Core Scheduling

Core scheduling support allows userspace to define groups of tasks that can share a core. These groups can be specified either for security usecases (one group of tasks don't trust another), or for performance usecases (some workloads may benefit from running on the same core as they don't need the same hardware resources of the shared core, or may prefer different cores if they do share hardware resource needs). This document only describes the security usecase.

Security usecase

A cross-HT attack involves the attacker and victim running on different Hyper Threads of the same core. MDS and L1TF are examples of such attacks. The only full mitigation of cross-HT attacks is to disable Hyper Threading (HT). Core scheduling is a scheduler feature that can mitigate some (not all) cross-HT attacks. It allows HT to be turned on safely by ensuring that only tasks in a user-designated trusted group can share a core. This increase in core sharing can also improve performance, however it is not guaranteed that performance will always improve, though that is seen to be the case with a number of real world workloads. In theory, core scheduling aims to perform at least as good as when Hyper Threading is disabled. In practice, this is mostly the case though not always: as synchronizing scheduling decisions across 2 or more CPUs in a core involves additional overhead - especially when the system is lightly loaded. When `total_threads <= N_CPUS/2`, the extra overhead may cause core scheduling to perform more poorly compared to SMT-disabled, where N_CPUS is the total number of CPUs. Please measure the performance of your workloads always.

Usage

Core scheduling support is enabled via the `CONFIG_SCHED_CORE` config option. Using this feature, userspace defines groups of tasks that can be co-scheduled on the same core. The core scheduler uses this information to make sure that tasks that are not in the same group never run simultaneously on a core, while doing its best to satisfy the system's scheduling requirements.

Core scheduling can be enabled via the `PR_SCHED_CORE` prctl interface. This interface provides support for the creation of core scheduling groups, as well as admission and removal of tasks from created groups:

```
#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3,
        unsigned long arg4, unsigned long arg5);
```

option:
: `PR_SCHED_CORE`

arg2:
: Command for operation, must be one off:

  - `PR_SCHED_CORE_GET` -- get core_sched cookie of `pid`.
  - `PR_SCHED_CORE_CREATE` -- create a new unique cookie for `pid`.
  - `PR_SCHED_CORE_SHARE_TO` -- push core_sched cookie to `pid`.
  - `PR_SCHED_CORE_SHARE_FROM` -- pull core_sched cookie from `pid`.

arg3:
: `pid` of the task for which the operation applies.

arg4:
: `pid_type` for which the operation applies. It is one of `PR_SCHED_CORE_SCOPE_`-prefixed macro constants. For example, if arg4 is `PR_SCHED_CORE_SCOPE_THREAD_GROUP`, then the operation of this command will be performed for all tasks in the task group of `pid`.

arg5:
: userspace pointer to an unsigned long for storing the cookie returned by `PR_SCHED_CORE_GET` command. Should be 0 for all other commands.

In order for a process to push a cookie to, or pull a cookie from a process, it is required to have the ptrace access mode: *PTRACE_MODE_READ_REALCREDS* to the process.

Building hierarchies of tasks

The simplest way to build hierarchies of threads/processes which share a cookie and thus a core is to rely on the fact that the core-sched cookie is inherited across forks/clones and execs, thus setting a cookie for the 'initial' script/executable/daemon will place every spawned child in the same core-sched group.

Cookie Transferral

Transferring a cookie between the current and other tasks is possible using PR_SCHED_CORE_SHARE_FROM and PR_SCHED_CORE_SHARE_TO to inherit a cookie from a specified task or a share a cookie with a task. In combination this allows a simple helper program to pull a cookie from a task in an existing core scheduling group and share it with already running tasks.

Design/Implementation

Each task that is tagged is assigned a cookie internally in the kernel. As mentioned in Usage, tasks with the same cookie value are assumed to trust each other and share a core.

The basic idea is that, every schedule event tries to select tasks for all the siblings of a core such that all the selected tasks running on a core are trusted (same cookie) at any point in time. Kernel threads are assumed trusted. The idle task is considered special, as it trusts everything and everything trusts it.

During a schedule() event on any sibling of a core, the highest priority task on the sibling's core is picked and assigned to the sibling calling schedule(), if the sibling has the task enqueued. For rest of the siblings in the core, highest priority task with the same cookie is selected if there is one runnable in their individual run queues. If a task with same cookie is not available, the idle task is selected. Idle task is globally trusted.

Once a task has been selected for all the siblings in the core, an IPI is sent to siblings for whom a new task was selected. Siblings on receiving the IPI will switch to the new task immediately. If an idle task is selected for a sibling, then the sibling is considered to be in a *forced idle* state. I.e., it may have tasks on its on runqueue to run, however it will still have to run idle. More on this in the next section.

Forced-idling of hyperthreads

The scheduler tries its best to find tasks that trust each other such that all tasks selected to be scheduled are of the highest priority in a core. However, it is possible that some runqueues had tasks that were incompatible with the highest priority ones in the core. Favoring security over fairness, one or more siblings could be forced to select a lower priority task if the highest priority task is not trusted with respect to the core wide highest priority task. If a sibling does not have a trusted task to run, it will be forced idle by the scheduler (idle thread is scheduled to run).

When the highest priority task is selected to run, a reschedule-IPI is sent to the sibling to force it into idle. This results in 4 cases which need to be considered depending on whether a VM or a regular usermode process was running on either HT:

```
          HT1 (attack)            HT2 (victim)
   A      idle -> user space      user space -> idle
   B      idle -> user space      guest -> idle
   C      idle -> guest           user space -> idle
   D      idle -> guest           guest -> idle
```

Note that for better performance, we do not wait for the destination CPU (victim) to enter idle mode. This is because the sending of the IPI would bring the destination CPU immediately into kernel mode from user space, or VMEXIT in the case of guests. At best, this would only leak some scheduler metadata which may not be worth protecting. It is also possible that the IPI is received too late on some architectures, but this has not been observed in the case of x86.

Trust model

Core scheduling maintains trust relationships amongst groups of tasks by assigning them a tag that is the same cookie value. When a system with core scheduling boots, all tasks are considered to trust each other. This is because the core scheduler does not have information about trust relationships until userspace uses the above mentioned interfaces, to communicate them. In other words, all tasks have a default cookie value of 0. and are considered system-wide trusted. The forced-idling of siblings running cookie-0 tasks is also avoided.

Once userspace uses the above mentioned interfaces to group sets of tasks, tasks within such groups are considered to trust each other, but do not trust those outside. Tasks outside the group also don't trust tasks within.

Limitations of core-scheduling

Core scheduling tries to guarantee that only trusted tasks run concurrently on a core. But there could be small window of time during which untrusted tasks run concurrently or kernel could be running concurrently with a task not trusted by kernel.

IPI processing delays

Core scheduling selects only trusted tasks to run together. IPI is used to notify the siblings to switch to the new task. But there could be hardware delays in receiving of the IPI on some arch (on x86, this has not been observed). This may cause an attacker task to start running on a CPU before its siblings receive the IPI. Even though cache is flushed on entry to user mode, victim tasks on siblings may populate data in the cache and micro architectural buffers after the attacker starts to run and this is a possibility for data leak.

Open cross-HT issues that core scheduling does not solve

1. For MDS

Core scheduling cannot protect against MDS attacks between the siblings running in user mode and the others running in kernel mode. Even though all siblings run tasks which trust each other, when the kernel is executing code on behalf of a task, it cannot trust the code running in the sibling. Such attacks are possible for any combination of sibling CPU modes (host or guest mode).

2. For L1TF

Core scheduling cannot protect against an L1TF guest attacker exploiting a guest or host victim. This is because the guest attacker can craft invalid PTEs which are not inverted due to a vulnerable guest kernel. The only solution is to disable EPT (Extended Page Tables).

For both MDS and L1TF, if the guest vCPU is configured to not trust each other (by tagging separately), then the guest to guest attacks would go away. Or it could be a system admin policy which considers guest to guest attacks as a guest problem.

Another approach to resolve these would be to make every untrusted task on the system to not trust every other untrusted task. While this could reduce parallelism of the untrusted tasks, it would still solve the above issues while allowing system processes (trusted tasks) to share a core.

3. Protecting the kernel (IRQ, syscall, VMEXIT)

Unfortunately, core scheduling does not protect kernel contexts running on sibling hyperthreads from one another. Prototypes of mitigations have been posted to LKML to solve this, but it is debatable whether such windows are practically exploitable, and whether the performance overhead of the prototypes are worth it (not to mention, the added code complexity).

Other Use cases

The main use case for Core scheduling is mitigating the cross-HT vulnerabilities with SMT enabled. There are other use cases where this feature could be used:

- Isolating tasks that needs a whole core: Examples include realtime tasks, tasks that uses SIMD instructions etc.
- Gang scheduling: Requirements for a group of tasks that needs to be scheduled together could also be realized using core scheduling. One example is vCPUs of a VM.