Why I seldom file bugs against SELinux policy

Jan Pazdziora
Sr. Principal Software Engineer
OpenShift Security, Red Hat
jpazdziora@redhat.com

DEVCONF.cz
26th January 2018

SELinux policy in distribution

- Distributions like Fedora, CentOS, or RHEL ship software in packages (rpm).
- By default, `selinux-policy` and `-targeted` (or `-mls`) packages with SELinux policy also get installed.
  - Rules for labeling files when rpm installs them (file contexts):

    `/etc/selinux/targeted/contexts/files/`

    - `rpm -ql` *`package`* `| xargs ls -d`**`Z`**
    - `restorecon(8)`
  - Rules that permit or deny interactions of processes with files, directories, ports, and other components of the operating system.
    - `sesearch(1)`

AVC denials

- When SELinux policy does not allow application to do something ...
  - things start to fail;
  - AVC (Access Vector Cache) denial is often logged.
- Sometimes, nothing is visibly broken, things work.
  - Yet AVC denials are still logged.
    - Is that the better situation ... or worse?
- Who decides if what application attempted to do was correct?
- Maybe the SELinux policy is right?

Where to find AVC denials?

- `/var/log/audit/audit.log`, search for `type=AVC`
- journal via `journalctl`, search for `AVC avc`
- `setroubleshootd` and `sealert`
- in beaker jobs in `/avc` subtests

- Let's look at some AVC denials.
- The goal is **not** to point fingers but to sharpen instincts.

Application and policy mismatch

```
avc:  denied  { read } for  pid=3728 comm="abc.cgi"
    name="objects.cache" dev="dm-0" ino=2364515
    scontext=system_u:system_r:xyz_script_t:s0
    tcontext=system_u:object_r:xyz_spool_t:s0 tclass=file permissive=0
```

```
avc:  denied  { open } for  pid=4311 comm="def.cgi"
    path="/var/spool/xyz/objects.cache" dev="dm-0" ino=2364515
    scontext=system_u:system_r:xyz_script_t:s0
    tcontext=system_u:object_r:xyz_spool_t:s0 tclass=file permissive=0
```

- Someone needs to decide if their CGI scripts should be allowed to work with the spool directory directly.
- To check if these are allowed on the current distribution:

```
$ sesearch --allow -s xyz_script_t -t xyz_spool_t -c file
```

Application and policy mismatch

```
avc:  denied  { execute_no_trans } for  pid=9992 comm="xyzd"
    path="/usr/libexec/xyz/xyz-action" dev="dm-0" ino=5833743
    scontext=system_u:system_r:xyz_t:s0
    tcontext=system_u:object_r:xyz_exec_t:s0 tclass=file permissive=0
```

- Someone needs to decide if their daemon should be allowed to call its utility directly.

```
avc:  denied  { execute } for  pid=19919 comm="xyz"
    name="prog" dev="md1" ino=57787811
    scontext=system_u:system_r:xyz_t:s0-s0:c0.c1023
    tcontext=system_u:object_r:prog_exec_t:s0 tclass=file permissive=0
```

- New feature to process output of external program instead of just files.
- Someone has to drive reflecting the new feature in the SELinux policy.
- Possibly via SELinux boolean.

Application and policy mismatch

```
avc:  denied  { create } for  pid=1913 comm="xyz-actiond"
    name=".#abc.confd7rHli"
    scontext=system_u:system_r:xyz_actiond_t:s0
    tcontext=system_u:object_r:etc_t:s0 tclass=file permissive=0
```

- If the utility should be creating config files in `/etc`, that needs to be reflected in the SELinux policy.

```
avc:  denied  { create } for  pid=24914 comm="xyz" name="ABC"
    scontext=system_u:system_r:xyz_t:s0
    tcontext=system_u:object_r:tmp_t:s0 tclass=file
```

- Different application, creating cache under `/var/tmp`.
- In the end, `xyz` was changed to use `/var/lib/xyz/cache` for its data.

Aligning application and policy

- Very often, the name of the game is not to “fix SELinux policy”.
  - Often, the policy is not broken, to be fixed.
- The task is to match the security policy with the service or application deployment expectations.
  - Yes, often yet another `allow` (or appropriate macro) gets added to the policy to allow the interaction.
  - But sometimes, AVC denials can point to suboptimal application setup.

Homework assignment

- Pick software dear to your heart.
- Figure out what SELinux types it uses for its files on disk ...
  - ... and for processes that it starts.
- Check what operations are allowed for those processes.
- Focus on “weird” things like

```
allow xyz_t xyz_log_t:file { ... unlink write };
```

- Get curious.
  - Why should an attacker be able to overwrite or remove the logs?

Unnecessary functionality

```
avc:  denied  { write } for  pid=977 comm="xyzd"
    name="root" dev="dm-0" ino=13
    scontext=system_u:system_r:xyzd_t:s0
    tcontext=system_u:object_r:admin_home_t:s0 tclass=dir permissive=0
```

- The gvfs creates `~/.cache` at startup if `XDG_RUNTIME_DIR` is not set.
- But xyz does not need gvfs at all, in the daemon.
- Patched with

```
+Environment=GVFS_DISABLE_FUSE=1
+Environment=GIO_USE_VFS=local
+Environment=GVFS_REMOTE_VOLUME_MONITOR_IGNORE=1
```

Unfortunate config defaults

```
avc:  denied  { create } for  pid=13354 comm="xyz" name="xyz"
    scontext=system_u:system_r:xyz_t:s0
    tcontext=system_u:object_r:xyz_log_t:s0 tclass=sock_file permissive=0
```

- Upstream's configuration sets its socket location under `/var/log/xyz`.
- Changed to configure it under `/var/spool/xyz`.
- Following the general purpose of directories brings order and makes the setup more secure.

Executable memory

```
avc:  denied  { execmem } for  pid=805 comm="xyzd"
    scontext=system_u:system_r:xyz_t:s0
    tcontext=system_u:system_r:xyz_t:s0 tclass=process permissive=0
```

- Patched with

```
-LINKFLAGS="%{?__global_ldflags}"
+LINKFLAGS="%{?__global_ldflags} -Wl,-z,noexecstack"
```

How much should the application do?

```
avc:  denied  { module_load } for  pid=12682 comm="modprobe"
    scontext=system_u:system_r:xyz_t:s0
    tcontext=system_u:system_r:xyz_t:s0 tclass=system permissive=0
```

- The daemon calls `modprobe` directly.
  - If compromised, different module name can be passed to it.
- Perhaps `xyz_t` should not be allowed to load kernel modules at all.
  - A single-purpose helper program with domain transition just to load specific module might be safer approach.
  - Or the module can be loaded before the daemon gets started (systemd service setup).

Application too eager to list directory

```
avc:  denied  { getattr } for  pid=28229 comm="xyz"
    path="/etc/group.lock" dev="dm-1" ino=135358895
    scontext=system_u:system_r:xyz_t:s0
    tcontext=unconfined_u:object_r:shadow_t:s0 tclass=file
```

- AVC denial was logged while no requests were being processed by `xyz`.
- Application used `inotify` on `/etc`.
- It tried to `stat` any file that got changed, even if it only cared about `resolv.conf`.

Potential shell command injection

```
avc:  denied  { execute } for  pid=1938 comm="xyz"
    name="bash" dev="vda1" ino=5442
    scontext=system_u:system_r:abc_t:s0
    tcontext=system_u:object_r:shell_exec_t:s0 tclass=file permissive=0
```

- I've grepped xyz sources and I do not see an explicit call to shell.
- Maybe single-parameter `system` or exec call in scripting language is used where multi-parameter one would avoid shell invocation?

Filehandle leak

```
avc:  denied  { write } for  pid=11813 comm="abc"
    path="/tmp/xyz.lock" dev=dm-0 ino=1048770
    scontext=unconfined_u:unconfined_r:abc_t:s0-s0:c0.c1023
    tcontext=unconfined_u:object_r:user_tmp_t:s0 tclass=file
```

- The xyz created lock file and let its filehandle leak to abc.
- Patched with

```
-LOG_LOCK = open(lockfile(), 'w')
+LOG_LOCK = open(lockfile(), 'we')
```

Bug in config code

```
avc:  denied  { read } for  pid=26234 comm="xyz"
    path="/.xyz_data/7Ggn3Ecq/data" dev="dm-1" ino=1835032
    scontext=system_u:system_r:abc_t:s0-s0:c0.c1023
    tcontext=system_u:object_r:root_t:s0 tclass=file permissive=0
```

- The setup script was using uninitialized $HOME
- Patched with

```
+[ "$HOME" ] || HOME=`getent passwd $ID | cut -d: -f6`
 datadir="$HOME/.xyz_dir"
-[ "$HOME" ] || HOME=`getent passwd $ID | cut -d: -f6`
```

Bug in application C code

```
avc:  denied  { module_request } for  pid=25312 comm="xyz"
    kmod="net-pf-0" scontext=system_u:system_r:container_t:s0:c40,c45
    tcontext=system_u:system_r:kernel_t:s0 tclass=system permissive=0
```

- Application code called `socket` on zeroed `sockaddr`.
- Patched with

```
+       if (!AF(addr))
+               return NULL;
        s = socket(AF(addr), SOCK_DGRAM, 0);
```

- Only caught in container because containerized domains are more restricted.

Who should get notified?

- These days, when the admin did not play the defaults too much, things work with the default SELinux enforcing setup just fine.
- Who should get notified when you see an AVC denial / “SELinux bug”?
- Remember, the task is to ...
  - Match the security policy with the service or application deployment expectations.
- Package maintainers know their applications and changes in them much better than SELinux policy maintainers.
- When you see an AVC denial, you should notify ...
  - of all people ...

I don't file bugs against SELinux policy

- When I see an AVC denial, I notify ...
  - Package maintainers of the software package.
  - By filing bugzilla against the appropriate component.

And so should you!
;-)

- Note: It's OK when after reviewing and assessing it, the component maintainers reassign it to `selinux-policy` with an RFE to align the policy with the latest application behaviour.

Further work

- The midterm homework: check SELinux policy of your favourite service.
- Paul Moore's SELinux Loves Modularity DevConf.cz talk on Sunday, January 28, 2018, at 11:00 am CET in C-D0207.
- Provide feedback about this session:
  - sched.co/DJXc
  - jpazdziora@redhat.com