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_USER_NOTIF:

Results in a struct seccomp_notif message sent on the userspace notification fd, if it is attached, or -ENOSYS if it is not. See below on discussion of how to handle user notifications.

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_TRACESECCOMP 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.

Userspace Notification

The SECCOMP_RET_USER_NOTIF return code lets seccomp filters pass a particular syscall to userspace to be handled. This may be useful for applications like container managers, which wish to intercept particular syscalls (mount(), finit_module(), etc.) and change their behavior.

To acquire a notification FD, use the SECCOMP_FILTER_FLAG_NEW_LISTENER argument to the seccomp() syscall:

```
fd = seccomp(SECCOMP_SET_MODE_FILTER, SECCOMP_FILTER_FLAG_NEW_LISTENER, &prog);
```

which (on success) will return a listener fd for the filter, which can then be passed around via `SCM_RIGHTS` or similar. Note that filter fds correspond to a particular filter, and not a particular task. So if this task then forks, notifications from both tasks will appear on the same filter fd. Reads and writes to/from a filter fd are also synchronized, so a filter fd can safely have many readers.

The interface for a seccomp notification fd consists of two structures:

```
struct seccomp_notif_sizes {
    __u16 seccomp_notif;
    __u16 seccomp_notif_resp;
    __u16 seccomp_data;
};

struct seccomp_notif {
    __u64 id;
    __u32 pid;
    __u32 flags;
    struct seccomp_data data;
};

struct seccomp_notif_resp {
    __u64 id;
    __s64 val;
    __s32 error;
    __u32 flags;
};
```

The `struct seccomp_notif_sizes` structure can be used to determine the size of the various structures used in seccomp notifications. The size of `struct seccomp_data` may change in the future, so code should use:

```
struct seccomp_notif_sizes sizes;
seccomp(SECCOMP_GET_NOTIF_SIZES, 0, &sizes);
```

to determine the size of the various structures to allocate. See samples/seccomp/user-trap.c for an example.

Users can read via `ioctl(SECCOMP_IOCTL_NOTIF_RECV)` (or `poll()`) on a seccomp notification fd to receive a `struct seccomp_notif`, which contains five members: the input length of the structure, a unique-per-filter `id`, the `pid` of the task which triggered this request (which may be 0 if the task is in a pid ns not visible from the listener's pid namespace). The notification also contains the `data` passed to seccomp, and a filters flag. The structure should be zeroed out prior to calling the ioctl.

Userspace can then make a decision based on this information about what to do, and `ioctl(SECCOMP_IOCTL_NOTIF_SEND)` a response, indicating what should be returned to userspace. The `id` member of `struct seccomp_notif_resp` should be the same `id` as in `struct seccomp_notif`.

Userspace can also add file descriptors to the notifying process via `ioctl(SECCOMP_IOCTL_NOTIF_ADDFD)`. The `id` member of `struct seccomp_notif_addfd` should be the same `id` as in `struct seccomp_notif`. The `newfd_flags` flag may be used to set flags like O_CLOEXEC on the file descriptor in the notifying process. If the supervisor wants to inject the file descriptor with a specific number, the `SECCOMP_ADDFD_FLAG_SETFD` flag can be used, and set the `newfd` member to the specific number to use. If that file descriptor is already open in the notifying process it will be replaced. The supervisor can also add an FD, and respond atomically by using the `SECCOMP_ADDFD_FLAG_SEND` flag and the return value will be the injected file descriptor number.

It is worth noting that `struct seccomp_data` contains the values of register arguments to the syscall, but does not contain pointers to memory. The task's memory is accessible to suitably privileged traces via `ptrace()` or `/proc/pid/mem`. However, care should be taken to avoid the TOCTOU mentioned above in this document: all arguments being read from the tracee's memory should be read into the tracer's memory before any policy decisions are made. This allows for an atomic decision on syscall arguments.

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.
>
> 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.