Subject: design of user namespaces
Posted by ebiederm on Fri, 20 Jun 2008 02:01:08 GMT
View Forum Message <> Reply to Message

I got to thinking about the user namespace again
today and I wanted to summarize my thoughts.

First and foremost I want a design that solves the suid problem,
and I think I have.

In particular if we can let an unprivileged user run a process
in a different user namespace, and become a user with 0 == uid == gid
and have all capabilities except CAP_SYS_RAWIO and they still can not
do anything security wise they could not before. We have a very useful design.

My thinking is as follows.
- All user visible security credentials uid, gid, selinux filesystem lables,
  etc will be relative to a user namespace.

  This makes security easier, and it the only thing that makes
  sense in the concept of separate administrative domain.

- Each struct user will live in exactly one user namespace.

- Each user namespace will hold the struct user of the user who
  created the user namespace.

- capable() will be extended to take a user namespace parameter. So
  the question capable asks is does the current process have the
  capability with respect to the current user namespace.

  This makes security namespaces loosely hierarchical. And the rule
  would be that if you could do something to the files and directories
  and other objects owned by the creator of the namespace you can do
  it to objects owned by users in the user namespace.

  Going the other direction from child to parent user namespaces,
  users in user namespace no matter what capabilities they have will
  have no permissions except those given to every process.

- Each user namespace will have a bounding set of capabilities that
  suid executables and the initial process get. Basically something
  so fundamentally machine specific capabilities like CAP_SYS_RAWIO
  can never be set inside of a user namespace unless the creator of
  the user namespace had those capabilities.

- The initial user in a user namespace will be uid 0 and have all
  capabilities, non machine specific cpabilities CAP_NET_ADMIN,

CAP_SYS_ADMIN, CAP_SYS_CHROOT etc. Since those capabilities will be restricted to objects in the user namespace they will have no security implications.

- struct vfsmount will include a user namespace pointer. Recording in which mount namespace the mount occurred, and not changing when struct vfsmount is copied or propagated (including through bind mounts). This user namespace will act as the user namespace to map all security credentials on the filesystem into. If you are accessing the vfsmount from another user namespace the best you can get is a the world readable permissions.

  struct vfsmount will also include a filesystem data pointer. Not changing except when struct user changes. Giving filesystems a place to store their per user namespace credential translation state.

  This allows is shared filesystem caches between user namespaces.

- For filesystems to be safely shared between 2 user namespaces we need two things. The owner of the filesystem has to allow it. The user of the filesystem needs to request it.

  For user namespaces we have two cases. Allowing native mounts to a user namespace of a previously mounted filesystem. Allowing native mounts to of an unmounted filesystem.

  Working with previously mounted filesystems is the safest as we don't have to deal with the hard problem of poisoned filesystem data trying to crash our filesystem implementation.

  For previously filesystems the simplest and most comprehensive way I can see to do this is to implement a special case of mount -o remount. In which a parameter is passed telling the filesystem which credential mapping strategy to employ. The credential mapping strategies I have seen to date are
  - identity mapping read-only.
  - identity mapping read-write (this is a problem with quotas)
  - uid/gid offset by a constant (solves the quota problem)
  - security label for namespaces (can solve the quota problem).

  Then the already mounted filesystem can either performs an upcall or examine it's configuration (potentially stored in a normal file in the filesystem like the quotas are) to see if the remount into native mode for the user namespace can be allowed.

  For the specific and very common case of identity mapping read-only filesystems we could even have a completely generic mount flag you

can set a priori to allow any user namespace to remount it native.

Most unix filesystems have common enough properties that we can implement a library they can wire up to implement this functionality without requiring an on disk format change. So despite it being filesystem specific functionality we can extensive share the code.

For network filesystems like nfs I expect the request would go to the server and authenticating a new mount. It is still safer then a totally new nfs mount because an unprivileged user can not specify the server.

- Previously I really wanted to say just do something like idmapd and we would be golden. The problem of quotas in different namespaces would be solved, etc. After thinking about it I realized there is no same place to run such a mapping daemon that could map between arbitrary user namespaces that could do something useful and not also compromise security.

  Mapping daemons are good at changing the form of security tokens. Say by looking up user names in /etc/passwd. They are not sufficient to perform a general mapping.

  The owner of the filesystem has to configure what you are allowed to see and access. Which uids you can use, which directories you can use etc. While it is the job of the mapping daemon to map the resources it has available to it (username strings typically) to something the users of the filesystem can actually use.

Eric

_____