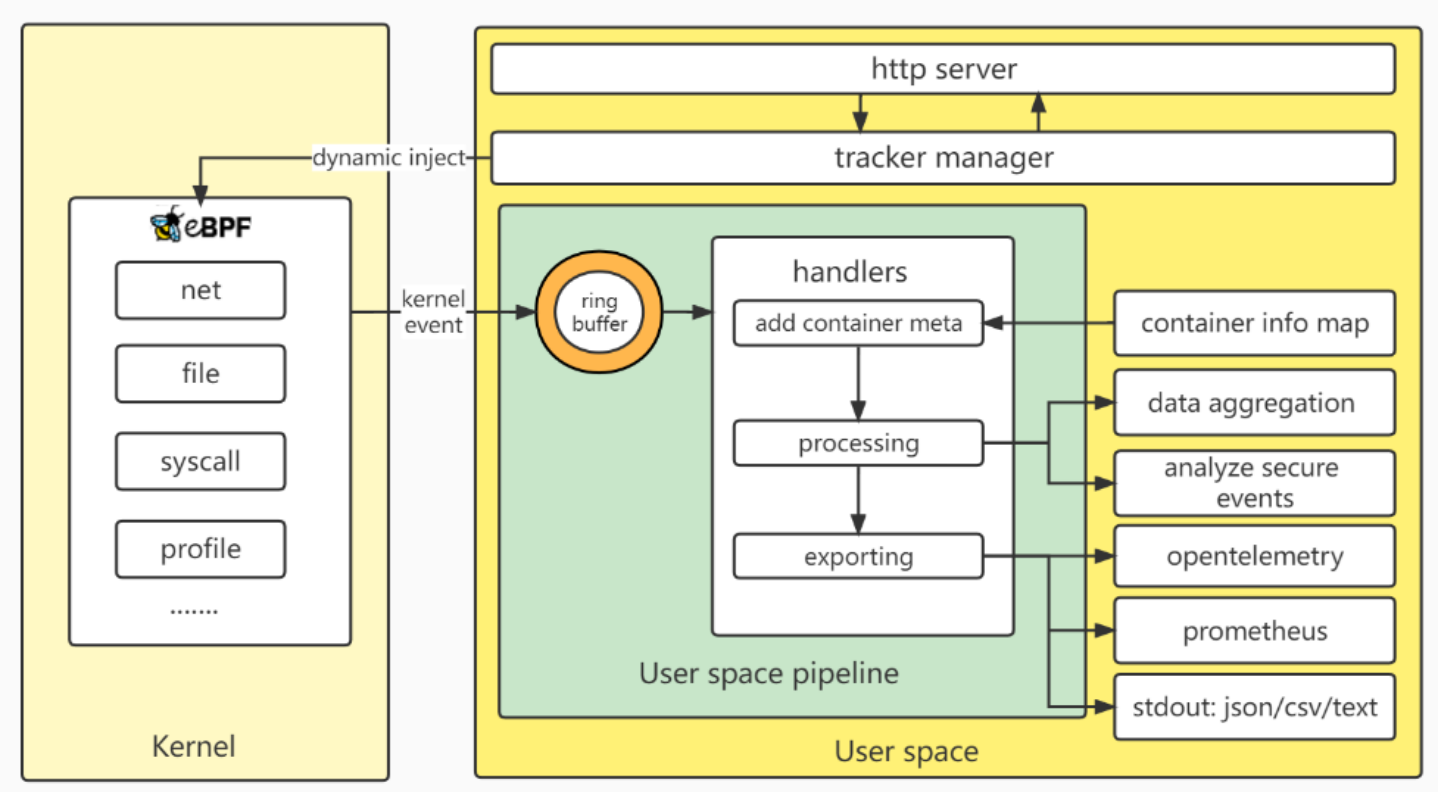
系统框架与ebpf探针设计

模块设计



tracker_manager定义

定义在include/agent/tracker_manager.h中

定义了一个名为 `tracker_manager` 的 C++ 类，用于管理和控制多个类型为 `tracker_base` 的跟踪器实例。该类提供了添加、删除、列出和启动跟踪器实例的功能。具体作用如下：

1. **类设计**：
   - 使用 RAII（Resource Acquisition Is Initialization）风格，确保在对象销毁时自动清理资源。
   - 使用 `unique_ptr` 以确保跟踪器实例的所有权和生命周期管理。
2. **内部数据结构**：
   - `tracker_base_data`：结构体，用于保存单个跟踪器的信息，包括其名称和实际的跟踪器实例。
   - `id_count`：用于生成唯一的跟踪器 ID。
   - `trackers`：存储所有跟踪器实例的映射，键是唯一的 ID，值是 `tracker_base_data` 结构体。
3. **公共接口**：

- `~tracker_manager()`：析构函数，销毁时输出信息（假设使用 `spdlog` 作为日志库）。
- `remove_tracker(int id)`：通过 ID 删除特定的跟踪器。
- `get_tracker_list()`：返回所有跟踪器的 ID 和名称列表。
- `start_tracker(std::unique_ptr<tracker_base> tracker_ptr, const std::string &name)`：
  - 启动一个跟踪器，将其放入映射中，并返回新添加的跟踪器数量。
- `remove_all_trackers()`：清空所有跟踪器。

4. **线程管理**：
   - 每个跟踪器实例使用 `std::jthread` 进行并行运行，通过 `tracker_base::start_tracker` 方法启动。

主要有五个ebpf探针

- process
- syscall
- tcp
- files
- ipc

container_manager定义

定义在include/agent/container_manager.h中

这段代码定义了一个名为 `container_manager` 的 C++ 类，用于管理和追踪容器或 Kubernetes 集群中的进程和容器信息。它提供了一套接口来查询、获取和管理容器信息，并且通过事件处理器来跟踪容器内的进程变化。代码的具体作用如下：

类设计

1. **`container_manager` 类**：
   - 负责管理容器和 Kubernetes 集群中的进程和容器信息。
   - 包含内部类 `container_tracking_handler` 和 `container_info_handler` 用于处理不同事件类型。
   - 提供查询进程所属容器的接口，并且通过 `container_client` 类与容器管理服务通信。

内部类

1. **`container_tracking_handler`**：
   - 继承自 `event_handler<process_event>`。

- 用于处理进程事件并更新 `container_manager` 中的容器信息。
- 通过 `handle` 方法将事件和 `container_manager` 关联起来。

2. **`container_info_handler<EVENT>`**:
   - 继承自 `event_handler<EVENT>`，也是一个模板类。
   - 用于处理与进程相关的事件，并根据事件的 PID 获取容器信息。
   - 在 `handle` 方法中，使用 `container_manager` 来获取容器信息并关联到事件数据中。
3. **`container_client`**:
   - 提供与容器服务交互的接口，使用 `httplib::Client` 与 Docker API 通信。
   - 实现了获取所有容器信息、获取容器内进程信息、检查容器信息等方法。
   - `get_os_container_info` 返回操作系统容器信息（例如未在任何容器中的进程）。
4. **`container_info_map`**:
   - 使用 `std::shared_mutex` （即读写锁）来保护 `std::unordered_map`，确保线程安全。
   - 提供插入、获取和删除 PID 对应的容器信息的方法。

私有数据

1. **`info_map`**:
   - `container_info_map` 类的实例，用于存储 PID 到容器信息的映射。
2. **`client`**:
   - `container_client` 类的实例，用于与容器服务交互。
3. **`os_info`**:
   - `container_info` 类型的实例，用于表示不在任何容器中的进程的默认信息。

ebpf 探针设计

采用 ebpf 探针的方式，可以获取到安全事件的相关信息，并且可以通过 prometheus 监控指标进行监控和分析。

探针代码分为两个部分:

其一是在 `bpftools` 中，是针对相关 ebpf 程序的 libbpf 具体探针接口实现，负责ebpf 程序的加载、配置、以及相关用户态和内核态通信的代码,运行在内核态;

另外一部分是在 src 中，针对 ebpf 探针上报的信息进行具体处理的 C++ 类实现，负责根据配置决定ebpf上报的信息将会被如何处理,运行在用户态。

1. 内核态：ebpf 探针相关 C 代码设计,以tcp为例：

代码位于bpftools/tcpconnect/tcp.bpf.c,仅截取关键的探针点代码展示：

```
SEC("kprobe/tcp_v4_connect")
int BPF_KPROBE(tcp_v4_connect, struct sock *sk) {
  return enter_tcp_connect(ctx, sk);
}

SEC("kretprobe/tcp_v4_connect")
int BPF_KRETPROBE(tcp_v4_connect_ret, int ret) {
  return exit_tcp_connect(ctx, ret, 4);
}

SEC("kprobe/tcp_v6_connect")
int BPF_KPROBE(tcp_v6_connect, struct sock *sk) {
  return enter_tcp_connect(ctx, sk);
}

SEC("kretprobe/tcp_v6_connect")
int BPF_KRETPROBE(tcp_v6_connect_ret, int ret) {
  return exit_tcp_connect(ctx, ret, 6);
}
```

通过追踪 `tcp_v4_connect` 和 `tcp_v6_connect` 内核函数，获取相关的 TCP 连接信息。

主要作用

- **追踪 TCP 连接**：该程序追踪 IPv4 和 IPv6 上的 TCP 连接，并在连接开始和结束时记录相关数据。
- **过滤**：可以按特定端口、UID 或 PID 过滤追踪。
- **统计和事件输出**：
  - **统计**：统计各个 TCP 流的连接次数。
  - **事件输出**：将 TCP 连接事件输出到用户态，通过 `perf_event` 输出。

使用**kprobe 和 kretprobe来监测挂载点tcp_v4_connect、tcp_v6_connect来检测：**

- `tcp_v4_connect` 和 `tcp_v6_connect`：
  - 使用 `kprobe` 追踪 `tcp_v4_connect` 和 `tcp_v6_connect` 函数调用。
  - 调用 `enter_tcp_connect` 记录 socket 信息。
- `tcp_v4_connect_ret` 和 `tcp_v6_connect_ret`：

- 使用 `kretprobe` 追踪 `tcp_v4_connect` 和 `tcp_v6_connect` 函数返回值。
- 调用 `exit_tcp_connect` 进行过滤、统计或事件输出。

得到的数据放进tcp.h中定义的数据结构tcp_event,该数据结构定义如下所示，包含了协议号以及对应的进程uid，pid，ip，端口等等信息。

```
struct tcp_event {
  union {
    unsigned int saddr_v4;
    unsigned char saddr_v6[16];
  };
  union {
    unsigned int daddr_v4;
    unsigned char daddr_v6[16];
  };
  char task[TASK_COMM_LEN];
  unsigned long ts_us;
  unsigned int af; // AF_INET or AF_INET6
  unsigned int pid;
  unsigned int uid;
  unsigned short dport;
};
```

每个 ebpf 探针会被当做一个独立的线程运行，这个线程会被放到一个单独的线程池中，这样就可以保证每个 ebpf 探针都是独立的进程：

- 我们可以在同一个二进制程序或者进程中同时运行多个探针，例如可以同时运行 process 和 tcp，通过 process 获取的容器元信息，以 pid 作为主键来查询 tcp 每个连接相关的容器信息。
- 探针可以在 eunomia 运行的任意时刻被启动，也可以在任意时刻被关闭。
- 同一种类型的探针可以被运行多个实例，比如来监测不同的 cgroups 或者不同的进程。

每个探针有两个重要的数据结构， event 和 env。event 上报给用户态的信息结构体， env是对应的 tracker 的配置：

2. 用户态：ebpf探针相关cpp代码设计，handler设计，以tcp为例

设计思想：

- 将每个不同的ebpf探针当作单独的类

- 每个探针类都可以有数量不限的事件处理方式（handler 类）：
  - 转换成 json 类型
  - 上报给 prometheus
  - 打印输出
  - 保存文件等

以tcp为例，该部分的cpp代码详见include\agent\tcp.h

```
/// trace tcp start and exit
class tcp_tracker : public tracker_with_config<tcp_env, tcp_event>
{
public:
  tcp_tracker(config_data config);

  /// create a tracker with deafult config
  static std::unique_ptr<tcp_tracker>
create_tracker_with_default_env(tracker_event_handler handler);
  static std::unique_ptr<tcp_tracker> create_tracker_with_args(
      tracker_event_handler handler,
      const std::vector<std::string> &args)
  {
    return create_tracker_with_default_env(handler);
  }

  // start tcp tracker
  void start_tracker();

  // used for prometheus exporter
  struct prometheus_event_handler : public event_handler<tcp_event>
  {
    prometheus::Family<prometheus::Counter> &eunomia_tcp_v4_counter;
    prometheus::Family<prometheus::Counter> &eunomia_tcp_v6_counter;
    void report_prometheus_event(tracker_event<tcp_event> &e);

    prometheus_event_handler(prometheus_server &server);
    void handle(tracker_event<tcp_event> &e);
  };
  static int fill_src_dst(sender &s, sender &d,const tcp_event &e);

  // convert event to json
  struct json_event_handler_base : public event_handler<tcp_event>
  {
    std::string to_json(const struct tcp_event &e);
```

```
35    };
36
37    // used for json exporter, inherits from json_event_handler
38    struct json_event_printer : public json_event_handler_base
39    {
40      void handle(tracker_event<tcp_event> &e);
41    };
42
43    struct plain_text_event_printer : public event_handler<tcp_event>
44    {
45      void handle(tracker_event<tcp_event> &e);
46    };
47
48    struct csv_event_printer : public event_handler<tcp_event>
49    {
50      void handle(tracker_event<tcp_event> &e);
51    };
52
53  private:
54      static void handle_tcp_sample_event(void *ctx, int cpu, void *data,
    unsigned int data_sz);
55  };
```

这部分代码继承自 tracker_with_config，每个 ebpf 探针的代码都会继承自 tracker_base 和 tracker_with_config（本身继承自tracker_base)

这段代码定义了一个名为 `tcp_tracker` 的 C++ 类，用于追踪 TCP 连接的开始和结束事件，并将这些事件输出为多种格式（如 JSON、纯文本、CSV、Prometheus 指标等）。它还支持通过事件处理器处理这些事件。让我们分解它的结构并解释其功能。

代码功能概述

1. **基类**：
   - `tracker_with_config<tcp_env, tcp_event>`：泛型基类，提供配置和事件类型的基础设施。
2. **类的公开接口**：
   - **构造函数**：
     - `tcp_tracker(config_data config)`：通过配置数据初始化 `tcp_tracker` 对象。
   - **静态工厂方法**：
     - `create_tracker_with_default_env(tracker_event_handler handler)`：创建具有默认环境配置的跟踪器。

- `create_tracker_with_args(tracker_event_handler handler, const std::vector<std::string> &args)`：通过提供的参数列表创建跟踪器。

- **启动跟踪器**：
  - `start_tracker()`：启动 TCP 跟踪器，开始追踪 TCP 连接事件。
- **静态辅助函数**：
  - `fill_src_dst(sender &s, sender &d, const tcp_event &e)`：填充发送者和接收者信息。

3. **事件处理器**：
   - **`prometheus_event_handler`**：
     - 继承自 `event_handler<tcp_event>`，用于将 TCP 事件报告为 Prometheus 指标。
     - `report_prometheus_event(tracker_event<tcp_event> &e)`：将 TCP 事件报告给 Prometheus。
     - `handle(tracker_event<tcp_event> &e)`：处理 TCP 事件。
   - **`json_event_handler_base`**：
     - 继承自 `event_handler<tcp_event>`，提供事件到 JSON 格式的转换。
     - `to_json(const tcp_event &e)`：将事件转换为 JSON 字符串。
   - **`json_event_printer`**：
     - 继承自 `json_event_handler_base`，将 TCP 事件以 JSON 格式打印。
     - `handle(tracker_event<tcp_event> &e)`：处理 TCP 事件并输出 JSON。
   - **`plain_text_event_printer`**：
     - 继承自 `event_handler<tcp_event>`，将 TCP 事件以纯文本格式打印。
     - `handle(tracker_event<tcp_event> &e)`：处理 TCP 事件并输出纯文本。
   - **`csv_event_printer`**：
     - 继承自 `event_handler<tcp_event>`，将 TCP 事件以 CSV 格式打印。
     - `handle(tracker_event<tcp_event> &e)`：处理 TCP 事件并输出 CSV。
4. **私有方法**：
   - `handle_tcp_sample_event(void *ctx, int cpu, void *data, unsigned int data_sz)`：静态方法，用于处理从 eBPF 程序收到的 TCP 事件样本。

handler定义

handler 的具体实现在 include\agent\model\event_handler.h 中，关键定义部分如下：

```cpp
/// T is the event from C code
template <typename T>
struct tracker_event
{
    T data;
    container_info ct_info;
    // TODO: add more data options here?
};

/// the event handler for share_ptr
template <typename T>
struct event_handler_base
{
public:
    virtual ~event_handler_base() = default;
    virtual void handle(tracker_event<T> &e) = 0;
    virtual void do_handle_event(tracker_event<T> &e) = 0;
};

/// the event handler for single type

/// all single type event hanlder should inherit from this class
template <typename T>
struct event_handler : event_handler_base<T>
{
std::shared_ptr<event_handler_base<T>> next_handler = nullptr;
public:
    virtual ~event_handler() = default;

    /// implement this function to handle the event
    virtual void handle(tracker_event<T> &e) = 0;

    /// add a next handler after this handler
    std::shared_ptr<event_handler<T>>
add_handler(std::shared_ptr<event_handler<T>> handler)
    {
        next_handler = handler;
        return handler;
    }
    /// do the handle event
    /// pass the event to next handler
    void do_handle_event(tracker_event<T> &e)
    {
        bool is_catched = false;
        try {
            handle(e);
```

```
46          } catch (const std::exception& error) {
47              // std::cerr << "exception: " << error.what() << std::endl;
48              is_catched = true;
49          }
50          if (!is_catched && next_handler)
51              next_handler->do_handle_event(e);
52          return;
53      }
54  };
```

每个探针类可以拥有多个事件处理器（handler）类，这些处理器可以执行各种操作，例如将事件转换为 JSON 格式、上报给 Prometheus、打印输出、保存文件或进行数据聚合等。这些 handler 类通过链表结构组织，可以在运行时动态地组装。

当 eBPF 上报的事件被处理时，这些事件将按照 handler 的顺序依次处理。如果某个 handler 返回 false，则该事件不会继续传递给后续的 handler；否则，事件将被传递到最后一个 handler 进行最终处理（捕获机制）。

上报的事件可以被转换为不同的类型，例如进行聚合操作，或者将事件结构体转换为 JSON 格式。多个不同的 eBPF 探针可以将事件发送到同一个 handler，例如，将文件访问信息和进程执行信息合并为一个事件，获取每个文件访问的进程的 Docker ID 和 Docker 名称，并将其发送到 Prometheus。

handler 还可以用来匹配相应的安全规则，并在检测到可能的安全风险时触发警告操作。

在tcp部分的代码中，定义了如下的handler规则：

- **`prometheus_event_handler`**：
  - 继承自 `event_handler<tcp_event>`，用于将 TCP 事件报告为 Prometheus 指标。
  - `report_prometheus_event(tracker_event<tcp_event> &e)`：将 TCP 事件报告给 Prometheus。
- **`json_event_handler_base`**：
  - 继承自 `event_handler<tcp_event>`，提供事件到 JSON 格式的转换。
- **`json_event_printer`**：
  - 继承自 `json_event_handler_base`，将 TCP 事件以 JSON 格式打印。
- **`plain_text_event_printer`**：
  - 继承自 `event_handler<tcp_event>`，将 TCP 事件以纯文本格式打印。
- **`csv_event_printer`**：
  - 继承自 `event_handler<tcp_event>`，将 TCP 事件以 CSV 格式打印。