

NAME

apparmor.d – syntax of security profiles for AppArmor.

DESCRIPTION

AppArmor profiles describe mandatory access rights granted to given programs and are fed to the AppArmor policy enforcement module using **apparmor_parser** (8). This man page describes the format of the AppArmor configuration files; see **apparmor** (7) for an overview of AppArmor.

FORMAT

The following is a BNF-style description of AppArmor policy configuration files; see below for an example AppArmor policy file. AppArmor configuration files are line-oriented; **#** introduces a comment, similar to shell scripting languages. The exception to this rule is that **#include** will *include* the contents of a file inline to the policy; this behaviour is modelled after **cpp** (1).

PROFILE FILE = ([*PREAMBLE*] [*PROFILE*])*

PREAMBLE = (*COMMENT* | *VARIABLE ASSIGNMENT* | *ALIAS RULE* | *INCLUDE*)*

Variable assignment and alias rules must come before the profile.

VARIABLE ASSIGNMENT = *VARIABLE* ('=' | '+=') (space separated values)

VARIABLE = '@{ ' *ALPHA* [(*ALPHANUMERIC* | '_') ...] }'

ALIAS RULE = 'alias' *ABS PATH* '->' *REWRITTEN ABS PATH* ','

INCLUDE = ('#include' | 'include') ['if exists'] (*ABS PATH* | *MAGIC PATH*)

ABS PATH = "" path "" (the path is passed to **open** (2))

MAGIC PATH = '<' relative path '>'

The path is relative to */etc/apparmor.d/*.

COMMENT = '#' *TEXT* ['\r'] '\n'

TEXT = any characters

PROFILE = (*PROFILE HEAD*) [*ATTACHMENT SPECIFICATION*] [*PROFILE FLAG CONDS*] '{' (*RULES*)* '}'

PROFILE HEAD = ['profile'] *FILEGLOB* | 'profile' *PROFILE NAME*

PROFILE NAME (*UNQUOTED PROFILE NAME* | *QUOTED PROFILE NAME*)

QUOTED PROFILE NAME = "" *UNQUOTED PROFILE NAME* ""

UNQUOTED PROFILE NAME = (must start with alphanumeric character (after variable expansion), or '/' AARE have special meanings; see below. May include *VARIABLE*. Rules with embedded spaces or tabs must be quoted.)

ATTACHMENT SPECIFICATION = *FILEGLOB*

PROFILE FLAG CONDS = ['flags='] '(' comma or white space separated list of *PROFILE FLAGS* ')'

PROFILE FLAGS = 'complain' | 'audit' | 'enforce' | 'mediate_deleted' | 'attach_disconnected' | 'chroot_relative'

RULES = [(*LINE RULES* | *COMMA RULES* ',' | *BLOCK RULES*)

LINE RULES = (*COMMENT* | *INCLUDE*) ['\r'] '\n'

COMMA RULES = (*CAPABILITY RULE* | *NETWORK RULE* | *MOUNT RULE* | *PIVOT ROOT RULE* | *UNIX RULE* | *FILE RULE* | *LINK RULE* | *CHANGE_PROFILE RULE* | *RLIMIT RULE* | *DBUS RULE*)

BLOCK RULES = (*SUBPROFILE* | *HAT* | *QUALIFIER BLOCK*)

SUBPROFILE = 'profile' *PROFILE NAME* [*ATTACHMENT SPECIFICATION*] [*PROFILE FLAG CONDS*] '{' (*RULES*)* '}'

HAT = ('hat' | '^') *HATNAME* [*PROFILE FLAG CONDS*] '{' (*RULES*)* '}'

HATNAME = (must start with alphanumeric character. See **aa_change_hat** (2) for a description of how this "hat" is used. If '^' is used to start a hat then there is no space between the '^' and *HATNAME*)

QUALIFIER BLOCK = *QUALIFIERS BLOCK*

ACCESS TYPE = ('allow' | 'deny')

QUALIFIERS = ['audit'] [*ACCESS TYPE*]

CAPABILITY RULE = [*QUALIFIERS*] 'capability' [*CAPABILITY LIST*]

CAPABILITY LIST = (*CAPABILITY*)+

CAPABILITY = (lowercase capability name without 'CAP_' prefix; see **capabilities** (7))

NETWORK RULE = [*QUALIFIERS*] 'network' [*DOMAIN*] [*TYPE* | *PROTOCOL*]

DOMAIN = ('unix' | 'inet' | 'ax25' | 'ipx' | 'appletalk' | 'netrom' | 'bridge' | 'atmpvc' | 'x25' | 'inet6' | 'rose' | 'netbeui' | 'security' | 'key' | 'netlink' | 'packet' | 'ash' | 'econet' | 'atmsvc' | 'rds' | 'sna' | 'irda' | 'pppox' | 'wanpipe' | 'llc' | 'ib' | 'mpls' | 'can' | 'tipc' | 'bluetooth' | 'iucv' | 'rxrpc' | 'isdn' | 'phonet' | 'ieee802154' | 'caif' | 'alg' | 'nfc' | 'vsock' | 'kcm' | 'qipctr' | 'smc' | 'xdp') ','

TYPE = ('stream' | 'dgram' | 'seqpacket' | 'rdm' | 'raw' | 'packet')

PROTOCOL = ('tcp' | 'udp' | 'icmp')

MOUNT RULE = (*MOUNT* | *REMOUNT* | *UMOUNT*)

MOUNT = [*QUALIFIERS*] 'mount' [*MOUNT CONDITIONS*] [*SOURCE FILEGLOB*] ['->' [*MOUNTPPOINT FILEGLOB*]

REMOUNT = [*QUALIFIERS*] 'remount' [*MOUNT CONDITIONS*] *MOUNTPPOINT FILEGLOB*

UMOUNT = [*QUALIFIERS*] 'umount' [*MOUNT CONDITIONS*] *MOUNTPPOINT FILEGLOB*

MOUNT CONDITIONS = [('fstype' | 'vfstype') ('=' | 'in') *MOUNT FSTYPE EXPRESSION*] ['options' ('=' | 'in') *MOUNT FLAGS EXPRESSION*]

MOUNT FSTYPE EXPRESSION = (*MOUNT FSTYPE LIST* | *MOUNT EXPRESSION*)

MOUNT FSTYPE LIST = Comma separated list of valid filesystem and virtual filesystem types (eg ext4, debugfs, devfs, etc)

MOUNT FLAGS EXPRESSION = (*MOUNT FLAGS LIST* | *MOUNT EXPRESSION*)

MOUNT FLAGS LIST = Comma separated list of *MOUNT FLAGS*.

MOUNT FLAGS = ('ro' | 'rw' | 'nosuid' | 'suid' | 'nodev' | 'dev' | 'noexec' | 'exec' | 'sync' | 'async' | 'remount' | 'mand' | 'nomand' | 'dirsync' | 'noatime' | 'atime' | 'nodiratime' | 'diratime' | 'bind' | 'rbind' | 'move' | 'verbose' | 'silent' | 'loud' | 'acl' | 'noacl' | 'unbindable' | 'runbindable' | 'private' | 'rprivate' | 'slave' | 'rslave' | 'shared' | 'rshared' | 'relatime' | 'norelatime' | 'iversion' | 'noiversion' | 'strictatime' | 'nouser' | 'user')

MOUNT EXPRESSION = (*ALPHANUMERIC* | *AARE*) ...

PIVOT ROOT RULE = [*QUALIFIERS*] pivot_root [oldroot=*OLD PUT FILEGLOB*] [*NEW ROOT FILEGLOB*] ['->' *PROFILE NAME*]

SOURCE FILEGLOB = *FILEGLOB*

MOUNTPPOINT FILEGLOB = *FILEGLOB*

OLD PUT FILEGLOB = *FILEGLOB*

PTRACE_RULE = [*QUALIFIERS*] 'ptrace' [*PTRACE ACCESS PERMISSIONS*] [*PTRACE PEER*]

PTRACE ACCESS PERMISSIONS = *PTRACE ACCESS* | *PTRACE ACCESS LIST*

PTRACE ACCESS LIST = '(Comma or space separated list of *PTRACE ACCESS*)'

PTRACE ACCESS = ('r' | 'w' | 'rw' | 'read' | 'readby' | 'trace' | 'tracedby')

PTRACE PEER = 'peer' '=' *AARE*

SIGNAL_RULE = [*QUALIFIERS*] 'signal' [*SIGNAL ACCESS PERMISSIONS*] [*SIGNAL SET*] [*SIGNAL PEER*]

SIGNAL ACCESS PERMISSIONS = *SIGNAL ACCESS* | *SIGNAL ACCESS LIST*

SIGNAL ACCESS LIST = '(Comma or space separated list of *SIGNAL ACCESS*)'

SIGNAL ACCESS = ('r' | 'w' | 'rw' | 'read' | 'write' | 'send' | 'receive')

SIGNAL SET = 'set' '=' '(*SIGNAL LIST*)'

SIGNAL LIST = Comma or space separated list of *SIGNALS*

SIGNALS = ('hup' | 'int' | 'quit' | 'ill' | 'trap' | 'abrt' | 'bus' | 'fpe' | 'kill' | 'usr1' | 'segv' | 'usr2' | 'pipe' | 'alarm' | 'term' | 'stkflt' | 'chld' | 'cont' | 'stop' | 'stp' | 'ttin' | 'ttou' | 'urg' | 'xcpu' | 'xfsz' | 'vtalrm' | 'prof' | 'winch' | 'io' | 'pwr' | 'sys' | 'emt' | 'exists' | 'rtmin+0' ... 'rtmin+32')

SIGNAL PEER = 'peer' '=' *AARE*

DBUS RULE = (*DBUS MESSAGE RULE* | *DBUS SERVICE RULE* | *DBUS EAVESDROP RULE* | *DBUS COMBINED RULE*)

DBUS MESSAGE RULE = [*QUALIFIERS*] 'dbus' [*DBUS ACCESS EXPRESSION*] [*DBUS BUS*] [*DBUS PATH*] [*DBUS INTERFACE*] [*DBUS MEMBER*] [*DBUS PEER*]

DBUS SERVICE RULE = [*QUALIFIERS*] 'dbus' [*DBUS ACCESS EXPRESSION*] [*DBUS BUS*] [*DBUS NAME*]

DBUS EAVESDROP RULE = [*QUALIFIERS*] 'dbus' [*DBUS ACCESS EXPRESSION*] [*DBUS BUS*]

DBUS COMBINED RULE = [*QUALIFIERS*] 'dbus' [*DBUS ACCESS EXPRESSION*] [*DBUS BUS*]

DBUS ACCESS EXPRESSION = (*DBUS ACCESS* | '(*DBUS ACCESS LIST*)')

DBUS BUS = 'bus' '=' '('system' | 'session' | ''' *AARE* ''' | *AARE*)'

DBUS PATH = 'path' '=' '(''' *AARE* ''' | *AARE*)'

DBUS INTERFACE = 'interface' '=' '(''' *AARE* ''' | *AARE*)'

DBUS MEMBER = 'member' '=' '(''' *AARE* ''' | *AARE*)'

DBUS PEER = 'peer' '=' '([*DBUS NAME*] [*DBUS LABEL*])'

DBUS NAME = 'name' '=' '(''' *AARE* ''' | *AARE*)'

DBUS LABEL = 'label' '=' '(''' *AARE* ''' | *AARE*)'

DBUS ACCESS LIST = Comma separated list of *DBUS ACCESS*

DBUS ACCESS = ('send' | 'receive' | 'bind' | 'eavesdrop' | 'r' | 'read' | 'w' | 'write' | 'rw')

Some accesses are incompatible with some rules; see below.

AARE = *?*[]{}^*

See below for meanings.

UNIX RULE = [*QUALIFIERS*] 'unix' [*UNIX ACCESS EXPR*] [*UNIX RULE CONDS*] [*UNIX LOCAL EXPR*] [*UNIX PEER EXPR*]

UNIX ACCESS EXPR = (*UNIX ACCESS* | *UNIX ACCESS LIST*)

UNIX ACCESS = ('create' | 'bind' | 'listen' | 'accept' | 'connect' | 'shutdown' | 'getattr' | 'setattr' | 'getopt' | 'setopt' | 'send' | 'receive' | 'r' | 'w' | 'rw')

Some access modes are incompatible with some rules or require additional parameters.

UNIX ACCESS LIST = '(*UNIX ACCESS* ([,] *UNIX ACCESS*) *)'

UNIX RULE CONDS = (*TYPE COND* | *PROTO COND*)

Each cond can appear at most once.

TYPE COND = 'type' '=' (*AARE* | '('"" *AARE* ""' | *AARE*) +)'

PROTO COND = 'protocol' '=' (*AARE* | '('"" *AARE* ""' | *AARE*) +)'

UNIX LOCAL EXPR = (*UNIX ADDRESS COND* | *UNIX LABEL COND* | *UNIX ATTR COND* | *UNIX OPT COND*)*

Each cond can appear at most once.

UNIX PEER EXPR = 'peer' '=' (*UNIX ADDRESS COND* | *UNIX LABEL COND*) +

Each cond can appear at most once.

UNIX ADDRESS COND 'addr' '=' (*AARE* | '('"" *AARE* ""' | *AARE*))'

UNIX LABEL COND 'label' '=' (*AARE* | '('"" *AARE* ""' | *AARE*))'

UNIX ATTR COND 'attr' '=' (*AARE* | '('"" *AARE* ""' | *AARE*))'

UNIX OPT COND 'opt' '=' (*AARE* | '('"" *AARE* ""' | *AARE*))'

RLIMIT RULE = 'set' 'rlimit' [*RLIMIT* '<=' *RLIMIT VALUE*]

RLIMIT = ('cpu' | 'fsize' | 'data' | 'stack' | 'core' | 'rss' | 'nofile' | 'ofile' | 'as' | 'nproc' | 'memlock' | 'locks' | 'sigpending' | 'msgqueue' | 'nice' | 'rtprio' | 'rttime')

RLIMIT VALUE = (*RLIMIT SIZE* | *RLIMIT NUMBER* | *RLIMIT TIME* | *RLIMIT NICE*)

RLIMIT SIZE = *NUMBER* ('K' | 'M' | 'G')

Only applies to RLIMIT of 'fsize', 'data', 'stack', 'core', 'rss', 'as', 'memlock', 'msgqueue'.

RLIMIT NUMBER = number from 0 to max rlimit value.

Only applies to RLIMIT of 'ofile', 'nofile', 'locks', 'sigpending', 'nproc', 'rtprio'.

RLIMIT TIME = *NUMBER* ('us' | 'microsecond' | 'microseconds' | 'ms' | 'millisecond' | 'milliseconds' | 's' | 'sec' | 'second' | 'seconds' | 'min' | 'minute' | 'minutes' | 'h' | 'hour' | 'hours' | 'd' | 'day' | 'days' | 'week' | 'weeks')

Only applies to RLIMIT of 'cpu' and 'rttime'. RLIMIT 'cpu' only allows units >= 'seconds'.

RLIMIT NICE = a number between -20 and 19.

Only applies to RLIMIT of 'nice'.

FILE RULE = [*QUALIFIERS*] ['owner'] ('file' | ['file'] (*FILEGLOB ACCESS* | *ACCESS FILEGLOB*) ['->' *EXEC TARGET*])

FILEGLOB = (*QUOTED FILEGLOB* | *UNQUOTED FILEGLOB*)

QUOTED FILEGLOB = '"" *UNQUOTED FILEGLOB* ""'

UNQUOTED FILEGLOB = (must start with '/' (after variable expansion), **AARE** have special meanings; see below. May include *VARIABLE*. Rules with embedded spaces or tabs must be quoted. Rules must end with '/' to apply to directories.)

ACCESS = ('r' | 'w' | 'a' | 'l' | 'k' | 'm' | *EXEC TRANSITION*) + (not all combinations are allowed; see below.)

EXEC TRANSITION = ('ix' | 'ux' | 'Ux' | 'px' | 'Px' | 'cx' | 'Cx' | 'pix' | 'Pix' | 'cix' | 'Cix' | 'pux' | 'PUx' | 'cux' | 'CUx' | 'x')

A bare 'x' is only allowed in rules with the deny qualifier, everything else only without the deny

qualifier.

EXEC TARGET = name

Requires *EXEC TRANSITION* specified.

LINK RULE = *QUALIFIERS* ['owner'] 'link' ['subset'] *FILEGLOB* '->' *FILEGLOB*

ALPHA = ('a', 'b', 'c', ... 'z', 'A', 'B', ... 'Z')

ALPHANUMERIC = ('0', '1', '2', ... '9', 'a', 'b', 'c', ... 'z', 'A', 'B', ... 'Z')

CHANGE_PROFILE RULE = 'change_profile' [[*EXEC MODE*] *EXEC COND*] ['->' *PROFILE NAME*]

EXEC_MODE = ('safe' | 'unsafe')

EXEC COND = *FILEGLOB*

All resources and programs need a full path. There may be any number of subprofiles (aka child profiles) in a profile, limited only by kernel memory. Subprofile names are limited to 974 characters. Child profiles can be used to confine an application in a special way, or when you want the child to be unconfined on the system, but confined when called from the parent. Hats are a special child profile that can be used with the **aa_change_hat**(2) API call. Applications written or modified to use **aa_change_hat**(2) can take advantage of subprofiles to run under different confinements, dependent on program logic. Several **aa_change_hat**(2)-aware applications exist, including an Apache module, **mod_apparmor**(5); a PAM module, **pam_apparmor**; and a Tomcat valve, **tomcat_apparmor**. Applications written or modified to use **change_profile**(2) transition permanently to the specified profile. **libvirt** is one such application.

Access Modes

File permission access modes consists of combinations of the following modes:

- r** – read
- w** – write — conflicts with append
- a** – append — conflicts with write
- ux** – unconfined execute
- Ux** – unconfined execute — scrub the environment
- px** – discrete profile execute
- Px** – discrete profile execute — scrub the environment
- cx** – transition to subprofile on execute
- Cx** – transition to subprofile on execute — scrub the environment
- ix** – inherit execute
- pix** – discrete profile execute with inherit fallback
- Pix** – discrete profile execute with inherit fallback — scrub the environment
- cix** – transition to subprofile on execute with inherit fallback
- Cix** – transition to subprofile on execute with inherit fallback — scrub the environment
- puX** – discrete profile execute with fallback to unconfined
- PUx** – discrete profile execute with fallback to unconfined — scrub the environment
- cux** – transition to subprofile on execute with fallback to unconfined
- CUx** – transition to subprofile on execute with fallback to unconfined — scrub the environment
- deny x** – disallow execute (in rules with the deny qualifier)
- m** – allow **PROT_EXEC** with **mmap**(2) calls

l – link

k – lock

Access Modes Details

r – Read mode

Allows the program to have read access to the file or directory listing. Read access is required for shell scripts and other interpreted content.

w – Write mode

Allows the program to have write access to the file. Files and directories must have this permission if they are to be unlinked (removed.) Write mode is not required on a directory to rename or create files within the directory.

This mode conflicts with append mode.

a – Append mode

Allows the program to have a limited appending only write access to the file. Append mode will prevent an application from opening the file for write unless it passes the `O_APPEND` parameter flag on open.

The mode conflicts with Write mode.

ux – Unconfined execute mode

Allows the program to execute the program without any AppArmor profile being applied to the program.

This mode is useful when a confined program needs to be able to perform a privileged operation, such as rebooting the machine. By placing the privileged section in another executable and granting unconfined execution rights, it is possible to bypass the mandatory constraints imposed on all confined processes. For more information on what is constrained, see the **apparmor** (7) man page.

WARNING 'ux' should only be used in very special cases. It enables the designated child processes to be run without any AppArmor protection. 'ux' does not scrub the environment of variables such as `LD_PRELOAD`; as a result, the calling domain may have an undue amount of influence over the callee. Use this mode only if the child absolutely must be run unconfined and `LD_PRELOAD` must be used. Any profile using this mode provides negligible security. Use at your own risk.

Incompatible with other exec transition modes and the deny qualifier.

Ux – unconfined execute — scrub the environment

'Ux' allows the named program to run in 'ux' mode, but AppArmor will invoke the Linux Kernel's **unsafe_exec** routines to scrub the environment, similar to setuid programs. (See **ld.so** (8) for some information on setuid/setgid environment scrubbing.)

WARNING 'Ux' should only be used in very special cases. It enables the designated child processes to be run without any AppArmor protection. Use this mode only if the child absolutely must be run unconfined. Use at your own risk.

Incompatible with other exec transition modes and the deny qualifier.

px – Discrete Profile execute mode

This mode requires that a discrete security profile is defined for a program executed and forces an AppArmor domain transition. If there is no profile defined then the access will be denied.

WARNING 'px' does not scrub the environment of variables such as `LD_PRELOAD`; as a result, the calling domain may have an undue amount of influence over the callee.

Incompatible with other exec transition modes and the deny qualifier.

Px – Discrete Profile execute mode — scrub the environment

'Px' allows the named program to run in 'px' mode, but AppArmor will invoke the Linux Kernel's **unsafe_exec** routines to scrub the environment, similar to setuid programs. (See **ld.so** (8) for some information on setuid/setgid environment scrubbing.)

Incompatible with other exec transition modes and the deny qualifier.

cx – Transition to Subprofile execute mode

This mode requires that a local security profile is defined and forces an AppArmor domain transition to the named profile. If there is no profile defined then the access will be denied.

WARNING 'cx' does not scrub the environment of variables such as LD_PRELOAD; as a result, the calling domain may have an undue amount of influence over the callee.

Incompatible with other exec transition modes and the deny qualifier.

Cx – Transition to Subprofile execute mode — scrub the environment

'Cx' allows the named program to run in 'cx' mode, but AppArmor will invoke the Linux Kernel's **unsafe_exec** routines to scrub the environment, similar to setuid programs. (See **ld.so**(8) for some information on setuid/setgid environment scrubbing.)

Incompatible with other exec transition modes and the deny qualifier.

ix – Inherit execute mode

Prevent the normal AppArmor domain transition on **execve**(2) when the profiled program executes the named program. Instead, the executed resource will inherit the current profile.

This mode is useful when a confined program needs to call another confined program without gaining the permissions of the target's profile, or losing the permissions of the current profile. There is no version to scrub the environment because 'ix' executions don't change privileges.

Incompatible with other exec transition modes and the deny qualifier.

Profile transition with inheritance fallback execute mode

These modes attempt to perform a domain transition as specified by the matching permission (shown below) and if that transition fails to find the matching profile the domain transition proceeds using the 'ix' transition mode.

```
'Pix' == 'Px' with fallback to 'ix'
'pix' == 'px' with fallback to 'ix'
'Cix' == 'Cx' with fallback to 'ix'
'cix' == 'cx' with fallback to 'ix'
```

Incompatible with other exec transition modes and the deny qualifier.

Profile transition with unconfined fallback execute mode

These modes attempt to perform a domain transition as specified by the matching permission (shown below) and if that transition fails to find the matching profile the domain transition proceeds using the 'ux' transition mode if 'pux', 'cux' or the 'Ux' transition mode if 'PUx', 'CUx' is used.

```
'PUx' == 'Px' with fallback to 'Ux'
'pux' == 'px' with fallback to 'ux'
'CUx' == 'Cx' with fallback to 'Ux'
'cux' == 'cx' with fallback to 'ux'
```

Incompatible with other exec transition modes and the deny qualifier.

deny x – Deny execute

For rules including the deny modifier, only 'x' is allowed to deny execute.

The 'ix', 'Px', 'px', 'Cx', 'cx' and the fallback modes conflict with the deny modifier.

Directed profile transitions

The directed ('px', 'Px', 'pix', 'Pix', 'pux', 'PUx') profile and subprofile ('cx', 'Cx', 'cix', 'Cix', 'cux', 'CUx') transitions normally determine the profile to transition to from the executable name. It is however possible to specify the name of the profile that the transition should use.

The name of the profile to transition to is specified using the '->' followed by the name of the profile to transition to. Eg.

```
/bin/** px -> profile,
```

Incompatible with other exec transition modes.

m – Allow executable mapping

This mode allows a file to be mapped into memory using **mmap**(2)'s PROT_EXEC flag. This flag marks the pages executable; it is used on some architectures to provide non-executable data pages, which can complicate exploit attempts. AppArmor uses this mode to limit which files a well-behaved program (or all programs on architectures that enforce non-executable memory access controls) may use as libraries, to limit the effect of invalid **-L** flags given to **ld**(1) and **LD_PRELOAD**, **LD_LIBRARY_PATH**, given to **ld.so**(8).

l – Link mode

Allows the program to be able to create a link with this name. When a link is created, the new link **MUST** have a subset of permissions as the original file (with the exception that the destination does not have to have link access.) If there is an 'x' rule on the new link, it must match the original file exactly.

k – lock mode

Allows the program to be able lock a file with this name. This permission covers both advisory and mandatory locking.

leading OR trailing access permissions

File rules can be specified with the access permission either leading or trailing the file glob. Eg.

```
rw /**,                # leading permissions
/** rw,                # trailing permissions
```

When leading permissions are used further rule options and context may be allowed, Eg.

```
l /foo -> /bar,        # lead 'l' link permission is equivalent to link rules
```

Link rules

Link rules allow specifying permission to form a hard link as a link target pair. If the subset condition is specified then the permissions to access the link file must be a subset of the profiles permissions to access the target file. If there is an 'x' rule on the new link, it must match the original file exactly.

Eg.

```
/file1 r,
/file2 rwk,
/link* rw,
link subset /link* -> /**,
```

The link rule allows linking of /link to both /file1 or /file2 by name however because the /link file has 'rw' permissions it is not allowed to link to /file1 because that would grant an access path to /file1 with more permissions than the 'r' permissions the profile specifies.

A link of /link to /file2 would be allowed because the 'rw' permissions of /link are a subset of the 'rwk' permissions for /file1.

The link rule is equivalent to specifying the 'l' link permission as a leading permission with no other file access permissions. When this is done the link rule options can be specified.

The following link rule is equivalent to the 'l' permission file rule

```
link /foo -> bar,
l /foo -> /bar,
```

File rules that specify the 'l' permission and don't specify the extend link permissions map to link rules as follows.


```

/foo l,
l /foo,
link subset /foo -> /**,

```

Comments

Comments start with # and may begin at any place within a line. The comment ends when the line ends. This is the same comment style as shell scripts.

Capabilities

The only capabilities a confined process may use may be enumerated; for the complete list, please refer to **capabilities**(7). Note that granting some capabilities renders AppArmor confinement for that domain advisory; while **open**(2), **read**(2), **write**(2), etc., will still return error when access is not granted, some capabilities allow loading kernel modules, arbitrary access to IPC, ability to bypass discretionary access controls, and other operations that are typically reserved for the root user.

Network Rules

AppArmor supports simple coarse grained network mediation. The network rule restrict all **socket**(2) based operations. The mediation done is a coarse grained check on whether a socket of a given type and family can be created, read, or written. There is no mediation based of port number or protocol beyond tcp, udp, and raw. Network **netlink**(7) rules may only specify type 'dgram' and 'raw'.

AppArmor network rules are accumulated so that the granted network permissions are the union of all the listed network rule permissions.

AppArmor network rules are broad and general and become more restrictive as further information is specified.

eg.

```

network,                #allow access to all networking
network tcp,            #allow access to tcp
network inet tcp,       #allow access to tcp only for inet4 addresses
network inet6 tcp,      #allow access to tcp only for inet6 addresses
network netlink raw,    #allow access to AF_NETLINK SOCK_RAW

```

Mount Rules

AppArmor supports mount mediation and allows specifying filesystem types and mount flags. The syntax of mount rules in AppArmor is based on the **mount**(8) command syntax. Mount rules must contain one of the mount, remount or umount keywords, but all mount conditions are optional. Unspecified optional conditionals are assumed to match all entries (eg, not specifying fstype means all fstypes are matched). Due to the complexity of the mount command and how options may be specified, AppArmor allows specifying conditionals three different ways:

1. If a conditional is specified using '=', then the rule only grants permission for mounts matching the exactly specified options. For example, an AppArmor policy with the following rule:

```
mount options=ro /dev/foo -E<gt> /mnt/,
```

Would match:

```
$ mount -o ro /dev/foo /mnt
```

but not either of these:

```
$ mount -o ro,atime /dev/foo /mnt
```

```
$ mount -o rw /dev/foo /mnt
```

2. If a conditional is specified using 'in', then the rule grants permission for mounts matching any combination of the specified options. For example, if an AppArmor policy has the following rule:

```
mount options in (ro,atime) /dev/foo -> /mnt/,
```

all of these mount commands will match:

```
$ mount -o ro /dev/foo /mnt
```

```
$ mount -o ro,atime /dev/foo /mnt
```

```
$ mount -o atime /dev/foo /mnt
```

but none of these will:

```
$ mount -o ro,sync /dev/foo /mnt
```

```
$ mount -o ro,atime,sync /dev/foo /mnt
```

```
$ mount -o rw /dev/foo /mnt
```

```
$ mount -o rw,noatime /dev/foo /mnt
```

```
$ mount /dev/foo /mnt
```

3. If multiple conditionals are specified in a single mount rule, then the rule grants permission for each set of options. This provides a shorthand when writing mount rules which might help to logically break up a conditional. For example, if an AppArmor policy has the following rule:

```
mount options=ro options=atime
```

both of these mount commands will match:

```
$ mount -o ro /dev/foo /mnt
```

```
$ mount -o atime /dev/foo /mnt
```

but this one will not:

```
$ mount -o ro,atime /dev/foo /mnt
```

Note that separate mount rules are distinct and the options do not accumulate. For example, these AppArmor mount rules:

```
mount options=ro,
```

```
mount options=atime,
```

are not equivalent to either of these mount rules:

```
mount options=(ro,atime),
```

```
mount options in (ro,atime),
```

To help clarify the flexibility and complexity of mount rules, here are some example rules with accompanying matching commands:

mount,

the 'mount' rule without any conditionals is the most generic and allows any mount. Equivalent to 'mount fstype=** options=** ** -> /**'.

mount /dev/foo,

allow mounting of /dev/foo anywhere with any options. Some matching mount commands:

```
$ mount /dev/foo /mnt
```

```
$ mount -t ext3 /dev/foo /mnt
```

```
$ mount -t vfat /dev/foo /mnt
```

```
$ mount -o ro,atime,noexec,nodiratime /dev/foo /srv/some/mountpoint
```

mount options=ro /dev/foo,

allow mounting of /dev/foo anywhere, as read only. Some matching mount commands:

```
$ mount -o ro /dev/foo /mnt
```

```
$ mount -o ro /dev/foo /some/where/else
```

mount options=(ro,atime) /dev/foo,

allow mount of /dev/foo anywhere, as read only and using inode access times. Some matching mount commands:

```
$ mount -o ro,atime /dev/foo /mnt
```

```
$ mount -o ro,atime /dev/foo /some/where/else
```

mount options in (ro,atime) /dev/foo,

allow mount of /dev/foo anywhere using some combination of 'ro' and 'atime' (see above). Some matching mount commands:

```
$ mount -o ro /dev/foo /mnt
```

```
$ mount -o atime /dev/foo /some/where/else
```

```
$ mount -o ro,atime /dev/foo /some/other/place
```

mount options=ro /dev/foo, mount options=atime /dev/foo,

allow mount of /dev/foo anywhere as read only, and allow mount of /dev/foo anywhere using inode access times. Note this is expressed as two different rules. Matches:

```
$ mount -o ro /dev/foo /mnt/1
```

```
$ mount -o atime /dev/foo /mnt/2
```

mount -> /mnt/,**

allow mounting anything under a directory in /mnt/**. Some matching mount commands:

```
$ mount /dev/foo1 /mnt/1
```

```
$ mount -o ro,atime,noexec,nodiratime /dev/foo2 /mnt/deep/path/foo2
```

mount options=ro -> /mnt/,**

allow mounting anything under /mnt/**, as read only. Some matching mount commands:

```
$ mount -o ro /dev/foo1 /mnt/1
```

```
$ mount -o ro /dev/foo2 /mnt/deep/path/foo2
```

mount fstype=ext3 options=(rw,atime) /dev/sdb1 -> /mnt/stick/,

allow mounting an ext3 filesystem in /dev/sdb1 on /mnt/stick as read/write and using inode access times. Matches only:

```
$ mount -o rw,atime /dev/sdb1 /mnt/stick
```

mount options=(ro, atime) options in (nodev, user) /dev/foo -> /mnt/,

allow mounting /dev/foo on /mnt/ read only and using inode access times or allow mounting /dev/foo on /mnt/ with some combination of 'nodev' and 'user'. Matches only:

```
$ mount -o ro,atime /dev/foo /mnt
```

```
$ mount -o nodev /dev/foo /mnt
```

```
$ mount -o user /dev/foo /mnt
```

```
$ mount -o nodev,user /dev/foo /mnt
```

Pivot Root Rules

AppArmor mediates changing of the root filesystem through the **pivot_root(2)** system call. The syntax of 'pivot_root' rules in AppArmor is based on the **pivot_root(2)** system call parameters with the notable exception that the ordering is reversed. The path corresponding to the `put_old` parameter of **pivot_root(2)** is optionally specified in the 'pivot_root' rule using the 'oldroot=' prefix.

AppArmor 'pivot_root' rules can specify a profile transition to occur during the **pivot_root(2)** system call. Note that AppArmor will only transition the process calling **pivot_root(2)** to the new profile.

The paths specified in 'pivot_root' rules must end with '/' since they are directories.

Here are some example 'pivot_root' rules:

```
# Allow any pivot
pivot_root,

# Allow pivoting to any new root directory and putting the old root
# directory at /mnt/root/old/
pivot_root oldroot=/mnt/root/old/,

# Allow pivoting the root directory to /mnt/root/
pivot_root /mnt/root/,

# Allow pivoting to /mnt/root/ and putting the old root directory at
# /mnt/root/old/
pivot_root oldroot=/mnt/root/old/ /mnt/root/,

# Allow pivoting to /mnt/root/, putting the old root directory at
# /mnt/root/old/ and transition to the /mnt/root/sbin/init profile
pivot_root oldroot=/mnt/root/old/ /mnt/root/ -> /mnt/root/sbin/init,
```

PTrace rules

AppArmor supports mediation of **ptrace(2)**. AppArmor PTrace rules are accumulated so that the granted PTrace permissions are the union of all the listed PTrace rule permissions.

AppArmor PTrace permissions are implied when a rule does not explicitly state an access list. By default, all PTrace permissions are implied.

The trace and tracedby permissions govern **ptrace(2)** while read and readby govern certain **proc(5)** filesystem accesses, **kcmp(2)**, futexes (**get_robust_list(2)**) and perf trace events.

For a ptrace operation to be allowed the profile of the tracing process and the profile of the target task must both have the correct permissions. For example, the profile of the process attaching to another task must have the trace permission for the target task's profile, and the task being traced must have the tracedby permission for the tracing process' profile.

Example AppArmor PTrace rules:

```
# Allow all PTrace access
ptrace,

# Explicitly allow all PTrace access,
ptrace (read, readby, trace, tracedby),

# Explicitly deny use of ptrace(2)
deny ptrace (trace),
```

```
# Allow unconfined processes (eg, a debugger) to ptrace us
ptrace (readby, tracedby) peer=unconfined,

# Allow ptrace of a process running under the /usr/bin/foo profile
ptrace (trace) peer=/usr/bin/foo,
```

Signal rules

AppArmor supports mediation of **signal** (7). AppArmor signal rules are accumulated so that the granted signal permissions are the union of all the listed signal rule permissions.

AppArmor signal permissions are implied when a rule does not explicitly state an access list. By default, all signal permissions are implied.

For the sending of a signal to be allowed, the profile of the sending process and the profile of the target task must both have the correct permissions. For example, the profile of a process sending a signal to another task must have the send permission for the target task's profile, and the task receiving the signal must have a receive permission for the sending process' profile.

Example AppArmor signal rules:

```
# Allow all signal access
signal,

# Explicitly deny sending the HUP and INT signals
deny signal (send) set=(hup, int),

# Allow unconfined processes to send us signals
signal (receive) peer=unconfined,

# Allow sending of signals to a process running under the /usr/bin/foo
# profile
signal (send) peer=/usr/bin/foo,

# Allow checking for PID existence
signal (receive, send) set=("exists"),

# Allow us to signal ourselves using the built-in @{profile_name} variable
signal peer=@{profile_name},

# Allow two real-time signals
signal set=(rtmin+0 rtmin+32),
```

DBus rules

AppArmor supports DBus mediation. The mediation is performed in conjunction with the DBus daemon. The DBus daemon verifies that communications over the bus are permitted by AppArmor policy.

AppArmor DBus rules are accumulated so that the granted DBus permissions are the union of all the listed DBus rule permissions.

AppArmor DBus rules are broad and general and become more restrictive as further information is specified. Policy may be specified down to the interface member level (method or signal name), however the contents of messages are not examined.

Some AppArmor DBus permissions are not compatible with all AppArmor DBus rules. The 'bind' permission cannot be used in message rules. The 'send' and 'receive' permissions cannot be used in service rules. The 'eavesdrop' permission cannot be used in rules containing any conditionals outside of the 'bus' conditional.

'r' and 'read' are synonyms for 'receive'. 'w' and 'write' are synonyms for 'send'. 'rw' is a synonym for both 'send' and 'receive'.

AppArmor Dbus permissions are implied when a rule does not explicitly state an access list. By default, all Dbus permissions are implied. Only message permissions are implied for message rules and only service permissions are implied for service rules.

Example AppArmor Dbus rules:

```
# Allow all Dbus access
dbus,

# Explicitly allow all Dbus access,
dbus (send, receive, bind),

# Deny send/receive/bind access to the session bus
deny dbus bus=session,

# Allow bind access for a particular name on any bus
dbus bind name=com.example.ExampleName,

# Allow receive access for a particular path and interface
dbus receive path=/com/example/path interface=com.example.Interface,

# Deny send/receive access to the system bus for a particular interface
deny dbus bus=system interface=com.example.ExampleInterface,

# Allow send access for a particular path, interface, member, and pair of
# peer names:
dbus send
    bus=session
    path=/com/example/path
    interface=com.example.Interface
    member=ExampleMethod
    peer=(name=(com.example.ExampleName1|com.example.ExampleName2)),

# Allow receive access for all unconfined peers
dbus receive peer=(label=unconfined),

# Allow eavesdropping on the system bus
dbus eavesdrop bus=system,

# Allow and audit all eavesdropping
audit dbus eavesdrop,
```

Unix socket rules

AppArmor supports fine grained mediation of unix domain abstract and anonymous sockets. Unix domain sockets with file system paths are mediated via file access rules.

Abstract unix domain sockets is a nonportable Linux extension of unix domain sockets, see **unix**(7) for more information.

Unix socket address paths

The `sun_path` component (aka the socket address) of a unix domain socket is specified by the

```
addr=
```

conditional. If an address conditional is not specified as part of a rule then the rule matches both abstract and anonymous sockets.

In apparmor the address of an abstract unix domain socket begins with the `@` character, similar to how they are reported (as paths) by `netstat -x`. The address then follows and may contain pattern matching and any

characters including the null character. In apparmor null characters must be specified by using an escape sequence `\000` or `\x00`. The pattern matching is the same as is used by file path matching so `*` will not match `/` even though it has no special meaning with in an abstract socket name. Eg.

```
unix addr=@*,
```

Anonymous unix domain sockets have no `sun_path` associated with the socket address, however it can be specified with the special *none* keyword to indicate the rule only applies to anonymous unix domain sockets. Eg.

```
unix addr=none,
```

If the address component of a rule is not specified then the rule applies to both abstract and anonymous sockets.

Unix socket permissions

Unix domain socket rules are accumulated so that the granted unix socket permissions are the union of all the listed unix rule permissions.

Unix domain socket rules are broad and general and become more restrictive as further information is specified. Policy may be specified down to the socket address (aka `sun_path`) and label level. The content of the communication is not examined.

Unix socket rule permissions are implied when a rule does not explicitly state an access list. By default if a rule does not have an access list all permissions that are compatible with the specified set of local and peer conditionals are implied.

The `create`, `bind`, `listen`, `shutdown`, `getattr`, `setattr`, `getopt`, and `setopt` permissions are local socket permissions. They are only applied to the local socket and can't be specified in rules that have a peer component. The `accept` permission applies to the combination of a local and peer socket. The `connect`, `send`, and `receive` permissions are peer socket permissions.

Only the peer socket permissions will be applied to rules that don't specify permissions and contain a peer component.

Example Unix domain socket rules:

```
# Allow all permissions to unix sockets
unix,

# Explicitly allow all unix permissions
unix (create, listen, accept, connect, send, receive, getattr, setattr, setopt, ge

# Explicitly deny unix socket access
deny unix,

# Allow create and use of abstract and anonymous sockets for profile_name
unix peer=(label=@{profile_name}),

# Allow receiving via unix sockets from unconfined
unix (receive) peer=(label=unconfined),

# Allow getattr and shutdown on anonymous sockets
unix (getattr, shutdown) addr=none,

# Allow SOCK_STREAM connect, receive and send on an abstract socket @bar
# with peer running under profile '/foo'
unix (connect, receive, send) type=stream peer=(label=/foo,addr="@bar"),

# Allow accepting connections from and receiving from peer running under
# profile '/bar' on abstract socket '@foo'
```

```
unix (accept, receive) addr=@foo peer=(label=/bar),
```

Abstract unix domain sockets autobind

Abstract unix domain sockets can autobind to an address. The autobind address is a unique 5 digit string of decimal numbers, eg. @00001. There is nothing that prevents a task from manually binding to addresses with a similar pattern so it is impossible to reliably identify autobind addresses from a regular address.

Interaction of network rules and fine grained unix domain socket rules

The coarse grained networking rules can be used to control unix domain sockets as well. When fine grained unix domain socket mediation is available the coarse grained network rule is mapped into the equivalent unix socket rule.

E.G.

```
network unix, => unix,
```

```
network unix stream, => unix stream,
```

Fine grained mediation rules however can not be lossly converted back to the coarse grained network rule; e.g.

```
unix bind addr=@example,
```

Has no exact match under coarse grained network rules, the closest match is the much wider permission rule of

```
network unix,
```

change_profile rules

AppArmor supports self directed profile transitions via the `change_profile` api. `Change_profile` rules control which permissions for which profiles a confined task can transition to. The profile name can contain apparmor pattern matching to specify different profiles.

```
change_profile -> **,
```

The `change_profile` api allows the transition to be delayed until when a task executes another application. If an `exec` rule transition is specified for the application and the `change_profile` api is used to make a transition at `exec` time, the transition specified by the `change_profile` api takes precedence.

The `Change_profile` permission can restrict which profiles can be transitioned to based off of the executable name by specifying the `exec` condition.

```
change_profile /bin/bash -> new_profile,
```

The restricting of the transition profile to a given executable at `exec` time is only useful when then current task is allowed to make dynamic decisions about what confinement should be, but the decision set needs to be controlled. A list of profiles or multiple rules can be used to specify the profiles in the set. Eg.

```
change_profile /bin/bash -> {new_profile1,new_profile2,new_profile3},
```

An `exec` rule can be used to specify a transition for the executable, if the transition should be allowed even if the `change_profile` api has not been used to select a transition for those available in the `change_profile` rule set. Eg.

```
/bin/bash Px -> new_profile1,
```

```
change_profile /bin/bash -> {new_profile1,new_profile2,new_profile3},
```

The `exec` mode dictates whether or not the Linux Kernel's **unsafe_exec** routines should be used to scrub the environment, similar to `setuid` programs. (See `ld.so(8)` for some information on `setuid/setgid` environment scrubbing.) The **safe** mode sets up environment scrubbing to occur when the new application is executed and **unsafe** mode disables AppArmor's requirement for environment scrubbing (the kernel and/or libc may still require environment scrubbing). An `exec` mode can only be specified when an `exec` condition is present.


```
change_profile safe /bin/bash -> new_profile,
```

Not all kernels support **safe** mode and the parser will downgrade rules to **unsafe** mode in that situation. If no exec mode is specified, the default is **safe** mode in kernels that support it.

rlimit rules

AppArmor can set and control the resource limits associated with a profile as described in the **setrlimit**(2) man page.

The AppArmor rlimit controls allow setting of limits and restricting changes of them and these actions can be audited. Enforcement of the set limits is handled by the standard kernel enforcement mechanism for rlimits and will not result in an audited apparmor message if the limit is enforced.

If a profile does not have an rlimit rule associated with a given rlimit then the rlimit is left alone and regular access, including changing the limit, is allowed. However if the profile sets an rlimit then the current limit is checked and if greater than the limit specified in the rule it will be changed to the specified limit.

AppArmor rlimit rules control the hard limit of an application and ensure that if the hard limit is lowered that the soft limit does not exceed the hard limit value.

Eg.

```
set rlimit data <= 100M,
set rlimit nproc <= 10,
set rlimit nice <= 5,
```

Variables

AppArmor's policy language allows embedding variables into file rules to enable easier configuration for some common (and pervasive) setups. Variables may have multiple values assigned, but any variable assignments must be made before the start of the profile.

The parser will automatically expand variables to include all values that they have been assigned; it is an error to reference a variable without setting at least one value. You can use empty quotes ("") to explicitly add an empty value.

At the time of this writing, the following variables are defined in the provided AppArmor policy:

```
@{HOME}
@{HOMEDIRS}
@{multiarch}
@{pid}
@{pids}
@{PROC}
@{securityfs}
@{apparmorfs}
@{sys}
@{tid}
@{XDG_DESKTOP_DIR}
@{XDG_DOWNLOAD_DIR}
@{XDG_TEMPLATES_DIR}
@{XDG_PUBLICSHARE_DIR}
@{XDG_DOCUMENTS_DIR}
@{XDG_MUSIC_DIR}
@{XDG_PICTURES_DIR}
@{XDG_VIDEOS_DIR}
```

These are defined in files in */etc/apparmor.d/tunables* and are used in many of the abstractions described later.

You may also add files in */etc/apparmor.d/tunables/home.d* for site-specific customization of **@{HOMEDIRS}**, */etc/apparmor.d/tunables/multiarch.d* for **@{multiarch}** and */etc/apparmor.d/tunables/xdg-user-dirs.d* for **@{XDG_*}**.

The special `@{profile_name}` variable is set to the profile name and may be used in all policy.

Alias rules

AppArmor also provides alias rules for remapping paths for site-specific layouts. They are an alternative form of path rewriting to using variables, and are done after variable resolution. Alias rules must occur within the preamble of the profile. System-wide aliases are found in `/etc/apparmor.d/tunables/alias`, which is included by `/etc/apparmor.d/tunables/global`. `/etc/apparmor.d/tunables/global` is typically included at the beginning of an AppArmor profile.

Globbering

File resources may be specified with a globbing syntax similar to that used by popular shells, such as `cs`h (1), `ba`sh (1), `z`sh (1).

* can substitute for any number of characters, excepting `'`

** can substitute for any number of characters, including `'`

? can substitute for any single character excepting `'`

[abc]

will substitute for the single character a, b, or c

[a-c]

will substitute for the single character a, b, or c

[^a-c]

will substitute for any single character not matching a, b or c

{ab,cd}

will expand to one rule to match ab, one rule to match cd

When AppArmor looks up a directory the pathname being looked up will end with a slash (e.g., `/var/tmp/`); otherwise it will not end with a slash. Only rules that match a trailing slash will match directories. Some examples, none matching the `/tmp/` directory itself, are:

`/tmp/*`

Files directly in `/tmp`.

`/tmp*/`

Directories directly in `/tmp`.

`/tmp/**`

Files and directories anywhere underneath `/tmp`.

`/tmp/**/`

Directories anywhere underneath `/tmp`.

Rule Qualifiers

There are several rule qualifiers that can be applied to permission rules. Rule qualifiers can modify the rule and/or permissions within the rule.

allow

Specifies that permissions requests that match the rule are allowed. This is the default value for rules and does not need to be specified. Conflicts with the `deny` qualifier.

audit

Specifies that permissions requests that match the rule should be recorded to the audit log.

deny

Specifies that permissions requests that match the rule should be denied without logging. Can be combined with `'audit'` to enable logging. Conflicts with the `allow` qualifier.

owner

Specifies that the task must have the same euid/fsuid as the object being referenced by the permission check.

Qualifier Blocks

Rule Qualifiers can be applied to multiple rules at a time by grouping the rules into a rule block.

```
audit {
    /foo r,
    network,
}
```

#include mechanism

AppArmor provides an easy abstraction mechanism to group common access requirements; this abstraction is an extremely flexible way to grant site-specific rights and makes writing new AppArmor profiles very simple by assembling the needed building blocks for any given program.

The use of `'#include'` is modelled directly after `cpp` (1); its use will replace the `'#include'` statement with the specified file's contents. The leading `'#'` is optional, and the `'#include'` keyword can be followed by an option conditional `'if exists'` that specifies profile compilation should continue if the specified file or directory is not found.

#include `"/absolute/path"` specifies that `/absolute/path` should be used. **#include `"relative/path"`** specifies that `relative/path` should be used, where the path is relative to the current working directory. **#include `<magic/path>`** is the most common usage; it will load `magic/path` relative to a directory specified to `apparmor_parser` (8). `/etc/apparmor.d/` is the AppArmor default.

The supplied AppArmor profiles follow several conventions; the abstractions stored in `/etc/apparmor.d/abstractions/` are some large clusters that are used in most profiles. What follows are short descriptions of how some of the abstractions are used.

abstractions/audio

Includes accesses to device files used for audio applications.

abstractions/authentication

Includes access to files and services typically necessary for services that perform user authentication.

abstractions/base

Includes files that should be readable and writable in all profiles.

abstractions/bash

Includes many files used by bash; useful for interactive shells and programs that call `system` (3).

abstractions/consoles

Includes read and write access to the device files controlling the virtual console, `sshd` (8), `xterm` (1), etc. This abstraction is needed for many programs that interact with users.

abstractions/fonts

Includes access to fonts and the font libraries.

abstractions/gnome

Includes read and write access to GNOME configuration files, as well as read access to GNOME libraries.

abstractions/kde

Includes read and write access to KDE configuration files, as well as read access to KDE libraries.

abstractions/kerberosclient

Includes file access rules needed for common kerberos clients.

abstractions/nameservice

Includes file rules to allow DNS, LDAP, NIS, SMB, user and group password databases, services, and protocols lookups.

abstractions/perl

Includes read access to perl modules.

abstractions/user-download
abstractions/user-mail
abstractions/user-manpages
abstractions/user-tmp
abstractions/user-write

Some profiles for typical “user” programs will use these include files to describe rights that users have in the system.

abstractions/wutmp

Includes write access to files used to maintain **wtmp** (5) and **utmp** (5) databases, used with the **w(1)** and associated commands.

abstractions/X

Includes read access to libraries, configuration files, X authentication files, and the X socket.

Some of the abstractions rely on variables that are set in files in the */etc/apparmor.d/tunables/* directory. These variables are currently **@{HOME}** and **@{HOMEDIRS}**. Variables cannot be set in profile scope; they can only be set before the profile. Therefore, any profiles that use abstractions should either **#include <tunables/global>** or otherwise ensure that **@{HOME}** and **@{HOMEDIRS}** are set before starting the profile definition. The **aa-autodep** (8) and **aa-genprof** (8) utilities will automatically emit **#include <tunables/global>** in generated profiles.

EXAMPLE

An example AppArmor profile:

```
# a variable definition in the preamble
@{HOME} = /home/*/ /root/

# a comment about foo.
/usr/bin/foo {
    /bin/mount          ux,
    /dev/{,u}random     r,
    /etc/ld.so.cache    r,
    /etc/foo.conf       r,
    /etc/foo/*          r,
    /lib/ld-*.so*       rmix,
    /lib/lib*.so*       r,
    /proc/[0-9]**      r,
    /usr/lib/**         r,
    /tmp/foo.pid        wr,
    /tmp/foo.*          lrw,
    /@{HOME}/.foo_file rw,
    /usr/bin/baz        Cx -> baz,

# a comment about foo's hat (subprofile), bar.
^bar {
    /lib/ld-*.so*       rmix,
    /usr/bin/bar         rmix,
    /var/spool/*        rwl,
}

# a comment about foo's subprofile, baz.
profile baz {
    #include <abstractions/bash>
    owner /proc/[0-9]*/stat r,
    /bin/bash ixr,
    /var/lib/baz/ r,
```

```
        owner /var/lib/baz/* rw,  
    }  
}
```

FILES

/etc/init.d/boot.apparmor
/etc/apparmor.d/

KNOWN BUGS

- Mount options support the use of pattern matching but mount flags are not correctly intersected against specified patterns. Eg, 'mount options=**,' should be equivalent to 'mount,', but it is not. (LP: #965690)
- The fstype may not be matched against when certain mount command flags are used. Specifically fstype matching currently only works when creating a new mount and not remount, bind, etc.
- Mount rules with multiple 'options' conditionals are not applied as documented but instead merged such that 'options in (ro,nodev) options in (atime)' is equivalent to 'options in (ro,nodev,atime)'.
- When specifying mount options with the 'in' conditional, both the positive and negative values match when specifying one or the other. Eg, 'rw' matches when 'ro' is specified and 'dev' matches when 'nodev' is specified such that 'options in (ro,nodev)' is equivalent to 'options in (rw,dev)'.

SEE ALSO

apparmor (7), **apparmor_parser** (8), **aa-complain** (1), **aa-enforce** (1), **aa_change_hat** (2), **mod_apparmor** (5), and <<https://wiki.apparmor.net>>.