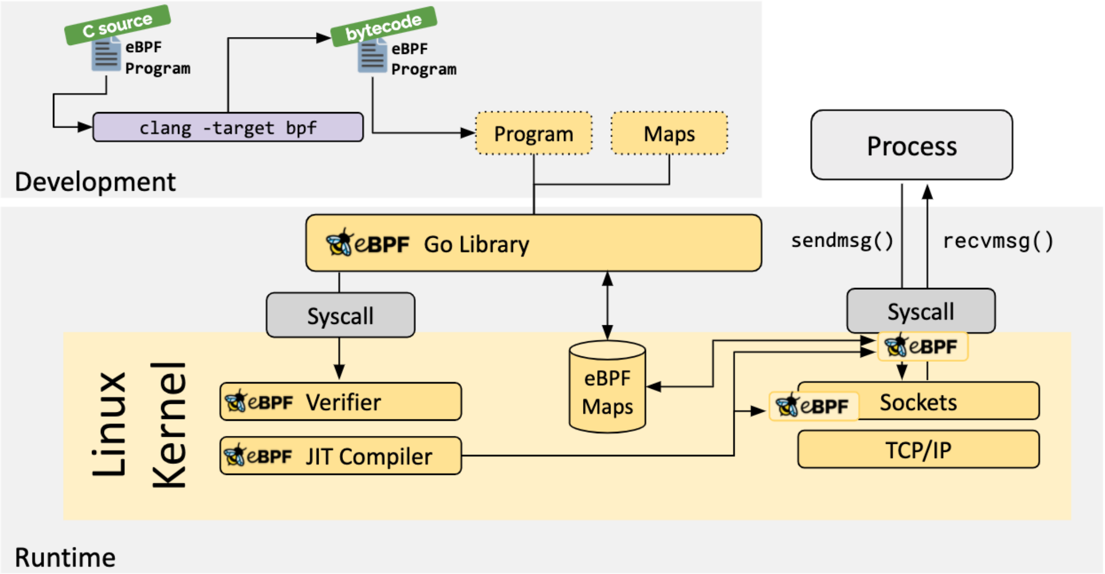
1. BPF 的前世今生

bpf 全称伯克利包过滤器（Berkeley Packet Filte），bpf 技术诞生于 1992 年，早期主要用来提升对数据包的过滤性能，但是早期的 bpf 提供的指令较少，限制了它的应用范围。本文要介绍的 ebpf 是 bpf 的扩展版本，相比早期版本的 bpf 功能变得更加强大，自 2014 年引入内核以来，BPF 现在已经成发展成内核中一个通用的引擎，通过相关 API 我们可以方便的读取到内核态的内存内容，也能够通过 BPF 改写运行时内存，具有强大的编程能力。毫不夸张的说，BPF 技术就是内核中的脚本语言。

2. BPF 的应用场景

BPF 的应用场景非常广泛，总结下来主要有下面几大领域。
性能分析： BPF 提供了对内核和应用程序极高的观察能力，通过编程可以实现比较丰富的统计功能，极高性能，能够避免对观测程序产生影响。
提高程序的观测性： 通过 BPF 技术，可以在函数调用的不同阶段进行插桩监控，能够观测运行时的程序内存，
通过提高程序的观测性，方便我们定位低概率复现的 bug。
安全 ： bpf 技术能够观测系统系统调用，感知哪些进程在运行，哪些文件被改写，面对 0day 漏洞，也能够快速的编写出对应的检测程序。
除了上面列出的几个大的方向。K8s 已经通过 BPF 技术来提高容器的网络安全性能和丰富的监控排错能力。甚至提供了基于 Cilium 的负载均衡能力。（k8s 中用的是 Cilium 框架）

3. BPF 的基本原理



BPF 程序首先由 BPF 验证器校验通过，再由编译器编译成特定的字节码。最后由运行在内核中的 bpf 虚拟机进行解释执行。内核也提供了一些 helper 函数，这里有详细的文档：

https://man7.org/linux/man-pages/man7/bpf-helpers.7.html
maps 是 eBPF 提供的存储结构。我们可以使用 maps 进行统计，存储一些自定义变量，然后可以在用户态通过 API 读取这些数据。
BPF 程序运行在沙箱当中，并且由验证器进行安全校验，可以防止 bpf 程序对系统造成破坏。如果在 bpf 程序中写了死循环，或者有数组越界的逻辑，验证器会直接报失败，防止程序对内核造成伤害。

4. 开始你的第一个 BPF 程序

BPF 程序的编译，一般使用 LLVM.LLVM 是一种 genreal-purpose 的编译器，LLVM 可以 emit 不同的字节码。在本章中，LLVM 将生成 bpf 的字节码，然后我们会 load 到内核的虚拟中。
内核提供了一个系统调用，专门用于 load bpf 的程序，除了 load bpf 的程序，这个系统调用，还可以有一些其他的操作，后面我们会看到它的用法，接下来，我们来看下 Hello world:

```
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

static int (*bpf_trace_printk)(const char *fmt, int fmt_size, ...) =
(void *)BPF_FUNC_trace_printk;

SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(void *ctx) {
    char msg[] = "Hello, BPF World!";
      bpf_trace_printk(msg, sizeof(msg));
        return 0;
}

char _license[] SEC("license") = "GPL";
```

编译命令:

```
$ clang -O2 -target bpf -c bpf_program.c  -I /usr/include/x86_64-linux-gnu/ -o bpf_program.o
```

我们使用 SEC 属性，告诉 BPF VM,我们想在什么时候运行我们写的这个程序。在上面的代码中，我们指定在 kernel 调用到 execve 的时候，来调用我们自己写的程序。即：SEC 中定义的是一个 Tracepoints,是 kernel 预先定义好的，允许开发者，在这里 injet 进去自己的代码。那你可能会问，我怎么知道都有哪些 tracepoints 呢？这个可以在 /sys/kernel/debug/tracing/events/syscalls/这个目录下找到系统预留的所有的 tracepoints.
另外，我们需要使用指定 GPL 的协议。因为 kernel 本身就是 GPL 的。我们使用 bpf_trace_printk 来打印在内核中生成的日志。当然你也可以通过 /sys/kernel/debug/tracing/trace_pipe 来查看内核的日志。

我们需要一份源码来编译 libbpf

```
cd /tmp
git clone --depth 1 git://kernel.ubuntu.com/ubuntu/ubuntu-bionic.git
```

可以将源码拷贝到/kernel-src 目录下，然后编译 libbpf

```
sudo mv ubuntu-bionic /kernel-src
cd /kernel-src/tools/lib/bpf
sudo make && sudo make install prefix=/usr/local
```

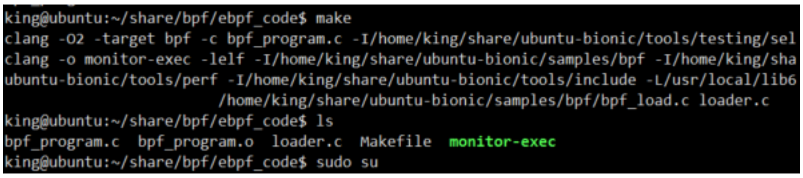
现在我们有了 bpf 的代码，需要一个程序把它 load 到内核中。

```
#include "bpf_load.h"
#include <stdio.h>

int main(int argc, char **argv) {
      if (load_bpf_file("bpf_program.o") != 0) {
              printf("The kernel didn't load the BPF program\n");
                 return -1;
                }

        read_trace_pipe();

          return 0;
}
```

Makefile

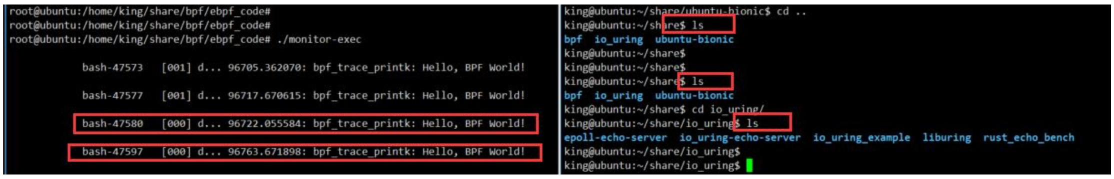
```
CLANG = clang

EXECABLE = monitor-exec

BPFCODE = bpf_program
BPFTOOLS = /home/king/share/ubuntu-bionic/samples/bpf
BPFLOADER = $(BPFTOOLS)/bpf_load.c

CCINCLUDE += -I/home/king/share/ubuntu-bionic/tools/testing/selftests/bpf

LOADINCLUDE += -I/home/king/share/ubuntu-bionic/samples/bpf
LOADINCLUDE += -I/home/king/share/ubuntu-bionic/tools/lib
LOADINCLUDE += -I/home/king/share/ubuntu-bionic/tools/perf
LOADINCLUDE += -I/home/king/share/ubuntu-bionic/tools/include
LIBRARY_PATH = -L/usr/local/lib64
BPFSO = -lbpf

.PHONY: clean $(CLANG) bpfload build

clean:
           rm -f *.o *.so $(EXECABLE)

build: ${BPFCODE.c} ${BPFLOADER}
           $(CLANG) -O2 -target bpf -c $(BPFCODE:=.c) $(CCINCLUDE) -o
${BPFCODE:=.o}

bpfload: build
           clang -o $(EXECABLE) -lelf $(LOADINCLUDE) $(LIBRARY_PATH)
$(BPFSO) \
                   $(BPFLOADER) loader.c

$(EXECABLE): bpfload

.DEFAULT_GOAL := $(EXECABLE)
```

```
$ make
```



```
king@ubuntu:~/share/bpf/ebpf_code$ make
clang -O2 -target bpf -c bpf_program.c -I/home/king/share/ubuntu-bionic/tools/testing/sel
clang -o monitor-exec -lelf -I/home/king/share/ubuntu-bionic/samples/bpf -I/home/king/sha
ubuntu-bionic/tools/perf -I/home/king/share/ubuntu-bionic/tools/include -L/usr/local/lib6
                  /home/king/share/ubuntu-bionic/samples/bpf/bpf_load.c loader.c
king@ubuntu:~/share/bpf/ebpf_code$ ls
bpf_program.c  bpf_program.o  loader.c  Makefile  monitor-exec
king@ubuntu:~/share/bpf/ebpf_code$ sudo su
```

每执行一个 ls 就会有一行打印信息



```
root@ubuntu:/home/king/share/bpf/ebpf_code#
root@ubuntu:/home/king/share/bpf/ebpf_code#
root@ubuntu:/home/king/share/bpf/ebpf_code# ./monitor-exec

           bash-47573   [001] d... 96705.362070: bpf_trace_printk: Hello, BPF World!

           bash-47577   [001] d... 96717.670615: bpf_trace_printk: Hello, BPF World!

           bash-47580   [000] d... 96722.055584: bpf_trace_printk: Hello, BPF World!

           bash-47597   [000] d... 96763.671898: bpf_trace_printk: Hello, BPF World!
```

```
king@ubuntu:~/share/ubuntu-bionic$ cd ..
king@ubuntu:~/share$ ls
bpf  io_uring  ubuntu-bionic
king@ubuntu:~/share$
king@ubuntu:~/share$
king@ubuntu:~/share$ ls
bpf  io_uring  ubuntu-bionic
king@ubuntu:~/share$ cd io_uring/
king@ubuntu:~/share/io_uring$ ls
epoll-echo-server  io_uring-echo-server  io_uring_example  liburing  rust_echo_bench
king@ubuntu:~/share/io_uring$
king@ubuntu:~/share/io_uring$
```