Seccomp syscall filtering in Mesos containerizer

Status	Reviewable
Revision	0.1
Last Updated	7/24/2018
Authors	Andrei Budnik <abudnik@mesosphere.io> Jay Guo <guojiannan1101@gmail.com>

JIRA Epic: MESOS-9029 - Seccomp syscall filtering in Mesos containerizer

Introduction

The Linux kernel exposes a large number of system calls to userland processes. However, only a subset of all available system calls is used by a typical user-space program. There were cases of vulnerabilities in the implementation of system calls allowing to attack the OS kernel. Therefore, having a mechanism for filtering of certain system calls could significantly enhance system security. Since a modern Linux kernel supports syscall filtering, adding it to a Mesos containerizer is another step towards running applications in a secure sandboxed environment.

Background

Seccomp (Secure Computing) was initially introduced in the Linux kernel since 2.6.12 as a mechanism to restrict syscalls that a process can make. It reduces attack surface of the kernel. It is a tool for developers to build a sandbox.

Initially seccomp could only be set to strict mode that allows `read()`, `write()`, `_exit()` and `sigreturn()` syscalls. In Linux 3.5, filter mode (a.k.a. “seccomp2”) was added to control which syscalls are permitted. Before Linux 3.17 (2014), `prctl()` was multiplexed to set seccomp mode and filters, however new syscall `seccomp()` was added in 3.17 as a superset of `prctl()` functionality.

Once a process puts itself into seccomp mode, all subsequent syscalls made by this process and its subprocesses will be validated against a filter before making the actual syscalls. If validation fails, Linux kernel may either KILL the process or let it proceed but fail that specific syscall, depending on user-defined policy. Once a filter is installed, it can NOT be uninstalled and keep effective till the end of process. The rationale behind this is that all code running after

`seccomp()` is considered untrusted, therefore it should not be able to withdraw restrictiveness even if the process is compromised. If `fork()` or `clone()` is allowed by the filter, any child processes will be constrained to the same system call filters as the parent. If `execve()` is allowed, the existing filters will be preserved across a call to `execve()`.

Seccomp leverages BPF (Berkeley Packet Filter) to implement filters. Essentially BPF defines an in-kernel virtual machine with a very simple instruction set. A BPF program is jump-forward only, which guarantees its termination. Seccomp takes a BPF program and installs it into the kernel. BPF is a fairly old technology (20+ yrs), so deep dive into BPF is beyond the scope of this document.

Seccomp is used in some well-known projects including Chrome OS, Chrome browser, vsftpd, OpenSSH and Docker. Docker comes with a default profile which provides a modest protection. It is enabled in Docker by default, whereas user could explicitly disable it or supply with a customized profile. Some more info can be found here.

Libseccomp is a library that abstracts away Seccomp subsystem in the Linux kernel by providing an easy-to-use API. A user of this library declares a set of filtering rules instead of writing a low-level ASM-like BPF program. Moreover, the libseccomp library is designed to work on different architectures and Linux kernel versions.

A minimal example where libseccomp API is used to load seccomp filters:

```
int ret;
scmp_filter_ctx ctx; // Context data structure to store filter
// Initialize a seccomp filter with default action.
ctx = seccomp_init(SCMP_ACT_ALLOW);
// Add syscall and corresponding action to the filter created.
ret = seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(close), 0);
// Load the filter into kernel.
ret = seccomp_load(ctx);
// Release all memory associated with this filter.
seccomp_release(ctx);
```

Declaration and description of `libseccomp` functions and constants can be found here.

Goals

- Allow operator to enable Seccomp with a default profile on a particular agent.
- Allow framework to enforce a Seccomp profile on a particular container.
- Define default Seccomp profiles for Mesos container.

Use Cases

1. Framework enforces seccomp profile on a particular container while executing a task. The name of a Seccomp profile is passed on via `LinuxInfo` in `ContainerInfo` protobuf. ~~The specified Seccomp profile is used instead of the profile passed to an agent via command-line flag `--seccomp-default-profile`.~~

~~**2. Cluster Operator** enforces default seccomp filter for all containers launched on a Mesos Agent. Operator supplies both Seccomp profile via `--seccomp-default-profile` flag and an absolute path to the directory containing Seccomp profiles via `--seccomp-profiles-dir` flag. Note that `--seccomp-default-profile` should be a path relative to the directory containing Seccomp profiles, which is specified via the `--seccomp-profiles-dir` flag. With this configuration, syscalls made by any container are being filtered according to the rules defined in the Seccomp profile.~~

Design Details

Seccomp configuration format

A Seccomp profile is a JSON file containing Seccomp filtering rules. It must be fully compatible with the Docker Seccomp profile format. Docker profile format is not an industry standard, although it allows us to specify all the filtering rules provided by the `libseccomp` library. Its structure hasn't changed since Docker v1.13-rc1 (mid 2016). An example of Seccomp profile, which is used by default in Docker, can be found here. It consists of three sections:

- `syscalls` - contains a list of syscall filtering rules.
- `defaultAction` - specifies a default action to be taken for syscalls which match no rules in the filter.
- `archMap` - contains a mapping: Architecture -> [List of subarchitecture], e.g. `SCMP_ARCH_X86_64` -> [`SCMP_ARCH_X86`, `SCMP_ARCH_X32`].

`archMap` is used to specify architectures with its corresponding subarchitectures, which are used as a part of Seccomp filter. For example, when a process is launched on x86-64, then accordingly to the `archMap` all syscall filtering rules will be associated with `SCMP_ARCH_X86_64`, `SCMP_ARCH_X86` and `SCMP_ARCH_X32` architectures. In this case, the filtering rules will be applied to 32-bit programs as well as to 64-bit programs. All available architecture constants are defined in seccomp.h.

`syscalls` contains a list of syscall filtering rules, where a single rule can be described as follows:

```
{
     “names”: [List of syscall names, e.g. `clone()`, `bpf()`,
etc.],

     “action”: “SCMP_ACT_*”, // An action to be taken when the
given rule is matched up. E.g. SCMP_ACT_ALLOW.

     “args”: [List of filtering rules for syscall arguments. It can
be an empty list.],

     “includes”: [Inclusion rules for the given rule. It can be an
empty list],

     “excludes”: [Exclusion rules for the given rule. It can be an
empty list]
}
```

`args` contains a list of filtering rules for syscall arguments, where a single rule can be described as follows:

```
{
     // The number of the syscall argument we are checking,
starting at 0.
     “index”: Int,

      // The first comparison value. The rule will match if
argument
      // $syscall_arg[index] is $op the provided comparison value.
     “value”: Int,

     // Some comparison operators accept two values. Masked equals,
      // for example, will mask $syscall_arg[index] with the second
value
      // provided (via bitwise AND) and then compare against the
first
      // value provided
     “valueTwo”: Int,

     // The comparison operator, e.g. SCMP_CMP_EQ.
     “op”: “SCMP_CMP_*”
}
```

`includes` and `excludes` sections have the same format for their rules, which can be described as follows:

```
{
      “caps”: [List of capabilities], // E.g. [“CAP_SYS_ADMIN”, “CAP_SYS_CHROOT”]

      “arches”: [List of architectures] // E.g. [“arm”, “arm64”]
}
```

If the rule declared in `includes` or `excludes` section is matched up, then the related syscall filtering rule is included or excluded from syscall filtering rules, respectively. For example, if the rule is `”includes”: [{“caps”: [“CAP_SYS_ADMIN”]}]` and a running process doesn’t have `CAP_SYS_ADMIN`, then the given syscall filtering rule is not included to Seccomp filtering rules using `libseccomp` API. Note that exclusion rules have a higher priority than inclusion rules.

Justification of the Docker Seccomp Format

The format of the Docker seccomp config has the following main advantages:

1) It allows to specify all the filtering rules provided by the `libseccomp` library.
2) It precisely reflects the API of `libseccomp` library.
3) The Docker default config can be used without the necessity to convert it into our own format.

Note that implementing a sane Seccomp profile, while providing wide application compatibility, is difficult. The Docker default config can be seen as an ever-growing database of rules, which is used to alleviate many known security issues found in the Linux kernel at the moment.

Maintenance and Implementation Complexity of the Docker Seccomp Format

Since the Docker config format is a JSON with a deep structure, it’s a very non-trivial task to implement a parser for it. Moreover, it contains additional rules that extend `libseccomp` API. For example, the logic for inclusion/exclusion of filtering rules depends on properties like `Capabilities`, which are usually filled by isolators. The resulting process capabilities can be requested only in the containerizer launcher after calling functions `setuid()` and `capset()`. Hence, the full logic for inclusion/exclusion of filtering rules can’t be encapsulated in the `linux/seccomp` isolator.

The second potential problem is that Docker Seccomp config format might be changed in a backward-incompatible way. The format is not documented and doesn’t contain versioning information. However, if that happens, then the previous Docker format, which will be supported by Mesos, can be used as a base for our own format. This new format can be designed to

preserve backward-compatibility with the old one. But this requires implementing a tool for converting new docker format to our own format.

Design Overview

~~The name of the Seccomp profile can be defined via the agent's `--seccomp-default-profile` command line flag. In addition, a framework can override the name of the Seccomp profile for a particular container, so that it will be used instead of the name provided by the agent's `--seccomp-default-profile` flag. In both cases the agent will use `--seccomp-profiles-dir` command line flag to build an absolute path to the Seccomp profile.~~

The Seccomp profile is translated from the JSON to the `ContainerSeccompProfile` protobuf by a new `linux/seccomp` isolator. The `ContainerSeccompProfile` protobuf is passed to the containerizer launcher via `launch_info` flag as a part of `ContainerLaunchInfo` protobuf message.

The containerizer launcher translates `ContainerSeccompProfile` protobuf into invocations of `libseccomp` API to generate a BPF program which should be loaded into the kernel right before calling `execvp()`. Docker internal data structures and a module that converts Seccomp config into these data structures can be used as a reference.

Mesos source code is bundled with a `libseccomp` library, which is linked statically with a `mesos-containerizer` executable by default. Autotools configuration script should provide flags to disable Seccomp isolator and to specify where to locate the libseccomp library. The latter can be used to build Mesos agent with a non-bundled, OS-provided `libseccomp`.

Since a Mesos containerizer persists information about containers on disk, the `linux/seccomp` isolator can be implemented as a stateless class. Therefore, implementation of `MesosIsolator::recover()` method is not required for the `linux/seccomp` isolator.

Seccomp Profile Inheritance

It is allowed to have distinct Seccomp profiles enabled for a parent and child containers within a POD. Hence, a child container might have less restrictive Seccomp profile than its parent has. If a framework attempts to launch a nested container without Seccomp profile specified, then a nested container inherits parent's Seccomp profile. Mesos containerizer persists information about containers on disk via `ContainerLaunchInfo` proto, which includes `ContainerSeccompProfile` proto. So, a Mesos agent can use this proto to load the parent's profile for a child container. Therefore, when a child inherits the parent's Seccomp profile, Mesos agent doesn't have to re-read a Seccomp profile from the disk, which was used

for the parent container. Otherwise, we would have to check that a file content hasn't changed since the parent was launched.

Framework's Protobuf Message

The Seccomp profile can be specified via new `SeccompInfo` protobuf:

```
/**
 * Encapsulation for Seccomp configuration, which is Linux
specific.
 */
message SeccompInfo {
  // A filename of the Seccomp profile. This should be a path
  // relative to the directory containing Seccomp profiles, which
is specified on the agent via the command-line flag.
  optional string profile_name = 1 [default = "default.json"];
}
```

`SeccompInfo` protobuf is included into `LinuxInfo` in the `mesos.proto`:

```
/**
 * Encapsulation for Linux specific configuration.
 * E.g, capabilities, limits etc.
 */
message LinuxInfo {
  …
  // Represents Seccomp configuration, which is used for syscall
filtering.
  optional SeccompInfo seccomp = 5;
}
```

`LinuxInfo` is included into both `ContainerInfo/TaskInfo` and `ContainerInfo/ExecutorInfo`.

Agent's Protobuf Message

`ContainerSeccompProfile` is added to `slave/containerizer.proto`:

```
/**
 * Encapsulation of Linux seccomp filter
 * Reference:
https://github.com/seccomp/libseccomp/blob/master/include/seccomp.h
.in // NOLINT
 */
message ContainerSeccompProfile {
```

```
enum Architecture {
  UNKNOWN = 0;
  ARCH_X86 = 1;
  ARCH_X86_64 = 2;
  ARCH_X32 = 3;
  ARCH_ARM = 4;
  ARCH_AARCH64 = 5;
  ARCH_MIPS = 6;
  ARCH_MIPSEL = 7;
  ARCH_MIPS64 = 8;
  ARCH_MIPSEL64 = 9;
  ARCH_MIPS64N32 = 10;
  ARCH_MIPSEL64N32 = 11;
  ARCH_PPC = 12;
  ARCH_PPC64 = 13;
  ARCH_PPC64LE = 14;
  ARCH_S390 = 15;
  ARCH_S390X = 16;
}

message Syscall {
  enum Action {
    UNKNOWN = 0;
    ACT_KILL = 1;
    ACT_TRAP = 2;
    ACT_ERRNO = 3;
    ACT_TRACE = 4;
    ACT_LOG = 5;
    ACT_ALLOW = 6;
  }

  message Arg {
    enum Operator {
      UNKNOWN = 0;
      CMP_NE = 1;         // not equal
      CMP_LT = 2;         // less than
      CMP_LE = 3;         // less than or equal
      CMP_EQ = 4;         // equal
      CMP_GE = 5;         // greater than or equal
      CMP_GT = 6;         // greater than
      CMP_MASKED_EQ = 7; // masked equality
    }
```

```
      // The number of the argument we are checking, starting at 0.
      required uint32 index = 1;

      // The comparison operator, e.g. CMP_*.
      required Operator op = 2;

      // The first comparison value. The rule will match if
argument
      // $syscall_arg[index] is $COMPARE_OP the provided comparison
value.
      required uint64 value = 3;

      // Some comparison operators accept two values. Masked
equals,
      // for example, will mask $syscall_arg[index] with the second
value
      // provided (via bitwise AND) and then compare against the
first
      // value provided.
      required uint64 value_two = 4;
    }

    message Filter {
      optional CapabilityInfo capabilities = 1;
    }

    repeated string names = 1;
    required Action action = 2;
    repeated Arg args = 3;
    optional Filter includes = 4;
    optional Filter excludes = 5;
  }

  required Syscall.Action default_action = 1;
  repeated Architecture architectures = 2;
  repeated Syscall syscalls = 3;
}
```

`ContainerSeccompProfile` is added to `ContainerLaunchInfo`:

```
message ContainerLaunchInfo {
```

```
  …
  // (Linux only) The Seccomp profile for the container.
  // The profile is used to configure syscall filtering via
`libseccomp`.
  optional ContainerSeccompProfile seccomp_profile = 18;
}
```

Agent API for File Operations on Security Configs

Management of security configuration at cluster level, including Seccomp profiles, requires operations for adding, updating and removing configs via Agent Operator API.
Currently, there is only one type of security config supported: Seccomp. Since there are other security-related technologies like SELinux, AppArmor and others, it'd be beneficial to provide a generic API for uploading security configs.

All new messages are added to `agent/agent.proto` into the `Call` message:

```
enum SecurityConfigType {
  UNKNOWN = 0;
  SECCOMP = 1;
}
```

```
// Adds a new security config file.
//
// The content of `data` field will be written into a new config
file in
// the corresponding config directory. The config directory is
chosen based on the agent's flag that specifies path to the config
directory for the given config type. The name of the config file is
specified by the `name` field.
//
// Returns 200 OK if a new config file is created, or an identical
config file
//   exists.
// Returns 400 Bad Request if `data` is not well-formed.
// Returns 403 Forbidden if the call is not authorized.
// Returns 409 Conflict if a config file with the same name and
type exists, but the content is
//   not identical.
// Returns 500 Internal Server Error if anything goes wrong.
message AddSecurityConfig {
  required SecurityConfigType type = 1;
  required string name = 2;
  required bytes data = 3;
```

```
}
```

```
// Updates an existing security config file.
//
// The content of `data` field will be written into a new config
file in
// the corresponding config directory. The config directory is
chosen based on the agent's flag that specifies path to the config
directory for the given config type. The name of the config file is
specified by the `name` field.
//
// Returns 200 OK if an existing config file is updated, or there
is is no change
//   in the config file.
// Returns 400 Bad Request if `data` is not well-formed.
// Returns 403 Forbidden if the call is not authorized.
// Returns 404 Not Found if no config file of the same type and
name exists.
// Returns 500 Internal Server Error if anything goes wrong.
message UpdateSecurityConfig {
  required SecurityConfigType type = 1;
  required string name = 2;
  required bytes data = 3;
}
```

```
// Removes a security config file.
//
// The config file of specified type and name will be removed from
the corresponding config directory. The config directory is chosen
based on the agent's flag that specifies path to the config
directory for the given config type. The name of the config file is
specified by the `name` field.
//
// Returns 200 OK if the config file is removed, or no matching
config file
//   exists.
// Returns 403 Forbidden if the call is not authorized.
// Returns 500 Internal Server Error if anything goes wrong.
message RemoveSecurityConfig {
  required SecurityConfigType type = 1;
  required string name = 2;
}
```