Sandboxing in Operating Systems

Austin Kim

April 20, 2020

Abstract

Sandboxing is one of many critical components of operating systems protection. It allows for the partitioning of system resources and the practice of executing processes with the least amount of privilege required to accomplish tasks. This strict partitioning ensures that even if a process is malicious, its ability to do damage on the system at large is limited. Sandboxing is a part of the larger defense-in-depth philosophy of operating system protection concept.

1 Sandboxing basics

Sandboxing is the practice of placing immutable restrictions on a process's access to resources [1], such as system calls, privileges, storage space, and many others. This creates a set of rigid access rules that cannot be disobeyed and creates a secure environment upon which processes may be executed.

To accomplish sandboxing, the access matrix[1] is partitioned into "horizontal" slices (by domain) and then conferred to the appropriate processes [2], often as early as before the process's `main()` function. This ensures that the restrictions placed upon it cannot be changed at a later time.

It is also within the realm of possibility to sandbox processes and limit their access to system calls to a small set. For example, a process that only has read privileges but is not allowed to invoke the `write()` call cannot modify any of the disk contents allocated to it, effectively isolating it from communicating with other processes.

2 Why bother?

Sandboxing is an important component in an operating system's protection scheme, as it allows for *per-process* control over what they can and cannot access.

Each time a new process is forked from its parent, it inherits the same permissions that the parent held [3]. The dangers of a rogue process inheriting its parent's permissions is as blatant as it is debilitating. Sandboxing allows the operating system to take charge and section off process's access based on their needs such that even if a malicious piece of code is executed, the damage done to the system is either minimal or nonexistent.

3 Sandboxing examples

At the operating systems level, sandboxing has been a feature since many decades prior. In the beginning, there was the *namespace* feature in *Plan 9 from Bell Labs*. In Unix, there was `chroot` for isolating processes into a confined environment and still exists to this day. In modern day Linux, `cgroups` became the de-facto tool for limiting resource access for processes. Many others, from full-scale virtual machines for sandboxing at the OS-level, to containers like *Docker* and *systemd-nspawn*, sandboxing remains a staple of process compartmentalization and a general protection feature.

There are numerous examples of sandboxing – too many to list them all. In this section, we will delve into some prominent sandboxing examples in Linux: `chroot`, `cgroups`, and `seccomp`.

3.1 chroot

`chroot` is a lightweight sandboxing solution available to Linux, whose original Unix version was the predecessor for the much more capable `jail` in FreeBSD.

`chroot` uses Linux namespaces to accomplish sandboxing. Linux namespaces, derived from *Plan 9 From Bell Labs*'s namespaces [3], are a kernel feature where every global resource within the file system follows a unified namespace scheme, but those names point to completely different locations per process [4]. As such, `chroot`'s sandboxing is largely limited to file access and is unable to compartmentalize the process away from things like network sockets.

To see how `chroot` acts as a namespace-based sandbox, it's worthwhile to take a look at the generalized steps for setting up an environment for one [5]:

[1] An abstract model composed of domains and system objects and what kind of access each domain has on each object.

1. Create a top level directory to act as the root.
2. Copy over or symlink the essential directories necessary to run a bare minimum environment (path to `sh`, among others). This step is often automated by scripts.
3. Create symbolic links to any additional system resources that the `chroot` environment should have read/write access to, such as directories or files.
4. Run `chroot DIR CMD` to run a command under this new environment.

Steps 2 and 3 outline the gist of how `chroot` works as a sandbox. It's trivial to select which shell the environment should run with, which outside files it should be allowed to see, and which files it can write to. Any write operations done inside the environment is confined to the environment unless a link exists to an outside directory that the process can write to. It is, in a way, a very lightweight virtual machine with very little overhead.

This is not to say that `chroot` is perfect at its job. If anything, it's one of the weakest sandboxing tools available, albeit an easy one to use. There are many documented and proven ways in which any user can escape the `chroot` environment and gain direct access to all system resources [6].

3.2 cgroups

Control groups, abbreviated `cgroups`, are groups of processes that are bundled together into hierarchies and their resource usage monitored. Following the "everything is a file" philosophy of Unix and its derivatives, its interfaces are exposed as a set of files in a pseudo-filesystem called *cgroupfs* [7]. It is the underlying kernel feature upon which big-name applications like *Docker* are built.

`cgroups` consists of two main pieces:

- *cgroup* bundles – a collection of processes with limits imposed upon them with regards to resource usage.
- a *subsystem* (also known as *resource controllers* or *controllers*) – a kernel component that modifies the behavior of the processes in a cgroup.

The controller and the cgroup bundles are organized into a *hierarchy* that constantly changes the underlying cgroupfs subdirectories. It's under these subdirectories that the access control is defined: whether by adding processes to it by writing the pid to `cgroups.procs`, or defining the process's behavior by assigning it to a specific subdirectory.

There are two different versions of `cgroups`:

- Cgroups v1 – Released in 2008. Commonly mounted under `/sys/fs/cgroup`.
- Cgroups v2 – Attempts to unify the `cgroups` standard and make the interface more consistent. Commonly mounted under `/sys/fs/cgroup/unified`.

Both versions of `cgroups` may be active at the same time, as v2 supports only a portion of controllers found in v1. The only caveat is that the appropriate controllers be used for different versions of `cgroups`. For instance, using v1's controllers for v2's subdirectories would not be allowed.

3.2.1 Creating and deleting cgroups

Using `cgroups` is as easy as simply writing to a file. For the sake of example, let's assume that `cgroups` v1 is being used, and a process needs to be added under the `cpu` controller subdirectory:

1. Create a directory for the new cgroup: `/sys/fs/cgroup/cpu/mygroup`. This creates an empty cgroup named 'mygroup.'
2. Write the pid of the process to the `cgroups.procs` file: `/sys/fs/cgroup/cpu/mygroup/cgroups.procs`.
3. The process of writing pid to `cgroups.procs` automatically removes that process from all other cgroup subdirectories, and assigns all threads belonging to that process to the cgroup immediately.

Removing a cgroup is equally simple: Make sure that no active processes belong under the cgroup, and simply remove the directory [7].

3.2.2 Controller restrictions

So, how exactly does `cgroups` compartmentalize system resource usage?

There are several predefined controller groups that are allowed certain resources and priorities based on the directory structure. In the above example, one of them (`cpu`) was used to showcase how processes are assigned to individual controllers. Any processes belonging to a controller group must abide by the rules set for that controller group, and any descendant groups may not override them [7].

Listed below are some examples of v1 controllers:

1. `cpu` – This controller group is guaranteed to be assigned at least some CPU cycles even when the system is busy.
2. `devices` – This controller group allows its processes to read and write to devices, such as disks.
3. `freezer` – All processes and their children in this controller group are allowed to be frozen at-will depending on the need of the system.

4. `pids` – Any processes belonging to this controller group are limited in how many child processes they may spawn.

3.3 seccomp

Secure Computing state, or `seccomp`, is a kernel mode for processes that either limits said process to a small subset of system calls, or limits them based on *Berkeley Packet Filters* (BPFs) passed to it [8]. It is enabled by the kernel config flag `CONFIG_SECCOMP=y` at compile time.

3.3.1 `seccomp` and Berkeley Packet Filters

BPF is a packet filtering system that exists within the kernel, and is accessed with `<linux/filter.h>`. It has its own assembly-like syntax and explanations thereof are outside the scope of this paper: An entirely new paper would need to be written to delve into that subject.

The main takeaway of BPF with regards to `seccomp` is that:

1. BPF filter can be defined and enclosed inside a structure within the source code
2. The filter is often written statically (hand-write what system calls the process would need)
3. Any system calls outside the filter would result in the process being terminated.

3.3.2 `seccomp` modes and their usage

`seccomp` has two main modes:

1. `SECCOMP_SET_MODE_STRICT` – The calling thread only has access to a subset of system calls consisting of `read()`, `write()`, `exit()`, and `sigreturn()` only.
2. `SECCOMP_SET_MODE_FILTER` – The allowed list of system calls are passed by a pointer to a BPF filter.

All the `seccomp` related interfaces are exposed as `<linux/seccomp.h>`. Any process may "elevate" itself to one of these more secure modes by invoking the `prctl()` function during execution [9]:

- `prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);` for strict mode
- `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, filter);` for filter mode

Followed by "shutting the door" by disabling any future `prctl()` calls:

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

Enforcement of `seccomp` on the kernel end is simple: Any invocations of system calls outside the process's allowed list results in the process being killed outright. This way, no matter how malicious the program is, its ability to do damage is severely limited by the fact that it simply cannot issue the potentially more damaging system calls.

4 Conclusions

More than ever, the need to protect computer systems from malicious code is ever present. The ability to fragment physical machines into logical partitions that cannot easily broach the divide is indispensable in a world driven by sensitive data stored in centralized servers. To this end, sandboxing serves a critical purpose.

References

[1] Silberschatz, Abraham, et al. *Operating System Concepts*. 10th ed. Laurie Rosatone, 2018.

[2] Tanerbaum, Andrew S., and Albert S. Woodhull. *Operating Systems Design and Implementation*. 3rd ed., Prentice Hall, 2006.

[3] Ballesteros, Francisco J. *Introduction to Operating Systems Abstractions Using Plan 9 from Bell Labs*. Draft 9/28/2007.

[4] "namespaces(7)." The Linux man-pages project, 11 Apr. 2020, http://man7.org/linux/man-pages/man7/namespaces.7.html.

[5] "chroot: Run a command with a different root directory." GNU Operating System. https://www.gnu.org/software/coreutils/manual/html_node/chroot-invocation.html. Accessed 18 Apr. 2020.

[6] "How to break out of a chroot() jail." Simon's computing stuff. https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html.

[7] "cgroups(7)." The Linux man-pages project, 11 Apr. 2020, http://man7.org/linux/man-pages/man7/cgroups.7.html.

[8] "seccomp(2)." The Linux man-pages project, 19 Nov. 2019, http://man7.org/linux/man-pages/man2/seccomp.2.html.

[9] “Denying syscalls with seccomp.” Eigenstate. https://eigenstate.org/notes/seccomp. Accessed 19 Apr. 2020.