AppArmor Technical Documentation

Andreas Gruenbacher and Seth Arnold
{agruen,seth.arnold}@suse.de
SUSE Labs / Novell

Contents

1 Introduction

In this paper we describe AppArmor from a technical point of view, introduce its concepts, and explain the design decisions taken. This text is intended for people interested in understanding why AppArmor works the way it does. You may be looking for less detailed, low-level, or kernel centric documentation; in that case, please refer to the AppArmor documentation web site [1].

Sections 2 and 3 discuss the AppArmor security model, while Section 4 shows how to use it from a low-level point of view. Please be aware that lots of details are discussed here which the higher-level tools hide from the average user.

2 Overview

AppArmor protects systems from insecure or untrusted processes by running them in confinement, still allowing them to share files with other parts of the system, exercising privilege, and communicating with other processes, but with some restrictions. These restrictions are mandatory; they are not bound to identity, group membership, or object ownership. In particular, the restrictions also apply to processes running with superuser privileges. AppArmor achieves this by plugging into the Linux Security Module (LSM) framework. The protections provided are in addition to the kernel's regular access control mechanisms.

The AppArmor kernel module and accompanying user-space tools are available under the GPL license. (The exception is the libapparmor library, available under the LGPL license, which allows change_hat(2) to be used by non-GPL binaries.)

At the moment, AppArmor knows about two types of resources: files, and POSIX.1e (draft) capabilities. By controlling access to these resources, AppArmor can effectively prevent confined processes from accessing files in unwanted ways, from executing binaries which they are not meant to execute, and from exercising privileges such as acting on behalf of another user (which are traditionally restricted to the superuser).

One use case for this kind of protection is a network daemon: even if the daemon is broken into, the additional restrictions imposed by AppArmor will prevent the attacker from attaining additional privileges beyond what the daemon is normally allowed to do. Because AppArmor controls which files a process can access in which ways down to the individual file level, the potential damage is much limited.

There is work going on for teaching AppArmor about additional resources like ulimits, and interprocess and network communication, but at this time, these resource types are not covered. This is less severe than it might initially seem: in order to attack another process from a broken-into process like a network daemon, that other process has to actively listen. The set of actively listening processes is relatively small, and this sort of interprocess communication is a natural security boundary, so listening processes should be validating all their input already. For protection against bugs in the input validation of those processes, they should also be confined by AppArmor though, thus further limiting the potential damage.

AppArmor protection is selective: it only confines processes for which policies (referred to as profiles) have been defined. All other processes will continue to run unrestricted by AppArmor.

To confine a process, all it takes is to write a profile for it, take an existing profile, or automatically generate a profile: for the latter, the process can be run in *learning* or *complain* mode in which AppArmor allows all accesses, and logs all accesses that are not allowed by the current profile already. This log can then be used to automatically generate a suitable new profile, or refine an existing one. The application does not need to be modified.

An example profile together with a complete low-level walk-through of AppArmor can be found in Section 4. The apparmor.d(5) manual page contains further details.

AppArmor is not based on labeling or label-based access and transition rules, so it does not stick a label on each each file in the file system (or more generally, on each object). It identifies files by name rather than by label, so if a process is granted read access to /etc/shadow and the system administrator renames /etc/shadow to /etc/shadow.old and replaces it with a copy (that may have an additional user in it, for example), the process will have access to the new /etc/shadow, and not to /etc/shadow.old.

3 The AppArmor Security Model

When a file is accessed by name with open(2), mkdir(2), etc., the kernel looks up the location of the object associated with the specified pathname in the file system hierarchy. The lookup is relative to the root directory for pathnames starting with a slash, and to the current working directory otherwise. Different processes can have have different working directories as well as different root directories. See path_resolution(2) for a detailed discussion of how pathname resolution works.

Either way, the result of the lookup is a pair of (dentry, vfsmount) kernel-internal objects that uniquely identify the location of the file in the file system hierarchy. The dentry points to the object if the object already exists, and is a placeholder for the object to be created otherwise.

AppArmor uses the (dentry, vfsmount) pair to compute the pathname of the file within a process's filesystem namespace. The resulting pathname contains no relative pathname components (“.” or “..”), or symlinks.

AppArmor checks if the current profile contains rules that match this pathname, and if those rules allow the requested access. Accesses that are not explicitly allowed are denied.

3.1 Symbolic Links

When looking up the (dentry, vfsmount) pair of a file, the kernel resolves symlinks where appropriate (and fails the lookup where symlink resolution is inappropriate).

The pathname that AppArmor computes from a (dentry, vfsmount) pair never contains symlinks. This also means that if symlinks are used instead of directories for paths like /tmp, profiles need to be adjusted accordingly. A

future version of AppArmor may have built-in support for this kind of pathname rewriting.

3.2 Namespaces

Linux allows different processes to live in separate namespaces, each of which forms an independent file system hierarchy. A recent paper by Al Viro and Ram Pai [2] discusses all the intricate things possible with namespaces in recent 2.6 kernels.

From the point of view of a process, an absolute path is a path that goes all the way up to the root directory of that process. This is ambiguous if processes have different root directories. Therefore, instead of paths relative to process root directories, AppArmor uses paths relative to the namespace root.

Pathnames are meaningful only within a namespace. Each namespace has a root where all the files, directories, and mount points are hanging off from.

The privilege of creating new namespaces is bound to the CAP_SYS_ADMIN capability, which grants a multitude of other things that would allow a process to break out of AppArmor confinement, so confined processes are not supposed to have this privilege, and processes with this capability need to be considered trusted.

In this setup, privileged processes can still create separate namespaces and start processes in those namespaces; processes confinement will be relative to whatever namespace a process ends up in. It is unclear at this point how AppArmor should support separate namespaces — either by computing all pathnames relative to one particular namespace considered global (assuming that such a globally meaningful namespace will exist in all setups in which AppArmor is relevant), or by allowing different sets of profiles to be associated with different namespaces.

3.3 Disconnected Files and Pseudo File Systems

In some situations, a process can end up with a file descriptor or working directory that was looked up by name at some point, but is not connected to the process's namespace anymore (and hasn't been deleted, either). This can happen when file descriptors are passed between processes that do not share the same namespace, or when a file system has been lazily unmounted (see the MNT_DETACH flag of umount2(2)). Such files may still be visible to other processes, and they may become reconnected. AppArmor cannot compute the pathnames of such files. Granting unrestricted access would be insecure, and so AppArmor denies access to disconnected files.

As a special case, the kernel supports a number of file systems that users can have file descriptors open for, but that can never be mounted. Those files are by definition disconnected. Anonymous pipes, futexes, inotify, and epoll are all examples of that. Accesses to those files is always allowed.

Future versions of AppArmor will have better control over disconnected files by controlling file descriptor passing between processes.

3.4 Mount

Mounting can change a process's namespace in almost arbitrary ways. This is a problem because AppArmor's file access control is pathname based, and granting a process the right to arbitrarily change its namespace would subvert this protection mechanism. AppArmor therefore denies confined processes access to the mount(2), umount(2), and umount2(2) system calls.

Future versions of AppArmor may offer fine-grained control over mount, and may grant confined processes specific mount operations.

3.5 The Kernel NFS Daemon

The security model of the various versions of NFS is that files are looked up by name as usual, but after that lookup, each file is only identified by a file handle in successive acesses. The file handle at a minimum includes some sort of filesystem identifier and the file's inode number. In Linux, the file handles used by most filesystems also include the inode number of the parent directory; this may change in the future. File handles are persistent across server restarts.

This means that when the NFS daemon is presented with a file handle, clients must get access without having specified a pathname. A pathname can be computed from a (parent, child) inode pair that identifies the file down to the directory level if the dentry is properly connected to the dcache, but multiple hardlinks to the same file within the same directory cannot be distinguished, and properly connecting dentries comes at a cost in the NFS daemon. Because of this overhead and the questionable benefit, most setups do not guarantee that dentries will be connected, and so pathnames cannot always be computed. (See the no_subtree_check option in exports(5).)

In addition, the NFS daemon is implemented in the kernel rather than as a user space process. There is no memory separation or other protection between the daemon and the rest of the kernel. This means that at best, the NFS daemon could cooperate with an additional access control mechanism like AppArmor — but there would be no enforcement.

Because of all of this, it makes little sense to put the kernel NFS daemon under AppArmor control. Administrators are advised to not assign profiles to the kernel nfsd daemons.

3.6 Why are the computed pathnames meaningful?

Whenever a process performs a name-based file access, the pathname or pathname component always refers to a specific path to that file: the path is either relative to the chroot if an absolute path is used, or else relative to the current working directory. The chroot or current working directory always has a unique pathname up to the namespace root (even if the process itself has no direct access above the chroot). This means that each name-based file access maps to a unique, canonical, absolute pathname. There may be additional paths pointing to the same file, but a particular name-based access still always refers to only one of them. These are the pathnames that AppArmor uses for permission checks.

If directories along the path get renamed after a process changes into them (either with chroot(2) or with chdir(2)), the resulting pathname will differ from

the pathnames that the process used. Consider the following sequence of operations for example:

Process 1	**Process 2**
chdir("/var/tmp/foo");	
	rename("/var/tmp/foo", "/var/tmp/bar");
creat("baz", 0666);	

The creat operation will check against the path /var/tmp/*bar*/baz, even though Process 1 never used *bar*. This is the expected behavior; we are interested in the names of the objects along the path at the time of the access, not in their previous names.

As already mentioned, a path lookup results in a pair of (dentry, vfsmount) kernel-internal objects. The pathname that AppArmor checks against is computed from these two objects after these objects have been looked up. The lookup and the pathname computation are not atomic, which means that pathname components could even be renamed after the lookup but before the pathname has been computed.

It matters that the AppArmor access check is performed between the lookup and the actual access, but atomicity between the lookup and that access check is not necessary: there is no difference between a rename before the lookup and a rename after the lookup from AppArmor's point of view; all we care about is the current pathname at some point between the lookup and the access.

A special case occurs when the lookup succeeds, but the file is deleted before the AppArmor access check. In this case the access is denied and errno is set to ENOENT, the same behavior as if the lookup had failed.

3.7 Path Permission Checking

On UNIX systems, when files are looked up by name, the lookup starts either at the root or the current working directory of a process. From there, each directory reached is checked for search permission (x). The permissions on the directories leading to the current working directory are not checked. When a file is being created or deleted, the parent directory of that file is checked for write and search access (wx). When a file is being accessed, the permissions of that file are checked for r, w, or x access, or a combination thereof. Each check can result in a failure with errno set to EACCES (Permission denied).

In contrast, AppArmor first computes the pathname to a file. If a file is being created, the name being looked up is the name of the new file and not the name of the parent directory.

If the file being looked up is a directory, AppArmor appends a slash to the pathname so that directory pathnames always end in a slash; otherwise the pathname will not end in a slash.

It then checks for file access rules in the process's profile that match that pathname, and decides based on that. With some exceptions for execute modes as described in Section 3.15, the permissions granted are the union of permissions of all matching rules.

r	Read.
w	Write.
ix	Execute and inherit the current profile.
px	Execute under a specific profile.
Px	Execute secure and under a specific profile.
ux	Execute unconfined.
Ux	Execute secure and unconfined.
m	Memory map as executable.
l	Link.

Table 1: File Access Permissions in Profiles

3.8 Profile Permissions

AppArmor differentiates between slightly more permissions than UNIX does, as shown in Table 1: file access rules in AppArmor support the read (r), write (w), execute (x), memory map as executable (m), and link (l) permissions. The execute permission requires a modifier that further specifies which kind of execution is being granted: inherit the current profile (ix), use the profile defined for that executable (px), or execute unconfined without a profile (ux). In addition, the px and ux permissions have Px and Ux forms that will trigger Secure Execution (see Section 3.14 below). The different permissions are used as follows:

Read. The profile read permission is required by all system calls that require the UNIX read permission. This includes open with O_RDONLY, getdents (i.e., readdir), listxattr, getxattr, and mmap with PROT_READ.

Write. The profile write permission is required by all system calls that require the UNIX write permission, except for operations that create or remove files: while UNIX requires write access to the parent directory, AppArmor requires write access on the new file in this case (which does not exist at the time of the permission check for file creates). Operations that create files include open with O_CREAT, creat, mkdir, symlink, and mknod. Operations that remove files include rename, unlink and rmdir.

Operations that require write access in UNIX as well as AppArmor include open with O_WRONLY (O_RDWR requires read and write), setxattr, removexattr, and mmap with PROT_WRITE.

Other system calls such as chmod, chown, utime, and utimes are bound to file ownership or the respective capabilities in UNIX. AppArmor also requires profile write access for those operations.

Execute. As mentioned above, AppArmor distinguishes a few different ways how files may be executed as described above.

For directories, the UNIX execute permission maps to search access. AppArmor does not control directory search access. Traversing directories is always granted.

Memory map as executable. The Linux kernel only requires read access to files in order to memory map them for execution with the PROT_EXEC

flag. AppArmor makes a distinction here, and requires the m profile permission in order for files to be mapped as executable. That way, it is more obvious in profiles what applications are allowed to do even if from a security point of view, the m permission provides a similar level of protection as the ix permission — execute under the current profile.

Link. Creating a hardlink requires the profile link permission (l) on the new path. In addition, the new path must have a subset of the r, w, x, and m permissions of the old path, and if the new path has the x permission, the execute flags (i, u, U, p, and P) of the old and the new path must be equal.

Rename. A rename requires profile read and write access for the source file, and profile write access for the target file.

Stat. Retrieving information about files is always allowed. We believe that providing policy for file information retrieval is more troublesome than the benefit it would provide.

3.9 System Calls Taking File Handles, At System Calls

A number of system calls take file descriptors instead of pathnames as their parameters (ftruncate, fchmod, etc.), or take directory file descriptors, and resolve pathnames relative to those directories (openat, mkdirat, etc.). These system calls are treated like their non-f and non-at equivalents, and the same access checks are performed. At the point where AppArmor is asked to validate those file accesses, it is passed a (dentry, vfsmount) pair no matter which system call variant is used.

3.10 File Descriptor Passing and Revalidation

After a file descriptor has been obtained, the permitted accesses (read and/or write) are encoded in the file descriptor, and reads and writes are not revalidated against the profile for each access. This is consistent with how access checks are done in UNIX; such access checks would have a severe performance impact.

The picture changes when a file descriptor is passed between processes and the other process is running under a different profile, or when a process switches profiles: in that case, read and write accesses are revalidated under the new profile. If the new profile does not allow them, the access is denied and errno is set to EACCES (Permission denied).

File descriptors opened by unconfined processes are exempt from this rule. This is so that processes will still have access to their stdin, stdout, and stderr without having to list all possible sources of input and output in all profiles.

3.11 Deleted Files

Revalidation is problematic for deleted files for which a process still has an open file descriptor — after all, the idea of the pathname of a deleted file is somewhat peculiar: the file is no longer reachable by any pathname, and it also cannot become re-attached to the filesystem namespace again.

The traditional UNIX behavior is to determine access upon file access, and to never check again. Applications depend on this, particularly for temporary files. In addition to temporary files, deleted files can be used as an interprocess communication mechanism if the file descriptor is shared among multiple processes.

AppArmor grants access to deleted files, just like it grants access to files opened by unconfined processes. It may control interprocess communication, including file descriptor passing, in a future version.

3.12 The access System Call

This system call determines whether a process has a given mode of access to a file in terms of the read, write, and execute permissions. This is not a sufficient replacement for performing the access check at the time of access even under traditional UNIX, because the access system call and the subsequent access are not atomic, and the permissions might change between the two operations. Applications are not supposed to rely on access(2).

AppArmor introduces additional restrictions, some of which cannot be modeled in terms of read, write, and execute: for example, an AppArmor profile may allow a process to create files /tmp/foo-*, but not any other files in /tmp.

There is no way to express this with access(2); in traditional UNIX, all that is required for creating files is write access to the parent directory. Access(2) will indicate that some accesses are allowed even when AppArmor will eventually deny them.

3.13 The ptrace System Call

The ability to ptrace allows a process to look up information about another process, read and write the memory of that process, and attach to (or trace) that process in order to debug it, or analyze its behavior. This gives total control over the process being traced, and so the kernel employs some restrictions over which processes may ptrace with other processes.

In addition to these restrictions, AppArmor requires that if the tracing task is confined, it must either have the CAP_SYS_PTRACE capability, or be confined by the same profile and sub-profile as the process being traced. Attempts to switch to another profile or sub-profile by a process being traced is denied.

3.14 Secure Execution

In this mode, the kernel passes a flag to user space. When glibc finds this flag set, it unsets environment variables that are considered dangerous, and it prevents the dynamic loader from loading libraries controlled by the environment. With non-secure exec, the LD_LIBRARY_PATH environment variable can be used to switch to a different set of libraries, for example. The secure exec mechanism is not specific to AppArmor: set-user-id and set-group-id executables also use it, as well as SELinux, which introduced this glibc feature.

3.15 Exec Mode Merging in Profiles, Exact Matches

When more than one rule in a profile matches a given path, all the permissions accumulate except for ix, px, Px, ux, and Ux: those permissions would conflict with each other; it would be unclear how to execute the new binary if more than one of these flags was set. To deal with this situation, AppArmor differentiates between rules that define exact matches and wildcard rules (see Table 2 on page 14). Execute flags in exact matches override execute flags in wildcard matches.

If the execute flags of multiple rules still disagree, the profile is rejected at profile load time.

3.16 Capabilities

AppArmor uses the standard Linux capability mechanism. When the kernel checks if a certain capability can be exercised, AppArmor additionally checks if the current profile allows the requested capability, and rejects the use of the capability otherwise.

3.17 The sysctl System Call and /proc/sys

The sysctl system call and files below /proc/sys can be used to read and modify various kernel parameters. Root processes can easily bring the system down by setting kernel parameters to invalid values. To prevent against that, AppArmor denies confined processes that do not have the CAP_SYS_ADMIN capability write access to kernel parameters.

3.18 Subprofiles aka. Hats

Profiles can contain subprofiles that processes may switch to from the main profile. Switching from a subprofile into a sibling subprofile or back to the parent profile is allowed depending on how the subprofile was entered, and provided that the child knows a magic cookie.[1] See the change_hat(2) manual page for details.

Each process may consist of multiple tasks. Each task may only change its own subprofile. The superuser cannot put a task into a different hat, but he can replace the entire profile and its subprofiles, or he can put a process in a different top-level profile (see Section 3.19).

Internally, change_hat(2) is implemented by writing to a special kernel-provided file. This is equivalent to a command like:

```
$ echo "changehat 123^hat_name" > /proc/$PID/attr/current
```

Here, the number is the magic cookie value, and hat_name obviously is the name of the hat; either may be replaced by the empty string (but not both).

[1] **A word of warning about change_hat(2):** When used with a non-zero magic cookie for changing into a subprofile, that magic cookie can be used to change back out of the subprofile; in this mode, change_hat(2) is not a strong confinement mechanism. If the code running in the subprofile can guess the magic cookie, it can break out of the subprofile. Likewise, if that code can manipulate the processes' behavior beyond the point where the process returns from the subprofile, it can influence what is done under the parent profile. Therefore, change_hat(2) with a non-zero magic cookie is only safe in combination with restricted code environments, such as when the subprofile is used for executing Safe Perl (see Safe(3pm)), etc.

3.19 Association of Profiles with Processes

Profiles are associated with kernel tasks, which roughly correspond to threads in user space (see clone(2) for details). Currently there are two ways how a profile can be associated with a task: when an executable is started and a profile is defined for that executable, or when the administrator assigns a profile to a task explicitly.

In addition to that, once a task is confined by a profile, that profile determines which other executables may be executed, and under which profile they may run (under the profile defined for that executable, the same profile as the current task, or unconfined; see Section 3.8).

A process will consist of a single task after an exec, so in the exec case, the entire process will be confined. New tasks (threads as well as processes) inherit the same profile and subprofile as their parent task.

Unconfined processes with the CAP_SYS_ADMIN privilege may assign a profile to a task with a command like this:

```
$ echo "setprofile /name/of/profile" > \
  /proc/$PID/attr/current
```

After that, the task will be in the new top-level profile, even if the process was in a subprofile before.

Processes with the CAP_SYS_ADMIN privilege as well as the process itself can query the profile a process is in by reading from that file:

```
$ cat /proc/$PID/attr/current
unconfined

$ cat /proc/$PID/attr/current
/name/of/profile (complain)

$ cat /proc/$PID/attr/current
/name/of/profile^hat_name (enforce)
```

The output includes the name of the profile and subprofile as well as the mode the active profile is in. (When a task is in a subprofile, the subprofile is the active profile.)

3.20 Profile Loading, Replacement, and Removal

Before the kernel can use any profiles, they must be loaded. The profile sources consist of plain text. This text representation is converted into in a binary representation that the kernel can more easily deal with by the user-space profile loader.

Profiles contain potentially long lists of file access rules that may include wildcards. In order to make the lookup efficient, the AppArmor kernel module does not actually go through all the file access rules when checking for access. Instead, the profile loader takes those rules and compiles them into transition tables. Pathnames are then looked up in those tables with a simple and efficient algorithm, the theory behind which is explained in the Lexical Analysis section of the Dragon Book [3].

An init script loads all the known profiles into the kernel at an early boot stage. This happens automatically and the system administrator tools will take care of loading, reloading, or removing profiles after they manipulate them, so end users will not usually notice this step.

Profiles can be replaced at any time during runtime, and all processes running under old profiles will transparently be switched to the updated versions. Profiles can also be removed. All processes running under a profile that is removed will become unconfined.

Profiles are always replaced together with all their subprofiles. It may be that an updated profile no longer contains a specific subprofile. If that happens while processes are using that subprofile, those processes will be put in a profile that denies all accesses. Such processes may still change to sibling subprofiles or back to the parent profile subject to the change_hat(2) semantics.

4 AppArmor Walk-Through

AppArmor consists of a set of kernel patches and accompanying user-space tools, both of which are available at `http://developer.novell.com/wiki/index.php/Apparmor`.

4.1 Kernel Patches and Configuration

The AppArmor kernel patches are provided in a format convenient for use with quilt,[2] however, other tools for applying the patches can be used, too. The patches are supposed to apply against recent kernel.org git kernels. A copy of the current git tree can be obtained from `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git` with *git clone* (see the git-clone(1) manual page). In case the the differences between the latest git tree and the tree the AppArmor patches are based on is too big, the patches won't apply cleanly. In this case, trying an older git tree may work better.

After obtaining the AppArmor patches tarball and the git tree which will end up in the linux-2.6 directory by default, the AppArmor patches can be applied to the git tree as follows:

```
$ tar zxvf apparmor.tar.gz
$ cd linux-2.6/
$ ln -s ../apparmor patches
$ quilt push -a
```

When configuring the kernel, make sure that AppArmor is built in or as a module (CONFIG_SECURITY_APPARMOR must be 'y' or 'm'). AppArmor cannot be used together with other Linux Security Modules, so if CONFIG_SECURITY_CAPABILITIES or CONFIG_SECURITY_SELINUX is set to 'y', they must be disabled by adding `selinux=0` and/or `capability.disable=1` to the kernel command line (grub, lilo, yaboot, etc.). It is not sufficient to put SELinux into permissive mode — at this time, AppArmor cannot be combined with other LSMs.

[2] `http://savannah.nongnu.org/projects/quilt`

4.2 The securityfs file system

AppArmor uses securityfs for configuration and to report information. The usual mountpoint for securityfs is /sys/kernel/security. Unless your distribution automatically does so, you can mount securityfs with:

```
$ mount securityfs -t securityfs /sys/kernel/security
```

Once securityfs has been mounted and the apparmor module loaded, /sys/kernel/security/apparmor/profiles will show the profiles loaded into the kernel, as well as mark if the profiles are in enforcement mode or in learning mode:

```
$ cat /sys/kernel/security/apparmor/profiles
/usr/bin/opera (complain)
/usr/lib/firefox/firefox.sh (complain)
/sbin/lspci (enforce)
...
```

Profile loading, replacement, and unloading, as well as configuration of AppArmor is also done via securityfs.

4.3 Profile Loading

Profile loading, replacement, and removal is performed by the apparmor_parser utility from the apparmor-parser package. The package can easily be built by running make in the package's top-level directory. Once that is done and the AppArmor module loaded, you may use the parser to load profiles with:

```
$ echo "/tmp/ls { /tmp/ls rm, }" | apparmor_parser
```

Once a profile for a program has been loaded into the kernel, you must use the –replace option for replacing the existing profile with a new one (this option may be used even if no profile by that name exists):

```
$ echo "/tmp/ls { /tmp/ls rm, }" | apparmor_parser --replace
```

4.4 Anatomy of a Profile

AppArmor profiles use a simple declaritive language, fully described in the apparmor.d(5) manual page. By convention, profiles are stored in /etc/apparmor.d/. The AppArmor parser supports a simple cpp-style include mechanism to allow sharing pieces of policy. A simple profile looks like this:

```
/bin/ls flags=(complain) {
  /bin/ls rm,
  /lib/ld-2.5.so rmix,
  /etc/ld.so.cache rm,
  /lib/lib*.so* rm,

  /dev/pts/* w,

  /proc/meminfo r,
  /var/run/nscd/socket w,
  /var/run/nscd/passwd r,
```

?	Any single character except “/”.
*	Any number of characters except “/”.
**	Any number of characters including “/”.
[ab]	One of “a” or “b”.
[a-c]	One of “a”, “b”, or “c”.
{ab,cd}	Alternation: either “ab” or “cd”.

Table 2: Globbing in File Access Rules. Alternation counts as an exact match in file access rules; all others count as wildcards (see Section 3.15).

```
  /var/run/nscd/group  r,

  /tmp/ r,
}
```

Here, the first /bin/ls is the name of the profile. This profile will be automatically used whenever an unconfined process executes /bin/ls. The flags instruct AppArmor to put the profile in complain (aka. learning) mode: in this mode, all operations are allowed, and any events that would have been denied are logged. This helps users to incrementally deploy AppArmor in production environments. The default if no flags are specified is enforcement mode, in which all operations not allowed by the profile are logged and denied.

Complain mode can be enabled individually for profiles as shown above (followed by reloading the profile), or by globally putting all profiles in complain mode with:

```
$ echo 1 > /sys/kernel/security/apparmor/control/complain
```

The user-space tools also include two small utilities, enforce and complain, which will put profiles into enforce or complain mode:

```
$ enforce firefox
Setting /usr/lib/firefox/firefox.sh to enforce mode.
```

Inside the body of the profile are any number of rules consisting of a pathname expression that may include globbing, and a set of permissions. Table 2 shows the supported shell-inspired globbing constructs; Section 3.8 on page 7 describes the permissions.

When AppArmor looks up a directory the pathname being looked up will end with a slash (e.g., /var/tmp/), otherwise it will not. Only rules that match that 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.

As explained in Section 3, AppArmor does not require execute access to allow directory traversal, or write access on a directory to create or rename files inside the directory. Instead, write access is required on the specific files

that a confined process attempts to create, remove, rename, etc. Read access is required for reading the contents of a directory.

AppArmor also mediates the use of POSIX 1003.1e draft capabilities; capabilities that a process is allowed to use are listed in the profile by their name in lower-case (with "CAP_" stripped off), e.g.,

```
#include <tunables/global>

/sbin/lspci {
  #include <abstractions/base>
  #include <abstractions/consoles>

  capability sys_admin,

  /sbin/lspci mr,
  /sys/bus/pci/ r,
  /sys/bus/pci/devices/ r,
  /sys/devices/** r,
  /usr/share/pci.ids r,
}
```

This profile uses predefined include files which are part of the apparmor-profiles package.

4.5 Logging

AppArmor uses the kernel standard audit facility for reporting. When a profile is in complain mode, the log messages look like this:

```
type=APPARMOR msg=audit(1174506429.573:1789): PERMITTING r access
 to /home/sarnold/ (ls(16504) profile /tmp/ls active /tmp/ls)
```

When a profile is in enforcement mode, the log messages look like this:

```
type=APPARMOR msg=audit(1174508205.298:1791): REJECTING r access
to /bin/ (ls(16552) profile /tmp/ls active /tmp/ls)
```

These log messages are sent to the kernel auditing facility; if auditd is not running, the kernel will forward these messages to printk for collection by klogd. Auditd must be configured with –with-apparmor to enable the #defines to handle AppArmor's message type correctly.

AppArmor also logs some important events in the process lifecycle, such as when processes in learning mode fork and change domain via exec. These other events, while not strictly related to permissions requested by the process, help the genprof profile generation tool reconstruct when specific accesses are required by processes — this allows the tool to make more relevant and meaningful policy suggestions.

4.6 Generating Profiles By Hand

While the majority of our users are expected to generate profiles with the help of our profile tools, it is possible to write policy by hand. This final section gives a very quick walkthrough generating a simple profile for firefox.

Since the kernel resolves symlinks to their "final destinations" before presenting AppArmor with policy questions, we first must see if /usr/bin/firefox is a symlink or the shell script that starts firefox; on our system, it is a symlink:

```
$ ls -l /usr/bin/firefox
lrwxrwxrwx 1 root root 25 Mar 21 13:36 /usr/bin/firefox ->
 ../lib/firefox/firefox.sh
```

So we will start a profile for /usr/lib/firefox/firefox.sh. This shell script will execute firefox-bin, as we will see later; when it does so, we will tell AppArmor to inherit this profile. Thus, firefox-bin will be executing under the profile for /usr/lib/firefox/firefox.sh.

To get started, we can make some assumptions about the privileges that firefox will need (both as a shell script and as a fairly complex GUI application):

```
$ cat /etc/apparmor.d/usr.lib.firefox.firefox.sh
/usr/lib/firefox/firefox.sh flags=(complain) {
  /usr/lib/firefox/firefox.sh r,
  /bin/bash rmix,
  /lib/ld-2.5.so rmix,
  /etc/ld.so.cache rm,
  /lib/lib*.so* rm,
  /usr/lib/lib*.so* rm,
}
$ apparmor_parser --reload < \
  /etc/apparmor.d/usr.lib.firefox.firefox.sh
Replacement succeeded for "/usr/lib/firefox/firefox.sh".
```

The easiest way to see what accesses AppArmor allows, start a tail -F /var/log/audit/audit.log (or /var/log/messages, or wherever your audit messages are being sent). In another terminal, start firefox. tail will show a few hundred PERMITTING audit events like these:

```
type=APPARMOR msg=audit(1174512269.026:1804): PERMITTING rw access
  to /dev/tty (firefox(16950) profile /usr/lib/firefox/firefox.sh
  active /usr/lib/firefox/firefox.sh)

type=APPARMOR msg=audit(1174512269.026:1805): PERMITTING r access
  to /usr/share/locale/locale.alias (firefox(16950) profile
  /usr/lib/firefox/firefox.sh active /usr/lib/firefox/firefox.sh)

type=APPARMOR msg=audit(1174512269.026:1806): PERMITTING r access
  to /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION (firefox(16950)
  profile /usr/lib/firefox/firefox.sh active
  /usr/lib/firefox/firefox.sh)
```

Because we want this profile to be fairly simple we'll be fairly permissive, add a few more rules to the profile and reload:

```
/dev/tty rw,
/usr/share/locale/** r,
/usr/lib/locale/** r,
```

Now re-run firefox. There is no need to handle all log entries at once. In complain mode, AppArmor will only report accesses that are not in the profile. This makes it fairly easy to add a few rules and re-run the application to determine what privileges are still necessary. We get a few more messages:

```
type=APPARMOR msg=audit(1174512791.236:5356): PERMITTING r access
  to /usr/lib/gconv/gconv-modules.cache (firefox(17031) profile
  /usr/lib/firefox/firefox.sh active /usr/lib/firefox/firefox.sh)

type=APPARMOR msg=audit(1174512791.236:5357): PERMITTING r access
  to /proc/meminfo (firefox(17031) profile
  /usr/lib/firefox/firefox.sh active /usr/lib/firefox/firefox.sh)

type=APPARMOR msg=audit(1174512791.240:5358): PERMITTING x access
  to /bin/basename (firefox(17032) profile
  /usr/lib/firefox/firefox.sh active /usr/lib/firefox/firefox.sh)

type=APPARMOR msg=audit(1174512791.240:5359): LOGPROF-HINT
  changing_profile pid=17032

type=APPARMOR msg=audit(1174512791.240:5360): PERMITTING r access
  to /bin/basename (firefox(17032) profile
  null-complain-profile active null-complain-profile)

...

 type=APPARMOR msg=audit(1174512791.240:5364): PERMITTING mr access
   to /bin/basename (basename(17032) profile
   null-complain-profile active null-complain-profile)
```

So now, we add a few more rules:

```
/usr/lib/gconv/** r,
/proc/meminfo r,
/bin/basename rmix,
```

We selected "rmix" for /bin/basename — most small shell utilities should not have a profile for themselves. There's nothing wrong with giving basename a profile, but the value of such a profile would be very limited. Giving other shell utilities their own profiles would be worse: the profile would need read access to the whole filesystem for shell scripts to function reliably. In our case, basename simply inherits privileges from another profile, then it has no more and no fewer privileges than the calling program — which is often a fine tradeoff.

The loader will need r and m access to execute basename, and we use ix to execute basename in the same profile. The kernel logs only reported r, m and x access; we have to choose the execute mode ourselves. Again, the standard user tools would prompt users for this decision and give consequences of decisions.

We continue in this fashion, iteratively adding and changing rules as needed by the logs. Some of the logs report attribute modifications, such as:

```
type=APPARMOR msg=audit(1174519157.851:10357): PERMITTING
  attribute (mode,ctime,) change to
  /home/sarnold/.gnome2_private/ (firefox-bin(17338) profile
  /usr/lib/firefox/firefox.sh active /usr/lib/firefox/firefox.sh)
```

These need to be represented in the profile with simple w access.

```
/home/*/.gnome2_private/ w,
```

After nine iterations, the profile looks like this — we have inserted blank lines between each iteration:

```
/usr/lib/firefox/firefox.sh flags=(complain) {
  /usr/lib/firefox/firefox.sh r,
  /bin/bash rmix,
  /lib/ld-2.5.so rmix,
  /etc/ld.so.cache rm,
  /lib/lib*.so* rm,
  /usr/lib/lib*.so* rm,

  /dev/tty rw,
  /usr/share/locale/** r,
  /usr/lib/locale/** r,

  /usr/lib/gconv/** r,
  /proc/meminfo r,
  /bin/basename rmix,

  /usr/bin/file rmix,
  /etc/magic r,
  /usr/share/misc/magic.mgc r,
  /bin/gawk rmix,
  /usr/lib/firefox/firefox-bin rmix,

  /usr/lib/firefox/lib*so rm,
  /opt/gnome/lib/lib*so* rm,
  /usr/share/X11/locale/* r,
  /var/run/nscd/socket w,
  /var/run/nscd/passwd r,

  /usr/share/X11/locale/** r,
  /home/*/.Xauthority r,
  /usr/lib/gconv/*so m,
  /home/*/.mozilla/** rw,
  /etc/resolv.conf r,
  /usr/lib/firefox/**.so rm,
  /usr/lib/firefox/** r,

  /etc/opt/gnome/** r,
  /var/run/dbus/system_bus_socket w,
  /etc/localtime r,
  /opt/gnome/lib/**.so rm,
  /var/cache/libx11/compose/* r,
  /tmp/orbit-*/ w,
  /dev/urandom r,
  /tmp/ r,
  /dev/null rw,
  /opt/gnome/lib/GConf/2/gconfd-2 rmix,

  /dev/log w,
```

```
  /tmp/orbit-*/* w,
  /tmp/gconfd-*/ r,
  /tmp/gconfd-*/** rwl,
  /home/*/.gconf/ r,
  /home/*/.gconf/* rw,
  /etc/fonts/** r,
  /var/cache/fontconfig/* r,
  /home/*/.fontconfig/** r,
  /usr/share/ghostscript/fonts/** r,
  /etc/passwd r,
  /var/tmp/ r,
  /bin/netstat rmix,

  /home/*/.gnome2_private/ w,
  /home/*/.gconfd/* rw,
  /proc/net/ r,
  /proc/net/* r,
  /usr/share/fonts/** r,
  /usr/lib/browser-plugins/ r,
  /usr/lib/browser-plugins/** rm,
}
```

Sorting the entries in the profile can help show areas that can be collapsed with even more generic rules. After doing that and making a few rules slightly more generic, we end up with:

```
/usr/lib/firefox/firefox.sh {
  /bin/basename rmix,
  /bin/bash rmix,
  /bin/gawk rmix,
  /bin/netstat rmix,
  /dev/log w,
  /dev/null rw,
  /dev/tty rw,
  /dev/urandom r,
  /etc/fonts/** r,
  /etc/ld.so.cache rm,
  /etc/localtime r,
  /etc/magic r,
  /etc/opt/gnome/** r,
  /etc/passwd r,
  /etc/resolv.conf r,
  /home/*/.fontconfig/** r,
  /home/*/.gconfd/* rw,
  /home/*/.gconf/ r,
  /home/*/.gconf/* rw,
  /home/*/.gnome2_private/ w,
  /home/*/.mozilla/** rw,
  /home/*/.Xauthority r,
  /lib/ld-2.5.so rmix,
  /lib/lib*.so* rm,
  /opt/gnome/lib/GConf/2/gconfd-2 rmix,
  /opt/gnome/lib/**.so* rm,
  /proc/meminfo r,
```

```
  /proc/net/ r,
  /proc/net/* r,
  /tmp/gconfd-*/ r,
  /tmp/gconfd-*/** rwl,
  /tmp/orbit-*/ w,
  /tmp/orbit-*/* w,
  /tmp/ r,
  /usr/bin/file rmix,
  /usr/lib/browser-plugins/ r,
  /usr/lib/browser-plugins/** rm,
  /usr/lib/firefox/firefox-bin rmix,
  /usr/lib/firefox/firefox.sh r,
  /usr/lib/firefox/** r,
  /usr/lib/firefox/**.so rm,
  /usr/lib/gconv/** r,
  /usr/lib/gconv/*so m,
  /usr/lib/lib*.so* rm,
  /usr/lib/locale/** r,
  /usr/share/** r,
  /var/cache/fontconfig/* r,
  /var/cache/libx11/compose/* r,
  /var/run/dbus/system_bus_socket w,
  /var/run/nscd/passwd r,
  /var/run/nscd/socket w,
  /var/tmp/ r,
}
```

References

[1] AppArmor documentation, `http://www.novell.com/documentation/apparmor/`

[2] Al Viro and Ram Pai: *Shared-Subtree Concept, Implementation and Applications in Linux,* Ottawa Linux Symposium, July 19-22, 2006, `http://www.linuxsymposium.org/2006/`

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools* (The "Dragon Book"), Addison-Wesley, 1986, ISBN 0-201-10088-6.

A second edition of this classic is available since August 2006 as ISBN 0-321-48681-1.