AppArmor

Kernel-level Process Confinement

AppArmor Introduction

AppArmor is a host-based intrusion prevention system.

It's specifically intended to stop an attacker who compromises one component of the system from being able to do anything else on the system.

AppArmor was implemented as a Linux kernel security module, using the same LSM interface as SELinux.

AppArmor's History

Immunix created AppArmor in 1998 as part of a commercial Linux distribution

Novell bought Immunix in 2005, integrated AppArmor into SUSE Linux, and released it as open source software.

Ubuntu made AppArmor its default technology in 2007.

In 2010, AppArmor became part of the mainstream kernel. (Linux kernel v2.6.36).

Security Model

AppArmor is anomaly prevention for application security

- Focus on *application* security
- Enforces normal, non-attacked application behaviour
- Name-based access control for ease of understanding policy
- Use full regular expression support for flexibility
- Automated tools for profile development
- Provides fine-grained security, even at *sub-process* level

Program-based Access Control

•Whenever a protected program runs regardless of UID, AppArmor controls:

- The POSIX capabilities it can have (even if it is running as root)
- The directories/files it can read/write/execute
- The network capabilities the program has.

Example security profile for **ntpd**

```
/usr/sbin/ntpd {
  #include <abstractions/base>
  #include <abstractions/nameservice>

  capability ipc_lock,
  capability net_bind_service,
  capability sys_time,
  capability sys_chroot,
  capability setuid,

  /etc/ntp.conf                r,
  /etc/ntp/drift*                              rwl,
  /etc/ntp/keys                r,
  /etc/ntp/step-tickers                        r,
  /tmp/ntp*                                    rwl,
  /usr/sbin/ntpd               rix,
  /var/log/ntp                 w,
  /var/log/ntp.log             w,
  /var/run/ntpd.pid            w,
  /var/lib/ntp/drift           rwl,
  /var/lib/ntp/drift.TEMP      rwl,
  /var/lib/ntp/var/run/ntp/ntpd.pid            w,
  /var/lib/ntp/drift/ntp.drift                 r,
  /drift/ntp.drift.TEMP        rwl,
  /drift/ntp.drift             rwl,
}
```

Native Linux Syntax and Semantics

Access controls reflect classic permission patterns

- Complements Linux file permissions rather than overlaying a new paradigm.

File globbing, in the traditional Linux/UNIX format.

- `/dev/{,u}random` matches `/dev/random` and `/dev/urandom`
- `/lib/ld-*.so*` matches most of the libraries in `/lib`
- `/home/*/.ssh/config` matches everyone's `.ssh.config` files
- `/home/*/public_html/**` matches everyone's public HTML directory tree

Profile Building Blocks

There are a set of “foundation class” rules that can be #include'd in your profiles.

Canonical and Novell help maintain these.

Here are some basic ones:

- **`base`**: needed by nearly all programs
- **`authentication`**: program will authenticate users
- **`console`**: program interacts with TTY consoles
- **`kerberos`**: uses Kerberos cryptography
- **`nameservice`**: program needs to look up domain names
- **`wutmp`**: program updates user login logs

AppArmor vs SELinux

AppArmor is far easier to learn and understand than SELinux.

SELinux can be used to create much better assurance, especially when you're working with the need for compartmentalization.

AppArmoring a Program

Here's a process for testing AppArmor on a program:

1. Pick an exploitable program.
2. Try the exploit on the program.
3. Develop an AppArmor profile for the program.
4. Repeat the exploitation against the protected program.

Profiling the Program

Use `aa-genprof`, which seeks to generate a profile for a program by running it and seeing what it does.

Just like SELinux's `audit2allow`, which we'll look at later, `aa-genprof` runs a program with the profile set to a non-enforcing "complain" mode. It then looks at the alerts and helps you build a profile to avoid the alerts.

Profiling Highlights

`aa-genprof` will notice that our program needs one or more POSIX capabilities.

`aa-genprof` may ask about an individual file the program wants to read. We can use globbing to allow that access to multiple files, or perhaps all files in a directory.

`aa-genprof` may generalize, asking if our program needs access to an entire include-file abstraction.

We can tell `aa-genprof` to generalize where we see fit as well.

Exercise

Please:

Open the Firefox browser on the class machine to:
http://localhost:10000/exercises/billubox-apparmor

Reloading a Profile

What if you missed something during profiling?

You can run `aa-genprof` again, to add to an existing profile.

Always restart the program when you modify the profile.

Putting a Profile in Complain Mode

If you put the program's profile into "complain" mode, it isn't enforced. Instead, the kernel will write log messages saying what it would have blocked.

```
aa-complain /usr/sbin/program
```

Add rules to the existing profile in `/etc/apparmor.d/usr.sbin.program`

Then reload the profile and restart the program:

```
aa-disable /usr/sbin/program
aa-enforce /usr/sbin/program
```

Final Note

You need to make sure AppArmor is activated on boot. For example, Debian versions before Debian 10 ("Buster") need manual AppArmor activation in the bootloader config:

```
$ sudo mkdir -p /etc/default/grub.d
$ echo 'GRUB_CMDLINE_LINUX_DEFAULT="$GRUB_CMDLINE_LINUX_DEFAULT apparmor=1 security=apparmor"' \
  | sudo tee /etc/default/grub.d/apparmor.cfg
$ sudo update-grub
$ sudo reboot
```

https://wiki.debian.org/AppArmor/HowToUse