插件+linx_ebpf 演示文档

V0.1

北京凝思软件股份有限公司

文档修订记录

VER	REV	修订人	修订日期	简要说明	批准人	批准日期
V0.1	C	周星宇	2025-06-13	初次创建		
VER（版本编号）：V——版本编号；R——修订编号。						
REV（修订状态）：C——创建；A——增加；M——修改；D——删除。						

目录

表格

插图

1 Falco 插件 + linx_ebpf

1.1 简介

1.1.1 概述

该实现是基于 Falco 插件功能完成的，主要利用 Falco 插件能提供事件源与字段解析能力，加载自研 linx_ebpf 并采集信息，采集的消息作为一个事件传递回 Falco 插件，在插件中解析消息并与字段绑定，此时字段便能应用在规则文件中，完成条件控制输出。

1.1.2 功能介绍

当前实现了以下功能：

1. ebpf 内核全量系统调用采集，并可通过 json 文件配置要采集哪些系统调用、要过滤掉哪些命令、任务等；
2. 插件提供众多可用规则字段，可在 yaml 规则文件中使用，配置在特定条件下输出特定内容；
3. 支持将 ebpf 内核采集到的信息以特定格式写入到指定文件中，是否写入文件以及文件路径可通过 JSON 进行配置。

1.2 成果演示

目前获取内核采集信息有两种途径：

1. 与 Falco 规则文件结合，根据插件提供的字段（参考 1.3.2.2 章）可以配置什么时候输出，输出什么内容，一般这些信息会直接显示在终端上，具体效果如下：

```
[root@localhost linx-behavior-detection]# ./build/userspace/falco/falco -c ./build/falco.yaml -r test.yaml --enable-
source=linx_ebpf
Thu Jun 12 13:17:21 2025: Falco version: 0.0.0 (x86_64)
Thu Jun 12 13:17:21 2025: Falco initialized with configuration files:
Thu Jun 12 13:17:21 2025:    ./build/falco.yaml | schema validation: ok
Thu Jun 12 13:17:21 2025: System info: Linux version 6.6.0-72.0.0.76.vlx16.v99_2403sp1.x86_64 (mockbuild@05f9c1ea16f
b4c34b02636761a857e1f) (gcc (GCC) 12.3.1 (openEuler 12.3.1-62.v99_2403sp1), GNU ld (GNU Binutils) 2.41) #1 SMP Mon M
ar 10 13:34:38 CST 2025
Thu Jun 12 13:17:21 2025: Loading plugin 'syscall_sequence_plugin' from file /falco/syscall_sequence_plugin/build/li
bsyscall_sequence_plugin.so
Thu Jun 12 13:17:21 2025: [libs]: syscall_sequence_plugin: Init plugin ...
Thu Jun 12 13:17:21 2025: Loading rules from:
Thu Jun 12 13:17:21 2025:    test.yaml | schema validation: ok
Thu Jun 12 13:17:21 2025: The chosen syscall buffer dimension is: 8388608 bytes (8 MBs)
Thu Jun 12 13:17:21 2025: Starting health webserver with threadiness 10, listening on 0.0.0.0:8765
Thu Jun 12 13:17:21 2025: Loaded event sources: syscall, linx_ebpf
Thu Jun 12 13:17:21 2025: Enabled event sources: linx_ebpf
Thu Jun 12 13:17:21 2025: Opening 'linx_ebpf' source with plugin 'syscall_sequence_plugin'
Thu Jun 12 13:17:21 2025: [libs]: Trying to open the right engine!
Thu Jun 12 13:17:22 2025: [libs]: syscall_sequence_plugin: Open plugin ...

13:17:39.937345963: Notice 2025-06-12 13:17:39.937345963 (connect)(fd=0, uservaddr=0, addrlen=0) (unix_chkpwd) pid=(
6214) ppid=(6212) cmdline=(/usr/sbin/unix_chkpwd root chkexpiry)
13:17:39.937467975: Notice 2025-06-12 13:17:39.937467975 (connect)(fd=0, uservaddr=0, addrlen=0) (unix_chkpwd) pid=(
6214) ppid=(6212) cmdline=(/usr/sbin/unix_chkpwd root chkexpiry)
13:17:40.145709464: Notice 2025-06-12 13:17:40.145709464 (socket)(family=0, type=0, protocol=0) (systemd-logind) pid
=(1159) ppid=(1) cmdline=(/usr/lib/systemd/systemd-logind)
```

插图 1: Falco 终端输出

2. 以自定义的固定格式输出到文件中，文件路径可配置（参考 1.4.1 章），当前自定义输出格式如下:

```
[时间戳]: 系统调用名, syscall_id=xx, dir=<|>, res=0, user=xxx,
comm=(xxx), cmdline=(xxx), pid=xx, tid=xx, ppid=x(xx), args=(xxxxxx),
fds=[xxxxxx]
```

具体输出效果如下:

```
[2025-06-12 13:18:10.347507525]: unlinkat, syscall_id=263, dir=<, res=0, user=root, comm=(sd-rmrf), cmdline=(sd-rmrf
), pid=6295, tid=6295, ppid=1(systemd), args=(dfd=-100, pathname=/var/tmp/systemd-private-3b9576e5c1cd491ba815acd406
734b6f-systemd-hostnamed.service-5j0skj, flag=512), fds=[0(/dev/null), 1(/dev/null), 2(/dev/null)]
[2025-06-12 13:18:10.348558453]: socket, syscall_id=41, dir=>, res=-1, user=root, comm=systemd-cgroups, cmdline=/usr
/lib/systemd/systemd-cgroups-agent /system.slice/systemd-hostnamed.service, pid=6292, tid=6292, ppid=2(kthreadd), ar
gs=(family=0, type=0, protocol=0), fds=[0(/dev/null), 1(/dev/null), 2(/dev/null)]
[2025-06-12 13:18:10.348567743]: socket, syscall_id=41, dir=<, res=3, user=root, comm=systemd-cgroups, cmdline=/usr/
lib/systemd/systemd-cgroups-agent /system.slice/systemd-hostnamed.service, pid=6292, tid=6292, ppid=2(kthreadd), arg
s=(family=1, type=524290, protocol=0), fds=[0(/dev/null), 1(/dev/null), 2(/dev/null), 3()]
```

插图 2: 自定义格式输出

1.3 整体架构

1.3.1 总体框架

该实现主要利用 Falco 插件机制实现，所以整体是以插件为主，然后向外扩展出许多功能，包括 ebpf 加载、事件丰富等功能。

具体框架如下所示:

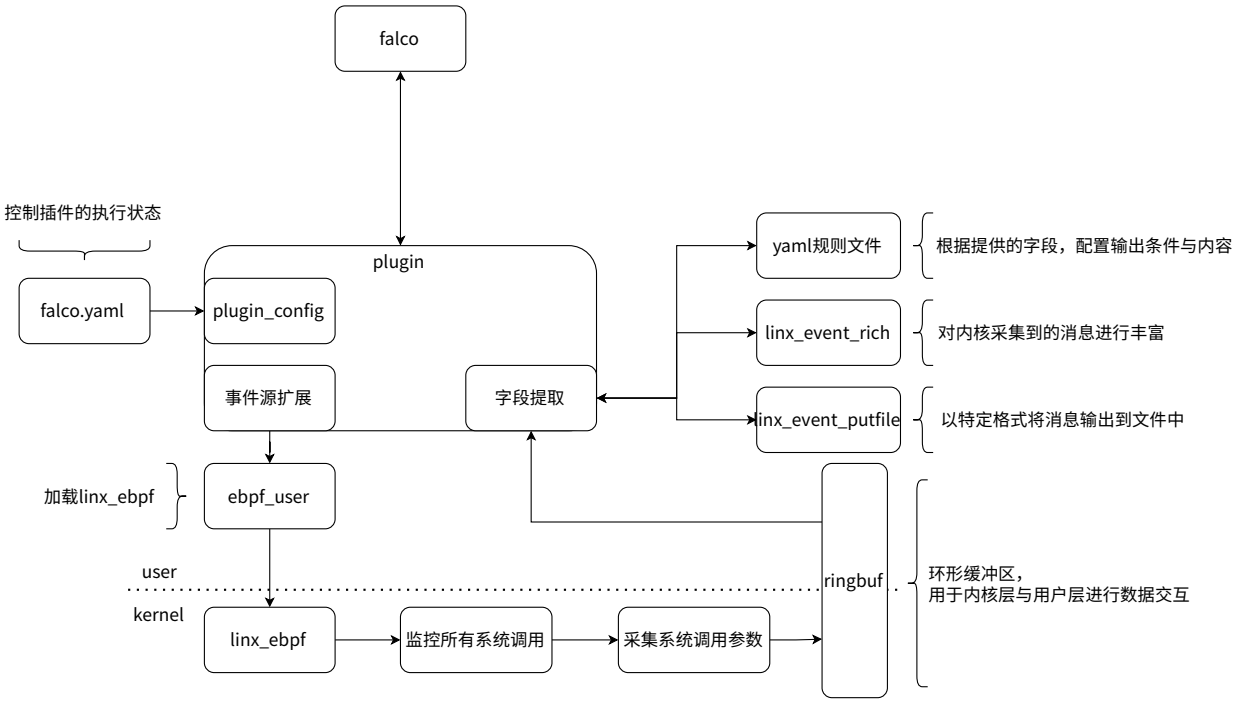

插图 3: 插件+linx_ebpf 总体框架

1.3.2 关键技术

1.3.2.1 插件 API

Falco 插件定义了使用插件需要提供哪些 API，并且具备不同能力的插件所需要的 API 也不相同。在 libs/userspace/plugin/plugin_api.h 文件中定义了所需 API 函数的签名。

下面罗列插件所必备的 API，只有将这些函数都实现了，Falco 才能正常加载该插件:

表格 1: 插件必备 API

序号	函数定义	描述
1	const char* (*get_required_api_version)();	返回该插件所使用的插件 API 版本
2	ss_plugin_t* (*init)(const ss_plugin_init_input* input, ss_plugin_rc*	初始化插件并分配其状态

	rc);	
3	void (*destroy)(ss_plugin_t* s);	销毁插件
4	const char* (*get_last_error)(ss_plugin_t* s);	返回插件最后生成的错误信息
5	const char* (*get_name)();	返回插件名称
6	const char* (*get_description)();	返回插件描述
7	const char* (*get_contact)();	返回插件作者联系信息
8	const char* (*get_version)();	返回插件本身版本

实现完上述 API 后，插件只是能正常加载与卸载，并没有具备相应的能力，扩展事件源需要实现以下 API：

表格 2: 扩展事件源插件必备 API

序号	函数定义	描述
1	uint32_t (*get_id)();	返回插件唯一 ID
2	const char* (*get_event_source)();	返回一个表示该插件生成的事件源名称的字符串
3	ss_instance_t* (*open)(ss_plugin_t* s, const char* params, ss_plugin_rc* rc);	打开事件源并开始捕获
4	void (*close)(ss_plugin_t* s, ss_instance_t* h);	关闭一个捕获
5	ss_plugin_rc (*next_batch)(ss_plugin_t* s, ss_instance_t* h, uint32_t* nevts, ss_plugin_event*** evts);	返回下一个事件

字段提取需要实现以下 API：

表格 3: 字段提取插件必备 API

序号	函数定义	描述
1	const char* (*get_fields)();	返回该插件导出的提取器字段列表，可用于 Falco 规则中。
2	ss_plugin_rc (*extract_fields)(ss_plugin_t* s, const ss_plugin_event_input* evt, const	从事件中提取一个或多个过滤字段值

	ss_plugin_field_extract_input* in);	

1.3.2.2 规则字段

当前插件提供了众多规则字段，在使用这些字段时都需要加上 linx.前缀。可用字段描述如下:

表格 4: 插件可用的规则字段

序号	名称	是否可判断	参数	输出示例	描述
1	type	使用 in 进行子集判断	可以不跟，也可从后面选一个: [key|value]	%linx.type: (3(close)) %linx.type[key]: (3) %linx.type[value]: (close)	触发事件的系统调用
2	user	使用 in 进行子集判断	可以不跟，也可从后面选一个: [key|value]	%linx.user: (0(root)) %linx.user[key]: (0) %linx.user[value]: (root)	触发事件的用户
3	group	使用 in 进行子集判断	可以不跟，也可从后面选一个: [key|value]	%linx.group: (0(root)) %linx.group[key]: (0) %linx.group[value]: (root)	触发事件的组
4	fds	使用 in 进行子集判断	可以不跟，也可从后面选一个: [key|value]	%linx.fds: (0(/dev/null), 1(/dev/null)) %linx.fds[key]: (0, 1) %linx.fds[value]: (/dev/null, /dev/null)	当前进程打开的所有文件
5	args	不能	不能	不同的系统调用输出不同，这里以 close 为例 %linx.args: (fd=43)	当前系统调用的所有参数
6	time	不能	不能	%linx.time: 2025-06-12 09:43:49.410583558	触发事件的时间
7	dir	使用 in 进行子集判断	不能	%linx.dir: (<)	标识进入系统调用(>)还是退出系统调用(<)
8	comm	使用 in 进行	不能	%linx.comm: (in:imjournal)	触发事件的命令

		子集判断			
9	pid	使用 in 进行子集判断	不能	%linx.pid: (814)	触发事件任务的 pid
10	tid	使用 in 进行子集判断	不能	%linx.tid: (814)	触发事件任务的 tid
11	ppid	使用 in 进行子集判断	不能	%linx.ppid: (1)	触发事件任务的 ppid
12	cmdline	使用 in 进行子集判断	不能	%linx.cmdline: (/usr/sbin/rsyslogd -n -i/var/run/rsyslogd.pid)	触发事件所执行的命令
13	fullpath	使用 in 进行子集判断	不能	%linx.fullpath: (/usr/sbin/rsyslogd)	命令的绝对路径

1.4 配置选项

当前存在三个不同的 JSON 字符串用于控制插件与 ebpf 内核程序的执行状态。

1.4.1 init_config

该 JSON 字符串位于 build/falco.yaml 文件中，主要用于控制插件本身的执行逻辑。可用字段及相关描述如下:

表格 5: init_config 可用字段

序号	名称	取值类型	取值范围	描述
1	rich_value_size	整数	0 ~ 2^32 - 1	控制丰富事件 buffer 的最大大小
2	str_max_size	整数	0 ~ 2^32 - 1	控制输出系统调用参数时，字符串的最大长度，当字符串超出设定值时，用...标识
3	putfile	字符串	true、false	控制是否按照特定消息格式输出到文件中
4	path	字符串	文件路径，最长 256	存放消息的文件路径
5	format	字符串	normal、json(目前未支持)	输出格式
6	log_level	字符串	DEBUG、INFO、WARNING、ERROR、FATAL	控制日志的输出等级

7	log_file	字符串	文件路径，最长 256，为空时自动输出到 stderr	存放日志的文件

1.4.2 open_params

该 JSON 字符串位于 build/falco.yaml 文件中，主要用于控制 ebpf 内核层的执行逻辑。可用字段及相关描述如下：

表格 6: open_params 可用字段

序号	名称	取值类型	取值范围	描述
1	filter_own	字符串	true、false	在 ebpf 内核采集时，是否过滤掉插件本身触发的系统调用
2	filter_falco	字符串	true、false	在 ebpf 内核采集时，是否过滤掉 falco 本身触发的系统调用
3	filter_pids	整数数组	最多过滤 64 个 pid，每个 pid 取值在 0 ~ 2 ^ 22 - 1 之间	在 ebpf 内核采集时，过滤哪些 PID
4	filter_comms	字符串数组	最多过滤 16 个命令，每个命令最多 16 个字符长度	在 ebpf 内核采集时，过滤哪些命令
5	drop_mode	字符串	true、false	在 ebpf 内核采集时，是否放弃采集所有系统调用
6	drop_failed	字符串	true、false	在 ebpf 内核采集时，是否放弃采集失败的系统调用
7	interest_syscall_file	字符串	文件路径，最长 256	指定特定 json 格式文件，文件内标识要采集哪些系统调用

1.4.3 json_config/interesting_syscalls.json

该文件主要描述了哪些系统调用需要在 ebpf 内核中进行参数采集，具体格式如下：

```
{
 "read": {
  "syscall_id": 0,
  "interesting": 0
```

```
},
"write": {
 "syscall_id": 1,
 "interesting": 1
},
"open": {
 "syscall_id": 2,
 "interesting": 0
},
 ……
}
```

在该文件中只需要关注 interesting 字段，其为 1 就表示需要采集该系统调用，为 0 则代表不采集该系统调用。

1.5 现存问题

1.5.1 scripts/get_syscalls_macro.sh 脚本问题

1.5.1.1 问题描述

目前脚本是读取/sys/kernel/debug/tracing/events/syscalls/sys_enter_$syscall/format 文件来获取系统调用的参数类型以及参数名称数据的，但目前获取的系统调用参数不是很齐全，导致没有以下系统调用的参数:

- uname、uselib、_sysctl、get_kernel_syms、query_module、nfsservctl、getpmsg、putpmsg、afs_syscall、tuxcall、security、set_thread_area、get_thread_area、lookup_dcookie、vserver、landlock_create_ruleset、landlock_add_rule、landlock_restrict_self

1.5.1.2 解决方案

后续思考是从其他内核文件中获取系统调用的参数数据或者手动添加。

1.5.2 eBPF 内核参数提取问题

1.5.2.1 问题描述

当前在 ebpf 内核中只将 char *字符串指针类型的实际字符串提出了出来，其余所有的指针类型都只提取了其地址，并没有提取其值。主要问题还是在于结构体指针的提取，不同结构体内部成员不同，并且可能存在成员又是指针的情况。

并且当前设计只在系统调用退出时才采集系统调用的参数。

1.5.2.2 解决方案

针对某些带指针入参的系统调用，选择性的只提取其中关键性的数据。

1.5.3 系统调用采集 ebpf 代码问题

1.5.3.1 问题描述

由于所有 ebpf 代码都是通过脚本生成的，并且数量众多，当前并没有进行完备的测试，某些系统调用的参数提取逻辑可能还存在问题。

1.5.3.2 解决方案

在后续开发中进行提取逻辑与提取正确性的验证。