

## New in-kernel privilege API: priv(9)

---

*Source:* <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/arch/2006-09/msg00045.html>

---

- *From:* Robert Watson <[rwatson@xxxxxxxxxxx](mailto:rwatson@xxxxxxxxxxx)>
  - *Date:* Wed, 13 Sep 2006 15:29:14 +0100 (BST)
- 

Dear all,

Over the past few weeks, I've been working on a replacement for the `suser(9)` API, used to check whether a thread or credential has the privilege to override discretionary access control or perform system configuration operations in the kernel. Currently, these checks use one of two kernel APIs:

`suser(thread)`

or:

`suser_cred(cred, flags)`

The former is the more common invocation, but the latter is also often used; this is largely because `jail(4)` requires limits of superuser privilege, so instances of privilege allowed in jail are explicitly marked via the `flags` field. There are also circumstances in which only a credential is available, perhaps cached from another context, and a very small number of instances (2) where a second flag, forcing use of the `ruid` instead of the `euid`, is used. The above API has served FreeBSD well for many years. However, it suffers from a number of architectural and functionality inadequacies. The goal of my work has been to address a particular functional lack: granularity. In particular, there are a number of things that finer granularity in the API would allow us to do:

– Make it easier to explore the finer-grained granting of privilege via policy, such as assigning specific useful privileges — the ability to bind a port, configure a SLIP interface, adjust the time, be exempt from audit requirements, be allowed to attach to a jail, override certain file permissions, set quotas, configure IP addresses, etc, which are cleanly separable (not to mention usefully assignable) privileges.

– Make it easier to explore the finer-grained denial of privilege. For example, `jail` is in large part based on a marking of different privilege checking points as being "allowed in jail" or "not allowed in jail". In some ways this is advantageous: the implementer of each `suser` check gets to decide whether it's in jail, and that information is available in the context of the check. However, this has several important disadvantages. Not least is that the implementation of `jail` is highly distributed rather than centralized, making auditing the implementation difficult. Another disadvantage is that configuration options that vary the behavior of `jail` are also distributed throughout the kernel rather than centralized, as they

## New in-kernel privilege API: priv(9)

must vary whether the `SUSER_ALLOWJAIL` flag is being passed into `suser`. It would be nice to be able to quickly and easily answer the question "what privileges are granted in jail", and to easily vary the list, which is not possible currently.

– Make it easier to identify, categorize, and audit the use of privilege throughout the kernel by actually having a list of the privileges and what they correspond to, as well as making it easier to identify all the places a specific privilege is used. This facilitates auditing of kernel privilege use, and easy comparison of the use of identical privileges in different subsystems. For example, while doing this work, I identified inconsistencies in the application of superuser privilege in different file systems, privileges that were sometimes allowed in jail, but sometimes not, etc. 200 anonymous `suser` checks are hard to analyze, 160 named privilege checks are much easier to analyze.

– Make it easier to modify the audit mechanism to capture a log of exactly what privileges are exercised during operation, a requirement for higher assurance evaluation.

What does this all mean in practice? It means replacing `suser(9)` and `suser_cred(9)` with calls that express the specific privilege being checked for. I took the most straight forward possible implementation: I reviewed all privilege checks in the kernel, identified all identical privileges and categorized all privileges by subsystem. I then assigned unique numeric constants to each unique privilege, and added a privilege identifier argument to the two new functions, `priv_check(9)` and `priv_check_cred(9)`. Here are a few sample snippet from the privilege list in `src/sys/priv.h`:

```
...
PRIV_ACCT, /* Manage process accounting. */
PRIV_MAXFILES, /* Exceed system open files limit. */
PRIV_MAXPROC, /* Exceed system processes limit. */
PRIV_KTRACE, /* Set/accept KTRFAC_ROOT on ktrace. */
PRIV_SETDUMPER, /* Configure dump device (XXX: needs work). */
PRIV_NFSD, /* Can become NFS daemon. */
PRIV_REBOOT, /* Can reboot system. */
PRIV_SWAPON, /* Can swapon(). */
PRIV_SWAPOFF, /* Can swapoff(). */
...
PRIV_PMC_MANAGE, /* Can administer PMC. */
PRIV_PMC_SYSTEM, /* Can allocate a system-wide PMC. */
PRIV_SCHED_DIFFCRED, /* Exempt scheduling other users. */
PRIV_SCHED_SETPRIORITY, /* Can set lower nice value for proc. */
PRIV_SCHED_RTPRIO, /* Can set real time scheduling. */
PRIV_SCHED_SETPOLICY, /* Can set scheduler policy. */
PRIV_SCHED_SET, /* Can set thread scheduler. */
PRIV_SCHED_SETPARAM, /* Can set thread scheduler params. */
...
PRIV_UFS_SETQUOTA, /* setquota(). */
PRIV_UFS_SETUSE, /* setuse(). */
PRIV_UFS_EXCEEDQUOTA, /* Exempt from quota restrictions. */
PRIV_VFS_READ, /* Override vnode DAC read perm. */
```

## New in-kernel privilege API: priv(9)

```
PRIV_VFS_WRITE, /* Override vnode DAC write perm. */
PRIV_VFS_ADMIN, /* Override vnode DAC admin perm. */
PRIV_VFS_EXEC, /* Override vnode DAC exec perm. */
PRIV_VFS_LOOKUP, /* Override vnode DAC lookup perm. */
PRIV_VFS_BLOCKRESERVE, /* Can use free block reserve. */
...
```

As you can see, they break down into both a set of system management privileges, relating to configuring kernel services, and then a set of specific privileges associated with (and sorted by) major kernel subsystems. None of this implies a change in underlying policy — just that a bit more contextual information is passed into the privilege check. This has some important specific functional benefits:

- It makes it possible to migrate the "allowed in jail" decision from the calling context to the privilege management code. This will allow us to gradually eliminate the passing of flags to the privilege check code under almost all circumstances. In my patch, I have added a new function to `kern_jail.c`, `prison_priv_check()`, which essentially contains a switch statement listing the privileges allowed in jail, and denying the rest. Configurable privileges, raw socket access, etc, can now occur in one place, and open the door to introducing more easy per-jail configuration of privilege. After these changes, the implementation is much more centralized in `kern_jail.c`.

- It makes it possible for the MAC Framework to restrict access to privilege, a feature required for the SEBSD policy module, which implements the FLASK/Type Enforcement policy environment as found in SELinux. Policy modules can register interest in privilege checks, and then specifically deny access to privileges as they see fit.

- It makes it possible for the MAC Framework to allow policies to grant privilege. Policy modules can register interest in privilege checks, and then specifically grant access to privileges as they see fit.

In order to demonstrate MAC Framework integration with the privilege system, I have implemented a sample policy module, `mac_privs`, which allows rule-based granting of privileges to specific uids. Using a command line tool, appropriately privileged processes can modify the rule list, granting named privileges to unprivileged users. This is not a particularly mature example of a privilege-granting policy, as ideally privilege is something that is available but not always exercised — i.e., similar to a setuid root binary that switches the effective uid to root only when it specifically needs privilege. However, it's quite useful in practice, and demonstrates how configurable policies can interact with kernel privilege decisions.

In the past, I've done similar work on two occasions: once in implementing POSIX.1e privileges for FreeBSD as part of the TrustedBSD Project (not merged), and once as part of the SEBSD implementation. This work is functionally similar, but there are several important ways in which this design differs from the POSIX.1e approach (also used in Linux):

- The identification of privileges is quite fine-grained. The Linux-extended POSIX.1e privilege set contains high level privileges like "Network privilege", which encapsulates a broad range of different network privilege checks. I have identified over 50 different specific network privileges, each separately named. It would be easy to map these into the POSIX.1e

## New in-kernel privilege API: priv(9)

privilege set, which is presumably what the SEBSD policy will need to do in order to produce the narrower set expected by the SELinux code.

- The approach is intended to allow the granting as well as denying of privilege. This is an important design choice, and has both some costs and some benefits. One important benefit is that it has historically proven difficult to take rights away from the root user without introducing security vulnerabilities associated with applications written to use root privilege expecting that all privileges be in place. Granting specific privileges implies a fairly different application and policy construction and may well be safer.
- Because of the fine-grained naming of privileges, it's possible to encapsulate jail in a way that was not previously possible: the POSIX.1e privilege set was simply too coarse to capture the requirements of jail.
- Privileges under this model are not treated as maskable values. In practice, there are very few situations in which it is useful to check multiple privileges at once, and permitting that encourages authors adding new privilege checks to combine privileges in a way that makes it opaque to the privilege mechanism as to which privilege was actually needed. This also has the benefit of making it much easier/more efficient to add new privileges as required, as it doesn't require expanding a bit string representing the privileges. Most POSIX.1e implementations limit the total number of privileges to 32 to 64 in order to have them fit in a bitmask easily.
- By assigning new privileges for every privilege with significantly different semantic, the question of "when to add a new privilege" is answered: unless there is an obvious match, you add one. With the POSIX.1e + Linux set, it is necessary to try to figure out how to fit a new check into one of many poorly matching privileges. The result was that almost all privileges not clearly matched to one of the POSIX.1e set ended up in the catch-all CAP\_SYS\_ADMIN.

The status of this work is that a pretty functional prototype can be found in Perforce:

`//depot/projects/trustedbsd/priv/...`

A snapshot patch from the branch, excluding mac\_privs, can be found here:

<http://www.watson.org/~robert/freebsd/20060913-trustedbsd-priv.diff>

In that tree, you'll want particularly to look at:

- sys/kern/kern\_jail.c Revised jail privilege behavior
- sys/kern/kern\_priv.c Privilege check implementation
- sys/security/mac/mac\_priv.c MAC extensions for privileges
- sys/security/mac\_privs/\* Sample MAC policy granting privileges
- sys/sys/priv.h Privilege list, API
- share/man/man9/priv.9 Draft man page

## New in-kernel privilege API: priv(9)

usr.sbin/mac\_privs/\* Management tool for sample MAC policy

It is my intent, following review, discussion, cleanup, etc, to commit the priv(9) work, sans mac\_privs, to the 7.x tree in the next couple of weeks. The mac\_privs policy is a sample policy that will continue to be maintained as part of the TrustedBSD Project, but not merged into the base tree at this point. Some remaining TODO items are:

- Review various XXX comments I added as part of this work.
- Complete modification of System V IPC code to properly check privileges.
- Update mac\_none.c sample policy to include privilege stubs.
- Possibly move securelevel support to kern\_priv.c, since it largely relates to privilege.
- Teach the audit subsystem to collect privilege information during a system call, and add it to audit records using privilege tokens (already present in Solaris).
- Complete man page updates, including finalize priv.9, trim down suser.9.
- Create further privilege-related regression tests.
- Finalize decision on using an enum or an int to identify privileges. Using an enum requires more namespace pollution, and requires hard-coded values anyway in order to avoid ABI issues. Possibly using #defines would be simpler.

I'd like to gratefully acknowledge the sponsorship of nCircle Network Security, Inc in performing this work.

Robert N M Watson  
Computer Laboratory  
University of Cambridge

---

freebsd-arch@xxxxxxxxxxx mailing list

<http://lists.freebsd.org/mailman/listinfo/freebsd-arch>

To unsubscribe, send any mail to "freebsd-arch-unsubscribe@xxxxxxxxxxx"