



第二届 eBPF开发者大会

www.ebpftravel.com

基于libbpf库的eBPF编程方案

主讲人：闻茂泉

2024-04-13

中国·西安



第二届 eBPF 开发者大会

www.ebpftravel.com

Content 目录

01

eBPF 编程方案简介

02

基于 libbpf 的编程方案

03

改进的 libbpf 编程方案

04

低版本内核编程方案

05

低版本编译环境编程方案



第二届 eBPF开发者大会

www.ebpftravel.com

01

eBPF编程方案简介

中国·西安

主流的C语言实现的eBPF编程方案

代际	方案指称	识别方法	备注
第1代	bpf_load.c文件方案	代码中有bpf_load.c文件，还有load_bpf_file函数。	Linux 4.x 系列早期内核版本的源码实例大多基于此文件，这个旧 API 方案已经在内核中被逐步废弃。
第2代	原生libbpf库方案	代码中有libbpf.c文件	Linux 5.x版本内核的源码实例很多使用以libbpf.c为核心的原生libbpf库方案，是本文重点阐述的方案。
第3代	libbpf-bootstrap骨架方案	代码中除了libbpf.c文件，还有libbpf-bootstrap、skeleton和*.skel.h关键词	最新版本内核的源码实例已经开始采用此方案。业界最新的eBPF介绍文章较多基于此方案。

原生libbpf库eBPF编程方案的一些独特优势：

- 1. 更深的控制和灵活性：**直接使用原生libbpf 库的方案意味着可以与更底层交互，实现更多的控制，定制加载和管理 eBPF 程序和 maps 过程，满足更复杂的需求。
- 2. 更好的学习和理解：**libbpf-bootstrap封装抽象屏蔽了很多细节，直接使用原生libbpf 可以对 eBPF 子系统有更深入的理解，有利于开发者对 eBPF 内部工作原理的理解。
- 3. 更细粒度的依赖管理：**直接使用原生libbpf库能够指定依赖的 libbpf 库版本和功能，进而更精细化地管理项目依赖关系。
- 4. 更好的低版本内核适应性：**基于原生libbpf库的方案，在低版本操作系统发行版和低版本内核上可以有更好的兼容性。

```
execve_bpf__open();  
execve_bpf__load();  
execve_bpf__attach();
```

最大硬伤：写死

C语言eBPF编程的基础环境准备

主流的linux发行版大多是基于rpm包或deb包的包管理系统。
不同的包管理系统，初始化eBPF开发环境时所依赖的包，也略有差别。

	rpm包基础环境初始化	deb包基础环境初始化
推荐发行版	Anolis 8.8、CentOS 8.5、Kylin V10、Fedora 33及以上	Debian 12、Ubuntu 21.04及以上
编译工具和依赖库包	<code>yum install clang llvm elfutils-libelf-devel</code>	<code>apt-get update</code> <code>apt install clang llvm libelf-dev</code>
基础包	<code>yum install git make</code>	<code>apt install git make</code>
用户态头文件	<code>yum install kernel-headers-\$(uname -r)</code>	<code>apt install linux-libc-dev</code>
bpftool工具	<code>yum install bpftool-\$(uname -r)</code>	<code>apt install linux-tools-common linux-tools-\$(uname -r)</code>

C语言eBPF编程方案的程序架构

第1代, 第2代编程方案

编译过程

execve.c
↓
execve

execve.bpf.c (execve_kern.c)
↓
execve.bpf.o (execve_kern.o)

第3代编程方案

execve.bpf.c
↓
execve.bpf.o
↓
execve.skel.h
↙ ↘
execve

执行过程

./execve

实时读取

← execve.bpf.o (execve_kern.o)

./execve



第二届 eBPF开发者大会

www.ebpftravel.com

02

基于libbpf的编程方案

中国·西安

Talk is cheap. Show me the code.

```
1  $ cd ~
2  $ git clone https://github.com/alibaba/sreworks-ext.git
3  $ cd ~/sreworks-ext/demos/native_libbpf_guide/trace_execve_libbpf130
4  $ make
5  $ sudo ./trace_execve
6  trace_execve 15836221 5501 bash 1534 bash 0 /usr/bin/ls
7  trace_execve 15914126 5502 bash 1534 bash 0 /usr/bin/ps
```

- **trace_execve_libbpf130**是一个基于libbpf库的第2代eBPF构建实例。
- eBPF初学者, 可以考虑选择跟踪 **execve** 系统调用产生的事件。
- 执行编译结果**trace_execve**命令, 完美验证通过。

eBPF项目的目录结构解析

项目目录	说明
./	项目用户态代码和主Makefile
./progs	项目内核态bpf程序代码
./include	项目的业务代码相关的头文件
./helpers	来自于libbpf库之外的helper文件
./tools/lib/bpf/	除Makefile外，来自于libbpf-1.3.0/src/
./tools/include/	所有都来自于libbpf-1.3.0/include/
./tools/build/	项目构建时一些feature探测代码
./tools/scripts/	项目Makefile所依赖的一些功能函数

```
1 $ find . -name "trace_execve*"
2  ./trace_execve.c
3  ./progs/trace_execve.bpf.c
4  ./include/trace_execve.h
```

在这个项目中添加ebpf的代码，可以遵循这样的目录结构。用户态加载文件放到根目录下，内核态bpf文件放到progs目录下，用户态和内核态公共的头文件放到include目录下。

eBPF项目的Makefile解析

trace_execve_libbpf130项目有4个Makefile，分别如下：

- 1) ./Makefile是主文件，用于生成用户态eBPF程序trace_execve。
- 2) ./progs/Makefile 用于生成内核态BPF程序trace_execve.bpf.o。
- 3) ./tools/lib/bpf/Makefile 用于生成libbpf.a静态库。
- 4) ./tools/build/feature/Makefile 用于一些feature的探测。

在项目空间的根目录运行make命令进行项目构建时，会首先执行Makefile文件。在Makefile文件中会通过make的-C选项间接触发progs/Makefile和tools/lib/bpf/Makefile的执行。

```
1  $ find . -name Makefile
2  ./Makefile
3  ./progs/Makefile
4  ./tools/lib/bpf/Makefile
5  ./tools/build/feature/Makefile
```

内核态bpf程序编译参数解析

```
1 clang -c trace_execve.bpf.c -o trace_execve.bpf.o \  
2   -iquote ../../include/ -iquote ../../helpers -I../../tools/lib/ -I../../tools/include/uapi \  
3   -D__KERNEL__ -D__BPF_TRACING__ -D__TARGET_ARCH_x86 -g -O2 -mlittle-endian -target bpf -mcpu=v3
```

```
1 $ cat progs/trace_execve.bpf.c  
2 #include <vmlinux.h>  
3 // #include <linux/bpf.h>  
4  
5 #include <bpf/bpf_helpers.h>  
6 #include <bpf/bpf_tracing.h>  
7  
8 #include "common.h"  
9 #include "trace_execve.h"
```

编译参数	参数解析
-I../../tools/lib/	由bpftool工具编译生成的vmlinux.h文件，包含了绝大多数bpf程序的内核态和用户态依赖，通过此选项可在./tools/lib/目录搜索到vmlinux.h头文件。 还可通过此选项在./tools/lib/目录下的bpf子目录中查找到bpf_helpers.h和bpf_tracing.h头文件，它们都是对vmlinux.h头文件内核态依赖的补充。
-I../../tools/include/uapi/	通过此选项，可在./tools/include/uapi/目录下的linux子目录中查找到bpf.h头文件。同时kernel-headers包引入的/usr/include/linux/目录下也有bpf.h，./tools/include/uapi下的bpf.h优先级会覆盖它。在一些简单使用场景，可以使用<linux/bpf.h>替代<vmlinux.h>。
-idirafter /usr/include/x86_64-linux-gnu	clang -target bpf时，标准系统目录中不包含/usr/include/x86_64-linux-gnu路径。
-iquote ../../include/	通过此编译选项，可以在./include/目录中查找到trace_execve.h和common.h头文件。
-D__KERNEL__ -D__BPF_TRACING__ -D__TARGET_ARCH_x86	以上头文件依赖的预处理过程中，会依赖宏变量来决定预处理的展开逻辑，此处编译命令中的宏就是起这些作用。比如在bpf_tracing.h头文件中，就有#if defined(__TARGET_ARCH_x86)的宏判断语句，来决定预处理展开逻辑走x86分支。
-g	用于生成execve.bpf.o目标文件中的.BTF和.BTF.ext节。（与低版本环境区别）
-target bpf	指示Clang将代码生成为针对eBPF目标的目标代码。（与低版本环境区别）

用户态加载程序编译参数解析

```
gcc -iquote ./include/ -I./tools/lib/ -I./tools/include/ -I./tools/include/uapi -c -o trace_execve.o trace_execve.c
```

```
1 $ cat trace_execve.c
2 // from kernel-headers
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <linux/limits.h>
6 #include <linux/perf_event.h>
7
8 // from libbpf
9 #include <linux/ring_buffer.h>
10 #include <bpf/libbpf.h>
11
12 #include "common.h"
13 #include "trace_execve.h"
```

```
1 $ gcc -xc /dev/null -Wp,-v
2 /usr/lib/gcc/x86_64-redhat-linux/8/include
3 /usr/local/include
4 /usr/include
```

编译参数	参数解析
-iquote ./include/	通过此选项，可在./include/目录中查找到trace_execve.h和common.h头文件。
-I./tools/lib/	通过此选项，可在./tools/lib/目录下的bpf子目录中查找到<bpf/libbpf.h>头文件。一些老代码中，有<libbpf.h>头文件使用用法，最新的ebpf项目实例，都会将libbpf库的libbpf.h等头文件放到bpf子目录下，推荐统一使用<bpf/libbpf.h>用法。
-I./tools/include/	通过此选项，可在./tools/include/目录下的linux子目录中查找到头文件<linux/ring_buffer.h>
-I./tools/include/uapi	同样是对kernel-headers包引入的/usr/include/linux/目录下头文件的更新升级。
标准系统目录 (Standard system directories)	除以上头文件外的其他头文件，均可以在kerne-headers包提供的标准系统目录（Standard system directories）的/usr/include/目录及子目录中查找到。因此，<linux/perf_event.h>最终会在/usr/include/linux/perf_event.h位置被查找到。这里同样是<linux/xxx.h>形式头文件，<linux/perf_event.h>和<linux/ring_buffer.h>却在两个完全不同的搜索路径查被查找到。一个系统上的标准系统目录如图示。

libbpf.a静态库编译参数解析

```
1 gcc -I. -I./tools/include -I./tools/include/uapi -o libbpf.o -c libbpf.c
2 ld -r -o libbpf-in.o btf.o ringbuf.o libbpf.o bpf.o .....
3 ar rcs libbpf.a libbpf-in.o
4
5 gcc -o trace_execve trace_execve.o ./tools/lib/bpf/libbpf.a -lelf
```

本项目针对./tools/lib/bpf目录中的*.c的libbpf库文件，进行了静态编译，最终生成libbpf.a静态库文件。之后把libbpf.a静态库文件编译进了所有ebpf应用程序中。

在本项目中，完全实现了libbpf库的自主可控编译构建过程。这给我们带来如下两方面好处：

- 1) 对于一些ebpf的资深人士，可以自主修改libbpf库，实现满足自己业务需求的优化。
- 2) 对于一些ebpf的初学者，完全可以在libbpf库中任意位置插入printf，学习libbpf库的原理。



第二届 eBPF开发者大会

www.ebpftravel.com

03

改进的libbpf编程方案

咱们中国人的原创

中国·西安

传统的第2代eBPF编程方案美中不足

```
1 $ cd ~/sreworks-ext/demos/native_libbpf_guide/trace_execve_libbpf130
2 $ make
3 $ rm -f progs/trace_execve.bpf.o
4 $ sudo ./trace_execve
5 libbpf: elf: failed to open progs/trace_execve.bpf.o: No such file or directory
6 ERROR: failed to open prog: 'No such file or directory'
```

- 从实验结果可以看出，当我们把bpf目标文件trace_execve.bpf.o删除后，trace_execve程序执行会报错，提示读取trace_execve.bpf.o文件不存在。
- 这说明，当前方案构建后，需要将trace_execve程序和bpf目标文件trace_execve.bpf.o这一组文件一起进行分发，才能正常执行。这给我们在工程的实现上带来了很大的挑战。
- 为了解决上面提到的问题，第3代ebpf编程方案libbpf-bootstrap框架发明了skeleton骨架，即使用*.skel.h头文件的方式，将bpf目标文件trace_execve.bpf.o的内容编译进trace_execve程序。这样后续只需分发trace_execve二进制程序文件即可。

关键突破：使用hexdump生成skel.h头文件

步骤	libbpf-bootstrap框架构建方法	可改进机会点
1	bpftool btf dump file vmlinux format c > vmlinux.h	
2	clang -O2 -target bpf -c trace_execve.bpf.c -g -o trace_execve.bpf.o	
3	bpftool gen skeleton trace_execve.bpf.o > trace_execve.skel.h	此步骤用hexdump替换bpftool
4	gcc -o trace_execve trace_execve.c -lbpf -lelf	此步骤更改加载函数为libbpf标准函数

```
$ hexdump -v -e '"\\x" 1/1 "%02x"' trace_execve.bpf.o > trace_execve.hex
```

- libbpf-bootstrap编程框架的第3步依靠bpftool工具将trace_execve.bpf.o这个目标文件转换成十六进制格式的文本，并将文本内容作为trace_execve.skel.h头文件中的一个变量的值，最后还需要让trace_execve.c用户态加载文件包含这个trace_execve.skel.h头文件。**此步骤，我们可以用hexdump命令将bpf目标文件转换成十六进制文本并继续生成skel.h头文件。**
- 有了使用hexdump命令成功头文件的基础，我们也可以使用libbpf库中的原生库函数完成bpf程序的用户态加载过程。原来加载函数trace_execve_bpf_load()必须包含程序的关键词trace_execve，但现在可以使用纯净的bpf_object_load()原生库函数完成同样的任务了。

Talk is cheap. Show me the code once again.

```
1 $ cd ~/sreworks-ext/demos/native_libbpf_guide/hexdump_skel_libbpf130
2 $ make
3 $ rm -fr progs/trace_execve.bpf.o
4 $ sudo ./trace_execve
5 trace_execve bash su 74113 74112 0 /usr/bin/bash
6 trace_execve bash su 74113 74112 0 /usr/bin/bash
```

- 从运行结果看，虽然删除了bpf目标文件trace_execve.bpf.o，仅仅依靠trace_execve文件即可成功执行。
- 也可以再尝试将trace_execve可执行文件拷贝到其他目录，结果依然可行。

改进的eBPF项目Makefile解析

```
1 SOURCES := $(wildcard *.c)
2 LOADER_OBJECT := $(patsubst %.c, %, $(SOURCES))
3 USER_OBJECT := $(patsubst %.c, %.o, $(SOURCES))
4 SKEL_OBJECT := $(patsubst %.c, %.skel.h, $(SOURCES))
5 HEX_OBJECT := $(patsubst %.c, %.hex, $(SOURCES))
6 BPF_OBJECT := $(patsubst %.c, ./progs/%.bpf.o, $(SOURCES))
```


```
1 $(BPF_OBJECT): ./progs/%.bpf.o : ./progs/%.bpf.c
2 $(HEX_OBJECT): %.hex : ./progs/%.bpf.o
3 $(SKEL_OBJECT): %.skel.h : %.hex
4 $(USER_OBJECT): %.o : %.c %.skel.h
5 $(LOADER_OBJECT): % : %.o
```

- 主Makefile中，为了实现目标依赖，我们连用了5个静态模式规则（Static Pattern Rules）。
- 任何一个静态模式规则的目标集合，都是通过项目根目录下*.c文件的集合，进行局部字符串替换获得。
- 用户态可执行加载程序的主要依赖链条如图。

```
1 trace_execve
2 |─ trace_execve.o
3 |   |─ trace_execve.c
4 |   |─ trace_execve.skel.h
5 |   |   |─ trace_execve.hex
6 |   |   |   |─ progs/trace_execve.bpf.o
7 |   |   |   |   |─ progs/trace_execve.bpf.c
```

从file到memory实现读取elf的转变

```
1 // 传统方式加载bpf目标文件
2 char filename[256] = "progs/trace_execve.bpf.o";
3 struct bpf_object * bpf_obj = bpf_object__open_file(filename, NULL);
```



```
1 // 改进方式加载bpf目标文件
2 #include "skeleton.skel.h"
3 struct bpf_object * bpf_obj = bpf_object__open_mem(obj_buf, obj_buf_sz, NULL);
```

```
1 static int bpf_object__elf_init(struct bpf_object *obj){
2     if (obj->efile.obj_buf_sz > 0) {
3         elf = elf_memory((char *)obj->efile.obj_buf, obj->efile.obj_buf_sz);
4     } else {
5         obj->efile.fd = open(obj->path, O_RDONLY | O_CLOEXEC);
6         elf = elf_begin(obj->efile.fd, ELF_C_READ_MMAP, NULL);
7     }
8 }
```

libbpf库提供了**bpf_object__open_file**和**bpf_object__open_mem**两个函数用于读取elf格式的bpf目标文件**trace_execve.bpf.o**。其中**bpf_object__open_file**是在**trace_execve**运行时，去读取**trace_execve.bpf.o**文件内容，而**bpf_object__open_mem**是在编译时，已经把elf内容编译进**trace_execve**程序。

这两个libbpf库函数，最终都是调用libbpf.c文件中的函数**bpf_object__elf_init**。在这个函数中，读取文件的场景会触发elf标准库函数**elf_begin**，而读取内存的场景会触发elf标准库函数**elf_memory**。

改进libbpf编程方案与行业主流方案比较

相比较第3代的 libbpf-bootstrap框架方案和第2代的传统libbpf库方案，使用hexdump命令的原生libbpf库第 2 代改进方案方案在实现方法上，有一些独特的优势。

比较项	第2代传统libbpf库方案	第3代libbpf-bootstrap方案	第2代hexdump的libbpf库改进方案
获取bpf.c的头文件	bpftool btf dump file /sys/kernel/btf/vmlinux format c >vmlinux.h	bpftool btf dump file /sys/kernel/btf/vmlinux format c >vmlinux.h	bpftool btf dump file /sys/kernel/btf/vmlinux format c >vmlinux.h
编译bpf.o目标文件	clang -O2 -target bpf -c trace_execve.bpf.c -g -o trace_execve.bpf.o	clang -O2 -target bpf -c trace_execve.bpf.c -g -o trace_execve.bpf.o	clang -O2 -target bpf -c trace_execve.bpf.c -g -o trace_execve.bpf.o
生成skel.h头文件	无	bpftool gen skeleton trace_execve.bpf.o > trace_execve.skel.h	hexdump
使用skel.h头文件	无	将程序名trace_execve添加到头文件名称中 trace_execve.skel.h	统一成一个固定的名称skeleton.skel.h
用户态加载函数	使用libbpf库标准加载函数 bpf_object__open_file(); bpf_object__load(); bpf_program__attach();	将程序名添加到加载函数名称中 trace_execve_bpf__open(); trace_execve_bpf__load(); trace_execve_bpf__attach();	使用libbpf库标准加载函数 bpf_object__open_mem(); bpf_object__load(); bpf_program__attach();

trace_execve_bpf__open()函数的实现，也是间接通过libbpf库的bpf_object__open_skeleton()函数，最终也调用了bpf_object__open_mem()函数。

使用attach_tracepoint替代attach

	trace_execve.c中相关代码	trace_execve.bpf.c中相关代码
attach方案A	bpf_program__attach(bpf_prog)	SEC("tracepoint/syscalls/sys_enter_execve")
attach方案B	bpf_program__attach_tracepoint(bpf_prog, "syscalls", "sys_enter_execve")	SEC("tracepoint")

- 方案A在内核态bpf.c文件的SEC的节名称中设置静态探针点信息。bpf_program__attach不用指定静态探针点的信息，会自动解析bpf.c目标文件中SEC的节名称信息来获取和确定静态探针点的信息的。
- 方案B在用户态bpf_program__attach_tracepoint函数的参数中指定静态探针点的具体信息。
- 在trace_execve.c和trace_execve.bpf.c的代码中，只要有一处设置静态探针点即可。若都设置，并且设置的静态探针点信息冲突的情况下，会以用户态的bpf_program__attach_tracepoint函数设置的信息为准。
- 特别推荐使用方案B中的bpf_program__attach_tracepoint替代方案A中的bpf_program__attach方法，这样方便在用户态代码中灵活的开关ebpf的采集。
- 除了专门用于静态探针点的bpf_program__attach_tracepoint()函数，还有适用于其他类型的专用的attach函数，例如bpf_program__attach_kprobe()、bpf_program__attach_kprobe()、bpf_program__attach_uprobe()和bpf_program__attach_usdt()等。

使用by_name替代by_title

```
1  SEC("tracepoint/syscalls/sys_enter_execve")
2  int trace_execve_enter(struct syscalls_enter_execve_args *ctx){
3      .....
4  }
```

- 早期libbpf库中提供2个函数用于获取bpf program类型数据结构，分别是bpf_object__find_program_by_name()函数和bpf_object__find_program_by_title()函数。
- 如图中，tracepoint/syscalls/sys_enter_execve这个字符串就称为title，trace_execve_enter这个函数名就称为name。上文推荐bpf内核态代码中都使用SEC("tracepoint")的语法格式，那么使用by_title函数将不再能做出区分。因此特别推荐大家今后使用by_name的函数替代by_title的函数。
- 并且，在最新版的libbpf库中，也彻底移除了bpf_object__find_program_by_title()函数支持。



第二届 eBPF 开发者大会

www.ebpftravel.com

04

低版本内核编程方案

中国·西安

获取BTF格式或DWARF格式的vmlinux

- 低版本内核推荐通过龙蜥社区获取BTF格式vmlinux。

高版本内核在/sys/kernel/btf/vmlinux路径会内建提供BTF格式的vmlinux文件，而低版本内核默认没有提供。在此，我们推荐通过龙蜥社区下载BTF格式的vmlinux。

<https://mirrors.openanolis.cn/coolbpf/btf/>

- 还可以安装dwarves包，获取pahole命令，进而提取到DWARF格式的vmlinux文件。

	CentOS环境	ubuntu环境
工具包	yum install dwarves	apt-get update apt install dwarves
DWARF包	kernel-debuginfo-\$(uname -r) kernel-debuginfo-common-x86_64-\$(uname -r)	linux-image-\$(uname -r)-dbgsym linux-image-unsigned-\$(uname -r)-dbgsym
推荐地址	https://mirrors.aliyun.com/centos-debuginfo/8/x86_64/Packages/	http://ddebs.ubuntu.com/pool/main/l/linux/ http://ddebs.ubuntu.com/pool/main/l/linux-signed/
提取命令	cp /usr/lib/debug/lib/modules/\$(uname -r)/vmlinux /tmp/vmlinux	cp /usr/lib/debug/boot/vmlinux-\$(uname -r) /tmp/vmlinux

依赖DWARF格式制作BTF格式vmlinux

➤ BTF文件制作步骤

```
1 $ cp /tmp/vmlinux /tmp/vmlinux.add
2 $ pahole -J /tmp/vmlinux.add
3 $ llvm-objcopy --dump-section .BTF=/tmp/vmlinux.btf /tmp/vmlinux.add
4 $ sudo cp /tmp/vmlinux.btf /boot/vmlinux-$(uname -r)
```

如果pahole版本大于等于1.24, 提取命令可简化为:
`pahole --btf_encode_detached=vmlinux.btf -J vmlinux`

➤ BTF文件制作简析

- 从图示看到, 原始的vmlinux文件是一个elf格式文件, 只有562M, 拥有82个节。pahole -J会依赖其中的dwarf格式信息生成第83个.BTF节, 此时vmlinux.add文件有565M。
- llvm-objcopy的dump-section选项, 会将vmlinux.add文件中的.BTF节独立保存到vmlinux.btf格式文件中, 单独保存的vmlinux.btf文件只有2.8M。
- 有些发行版环境, 还需要把vmlinux.btf转为elf格式。

```
$ ls -lh /tmp/ | grep vmlinux | awk '{print $5,$9}'
562M vmlinux
565M vmlinux.add
2.8M vmlinux.btf
```

```
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, x86-64,

$ file vmlinux.add
vmlinux.add: ELF 64-bit LSB executable, x86-64,

$ file vmlinux.btf
vmlinux.btf: data
```

使用BTF格式的vmlinux

➤ 运行过程使用BTF文件

用户态加载过程，libbpf库会在btf.c文件的btf_load_vmlinux_btf()函数中，尝试在如下文件路径查找BTF文件，并加载。

```
1 /sys/kernel/btf/vmlinux
2 /boot/vmlinux-$(uname -r)
3 /lib/modules/$(uname -r)/vmlinux-$(uname -r)
4 /lib/modules/$(uname -r)/build/vmlinux
5 /usr/lib/modules/$(uname -r)/kernel/vmlinux
6 /usr/lib/debug/boot/vmlinux-$(uname -r)
7 /usr/lib/debug/boot/vmlinux-$(uname -r).debug
8 /usr/lib/debug/lib/modules/$(uname -r)/vmlinux
```

➤ 编译过程使用BTF文件

ebpf编译过程，也需要使用bpftool命令，依赖BTF文件，生成vmlinux.h头文件。

```
bpftool btf dump file /boot/vmlinux-$(uname -r) format c >vmlinux.h
```

推荐选择此路径



第二届 eBPF 开发者大会

www.ebpftravel.com

05

低版本编译环境编程方案

中国·西安

改进libbpf编程方案在低版本环境的应用

第2代libbpf库改进方案在低版本环境也有很强的适用性：

- 1. 在低版本内核环境，需要用自制的BTF文件/boot/vmlinux-\$(uname -r)，替代/sys/kernel/btf/vmlinux。
- 2. clang版本低于9.0的属于低版本编译工具环境。此时可以尽量使用早期ebpf的开发方案，包括使用内核开发包kernel-devel头文件替代 vmlinux.h，使用map替代.map，使用llc -march=bpf替代clang -target bpf编译命令等。这些早期编译方案的组合将具有极强的低版本编译环境的兼容性。

比较项	第2代hexdump的libbpf库改进方案	低版本内核环境	低版本clang编译工具环境
获取bpf.c的头文件	bpftool btf dump file /sys/kernel/btf/vmlinux format c >vmlinux.h	bpftool btf dump file /boot/vmlinux-\$(uname -r) format c >vmlinux.h	安装kernel-devel rpm（或linux-headers deb）包
编译bpf.o目标文件	clang -O2 -target bpf -c trace_execve.bpf.c -g -o trace_execve.bpf.o	clang -O2 -target bpf -c trace_execve.bpf.c -o -g trace_execve.bpf.o	clang -O2 -emit-llvm -Xclang -c probe_execve.bpf.c -o - \n opt -O2 -mtriple=bpf-pc-linux \n llvm-dis \n llc -march=bpf -filetype=obj -o probe_execve.bpf.o
生成skel.h头文件	hexdump	hexdump	hexdump
使用skel.h头文件	统一成一个固定的名称skeleton.skel.h	统一成一个固定的名称skeleton.skel.h	统一成一个固定的名称skeleton.skel.h
用户态加载函数	使用libbpf库标准加载函数\n bpf_object__open_mem();\n bpf_object__load();\n bpf_program__attach();	使用libbpf库标准加载函数\n bpf_object__open_mem();\n bpf_object__load();\n bpf_program__attach();	使用libbpf库标准加载函数\n bpf_object__open_mem();\n bpf_object__load();\n bpf_program__attach();

Talk is cheap.Show me the code third time.

➤ 低版本编译环境示例:

```
1  $ cd ~/sreworks-ext/demos/native_libbpf_guide/skel_execve_libbpf081
2  $ make
3  $ rm -fr progs/trace_execve.bpf.o
4  $ sudo ./trace_execve
5  trace_execve 718254 4790 bash 4789 bash 0 /usr/bin/sed
6  trace_execve 735932 4791 bash 4776 bash 0 /usr/bin/tty
```

从运行结果看，不依赖vmlinux.h，仅依赖kernel-devel (linux-headers) 包的头文件编译的bpf目标文件trace_execve.bpf.o，嵌入trace_execve文件后，依然可以成功执行。

低版本编译环境内核态bpf程序编译参数解析

```
1 clang
2 -iquote ../../include/ -I../../helpers -I../../tools/lib/
3 -I/lib/modules/$(shell uname -r)/build/arch/x86/include
4 -I/lib/modules/$(shell uname -r)/build/arch/x86/include/generated
5 -I/lib/modules/$(shell uname -r)/build/include
6 -I/lib/modules/$(shell uname -r)/build/arch/x86/include/uapi
7 -I/lib/modules/$(shell uname -r)/build/arch/x86/include/generated/uapi
8 -I/lib/modules/$(shell uname -r)/build/include/uapi
9 -I/lib/modules/$(shell uname -r)/build/include/generated/uapi
10 -include /lib/modules/$(shell uname -r)/build//include/linux/kconfig.h
11 -D__KERNEL__ -D__BPF_TRACING__ -D__TARGET_ARCH_x86
12 -O2 -emit-llvm -Xclang -disable-llvm-passes -c trace_execve.bpf.c -o - | \
13 opt -O2 -mtriple=bpf-pc-linux | \
14 llvm-dis | \
15 llc -march=bpf -filetype=obj -o trace_execve.bpf.o
```

```
1 $ cat trace_execve.bpf.c
2 // SPDX-License-Identifier: GPL-2.0
3 #include <linux/ptrace.h>
4 #include <linux/sched.h>
5 #include <linux/trace_events.h>
6 #include <linux/version.h>
7 #include <linux/bpf_common.h>
8
9 #include <bpf/bpf_helpers.h>
10 #include <bpf/bpf_tracing.h>
11
12 #include "common.h"
13 #include "trace_execve.h"
```

编译参数	参数解析
-I/lib/modules/\$(shell uname -r)/build/arch/x86/include -I/lib/modules/\$(shell uname -r)/build/arch/x86/include/generated -I/lib/modules/\$(shell uname -r)/build/include -I/lib/modules/\$(shell uname -r)/build/arch/x86/include/uapi -I/lib/modules/\$(shell uname -r)/build/arch/x86/include/generated/uapi -I/lib/modules/\$(shell uname -r)/build/include/uapi -I/lib/modules/\$(shell uname -r)/build/include/generated/uapi	通过这一组的7个选项，可在这7个内核头文件的搜索路径下查找如下内核头文件。 #include <linux/ptrace.h> #include <linux/sched.h> #include <linux/trace_events.h> #include <linux/version.h>
-include /lib/modules/\$(shell uname -r)/build//include/linux/kconfig.h	等价于添加头文件 #include <linux/kconfig.h>
无需 -g参数	无需生成execve.bpf.o目标文件中的.BTF和.BTF.ext节。
llc -march=bpf	指示clang默认编译是x86格式，通过llc命令再转换为bpf格式。

BTF风格 and 传统风格map定义方式

➤ 传统风格map定义方式

```
1 SEC("maps") struct bpf_map_def perf_map = {
2     .type          = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
3     .key_size       = sizeof(int),
4     .value_size     = sizeof(u32),
5     .max_entries    = MAX_CPUS,
6 };
```

➤ BTF风格map定义方式

```
1 SEC(".maps") struct {
2     __uint(type,      BPF_MAP_TYPE_PERF_EVENT_ARRAY);
3     __uint(key_size,   sizeof(__u32));
4     __uint(value_size, sizeof(__u32));
5     __uint(max_entries, MAX_CPUS);
6 } perf_map;
```

1.0版本以上libbpf库已经禁用传统风格的map定义方式。如果依然需要使用传统风格map定义方式，需要使用0.8.1以下版本libbpf库。

➤ 0.8.1版本libbpf库对map节的提取逻辑

```
1 $ cat libbpf.c
2 #define MAPS_ELF_SEC ".maps"
3 static int bpf_object__elf_collect(struct bpf_object *obj){
4     .....
5     } else if (strcmp(name, "maps") == 0) {
6         obj->efile.maps_shndx = idx;
7     } else if (strcmp(name, MAPS_ELF_SEC) == 0) {
8         obj->efile.btfs_maps_shndx = idx;
9         .....
10 }
```

➤ 1.3.0版本libbpf库对map节的提取逻辑

```
1 $ cat libbpf.c
2 #define MAPS_ELF_SEC ".maps"
3 static int bpf_object__elf_collect(struct bpf_object *obj){
4     .....
5     } else if (strcmp(name, "maps") == 0) {
6         pr_warn("elf: legacy map definitions in 'maps' section are not supported");
7         return -ENOTSUP;
8     } else if (strcmp(name, MAPS_ELF_SEC) == 0) {
9         obj->efile.btfs_maps_shndx = idx;
10        .....
11 }
```



第二届 eBPF开发者大会

www.ebpftravel.com



钉钉加入龙蜥社区群

群号33304007



微信加sreworks小助手

邀请入群

Thanks