"Fun-reliable side-channels for cross-container communication"

"ivan"

Introduction

While exploring the Linux kernel we discovered a fun side-channel that allows for cross-container communication in the most common, default container deployment scenarios on modern kernels. This is cool because it doesn't require sharing volume mounts, nor does it involve modifying any of the default namespaces (`NET`, `PID`, `IPC`, etc.), or adding special privileges (no new `CAP_`-abilities, nor changes to `seccomp` or `AppArmor`). It works out of the box with default Docker and Kubernetes configurations, and it even works with no network at all, as we demonstrate in this post by using `docker run --network none sidechannel /h4x0rchat` to showcase a full cross-container IRC-style chatroom implemented on top of this side-channel.

We originally set out to find this side-channel because we wanted a way for a given container to know if another instance of its same image was already running on the host. Consider a scenario where you want to collect environmental telemetry from your containers when they first start running. Now consider that, to handle real workloads, container deployments are often scaled up with a given image running many times over simultaneously on the same host.

Humoring further consideration, if you scale the same container image thousands of times over, and the environmental telemetry is effectively the same for each instance on the same host, you'll probably want a way to throttle how many instances report back, to save compute time and bandwidth that would be otherwise wasted on duplicate reports. Finally, imagine that you work with many teams, all of which operate with varying requirements and constraints, and as such you can't always control (or, maybe even never control) how these containers are deployed. If only there was a way the container could identify the presence of itself already running on the same host?

Because this side-channel circumvents the intended isolation behavior of containers, it could technically be considered a vulnerability, even through we see it more as functionality we previously wished we had.

Components

The first component of this side-channel involves `nsfs` (the namespace filesystem), which is a special filesystem made available to userland through `/proc/<pid>/ns/`. The `nsfs` is similar to `procfs` in that its entries are not actual files, but instead special file-like objects which can be used for interfacing with the kernel. In partciular, `nsfs` entries are like magical symlinks that point to namespace inode identifiers, with each namespace type being represented by its own named entry in the `/proc/<pid>/ns/` directory. In practice, these magical symlinks can be used by opening a file descriptor to one and and passing it to `setns`, to enter a namespace, for example.

Unlike `procfs`, the `nsfs` entries are not unique across different mounts of the parent `procfs` containing the `ns/*` directory. This means that any namespace shared by multiple processes will result in them having the *same* file-like `nsfs` entry representing that namespace, reachable relative to each process at `/proc/self/ns/<namespacename>`.

The next component of this side-channel are `time` namespaces, which are for applying offsets to the system's monotonic and boot-time clocks. The issue is not with how `time` namespaces are used, but in the fact that they are generally not used.

The utility of the `time` namespace applies only to niche scenarios like cross-host container migration, which is probably why (as far as I can tell) Docker doesn't support setting the `time` namespace, and the documentation available instead instructs users to manually run `unshare`. In other words, not only are `time` namespaces shared, but there's no easy way for the average container user to unshare them.

The important result of all of this is that by default, container and host processes all share the same `/proc/self/ns/time` entry, which more-or-less behaves like a file resource (or enough like one that it enables our side-channel).

It's common that a single `user` namespace is shared by default across containers, and would

also lend itself for exercising this same side-channel. However, some security conscious users set up separate `user` namespaces to reduce kernel attack surface a tiny bit, so we don't expect it to be shared as ubiquitously as `time`.

Now, let's talk about **POSIX Advisory Locks**, the official Linux docs for which can be read by running `man fcntl` and scrolling down to the "Advisory record locking" section. In short, POSIX advisory locks provide a cooperative (vs mandatory) file locking mechanism that operates on byte offset ranges (intervals) within a given file. These locks are "process-associated", meaning that their acquisition and entire life-cycle is bound to a single process (and its threads). These locks are not inherited by child processes, and they clear once the owning process exits. By operating on intervals, these advisory locks allow for a more explicit expression of file content usage than whole-file locking mechanisms. For example, one process might hold a read-lock for byte range 10-200, and another might hold a write-lock for range 500-600 on the same file, and because those ranges don't overlap, neither lock would contend with the other. Since these are cooperative, holding a lock doesn't stop other processes from reading or writing the files, and instead only stops other processes from acquiring locks of a conflicting type that intersect with the same interval.

These advisory locks have some additional interesting properties, which, when combined with a shared file resource (or even pseudo-file resource, like `/proc/self/ns/time`) can facilitate a side-channel:

A user only needs to have a file-like resource open for reading in order to acquire a read-lock (and conversely must have the file open for writing, in order to acquire a write-lock).

The file doesn't need to actually have readable content (note that /proc/self/ns/time does not actually have anything to read, for example).

The lock intervals do not need to reflect the real size of the file, and are specified using `off_t`, which means there are effectively 63bits of space available in which a lock interval can be set (`off_t` is signed and locks cannot be placed below offset 0).

A file open for reading can be queried to determine if a write-lock would hypothetically contend with any other lock, even if the querying process does not actually possess the privileges to open the file for writing.

These properties combined are enough to provide a basic cross-container side-channel primitive, because a process in one container can set a read-lock at some interval on `/proc/self/ns/time`, and a process in another container can observe the presence of that lock by querying for a hypothetically intersecting write-lock.

There are still yet more properties about these locks that can be used for synchronization across this side-channel, but before getting into those, presented below are small programs demonstrating cross-container communication using the fundamentals discussed above.


```
// setlock.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf("usage: %s <offset> <len>\n", argv[0]);
    exit(1);
  }
  off_t  offset = atol(argv[1]);
  off_t  len    = atol(argv[2]);
  int    fd     = open("/proc/self/ns/time", O_RDONLY);
  if (fd < 0) {
    printf("failed to open /proc/self/ns/time\n");
    exit(1);
  }
  struct flock lock;
  memset(&lock, 0, sizeof(lock));
  lock.l_type   = F_RDLCK;
  lock.l_whence = SEEK_SET;
```

```
  lock.l_start  = offset;
  lock.l_len    = len;
  if (fcntl(fd, F_SETLK, &lock) < 0) {
    printf("fcntl() failed\n");
    exit(1);
  }
  printf("lock set at %ld:%ld, press enter to exit\n", offset,
offset+len);
  getchar();
}
```

The above `setlock.c` program takes two arguments, an offset and a length, which are used as the interval for an advisory read lock on `/proc/self/ns/time`. Below is a counterpart program which similarly takes two arguments, instead querying the interval for hypothetical contention, using an advisory write-lock:

```
// querylock.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf("usage: %s <offset> <len>\n", argv[0]);
    exit(1);
  }
  off_t  offset = atol(argv[1]);
  off_t  len    = atol(argv[2]);
  int    fd     = open("/proc/self/ns/time", O_RDONLY);
  if (fd < 0) {
    printf("failed to open /proc/self/ns/time\n");
    exit(1);
  }
  struct flock lock;
  memset(&lock, 0, sizeof(lock));
  lock.l_type   = F_WRLCK;
  lock.l_whence = SEEK_SET;
  lock.l_start  = offset;
  lock.l_len    = len;
  if (fcntl(fd, F_GETLK, &lock) < 0) {
```

```
    printf("fcntl() failed\n");
    exit(1);
  }
  if (F_UNLCK != lock.l_type) {
    printf("collision: %ld:%ld\n", lock.l_start,
lock.l_start+lock.l_len);
  } else {
    printf("no lock intersects with %ld:%ld\n", offset, offset+len);
  }
}
```

In `querylock.c` we set the `struct flock.l_type` to `F_WRLOCK`, but when calling `fcntl()` we specify the command argument as `F_GETLK` to get info about possible lock contention vs attempt to set a lock. If there is no contention, the `struct flock` member field `l_type` is updated by the kernel to contain `F_UNLCK`. Shown below, you can see that once a lock is set in one container, any other container (or any process on the host for that matter) can see it:

Synchronization

The ability to set and query for the presence of read-locks across containers is itself pretty cool, but to use this for proper communication, we would ideally have some way to synchronize how containers access the 63bit-space available in `/proc/self/ns/time`. Luckily POSIX advisory locks have some other nuances which we can use to achieve this:

When the presence of a contending lock is found, the kernel updates the `struct flock` member field `l_pid` to contain the `PID` of the process holding the lock, or `0` if that process is in another `PID` namespace.

If there are multiple processes with contending locks, the kernel selects and reports the `PID` of a "primary" lock holder. Ironically, the ordering for selecting this primary lock holder is not based on which of the contestants was first to acquire an intersecting lock, but instead by which of the contestants has held any advisory lock on the file the longest.

Given that the kernel imposes ordering when reporting the `PID` of lock owners, and that the ordering is preserved across `PID` namespaces (even if that means the owning `PID` is reported as `0`), for a process to know if it is the "primary" lock holder, all it needs to do is create a child process to query the lock, and see if the owning `PID` is the parent process. Also, given how "primary" lock holders are determined by the kernel, to participate with "fairness" in this race, the process competing to acquire this lock should not hold any prior locks. For example, let's say two containers both want to compete in a race for "ownership" of the offsets 500-501, they could each take the following steps to attempt to lock, and determine if they "won" the competition:

The locking process, here called P1, holding no prior locks, acquires a read-lock at offsets 500-501.

P1 `fork()`s to create process P2, which has no affiliation with P1's lock state.

P2 queries for the hypothetical write-lock contention at offset 500-501, the reply to which will always be true (given that P1 definitely has a lock, possibly others do too). P2 then compares the `struct flock.l_pid` field to see if it matches P1's `PID`, if so then P1 won the race for ownership, otherwise it did not. If, instead, it sees 0 for the `PID`, it means a process in another `PID` namespace was first to get the lock P1 is not the owner.

P2 tells P1 (either by pipe, or any form of IPC) the result, and now P1 is coordinated with all other container instances which are following this same protocol to race for ownership of byte offsets 500-501.

These additional properties provide us enough functionality to construct a sort of protocol for containers that normally are unaware of each other's existence to synchronize with each other, build more traditional read-write-lock mechanisms, and to perform tasks like leader selection. To demonstrate that this is not purely theoretical and can be used for practical communication, we've written a cross-container h4x0rchat program built on top of this side-channel:

To support an arbitrary number of users with proper message ordering, and provide a real-time chat without absolutely slamming the CPU in tight query loops, `h4x0rchat` expands on the described synchronization primitive to create a system of ever-forward-rolling message slots. Clients sync to claim slots in order to post messages, and check for new messages periodically between reading `stdin`. Messages are written bit-by-bit, with each byte offset in the message slot representing a 1 or 0 depending on if a lock is held at that offset. A "ready" bit is used to indicate when a claimed message slot has been fully written. As long as everyone is following the same protocol, this chat avoids racy data collisions…mostly. A full walk-through of the `h4x0rchat` protocol, and its shortcomings, would be too much to unpack in this post, but we're considering writing a follow-up if there is reader interest— as in, if both of our readers like it, ha!

NOTE: the demo `setlock.c` program shown earlier will interfere with this chat program, so if you see any error messages about there being "interference" when trying out `h4x0rchat`, make sure you're not also running `setlock`!

Fun-reliability & Defense Considerations

The `h4x0rchat`, or any other communication mechanism built atop this same side-channel, is open to disruption (and likely complete denial-of-service) because there are no security guarantees over how other processes apply locks. For example, any process can acquire a lock that spans the entire space of all lockable offsets, and if they're the first to hold such a lock they can ruin the party line for everyone. Maybe for a defender that's a good thing? While this side-channel doesn't present a dire threat to container security, there are definitely scenarios where it could support nefarious activity, and so we'll close with some security considerations:

You can use the demo setlock program to be the rude user who jams the whole party line by

running our demo program with `./setlock 0 9223372036854775807`, but if one or more users have held any lock before you, they might be able to devise a protocol for communicating still (just not with the same freedom and ease).

We thought that it would be possible to write a simple AppArmor profile using `deny /proc/*/ns/time rwklx`, to deny access to `/proc/*/ns/time`. But, from a first pass of experiments, it seems that this doesn't work. We will follow-up as we learn more—my gut is telling me that this some specific behavior related to `nsfs`, but who knows?

You could also put in the grueling work of manually invoking `unshare` on the `time` namespace (this feels tedious, who's got `time` for that??)

Special thanks to Robert Prast, Jay Beale, and Lee T. Hacker for their feedback.