label on a persistent object
- name is same as any valid label name
- when an explicit label is encountered it is looked up in system loaded policy
    - if not found it is created
        - if its a compound label its constituent profiles that don't exist are created (they can be replaced)
        - label is created
        - instance of label with the explicit label tag is created.
    - Else if found
        - instance of label with the explicit label tag is created.

Label preseeding (not currently implemented)

- profile renaming-replacement, invalidation, removal can preseed the label tree with the labels needed by label lookups to update the invalidated labels
    - this makes all the needed allocations occur in the write (profile replacement) path instead of the reader lookup path
- preseeded labels have a reference count of -1
    - requires a special read path tree get documented above in label
- invalidated labels that are being freed (refcount is 0) are responsible for checking tree for preseeds that are unused.
    - That is the free computes what the set of labels that it would be valid replacements
    - looks them up in the tree
    - and if they have a preseed refcount of -1 “put” them so they can be cleaned up
        - this should be generally okay as no references left on the invalid label means it can not be causing lookups of its replacement(s)
        - this can cause the lookup path to have to recreate the preseeded label if multiple invalid labels result in the same preseed (should not be common)

10. Secids

- Sids need to be a global value

- sids are not and should not be exposed to userspace.
- sids do not have proper lifetime management
- sids use a tiered table much like a page table with upper bits being indexes into the root table, and lower bits indexing subtables
- sub sid tables are an array of label pointers index by the sid # (properly masked)
- each new label gets a unique sid
  - when a label is allocated it is given a sid but the sid does NOT backreference to the label
  - NULL is the default value for an allocated sid entry
  - the sid table only gains a reference to its label when it is actually used (get_sid?). This prevents sids from pinning labels if the sid is not used.
    - If a sid is used it gets a refer count to the label, and the label is pinned
    - there needs to be more work into recycling and using pinned sids/labels, but lifetime management issues of sids make this difficult, and it will never be perfect
    - removal, revocation etc could update the sid with a new value to unpin the label
      - removal – sid would have to remain allocated but unused (discarded)
      - renaming replacement – could point to new profile but that would pin that profile unless profiles gained the ability to have a list of sids (that are not the primary sid). Hrmm could be primary sid for renaming replace
      - revocation – same as renaming replace
- if a sid is freed it, it is added to the list of sids that are free.
  - The list is just a set of indexes with the sid value providing the next sid in the free list
- sid table is protected by a simple spinlock,
  - read access is an “rcu” dereference
    - sid if outstanding should always map to a valid label, or NULL
    - free list sids should not be index by any sid → label routine as all sids that where gotten are not freed (at this time anyway)
      - would be good if we could do it on profile or namespace removal but we can't guarantee that the sid isn't on an object some where

Tasks

- tasks confinement is carried by task_context, which contains the stack (task label), exec set_profile info and change_hat info
- every task has a context and is labeled, some tasks use the NS's unconfined profile label
- task stack (labeling) is used to track which NS the task is in (even for “unconfined” tasks)

- task handle updating their cred/task_context when getting their confinement as part of LSM hook entry (this is used for profile replacement)

task context (stacking)

- The task context stores information needed for domain changes
- task labels should be strictly namespace hierarchical
- only the current label is used for mediation
  - there is no current profile, change_profile affects all profiles in the current namespace
  - current namespace is determined by the last profile in the label index (labels_ns())
- onexec label is used to store delayed change_profile and stack requests
  - the stack and change_profile labels are computed at the time of the api call
  - ????? should we allow both stack and change_profile to be called, or just fail
- previous and token used by change_hat
- setuid
  - changes current what gets stored???? what gets cleared
- domain transitions cause the task context to be clear

File/inode/socket Labeling

- We label file object instead of inodes becomes profile replacement, mounts, unmounts, dir moves can cause pass permission changes in our model
  - we don't revoke current fd access for files that have changed permission for regular profile replacement (we do for revoke), moves, dir moves, mounts, unmounts
  - if we did implicit labeling on inodes we would have to monitor moves, mounts etc and update labeling on those ops. This is possible but there is no good solution for tree moves requiring mass relabeling
- We label inodes for explicit labels because these don't have the consistency problems of implicit label permissions, it also lets us track the explicit label better
- We label inodes for network operation because the sock uses the inode, and network inodes don't have the mass relabel problem that files have
- We need to update file labels for taint when a task has access added
  - to avoid doing this at file_ op LSM hooks we do it proactively at barriers (exec, fd passing)
  - we used to just lazily revalidate file labels when needed
  - We replace fds that point to files that are not allowed to essentially /dev/null because we can't reliably free the fd and expect the app to work
- file label needs to carry permissions and delegation

- delegation set when file open is tracked?, so it doesn't have to be looked up again
  - what if profile is added after file is opened – argument for lookup
  - what if file doesn't have name to relookup – argument for when opened/set and against lookup
  - set full delegation set, or only add as new domains encountered
    - learning is as new domains are encountered
      - potentially more flexible
      - has broader permissions in that not all possible delegation perms must pass intersection
      - probably lazy
    - how to track delegation?
      - Pointer to list of names
      - pointer to list of profiles
      - need to know which profile the delegation comes from
    - hrmmm can a task that has been delegated to also delegate
      - that is A delegates to B, but B has permission for file and it self has a delegation
        - probably shouldn't carry delegation for B
          - would result in smaller delegation lists
          - then B could add its own delegation, without special rules etc.
          - do we do transitive closure, or just first level, and use revalidation if delegation is more than first level
        - incremental additions to label would cause new lookups
  - add all delegations to label means we don't have to track a separate list
    -

DFA – mostly needs to move to another doc

- is stored in a different format that it is used in. It is remapped at load time to get an optimal alignment for the processor in use.
- Used to have permissions directly encoded via two 32 bit accept tables, has been moved to a single 16 bit accept index table
- the accept index table indexes into accept tables that are per type, not per profile (though it might be stored per profile, with types in different segments)
- the dfa can be shared between profiles
  - ????
- pre dfa - tree optimization
  - change to sort and merge routine
- dfa creation
  - do early deny permission removal (reduces states early)
  - don't do second table because only accept states cause the variation?

- Improvements to comb compression?

Owner conditional profiles and namespaces

?????

Setuid/Setguid transitions

?????

need to store off old label (much like change_hat)

need to remove conditional profiles from new label

Others todo ????
access path vs object conditional encoding

todo
Labeling

Oh! Revoke on rename???? Could revoke/remove labels from an object on rename, unlink

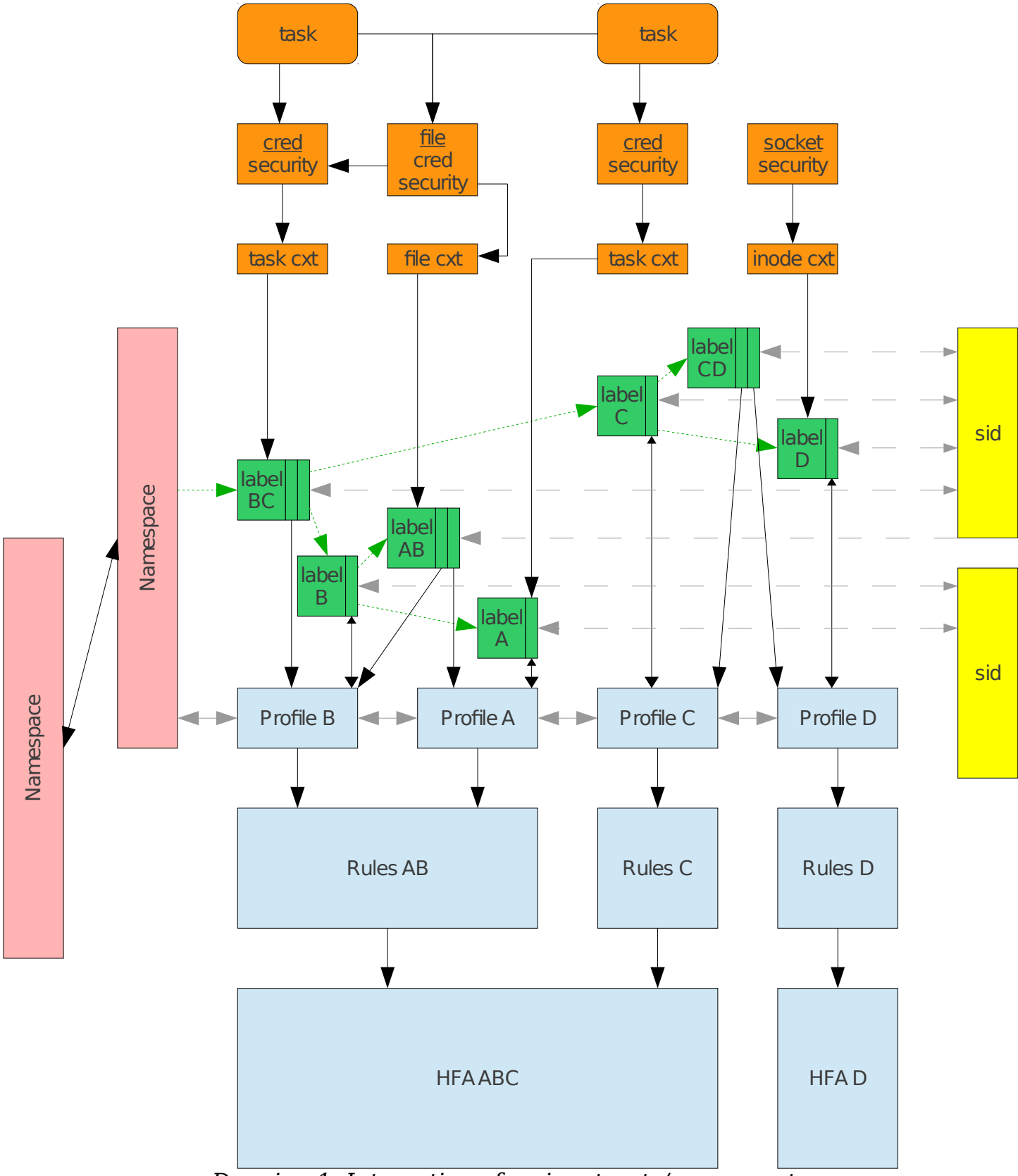
Interaction of the various components/structs and lifetimes

The interaction of the various components is fairly straight forward and a course overview is shown in ???.

There are a few important things to note

- Replacedby
  - the replacedby struct contains an rcu pointer to the current newest version of a profile
  - older versions of Profiles that still have references point to the shared replacedby struct
- Profiles:
  - profile reference obtained from walking the profile list must use the guarded (not0) version of aa_get_profile. This is because the writer can remove the profile and put the last reference between obtaining the profile reference and incrementing the count on the reference (see general pattern of rcu lookups and refcounting).
  - profiles on the profile list may or may not have a list reference count. If the list does not have a reference on the profile it will be removed when the last reference is put (this is used to auto cleanup null profiles). Again the guarded version of get reference prevent obtaining a bad profile pointer in this case.
  - profiles share a replacedby struct, that handles forwarding to the most

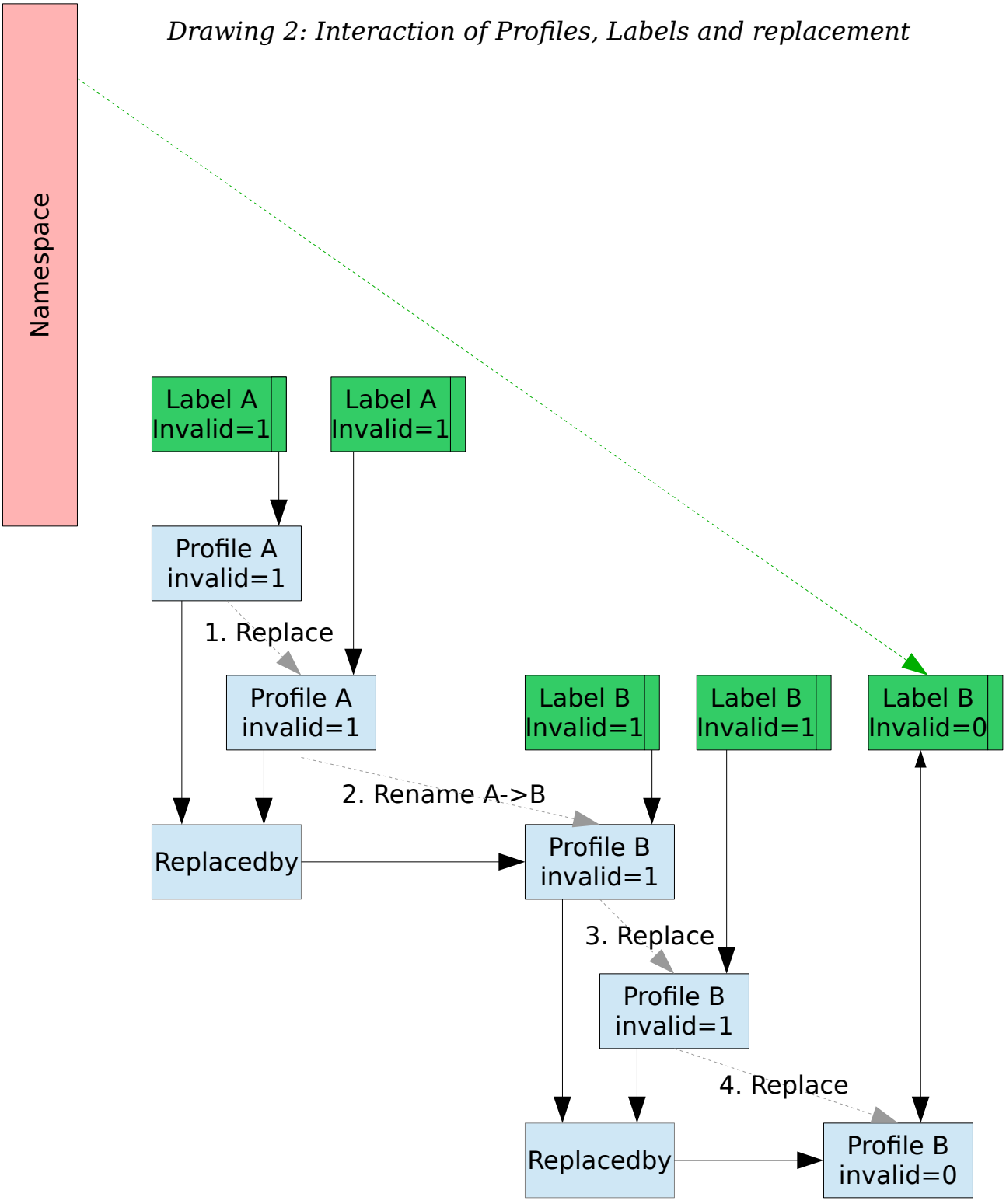
recent replacement.
        - It is only ever referenced if a profile has been marked invalid
    - profiles contain a back pointer to a single label that represents it. This pointer is refcounted (pinning the label in memory) as long as the profile is on the profile list. When the profile is removed from the list this reference is put to break the label → profile, profile → label cycle. This means the label will stay valid as long as the profile is on the list.
        - What of unconfined???? put it on the list too????
    - When the a label puts its last reference it will update the back reference in the profile setting it to NULL.
        - this is why that reference is __rcu and is required to aa_get_profile_not0
        - this will only happen after the profile has been removed from the profile list, other wise the profile is holding a reference to the label
        - the profile may still have references on it
    - A profiles reference to its rules can be replaced-by rcu so the rules must be used within an rcu critical section or get_guarded must be used to get a rules reference count
- Labels
    - labels hold references to their list of profiles and those references are not updated for the life of the label. When this reference is put it will NULL out the back reference in to the label in the profile.
    - Labels don't have a reference from their tree so that they can be auto removed when no longer referenced, this means get_guarded must be used on the reference obtain from a lockless lookup (see labels section for more)
    - labels when created allocate a sid #, this sid is valid for the life of the label
    - when a label is destroyed it frees its sid
        - Note if the sid has a reference to its label then the label is pinned and can not be removed
- Sids
    - Are a simple table of label references
    - Their label reference is NULL unless the sid is actually gotten/used and placed on a kernel object. At which point a refcounted label reference is stored



Drawing 1: Interaction of major structs/components

Replacement

Drawing 2: Interaction of Profiles, Labels and replacement



Policy Encoding

AppArmor policy is encoded in a architecturally independent binary format that is unpacked and verified at load time.

Encoding of Rule Paths

foo

Encoding of Conditionals

foo

Encoding of File Rule Paths

foo

Encoding of Network Rule Paths

foo

Encoding of Mount Rule Paths

foo

Encoding of the mount flags

foo

Encoding of DBus Rule Paths

foo

Encoding of X Window Rule Paths

foo

DFA/HFA

The DFA/HFA was first introduced in AppArmor 2.1 and has seen gradual improvements and extensions.

Format of the DFA/HFA

foo

PolicyDB the encoding of policy rules into the HFA

foo

Encoding permissions

foo

Putting it altogether relationship of Policy and DFA/HFA

foo

Steps in generating the DFA/HFA

foo