SafeSetID

SafeSetID is an LSM module that gates the setid family of syscalls to restrict UID/GID transitions from a given UID/GID to only those approved by a system-wide allowlist. These restrictions also prohibit the given UIDs/GIDs from obtaining auxiliary privileges associated with CAP_SET{U/G}ID, such as allowing a user to set up user namespace UID/GID mappings.

Background

In absence of file capabilities, processes spawned on a Linux system that need to switch to a different user must be spawned with CAP_SETUID privileges. CAP_SETUID is granted to programs running as root or those running as a non-root user that have been explicitly given the CAP_SETUID runtime capability. It is often preferable to use Linux runtime capabilities rather than file capabilities, since using file capabilities to run a program with elevated privileges opens up possible security holes since any user with access to the file can exec() that program to gain the elevated privileges.

While it is possible to implement a tree of processes by giving full CAP_SET{U/G}ID capabilities, this is often at odds with the goals of running a tree of processes under non-root user(s) in the first place. Specifically, since CAP_SETUID allows changing to any user on the system, including the root user, it is an overpowered capability for what is needed in this scenario, especially since programs often only call setuid() to drop privileges to a lesser-privileged user -- not elevate privileges. Unfortunately, there is no generally feasible way in Linux to restrict the potential UIDs that a user can switch to through setuid() beyond allowing a switch to any user on the system. This SafeSetID LSM seeks to provide a solution for restricting setid capabilities in such a way.

The main use case for this LSM is to allow a non-root program to transition to other untrusted uids without full blown CAP_SETUID capabilities. The non-root program would still need CAP_SETUID to do any kind of transition, but the additional restrictions imposed by this LSM would mean it is a "safer" version of CAP_SETUID since the non-root program cannot take advantage of CAP_SETUID to do any unapproved actions (e.g. setuid to uid 0 or create/enter new user namespace). The higher level goal is to allow for uid-based sandboxing of system services without having to give out CAP_SETUID all over the place just so that non-root programs can drop to even-lesser-privileged uids. This is especially relevant when one non-root daemon on the system should be allowed to spawn other processes as different uids, but its undesirable to give the daemon a basically-root-equivalent CAP_SETUID.

Other Approaches Considered

Solve this problem in userspace

For candidate applications that would like to have restricted setid capabilities as implemented in this LSM, an alternative option would be to simply take away setid capabilities from the application completely and refactor the process spawning semantics in the application (e.g. by using a privileged helper program to do process spawning and UID/GID transitions). Unfortunately, there are a number of semantics around process spawning that would be affected by this, such as fork() calls where the program doesn't immediately call exec() after the fork(), parent processes specifying custom environment variables or command line args for spawned child processes, or inheritance of file handles across a fork()/exec(). Because of this, as solution that uses a privileged helper in userspace would likely be less appealing to incorporate into existing projects that rely on certain process-spawning semantics in Linux.

Use user namespaces

Another possible approach would be to run a given process tree in its own user namespace and give programs in the tree setid capabilities. In this way, programs in the tree could change to any desired UID/GID in the context of their own user namespace, and only approved UIDs/GIDs could be mapped back to the initial system user namespace, affectively preventing privilege escalation. Unfortunately, it is not generally feasible to use user namespaces in isolation, without pairing them with other namespace types, which is not always an option. Linux checks for capabilities based off of the user namespace that "owns" some entity. For example, Linux has the notion that network namespaces are owned by the user namespace in which they were created. A consequence of this is that capability checks for access to a given network namespace are done by checking whether a task has the given capability in the context of the user namespace that owns the network namespace -- not necessarily the user namespace under which the given task runs. Therefore spawning a process in a new user namespace effectively prevents it from accessing the network namespace owned by the initial namespace. This is a deal-breaker for any application that expects to retain the CAP_NET_ADMIN capability for the purpose of adjusting network configurations. Using user namespaces in isolation causes problems regarding other system interactions, including use of pid namespaces and device creation.

Use an existing LSM

None of the other in-tree LSMs have the capability to gate setid transitions, or even employ the security_task_fix_setuid hook at all. SELinux says of that hook: "Since setuid only affects the current process, and since the SELinux controls are not based on the Linux identity attributes, SELinux does not need to control this operation."

Directions for use

This LSM hooks the setid syscalls to make sure transitions are allowed if an applicable restriction policy is in place. Policies are configured through securityfs by writing to the safesetid/uid_allowlist_policy and safesetid/gid_allowlist_policy files at the location where securityfs is mounted. The format for adding a policy is '<UID>:<UID>' or '<GID>:<GID>', using literal numbers, and ending with a newline character such as '123:456n'. Writing an empty string "" will flush the policy. Again, configuring a policy for a UID/GID will prevent that UID/GID from obtaining auxiliary setid privileges, such as allowing a user to set up user namespace UID/GID mappings.

Note on GID policies and setgroups()

In v5.9 we are adding support for limiting CAP_SETGID privileges as was done previously for CAP_SETUID. However, for compatibility with common sandboxing related code conventions in userspace, we currently allow arbitrary setgroups() calls for processes with CAP_SETGID restrictions. Until we add support in a future release for restricting setgroups() calls, these GID policies add no meaningful security. setgroups() restrictions will be enforced once we have the policy checking code in place, which will rely on GID policy configuration code added in v5.9.