The Linux kernel user-space API guide

Release

The kernel development community

Nov 02, 2017

While much of the kernel's user-space API is documented elsewhere (particularly in the man-pages project), some user-space information can also be found in the kernel tree itself. This manual is intended to be the place where this information is gathered.

Table of contents

CHAPTER

ONE

NO NEW PRIVILEGES FLAG

The execve system call can grant a newly-started program privileges that its parent did not have. The most obvious examples are setuid/setgid programs and file capabilities. To prevent the parent program from gaining these privileges as well, the kernel and user code must be careful to prevent the parent from doing anything that could subvert the child. For example:

- The dynamic loader handles LD_* environment variables differently if a program is setuid.
- chroot is disallowed to unprivileged processes, since it would allow `/etc/passwd` to be replaced from the point of view of a process that inherited chroot.
- The exec code has special handling for ptrace.

These are all ad-hoc fixes. The `no_new_privs` bit (since Linux 3.5) is a new, generic mechanism to make it safe for a process to modify its execution environment in a manner that persists across execve. Any task can set `no_new_privs`. Once the bit is set, it is inherited across fork, clone, and execve and cannot be unset. With `no_new_privs` set, `execve()` promises not to grant the privilege to do anything that could not have been done without the execve call. For example, the setuid and setgid bits will no longer change the uid or gid; file capabilities will not add to the permitted set, and LSMs will not relax constraints after execve.

To set `no_new_privs`, use:

```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
```

Be careful, though: LSMs might also not tighten constraints on exec in `no_new_privs` mode. (This means that setting up a general-purpose service launcher to set `no_new_privs` before execing daemons may interfere with LSM-based sandboxing.)

Note that `no_new_privs` does not prevent privilege changes that do not involve `execve()`. An appropriately privileged task can still call `setuid(2)` and receive SCM_RIGHTS datagrams.

There are two main use cases for `no_new_privs` so far:

- Filters installed for the seccomp mode 2 sandbox persist across execve and can change the behavior of newly-executed programs. Unprivileged users are therefore only allowed to install such filters if `no_new_privs` is set.
- By itself, `no_new_privs` can be used to reduce the attack surface available to an unprivileged user. If everything running with a given uid has `no_new_privs` set, then that uid will be unable to escalate its privileges by directly attacking setuid, setgid, and fcap-using binaries; it will need to compromise something without the `no_new_privs` bit set first.

In the future, other potentially dangerous kernel features could become available to unprivileged tasks if `no_new_privs` is set. In principle, several options to `unshare(2)` and `clone(2)` would be safe when `no_new_privs` is set, and `no_new_privs` + `chroot` is considerable less dangerous than chroot by itself.

CHAPTER

TWO

SECCOMP BPF (SECURE COMPUTING WITH FILTERS)

Introduction

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp filtering provides a means for a process to specify a filter for incoming system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters, except that the data operated on is related to the system call being made: system call number and the system call arguments. This allows for expressive filtering of system calls using a filter program language with a long history of being exposed to userland and a straightforward data set.

Additionally, BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. BPF programs may not dereference pointers which constrains all filters to solely evaluating the system call arguments directly.

What it isn't

System call filtering isn't a sandbox. It provides a clearly defined mechanism for minimizing the exposed kernel surface. It is meant to be a tool for sandbox developers to use. Beyond that, policy for logical behavior and information flow should be managed with a combination of other system hardening techniques and, potentially, an LSM of your choosing. Expressive, dynamic filters provide further options down this path (avoiding pathological sizes or selecting which of the multiplexed system calls in socketcall() is allowed, for instance) which could be construed, incorrectly, as a more complete sandboxing solution.

Usage

An additional seccomp mode is added and is enabled using the same prctl(2) call as the strict seccomp. If the architecture has `CONFIG_HAVE_ARCH_SECCOMP_FILTER`, then filters may be added as below:

PR_SET_SECCOMP: Now takes an additional argument which specifies a new filter using a BPF program. The BPF program will be executed over struct seccomp_data reflecting the system call number, arguments, and other metadata. The BPF program must then return one of the acceptable values to inform the kernel which action should be taken.

Usage:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, prog);
```

The 'prog' argument is a pointer to a struct sock_fprog which will contain the filter program. If the program is invalid, the call will return -1 and set errno to `EINVAL`.

If fork/clone and execve are allowed by @prog, any child processes will be constrained to the same filters and system call ABI as the parent.

Prior to use, the task must call prctl(PR_SET_NO_NEW_PRIVS,1) or run with CAP_SYS_ADMIN privileges in its namespace. If these are not true, -EACCES will be returned. This requirement ensures that filter programs cannot be applied to child processes with greater privileges than the task that installed them.

Additionally, if prctl(2) is allowed by the attached filter, additional filters may be layered on which will increase evaluation time, but allow for further decreasing the attack surface during execution of a process.

The above call returns 0 on success and non-zero on error.

Return values

A seccomp filter may return any of the following values. If multiple filters exist, the return value for the evaluation of a given system call will always use the highest precedent value. (For example, SECCOMP_RET_KILL_PROCESS will always take precedence.)

In precedence order, they are:

SECCOMP_RET_KILL_PROCESS: Results in the entire process exiting immediately without executing the system call. The exit status of the task (status & 0x7f) will be SIGSYS, not SIGKILL.

SECCOMP_RET_KILL_THREAD: Results in the task exiting immediately without executing the system call. The exit status of the task (status & 0x7f) will be SIGSYS, not SIGKILL.

SECCOMP_RET_TRAP: Results in the kernel sending a SIGSYS signal to the triggering task without executing the system call. siginfo->si_call_addr will show the address of the system call instruction, and siginfo->si_syscall and siginfo->si_arch will indicate which syscall was attempted. The program counter will be as though the syscall happened (i.e. it will not point to the syscall instruction). The return value register will contain an arch- dependent value – if resuming execution, set it to something sensible. (The architecture dependency is because replacing it with -ENOSYS could overwrite some useful information.)

The SECCOMP_RET_DATA portion of the return value will be passed as si_errno.

SIGSYS triggered by seccomp will have a si_code of SYS_SECCOMP.

SECCOMP_RET_ERRNO: Results in the lower 16-bits of the return value being passed to userland as the errno without executing the system call.

SECCOMP_RET_TRACE: When returned, this value will cause the kernel to attempt to notify a ptrace()-based tracer prior to executing the system call. If there is no tracer present, -ENOSYS is returned to userland and the system call is not executed.

A tracer will be notified if it requests PTRACE_O_TRACESECCOM``P using ``ptrace(PTRACE_SETOPTIONS). The tracer will be notified of a PTRACE_EVENT_SECCOMP and the SECCOMP_RET_DATA portion of the BPF program return value will be available to the tracer via PTRACE_GETEVENTMSG.

The tracer can skip the system call by changing the syscall number to -1. Alternatively, the tracer can change the system call requested by changing the system call to a valid syscall number. If the tracer asks to skip the system call, then the system call will appear to return the value that the tracer puts in the return value register.

The seccomp check will not be run again after the tracer is notified. (This means that seccomp-based sandboxes MUST NOT allow use of ptrace, even of other sandboxed processes, without extreme care; ptracers can use this mechanism to escape.)

SECCOMP_RET_LOG: Results in the system call being executed after it is logged. This should be used by application developers to learn which syscalls their application needs without having to iterate through multiple test and development cycles to build the list.

This action will only be logged if “log” is present in the actions_logged sysctl string.

SECCOMP_RET_ALLOW: Results in the system call being executed.

If multiple filters exist, the return value for the evaluation of a given system call will always use the highest precedent value.

Precedence is only determined using the SECCOMP_RET_ACTION mask. When multiple filters return values of the same precedence, only the SECCOMP_RET_DATA from the most recently installed filter will be returned.

Pitfalls

The biggest pitfall to avoid during use is filtering on system call number without checking the architecture value. Why? On any architecture that supports multiple system call invocation conventions, the system call numbers may vary based on the specific invocation. If the numbers in the different calling conventions overlap, then checks in the filters may be abused. Always check the arch value!

Example

The samples/seccomp/ directory contains both an x86-specific example and a more generic example of a higher level macro interface for BPF program generation.

Sysctls

Seccomp’s sysctl files can be found in the /proc/sys/kernel/seccomp/ directory. Here’s a description of each file in that directory:

actions_avail: A read-only ordered list of seccomp return values (refer to the SECCOMP_RET_* macros above) in string form. The ordering, from left-to-right, is the least permissive return value to the most permissive return value.

The list represents the set of seccomp return values supported by the kernel. A userspace program may use this list to determine if the actions found in the seccomp.h, when the program was built, differs from the set of actions actually supported in the current running kernel.

actions_logged: A read-write ordered list of seccomp return values (refer to the SECCOMP_RET_* macros above) that are allowed to be logged. Writes to the file do not need to be in ordered form but reads from the file will be ordered in the same way as the actions_avail sysctl.

It is important to note that the value of actions_logged does not prevent certain actions from being logged when the audit subsystem is configured to audit a task. If the action is not found in actions_logged list, the final decision on whether to audit the action for that task is ultimately left up to the audit subsystem to decide for all seccomp return values other than SECCOMP_RET_ALLOW.

The allow string is not accepted in the actions_logged sysctl as it is not possible to log SECCOMP_RET_ALLOW actions. Attempting to write allow to the sysctl will result in an EINVAL being returned.

Adding architecture support

See arch/Kconfig for the authoritative requirements. In general, if an architecture supports both ptrace_event and seccomp, it will be able to support seccomp filter with minor fixup: SIGSYS support and seccomp return value checking. Then it must just add CONFIG_HAVE_ARCH_SECCOMP_FILTER to its arch-specific Kconfig.

Caveats

The vDSO can cause some system calls to run entirely in userspace, leading to surprises when you run programs on different machines that fall back to real syscalls. To minimize these surprises on x86, make sure you test with `/sys/devices/system/clocksource/clocksource0/current_clocksource` set to something like `acpi_pm`.

On x86-64, vsyscall emulation is enabled by default. (vsyscalls are legacy variants on vDSO calls.) Currently, emulated vsyscalls will honor seccomp, with a few oddities:

- A return value of `SECCOMP_RET_TRAP` will set a `si_call_addr` pointing to the vsyscall entry for the given call and not the address after the 'syscall' instruction. Any code which wants to restart the call should be aware that (a) a ret instruction has been emulated and (b) trying to resume the syscall will again trigger the standard vsyscall emulation security checks, making resuming the syscall mostly pointless.
- A return value of `SECCOMP_RET_TRACE` will signal the tracer as usual, but the syscall may not be changed to another system call using the orig_rax register. It may only be changed to -1 order to skip the currently emulated call. Any other change MAY terminate the process. The rip value seen by the tracer will be the syscall entry address; this is different from normal behavior. The tracer MUST NOT modify rip or rsp. (Do not rely on other changes terminating the process. They might work. For example, on some kernels, choosing a syscall that only exists in future kernels will be correctly emulated (by returning `-ENOSYS`).

To detect this quirky behavior, check for `addr & ~0x0C00 == 0xFFFFFFFFFF600000`. (For `SECCOMP_RET_TRACE`, use rip. For `SECCOMP_RET_TRAP`, use `siginfo->si_call_addr`.) Do not check any other condition: future kernels may improve vsyscall emulation and current kernels in vsyscall=native mode will behave differently, but the instructions at `0xF...F600{0,4,8,C}00` will not be system calls in these cases.

Note that modern systems are unlikely to use vsyscalls at all – they are a legacy feature and they are considerably slower than standard syscalls. New code will use the vDSO, and vDSO-issued system calls are indistinguishable from normal system calls.

CHAPTER

THREE

UNSHARE SYSTEM CALL

This document describes the new system call, unshare(). The document provides an overview of the feature, why it is needed, how it can be used, its interface specification, design, implementation and how it can be tested.

Change Log

version 0.1 Initial document, Janak Desai (janak@us.ibm.com), Jan 11, 2006

Contents

1) Overview

Most legacy operating system kernels support an abstraction of threads as multiple execution contexts within a process. These kernels provide special resources and mechanisms to maintain these “threads”. The Linux kernel, in a clever and simple manner, does not make distinction between processes and “threads”. The kernel allows processes to share resources and thus they can achieve legacy “threads” behavior without requiring additional data structures and mechanisms in the kernel. The power of implementing threads in this manner comes not only from its simplicity but also from allowing application programmers to work outside the confinement of all-or-nothing shared resources of legacy threads. On Linux, at the time of thread creation using the clone system call, applications can selectively choose which resources to share between threads.

unshare() system call adds a primitive to the Linux thread model that allows threads to selectively ‘unshare’ any resources that were being shared at the time of their creation. unshare() was conceptualized by Al Viro in the August of 2000, on the Linux-Kernel mailing list, as part of the discussion on POSIX threads on Linux. unshare() augments the usefulness of Linux threads for applications that would like to control

shared resources without creating a new process. unshare() is a natural addition to the set of available primitives on Linux that implement the concept of process/thread as a virtual machine.

2) Benefits

unshare() would be useful to large application frameworks such as PAM where creating a new process to control sharing/unsharing of process resources is not possible. Since namespaces are shared by default when creating a new process using fork or clone, unshare() can benefit even non-threaded applications if they have a need to disassociate from default shared namespace. The following lists two use-cases where unshare() can be used.

2.1 Per-security context namespaces

unshare() can be used to implement polyinstantiated directories using the kernel's per-process namespace mechanism. Polyinstantiated directories, such as per-user and/or per-security context instance of /tmp, /var/tmp or per-security context instance of a user's home directory, isolate user processes when working with these directories. Using unshare(), a PAM module can easily setup a private namespace for a user at login. Polyinstantiated directories are required for Common Criteria certification with Labeled System Protection Profile, however, with the availability of shared-tree feature in the Linux kernel, even regular Linux systems can benefit from setting up private namespaces at login and polyinstantiating /tmp, /var/tmp and other directories deemed appropriate by system administrators.

2.2 unsharing of virtual memory and/or open files

Consider a client/server application where the server is processing client requests by creating processes that share resources such as virtual memory and open files. Without unshare(), the server has to decide what needs to be shared at the time of creating the process which services the request. unshare() allows the server an ability to disassociate parts of the context during the servicing of the request. For large and complex middleware application frameworks, this ability to unshare() after the process was created can be very useful.

3) Cost

In order to not duplicate code and to handle the fact that unshare() works on an active task (as opposed to clone/fork working on a newly allocated inactive task) unshare() had to make minor reorganizational changes to copy_* functions utilized by clone/fork system call. There is a cost associated with altering existing, well tested and stable code to implement a new feature that may not get exercised extensively in the beginning. However, with proper design and code review of the changes and creation of an unshare() test for the LTP the benefits of this new feature can exceed its cost.

4) Requirements

unshare() reverses sharing that was done using clone(2) system call, so unshare() should have a similar interface as clone(2). That is, since flags in clone(int flags, void *stack) specifies what should be shared, similar flags in unshare(int flags) should specify what should be unshared. Unfortunately, this may appear to invert the meaning of the flags from the way they are used in clone(2). However, there was no easy solution that was less confusing and that allowed incremental context unsharing in future without an ABI change.

unshare() interface should accommodate possible future addition of new context flags without requiring a rebuild of old applications. If and when new context flags are added, unshare() design should allow incremental unsharing of those resources on an as needed basis.

5) Functional Specification

NAME unshare - disassociate parts of the process execution context

SYNOPSIS #include <sched.h>

int unshare(int flags);

DESCRIPTION unshare() allows a process to disassociate parts of its execution context that are currently being shared with other processes. Part of execution context, such as the namespace, is shared by default when a new process is created using fork(2), while other parts, such as the virtual memory, open file descriptors, etc, may be shared by explicit request to share them when creating a process using clone(2).

The main use of unshare() is to allow a process to control its shared execution context without creating a new process.

The flags argument specifies one or bitwise-or'ed of several of the following constants.

CLONE_FS If CLONE_FS is set, file system information of the caller is disassociated from the shared file system information.

CLONE_FILES If CLONE_FILES is set, the file descriptor table of the caller is disassociated from the shared file descriptor table.

CLONE_NEWNS If CLONE_NEWNS is set, the namespace of the caller is disassociated from the shared namespace.

CLONE_VM If CLONE_VM is set, the virtual memory of the caller is disassociated from the shared virtual memory.

RETURN VALUE On success, zero returned. On failure, -1 is returned and errno is

ERRORS

EPERM CLONE_NEWNS was specified by a non-root process (process without CAP_SYS_ADMIN).

ENOMEM Cannot allocate sufficient memory to copy parts of caller's context that need to be unshared.

EINVAL Invalid flag was specified as an argument.

CONFORMING TO The unshare() call is Linux-specific and should not be used in programs intended to be portable.

SEE ALSO clone(2), fork(2)

6) High Level Design

Depending on the flags argument, the unshare() system call allocates appropriate process context structures, populates it with values from the current shared version, associates newly duplicated structures with the current task structure and releases corresponding shared versions. Helper functions of clone (copy_*) could not be used directly by unshare() because of the following two reasons.

1. clone operates on a newly allocated not-yet-active task structure, where as unshare() operates on the current active task. Therefore unshare() has to take appropriate task_lock() before associating newly duplicated context structures
2. unshare() has to allocate and duplicate all context structures that are being unshared, before associating them with the current task and releasing older shared structures. Failure do so will create race conditions and/or oops when trying to backout due to an error. Consider the case of unsharing both virtual memory and namespace. After successfully unsharing vm, if the system call encounters an error while allocating new namespace structure, the error return code will have to reverse the

unsharing of vm. As part of the reversal the system call will have to go back to older, shared, vm structure, which may not exist anymore.

Therefore code from copy_* functions that allocated and duplicated current context structure was moved into new dup_* functions. Now, copy_* functions call dup_* functions to allocate and duplicate appropriate context structures and then associate them with the task structure that is being constructed. unshare() system call on the other hand performs the following:

1. Check flags to force missing, but implied, flags
2. For each context structure, call the corresponding unshare() helper function to allocate and duplicate a new context structure, if the appropriate bit is set in the flags argument.
3. If there is no error in allocation and duplication and there are new context structures then lock the current task structure, associate new context structures with the current task structure, and release the lock on the current task structure.
4. Appropriately release older, shared, context structures.

7) Low Level Design

Implementation of unshare() can be grouped in the following 4 different items:

1. Reorganization of existing copy_* functions
2. unshare() system call service function
3. unshare() helper functions for each different process context
4. Registration of system call number for different architectures

7.1) Reorganization of copy_* functions

Each copy function such as copy_mm, copy_namespace, copy_files, etc, had roughly two components. The first component allocated and duplicated the appropriate structure and the second component linked it to the task structure passed in as an argument to the copy function. The first component was split into its own function. These dup_* functions allocated and duplicated the appropriate context structure. The reorganized copy_* functions invoked their corresponding dup_* functions and then linked the newly duplicated structures to the task structure with which the copy function was called.

7.2) unshare() system call service function

- Check flags Force implied flags. If CLONE_THREAD is set force CLONE_VM. If CLONE_VM is set, force CLONE_SIGHAND. If CLONE_SIGHAND is set and signals are also being shared, force CLONE_THREAD. If CLONE_NEWNS is set, force CLONE_FS.
- For each context flag, invoke the corresponding unshare_* helper routine with flags passed into the system call and a reference to pointer pointing the new unshared structure
- If any new structures are created by unshare_* helper functions, take the task_lock() on the current task, modify appropriate context pointers, and release the task lock.
- For all newly unshared structures, release the corresponding older, shared, structures.

7.3) unshare_* helper functions

For unshare_* helpers corresponding to CLONE_SYSVSEM, CLONE_SIGHAND, and CLONE_THREAD, return -EINVAL since they are not implemented yet. For others, check the flag value to see if the unsharing is required for that structure. If it is, invoke the corresponding dup_* function to allocate and duplicate the structure and return a pointer to it.

7.4) Finally

Appropriately modify architecture specific code to register the new system call.

8) Test Specification

The test for unshare() should test the following:

1. Valid flags: Test to check that clone flags for signal and signal handlers, for which unsharing is not implemented yet, return -EINVAL.
2. Missing/implied flags: Test to make sure that if unsharing namespace without specifying unsharing of filesystem, correctly unshares both namespace and filesystem information.
3. For each of the four (namespace, filesystem, files and vm) supported unsharing, verify that the system call correctly unshares the appropriate structure. Verify that unsharing them individually as well as in combination with each other works as expected.
4. Concurrent execution: Use shared memory segments and futex on an address in the shm segment to synchronize execution of about 10 threads. Have a couple of threads execute execve, a couple _exit and the rest unshare with different combination of flags. Verify that unsharing is performed as expected and that there are no oops or hangs.

9) Future Work

The current implementation of unshare() does not allow unsharing of signals and signal handlers. Signals are complex to begin with and to unshare signals and/or signal handlers of a currently running process is even more complex. If in the future there is a specific need to allow unsharing of signals and/or signal handlers, it can be incrementally added to unshare() without affecting legacy applications using unshare().