【云原生技术研究】 从bpftrace看如何利用eBPF实现内核追踪

原创 星云实验室 绿盟科技研究通讯 2020-08-28 17:00

收录于合集
#云原生安全和5G安全
50个

摘要

bpftrace提供了一种快速利用eBPF实现动态追踪的方法，可以作为简单的命令行工具或者入门级编程工具来使用。本文以bpftrace为例，介绍如何利用eBPF实现内核的动态追踪。

上一篇文章中[1]，我们介绍到eBPF提供了一种软件定义内核的方法，可以使用eBPF实现Linux的动态追踪以及Linux高速的网络数据包处理。

在容器环境或者是云原生环境中，各种不同的应用依托容器运行在主机上，从主机视角来看，其上运行的程序变得更加的复杂多变。因此，真实的了解系统实时状态，对于确定主机系统是否发生安全威胁以及安全入侵，有着重要的意义。本文将就如何使用eBPF进行动态追踪进行详细介绍。

一、什么是动态追踪

对计算机系统进行动态追踪，清晰的知道应用程序或者操作系统内核当前正在执行哪些操作，一直以来，都是开发者、系统运维者或者安全运维者十分关注和感兴趣的话题。

动态追踪（DynamicTracing）是一种高级的内核调试技术，通过探针机制，采集内核态或者用户态程序的运行信息，而不需要修改内核和应用程序的代码。这种机制性能损耗小，不会对系统运行构成任何危险。因此，能够以非常低的成本，在短时间内获得丰富的运行信息，进而可以快速的分析、排查、发现系统运行中的问题。

那么，动态追踪到底能追踪什么？我们知道，Linux是一个事件驱动的系统设计，因此，对于任何事件的发生，理论上都可以对其进行追踪。比如：追踪目标可以是一次“系统调用”，一个“函数的调用”，甚至是这种调用内部发生的一些细节。除此之外，还可以是一个计时器或硬件事件，比如：“发生了页面错误”、“发生了上下文切换”或“发生了CPU缓存丢失”等等。前面我们介绍到，这种追踪是通过探针机制实现，因此，具体的追踪目标，取决于系统中支持以及存在的探针内容。关于探针，后文会对其进行详细的介绍。

二、动态追踪工具

提到动态追踪，首先不得不说的就是DTrace[2]。DTrace作为动态追踪领域的鼻祖（the Father of Tracing），最初是由Sun开发的全系统动态跟踪框架，然后将其开源，支持Solaris、FreeBSD、Mac OS X等操作系统。遗憾的是，由于许可（License）问题而非技术问题，DTrace无法直接在Linux上运行，但其对Linux的动态追踪依然有着巨大的影响。

DTrace提供了一种很像C语言的脚本语言，叫做D语言，开发者可以使用D语言实现相应的追踪调试工具。它的运行时常驻在内核中，用户可以通过dtrace命令，把D语言编写的追踪脚本，提交到内核中的运行时

来执行。DTrace可以跟踪用户态和内核态的几乎所有事件，并通过一系列的优化措施，保证最小的性能开销。

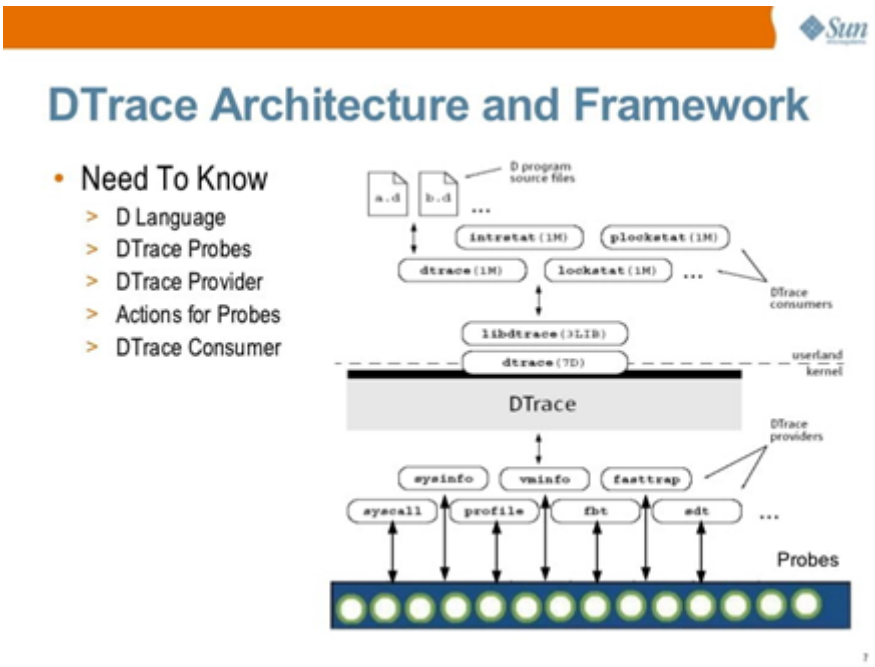

图1 DTrace架构与流程图

后文中我们会发现，本文将要介绍的bpftrace跟DTrace有着很多的相似之处，实际上，bpftrace和其相关生态的许多关键技术，著名的Brendan Gregg（System Performance，BPF Performance Tools作者）都做出了巨大的贡献，这也就解释了DTrace和bpftrace之间各种的相似之处。

尽管DTrace无法直接在Linux上运行，但是很多工程师都尝试过把DTrace 移植到Linux中，这其中，最著名的就是RedHat主推的SystemTap。同DTrace一样，SystemTap也定义了一种类似的脚本语言，方便用户根据需要自由扩展。不过，不同于DTrace，SystemTap并没有常驻内核的运行时，它需要先把脚本编译为内核模块，然后再插入到内核中执行，如下图所示。

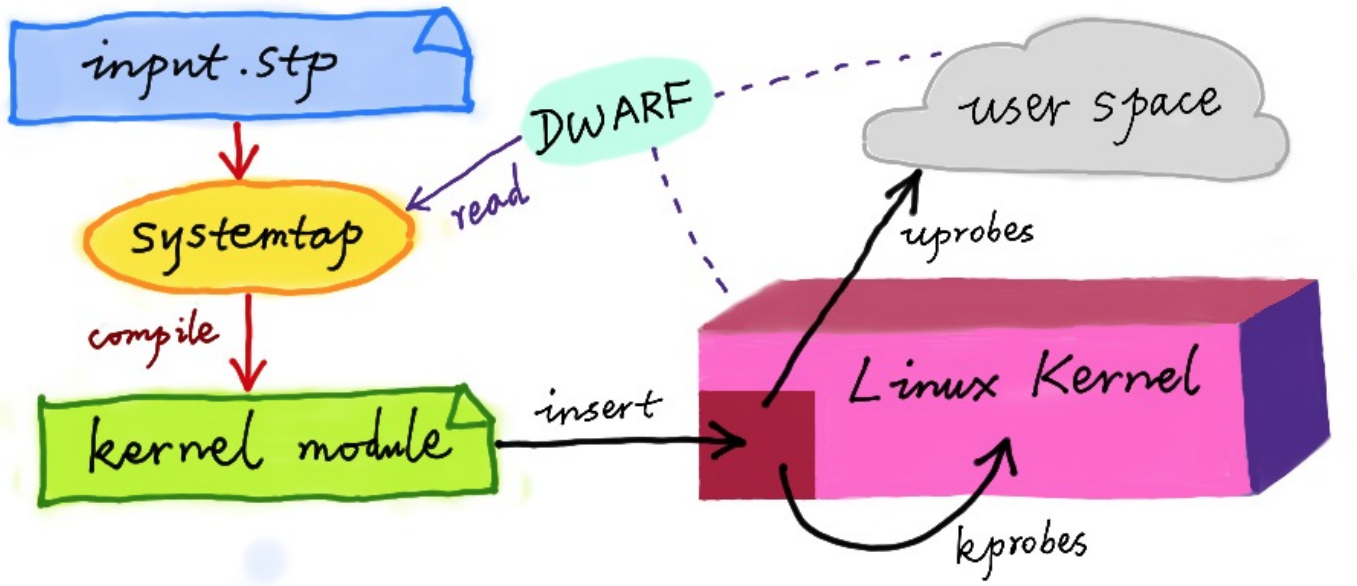

图2 SystemTap架构与流程图

因此，要实现动态追踪，通常需要在Linux中使用相应的探测手段，甚至涉及到编写并编译成内核模块，这可能会在生产系统中导致灾难性的后果。经过多年的发展，尽管它们的执行已经变的更加安全了，但是编写和测试仍然很麻烦。

eBPF似乎为上述问题找到了解决的福音，eBPF通过一种软件定义的方式,提供并支持了丰富的内核探针类型，提供了强大的动态追踪能力。开发者通过编写eBPF程序，实现相应的追踪脚本，eBPF利用自身的实现机制，保障了在内核执行动态追踪的效率以及安全性问题。

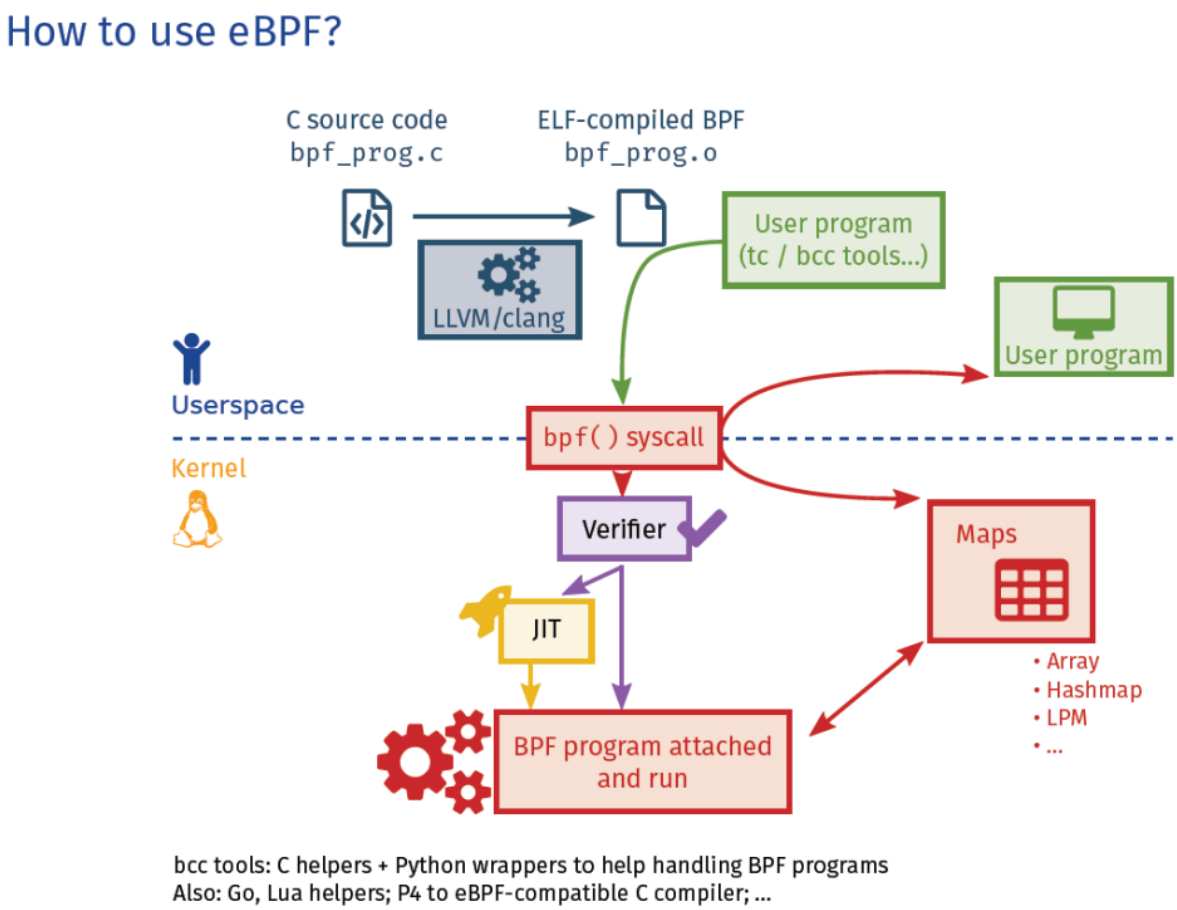

图3 eBPF架构与流程图

然而，编写eBPF程序对于开发者来说，门槛相对还是比较高，一方面需要开发者对内核有一个深入的了解，另一方面，需要使用LLVM/clang等编译程序去编译并手动的将其加载到内核中。那么像bpftrace、BCC这样的工具，就得到了开发者的青睐。相比较bpftrace，BCC已经是一套创建eBPF程序的工具包了，其所提供的功能会更加的强大，而且bpftrace在后端的处理上也依赖于BCC。作为初级使用者，我们先从bpftrace来看如何利用它实现基于eBPF的动态追踪。

bpftrace[3]是Linux中基于eBPF的高级追踪语言，使用LLVM作为后端来编译eBPF字节码脚本，并使用BCC与Linux BPF系统交互。它允许开发者用简洁的DSL（Domain Specific Language）编写eBPF程序，并将它们保存为脚本，开发者可以执行这些脚本，而不必在内核中手动编译和加载它们。

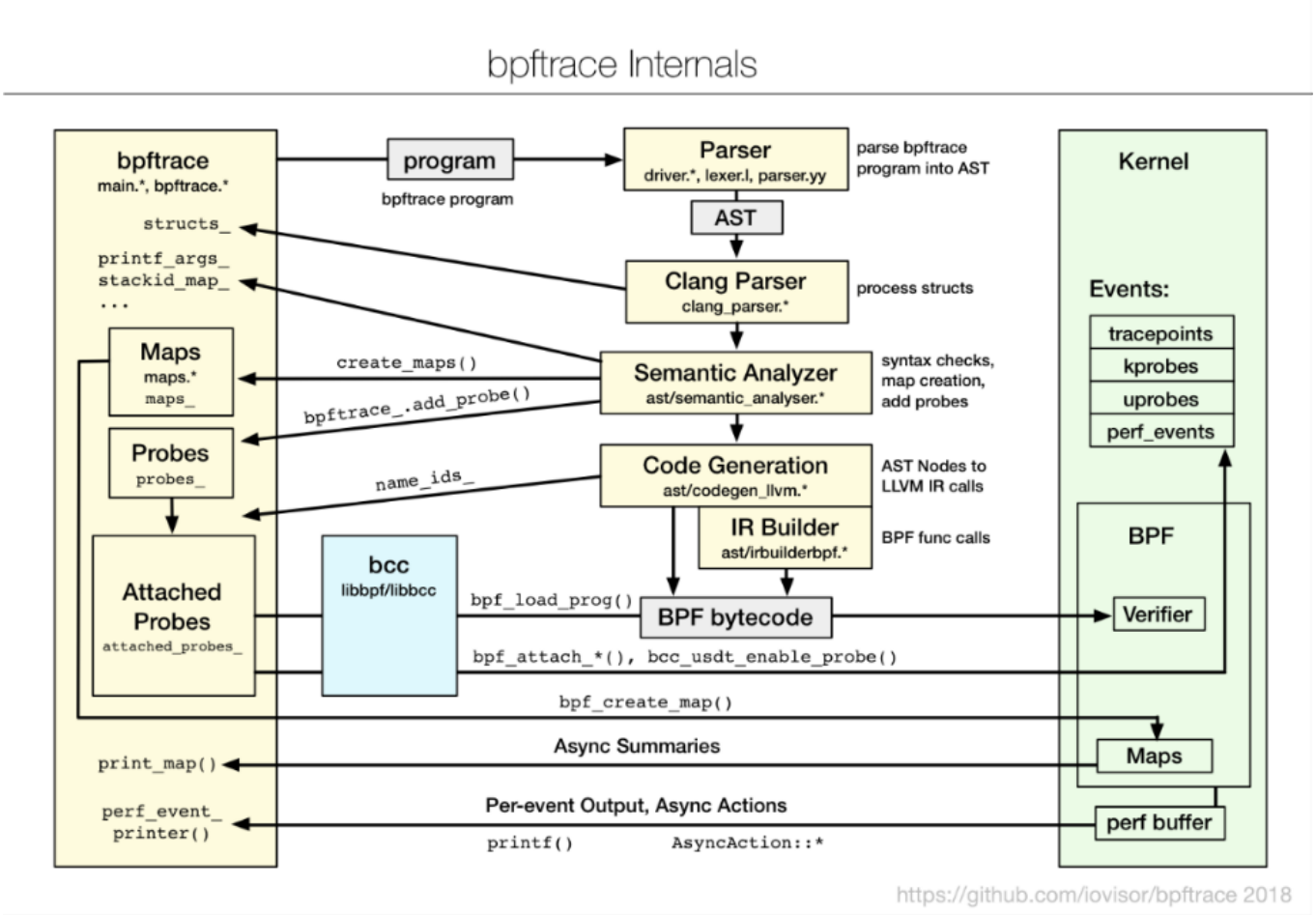

图4 bpftrace架构与流程图

bpftrace的灵感就是来自著名的Trace工具，比如awk和DTrace，bpftrace将会是DTrace的一个很好的替代品。与直接使用BCC或其他eBPF工具编写程序相比，使用bpftrace的一个优点是，它提供了许多不需要自己实现的内置功能，比如聚合信息和创建直方图等。

三、探针类型

无论是DTrace、SystemTap，还是bpftrace，其实现动态追踪都是通过探针的机制，依赖于在追踪点实现的探针，进而获取相应的追踪数据。本小节将着重介绍一下，基于eBPF的bpftrace在Linux上都支持哪些探针类型。

探针是用于捕获事件数据的检测点，bpftrace在实现内核行为追踪时使用的探针主要包括内核动态探针（Kprobes）和内核静态探针（Tracepoints）两种，这些探针延续了以往常见的动态追踪工具所使用的内核探针设计。

3.1 内核动态探针-Kprobes

eBPF支持的内核探针（Kernel probes）功能，允许开发者在几乎所有的内核指令中以最小的开销设置动态的标记或中断。当内核运行到某个标记的时候，就会执行附加到这个探测点上的代码，然后恢复正常的流程。对内核行为的追踪探测，可以获取内核中发生任何事件的信息，比如系统中打开的文件、正在执行的二进制文件、系统中发生的TCP连接等。

内核动态探针可以分为两种：kprobes 和 kretprobes。二者的区别在于，根据探针执行周期的不同阶段，来确定插入eBPF程序的位置。kprobes类型的探针用于跟踪内核**函数调用**，是一种功能强大的探针类型，让我们可以追踪成千上万的内核函数。由于它们用来跟踪底层内核的，开发者需要熟悉内核源代码，理解这些探针的参数、返回值的意义。

Kprobes通常在内核函数执行前插入eBPF程序，而kretprobes则在内核函数执行完毕返回之后，插入相应的eBPF程序。比如，tcp_connect() 是一个内核函数，当有TCP连接发生时，将调用该函数，那么如果对tcp_connect()使用kprobes探针，则对应的eBPF程序会在tcp_connect() 被调用时执行，而如果是使用kretprobes探针，则eBPF程序会在tcp_connect() 执行返回时执行。后文会举例说明如何使用Kprobes探针。

尽管Kprobes允许在执行任何内核功能之前插入eBPF程序。但是，它是一种“不稳定"的探针类型，开发者在使用Kprobes时，需要知道想要追踪的函数签名（Function Signature）。而Kprobes当前没有稳定的应用程序二进制接口（ABI），这意味着它们可能在内核不同的版本之间发生变化。如果内核版本不同，内核函数名、参数、返回值等可能会变化。如果尝试将相同的探针附加到具有两个不同内核版本的系统上，则相同的代码可能会停止工作。

因此，开发者需要确保使用Kprobe的eBPF程序与正在使用的特定内核版本是兼容的。

例如，我们可以通过bpftrace的以下命令，列出当前版本内核所支持Kprobes探针列表。

```
root@u18-181:/tmp# bpftrace -l 'kprobe:tcp*'
kprobe:tcp_mmap
kprobe:tcp_get_info_chrono_stats
kprobe:tcp_init_sock
kprobe:tcp_splice_data_recv
kprobe:tcp_push
kprobe:tcp_send_mss
kprobe:tcp_cleanup_rbuf
kprobe:tcp_set_rcvlowat
kprobe:tcp_recv_timestamp
kprobe:tcp_enter_memory_pressure
……
```

3.2 内核静态探针-Tracepoints

Tracepoints是在内核代码中所做的一种静态标记[4]，是开发者在内核源代码中散落的一些hook，开发者可以依托这些hook实现相应的追踪代码插入。

开发者在/sys/kernel/debug/tracing/events/目录下，可以查看当前版本的内核支持的所有Tracepoints，在每一个具体Tracepoint目录下，都会有一系列对其进行配置说明的文件，比如可以通过enable中的值，来设置该Tracepoint探针的开关等。

与Kprobes相比，他们的主要区别在于，Tracepoints是内核开发人员已经在内核代码中提前埋好的，这也是为什么称它们为静态探针的原因。而kprobes更多的是跟踪内核函数的进入和返回，因此将其称为动态的探针。但是内核函数会随着内核的发展而出现或者消失，因此kprobes对内核版本有着相对较强的依赖性，前文也有提到，针对某个内核版本实现的追踪代码，对于其它版本的内核，很有可能就不工作了。

那么，相比Kprobes探针，我们更加喜欢用Tracepoints探针，因为Tracepoints有着更稳定的应用程序编程接口，而且在内核中保持着前向兼容，总是保证旧版本中的跟踪点将存在于新版本中。

然而，Tracepoints的不足之处在于，这些探针需要开发人员将它们添加到内核中，因此，它们可能不会覆盖内核的所有子系统，只能使用当前版本内核所支持的探测点。

例如，我们可以通过bpftrace的以下命令，列出当前版本内核所支持的Tracepoints探针列表。

```
root@u18-181:/tmp# bpftrace -l 'tracepoint:*'
tracepoint:syscalls:sys_enter_socket
tracepoint:syscalls:sys_enter_socketpair
tracepoint:syscalls:sys_enter_bind
tracepoint:syscalls:sys_enter_listen
tracepoint:syscalls:sys_enter_accept4
tracepoint:syscalls:sys_enter_accept
tracepoint:syscalls:sys_enter_connect
tracepoint:syscalls:sys_enter_getsockname
……
```

3.3 其它探针

除了前面介绍的Kprobes/Kretprobes和Tracepoints内核探针外，eBPF还支持对用户态程序通过探针进行追踪。例如用户态的Uprobes/Uretprobes探针，在用户态对函数进行hook，实现与Kprobes/Kretprobes类似的功能；再比如USDTs（User Static Defined Tracepoints）探针，是用户态的Tracepoints，需要开发者在用户态程序中自己埋点Tracepoint，实现与内核Tracepoints类似的功能。

另外，bpftrace还支持内核软件事件（software）、处理器事件（hardware）等探针格式，具体可参考其github官方的介绍[5]，本文就不逐一进行分析了。

四、安装部署bpftrace

前文对动态追踪工具、bpftrace以及相应的探针类型做了介绍，接下来我们就部署运行bpftrace，看看如何使用其实现动态追踪。

由于bpftrace是建立在eBPF之上的一种编程语言，考虑到部分特性需满足Linux内核的支持，因此建议Linux的内核版本在4.9以上，部分功能在低版本内核中是不支持的，例如Tracepoints是从4.7开始支持，Uprobes从4.3开始支持，Kprobes从4.1开始支持。

bpftrace在github官方文档中提供了三种安装部署方式[6]：

（1）通过**安装包**安装，比如在Ubuntu19.04之后，可以运行sudo apt-get install -y bpftrace安装；

（2）通过**Docker镜像**安装，使用如下命令运行一个docker容器，并且执行对应的脚本。下面示例命令中，使用latest镜像，运行capable.bt脚本。

```
$ docker run -ti -v /usr/src:/usr/src:ro \
      -v/lib/modules/:/lib/modules:ro \
      -v/sys/kernel/debug/:/sys/kernel/debug:rw \
     --net=host --pid=host --privileged \
     quay.io/iovisor/bpftrace:latest \
      capable.bt
```

```
Attaching 3 probes...
Tracing cap_capable syscalls... Hit Ctrl-C to end.
TIME     UID    PID    COMM           CAP      NAME         AUDIT
02:49:04  0     501    systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     501    systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     501    systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     5585   systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     5576   runc:[2:INIT]    21  CAP_SYS_ADMIN      0
02:49:04  0     5576   runc:[2:INIT]    21  CAP_SYS_ADMIN      0
02:49:04  0     5577   systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     5587   systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     501    systemd-udevd    12  CAP_NET_ADMIN      0
02:49:04  0     5576   runc:[2:INIT]    21  CAP_SYS_ADMIN      0
02:49:04  0     5576   runc:[2:INIT]    21  CAP_SYS_ADMIN      0
……
```

（3）通过**源码**安装，笔者的测试环境就是通过源码进行的部署安装，相关信息如下所示。

操作系统：Ubuntu 18.04.4 LTS\n \l

内核版本：Linux5.5.7-050507-generic x86_64

安装步骤：

```
$ apt-get update
$ apt-get install -y bison cmake flex g++ gitlibelf-dev zlib1g-dev libfl-dev systemtap-sdt-dev binutils-dev
$ apt-get install -y llvm-7-dev llvm-7-runtimelibclang-7-dev clang-7

$ git clone https://github.com/iovisor/bpftrace

$ mkdir bpftrace/build; cd bpftrace/build;
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j8
$ make install
```

bpftrace的二进制可执行文件被安装在了/usr/local/bin/bpftrace路径下，相关的工具安装在/usr/local/share/bpftrace/tools/路径。另外，可以在cmake时通过参数指定安装目录，默认路径是-DCMAKE_INSTALL_PREFIX=/usr/local。

安装完毕后，可以运行bpftrace --help进行验证，同时可以简单看一下其用法。

```
root@u18-181:~# bpftrace --help
USAGE:
    bpftrace[options] filename
    bpftrace[options] - <stdin input>
    bpftrace[options] -e 'program'

OPTIONS:
    -BMODE        output buffering mode('full', 'none')
    -fFORMAT      output format ('text','json')
    -ofile        redirect bpftrace output tofile
    -d            debug info dry run
    -dd           verbose debug info dry run
```

```
    -b           force BTF (BPF type format)processing
    -e'program'   execute this program
    -h,--help     show this help message
    -IDIR         add the directory to the include search path
    --includeFILE add an #include file before preprocessing
    -l[search]    list probes
    -pPID         enable USDT probes on PID
    -c'CMD'       run CMD and enable USDTprobes on resulting process
   --usdt-file-activation
                activate usdt semaphores based on file path
   --unsafe       allow unsafebuiltin functions
    -v           verbose messages
   --info         Print informationabout kernel BPF support
    -k           emit a warning when a bpf helper returns anerror (except read functions)
    -kk          check all bpf helper functions
    -V,--version  bpftrace version
…… ……
```

五、如何进行追踪

bpftrace的一个方便之处在于，其既可以通过一个命令行，完成简单动态追踪，又可以按照其规定的语法结构，将追踪逻辑编辑成可执行的脚本。

5.1 命令行

bpftrace github官网给出了一个通过命令行进行使用的教程[7]，这里我们选择其中几个简要的进行介绍分析。

>>>> 5.1.1 列出支持的探针

```
root@u18-181:~# bpftrace -l 'tracepoint:syscalls:sys_enter_*'
```

这个我们在前面介绍探针的时候已经使用过了，bpftrace -l 可以列出支持的所有探针，后面可以使用上述命令中引号内的条件对结果进行搜索过滤，搜索条件支持*/?等通配符。也可以通过管道传递给grep，进行完整的正则表达式搜索。

>>>> 5.1.2 Hello World

```
root@u18-181:~# bpftrace -e 'BEGIN { printf("hello world\n"); }'
```

打印欢迎消息，运行后按Ctrl-C结束。

命令中的-e 'program'，表示将要执行这个程序。BEGIN是一个特殊的探针，在程序开始执行时触发探针执行，可以使用它设置变量和打印消息头。BEGIN探针后的{ }是与该探针关联的动作。

>>>> 5.1.3 追踪文件打开

```
root@u18-181:~#  bpftrace  -e'tracepoint:syscalls:sys_enter_openat  {  printf("%s  %s\n",
comm,str(args->filename)); }'
Attaching 1 probe...
ls /etc/ld.so.cache
ls /lib/x86_64-linux-gnu/libselinux.so.1
ls /lib/x86_64-linux-gnu/libc.so.6
```

```
ls /lib/x86_64-linux-gnu/libpcre.so.3
ls /lib/x86_64-linux-gnu/libdl.so.2
ls /lib/x86_64-linux-gnu/libpthread.so.0
ls /proc/filesystems
ls /usr/lib/locale/locale-archive
ls .
^C
```

这个命令可以在文件打开时，追踪并打印出进程名以及对应的文件名，运行后按Ctrl-C结束。

执行的程序中，tracepoint:syscalls:sys_enter_openat表示这是一个tracepoint探针，当进入openat()系统调用时执行该探针。该探针的动作是打印进程名和文件名，也就是后边{ }中的内容。

comm是内建变量，代表当前进程的名字，其它类似的变量还有pid和tid，分别表示进程标识和线程标识。

args是一个指针，指向该tracepoint的参数。这个结构时由bpftrace根据tracepoint信息自动生成的。这个结构的成员可以通过命令bpftrace -vltracepoint:syscalls:sys_enter_openat找到。

5.2 追踪脚本

除了上述命令行方式之外，我们还可以将复杂的追踪命令编写成特定的脚本，然后通过bpftrace命令执行这个脚本完成我们的追踪目标。

>>>> 5.2.1 文件执行追踪

下面这个示例脚本，跟踪了进程何时调用exec()。它可以用于识别新的通过fork()->exec()序列创建的进程。不过，这里当前没有对返回值进行跟踪，因此exec()可能已经执行失败。

该脚本同样是采用了tracepoint探针，当进入execve ()系统调用时执行该探针。该探针的动作是打印时间、PID和执行命令。

```
#!/usr/bin/env bpftrace
/*
 *execsnoop.bt   Trace new processes viaexec() syscalls.
 *             For Linux, uses bpftrace andeBPF.
 */

BEGIN
{
    printf("%-10s%-5s %s\n", "TIME(ms)", "PID", "ARGS");
}

tracepoint:syscalls:sys_enter_execve
{
    printf("%-10u%-5d ", elapsed / 1000000, pid);
    join(args->argv);
}
```

执行结果：

```
root@u18-181:~# bpftraceexecsnoop.bt
Attaching 2probes...
TIME(ms)   PID       ARGS
6135       15424     /usr/sbin/sshd -D -R
```

```
6589      15426    /usr/sbin/sshd -D -R
6590      15427
6592      15428    /usr/bin/env -iPATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
run-parts--lsbsysinit /etc/update-motd.d
6593      15428     run-parts --lsbsysinit /etc/update-motd.d
6593      15428     run-parts --lsbsysinit /etc/update-motd.d
6593      15428     run-parts --lsbsysinit /etc/update-motd.d
6593      15428    run-parts --lsbsysinit/etc/update-motd.d
6593      15428     run-parts --lsbsysinit /etc/update-motd.d
6593      15428     run-parts --lsbsysinit /etc/update-motd.d
6594      15429     /etc/update-motd.d/00-header
6596      15430    uname -o
6597      15431   uname -r
6599      15432    uname -m
……
```

运行上述追踪脚本，我们同时新创建了一个ssh连接，可以发现该脚本追踪到了在创建新的ssh连接过程中发生的部分execve ()系统调用情况。

>>>> 5.2.2 TCP连接追踪

我们再看一个kprobe探针的例子。下面这个脚本，使用了kprobe探针，当内核功能tcp_connect() 被调用时，执行脚本中的追踪程序，并且抓取时间、PID、命令以及源目的相关的信息。

从脚本中对抓取信息的解析和处理中我们可以看出，如前文所述，在使用kprobes探针时，需要知道想要追踪的函数签名（Function Signature），这里一方面需要开发者对内核函数有一个比较清晰的认识，同时对特定版本的依赖也较强。

```
#!/usr/bin/env bpftrace
/*
 *tcpconnect.bt  Trace TCP connect()s.
 *              For Linux, uses bpftrace andeBPF.
 */
#include <linux/socket.h>
#include <net/sock.h>

BEGIN
{
 printf("Tracing tcp connections. Hit Ctrl-C to end.\n");
 printf("%-8s %-8s %-16s ", "TIME", "PID","COMM");
 printf("%-39s %-6s %-39s %-6s\n", "SADDR","SPORT", "DADDR", "DPORT");
}

kprobe:tcp_connect
{
  $sk =((struct sock *) arg0);
 $inet_family = $sk->__sk_common.skc_family;

  if($inet_family == AF_INET || $inet_family == AF_INET6) {
    if($inet_family == AF_INET) {
      $daddr= ntop($sk->__sk_common.skc_daddr);
      $saddr= ntop($sk->__sk_common.skc_rcv_saddr);
```

```
    } else {
      $daddr= ntop($sk->__sk_common.skc_v6_daddr.in6_u.u6_addr8);
      $saddr= ntop($sk->__sk_common.skc_v6_rcv_saddr.in6_u.u6_addr8);
    }
    $lport =$sk->__sk_common.skc_num;
    $dport =$sk->__sk_common.skc_dport;

    //Destination port is big endian, it must be flipped
    $dport =($dport >> 8) | (($dport << 8) & 0x00FF00);

   time("%H:%M:%S ");
   printf("%-8d %-16s ", pid, comm);
   printf("%-39s %-6d %-39s %-6d\n", $saddr, $lport, $daddr,$dport);
  }
}
```

执行结果：

```
root@u18-181:/usr/local/share/bpftrace/tools#bpftrace tcpconnect.bt
Attaching 2 probes...
Tracing tcp connections. Hit Ctrl-C to end.
TIME      PID  COMM   SADDR      SPORT   DADDR      DPORT
17:56:02   14243   curl    192.168.19.181  55758 182.61.200.7     80
17:56:36   14255  http    192.168.19.181  48572 192.168.19.16    80
17:56:37   14254  https   192.168.19.181  36262 192.168.255.51   3128
17:56:37   14253   http   192.168.19.181  36264  192.168.255.51   3128
17:56:38   14494  https   192.168.19.181  36266 192.168.255.51   3128
^C
```

我们发现，脚本中相关的语法内容，跟上一节介绍的命令行方式，其实是一致的。命令行可以简单快速的追踪到一些简单的数据，而脚本的好处就是，我们可以把一些复杂、常用的追踪内容实现为特定的追踪工具来更方便的使用。

bpftrace中已经实现了一部分的追踪工具，前文安装部署部分已经介绍，默认情况下，这些工具在/usr/local/share/bpftrace/tools/路径下面。

5.3 追踪工具

笔者当前部署的bpftrace（bpftrace v0.10.0-156-ga840）版本，共提供35个可直接使用的工具脚本[8]。按照其实现的功能，对除了系统性能分析相关的14个工具外，其余的21个工具大致进行了一下分类，如下表所示。

>>>> 5.3.1 网络

编号	脚本	使用探针	实现功能
1	tcpaccept.bt	kretprobe:inet_csk_accept	追踪TCP套接字accept()操作
2	tcpconnect.bt	kprobe:tcp_connect	追踪所有的TCP连接操作
3	tcpdrop.bt	kprobe:tcp_drop	追踪TCP丢包详情
4	tcplife.bt	kprobe:tcp_set_state	追踪TCP连接生命周期详情
5	tcpretrans.bt	kprobe:tcp_retransmit_skb	追踪TCP重传

6	tcpsynbl.bt	kprobe:tcp_v4_syn_recv_sock, kprobe:tcp_v6_syn_recv_sock	以柱状图的形式显示TCP SYN backlog

>>>> 5.3.2 安全

编号	脚本	使用探针	实现功能
1	capable.bt	kprobe:cap_capable	追踪capabilitiy的使用
2	oomkill.bt	kprobe:oom_kill_process	追踪OOM killer
3	setuids.bt	tracepoint:syscalls:sys_enter_setuid, tracepoint:syscalls:sys_enter_setfsuid tracepoint:syscalls:sys_enter_setresuid tracepoint:syscalls:sys_exit_setuid tracepoint:syscalls:sys_exit_setfsuid tracepoint:syscalls:sys_exit_setresuid	跟踪通过setuid系统调用实现特权升级

>>>> 5.3.3 系统

编号	脚本	使用探针	实现功能
1	bashreadline.bt	uretprobe:/bin/bash:readline	打印从所有运行shell输入的bash命令
2	execsnoop.bt	tracepoint:syscalls:sys_enter_execve	追踪通过exec()系统调用产生新进程
3	killsnoop.bt	tracepoint:syscalls:sys_enter_kill tracepoint:syscalls:sys_exit_kill	追踪kill()系统调用
4	naptime.bt	tracepoint:syscalls:sys_enter_nanosleep	跟踪应用程序通过nanosleep(2)系统调用休眠
5	opensnoop.bt	tracepoint:syscalls:sys_enter_open tracepoint:syscalls:sys_enter_openat tracepoint:syscalls:sys_exit_open, tracepoint:syscalls:sys_exit_openat	追踪全系统范围内的open()系统调用，并打印详细信息
6	pidpersec.bt	tracepoint:sched:sched_process_fork	追踪新进程产生速率
7	statsnoop.bt	tracepoint:syscalls:sys_enter_statfs tracepoint:syscalls:sys_enter_statx, tracepoint:syscalls:sys_enter_newstat, tracepoint:syscalls:sys_enter_newlstat tracepoint:syscalls:sys_exit_statfs, tracepoint:syscalls:sys_exit_statx, tracepoint:syscalls:sys_exit_newstat, tracepoint:syscalls:sys_exit_newlstat	追踪系统范围内的不同stat()系统调用
8	swapin.bt	kprobe:swap_readpage	按进程计算交换次数，以显示哪个进程受到交换的影响
9	syncsnoop.bt	tracepoint:syscalls:sys_enter_sync, tracepoint:syscalls:sys_enter_syncfs, tracepoint:syscalls:sys_enter_fsync, tracepoint:syscalls:sys_enter_fdatasync, tracepoint:syscalls:sys_enter_sync_file_range, tracepoint:syscalls:sys_enter_msync	追踪sync相关的各种系统调用

10	syscount.bt	tracepoint:raw_syscalls:sys_enter	对系统调用进行追踪计数，并打印前10个系统调用id和前10个生成系统调用的进程名的摘要
11	threadsnoop.bt	uprobe:/lib/x86_64-linux-gnu/libpthread.so.0:pthread_create	追踪新线程
12	xfsdist.bt	kprobe:xfs_file_read_iter, kprobe:xfs_file_write_iter, kprobe:xfs_file_open, kprobe:xfs_file_fsync kretprobe:xfs_file_read_iter, kretprobe:xfs_file_write_iter, kretprobe:xfs_file_open, kretprobe:xfs_file_fsync	追踪XFS的读、写、打开和fsync，并将它们的延迟汇总为一个2次方直方图

六、总结

动态追踪是一种高级的内核追踪技术，在性能优化以及安全检测和防护上，有着重要的意义。尤其是在云原生环境中，面对容器化的基础设施、微服务架构下的应用程序，通过动态追踪，实现整个系统的可观察性。

本文从动态追踪入手，介绍了bpftrace及其使用的相关探针、追踪脚本的编写以及现有工具脚本的分类使用等内容。使用eBPF进行内核追踪，bpftrace是一种有效的入门工具和方法。

参考文献

[1] 【云原生技术研究】 BPF使能软件定义内核，https://mp.weixin.qq.com/s?__biz=MzIyODYzNTU2OA==&mid=2247487440&idx=1&sn=cb6379cfc4a1bba0881363840afc438a&chksm=e84fa90fdf38201990781e3c95d9857e6d4ad083ce40a575e1cbdc12aaabb4f58ba527dc58c1&token=436386698&lang=zh_CN#rd

[2] DTrace， http://dtrace.org/blogs/about

[3] bpftrace， https://github.com/iovisor/bpftrace

[4] tracepoints，https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[5] probes，https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md#probes

[6] https://github.com/iovisor/bpftrace/blob/master/INSTALL.md

[7] tutorial_one_liners_chinese，https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners_chinese.md

[8] bpftrace tools， https://github.com/iovisor/bpftrace/tree/master/tools

关于星云实验室

星云实验室专注于云计算安全、解决方案研究与虚拟化网络安全问题研究。基于IaaS环境的安全防护，利用SDN/NFV等新技术和新理念，提出了软件定义安全的云安全防护体系。承担并完成多个国家、省、市以及行业重点单位创新研究课题，已成功孵化落地绿盟科技云安全解决方案

内容编辑：星云实验室 江国龙　责任编辑：王星凯

往期回顾

- 【云原生攻防研究】Istio访问授权再曝高危漏洞
- 【云原生技术研究】BPF使能软件定义内核
- 【云原生攻防研究】容器逃逸技术概览
- 【云原生技术研究】Cilium网络概述

关于我们

绿盟科技研究通讯由绿盟科技创新中心负责运营，绿盟科技创新中心是绿盟科技的前沿技术研究部门。包括云安全实验室、安全大数据分析实验室和物联网安全实验室。团队成员由来自清华、北大、哈工大、中科院、北邮等多所重点院校的博士和硕士组成。

绿盟科技创新中心作为“中关村科技园区海淀园博士后工作站分站”的重要培养单位之一，与清华大学进行博士后联合培养，科研成果已涵盖各类国家课题项目、国家专利、国家标准、高水平学术论文、出版专业书籍等。

我们持续探索信息安全领域的前沿学术方向，从实践出发，结合公司资源和先进技术，实现概念级的原型系统，进而交付产品线孵化产品并创造巨大的经济价值。

长按上方二维码，即可关注我们

收录于合集 #云原生安全和5G安全 50

上一篇
【云原生攻防研究】容器逃逸技术概览

下一篇
【云原生安全】从分布式追踪看云原生应用安全

喜欢此内容的人还喜欢

创新方案，如何更有效地预防数据泄露?

绿盟科技研究通讯