第十章 系统集成开发eBPF

安卓系统的安全攻防技术日新月异，如今正进入一个全新的高度。随着eBPF技术的崛起，国内外的安全专家们也积极探索eBPF技术在安全领域中的应用场景。

本章将为安卓系统开发与定制人员提供一种结合代码修改和eBPF可观测性的系统定制细路。这旨在为安全行业的发展提供一点启示和引导作用，希望能够激发更多创意和思考。

10.1 eBPF概述

eBPF（扩展伯克利数据包过滤器）是一种现代化的Linux内核技术，它使开发人员能够对网络数据包进行更细粒度的过滤和修改。相较于传统的Berkeley Packet Filter (BPF)，eBPF具有更高的灵活性、可扩展性和安全性，因此在广泛应用中受到了认可。

实际上，通过使用eBPF技术可以实现多项功能，例如网络流量监控、日志记录、流量优化以及安全审计等。这使得它在各个领域都具备广泛的应用前景。

10.1.1 eBPF发展背景

在2008年，Linux内核开发者提出了BPF（Berkeley Packet Filter）的概念，它是一种用于过滤和修改网络数据包的内核模块。BPF是一个非常强大的工具，它允许开发人员对网络数据包进行细粒度的过滤和修改，从而实现网络流量监控、日志记录、性能优化等功能。

然而，BPF也存在一些限制。首先，编写和编译BPF模块需要由内核开发人员手动完成，这对非专业开发人员来说可能是具有挑战性的任务。其次，在编写BPF模块时需要一定的技术知识和经验，否则可能会导致内核崩溃或其他问题。最后，并且很重要的是，由于BPF模块拥有高级别的访问权限，意味着它们可以访问系统中所有的内存和网络资源，在安全上需要进行严格控制。

为了解决这些问题，在2014年Linux内核引入了eBPF（扩展伯克利数据包过滤器）技术。eBPF是BPF的扩展版本，并提供更多功能以及更加灵活可扩展与安全性方面优势。相比传统 BFP技术,eBFP得到广泛应用与认可。

eBPF最初的发展目标是实现高效网络数据包过滤。随着发展，除了扩展传统的数据包过滤字节码格式外，eBPF还支持在整个操作系统不同模块中运行多种类型的eBPF程序。最初提倡的可观测性领域也已扩展为支持对数据进行观测与修改（包括用户态数据和内核函数返回值）。这样的进展使得eBPF技术看起来更像一个现代化的Hook 技术框架，因此受到安全从业人员青睐。

可以预见，在内核版本更新中，eBPF内置功能将会越来越丰富。作为eBPF能力核心部分，eBFP 内核方法接口也会变得更加多样化。基于这些接口所实现的安全功能必定会影响整个行业的发展。

10.1.2 eBPF的工作原理

eBPF的工作原理可以概括为以下三个步骤：

1. **解析**：在系统启动时，内核会加载eBPF符号表到内存中。这个二进制文件包含了eBPF模块的所有符号和参数。同时，内核还会将eBPF模块的二进制代码转换为机器码，并加载到内存中。
2. **执行**：当网络数据包到达时，内核首先检查是否匹配到了与之关联的eBPF模块。如果匹配成功，内核会执行该模块中的代码来对网络数据包进行过滤或修改。在执行过程中，内核使用eBPF运行时数据结构体来传递参数和上下文信息。

3. **卸载**：当eBPF模块执行完毕后，内核会将其从内存中卸载并清除。此时，所有符号和参数都被还原成二进制码，并从内存中清除。

10.1.3 eBPF的应用场景

eBPF是一种非常强大的技术，可以实现许多网络流量监控、日志记录和性能优化等功能。下面列举了一些常见的eBPF应用场景：

1. **日志记录**：eBPF可用于记录网络流量、系统调用和错误事件等信息，实现全面的系统监控和日志记录。在云原生安全领域，安全监控工具如`sysdig`和`falco`使用eBPF来监控系统调用。

2. **流量控制**：通过eBPF可以实现网络流量的控制，例如限制同一主机或端口的网络流量。防火墙工具如著名的基于eBPF扩展版本的`iptables`就是典型应用。

3. **流量优化**：使用eBPF可以对网络流量进行优化，例如过滤重复数据包、压缩数据包以及优化TCP/IP协议栈等。这方面应用包括开发透明代理工具、网络数据镜像转发工具和流量优化工具等。

4. **安全审计**：利用eBPF可以实现安全审计功能，如记录系统用户操作并检查资源使用情况。主机安全类防护产品（如HIDS）在这个领域有着广泛应用空间。一个例子是安全工具`Tracee`。

总的来说，eBPF技术在各个领域都有广泛应用，并且具备强大的灵活性和可扩展性。

10.2 eBPF相关的开发工具

eBPF是一种现代化的Linux内核技术，允许开发者在内核中安全地运行外部程序，用于处理网络数据包、系统调用等场景。相较于传统的内核模块，eBPF具有更高的安全性和可移植性，因此得到了越来越广泛的应用。虽然eBPF运行在内核中，但控制它的程序却是运行在用户态。下面将介绍一些开发eBPF工具时常用的方法和库。

1. bcc：bcc（BPF Compiler Collection）是一个基于LLVM编译器框架构建而成的工具集合。它提供了一组功能强大且易于使用的命令行工具和库来开发、测试和分析eBPF程序。通过bcc可以编写高级语言（如C/C++）来生成eBPF代码，并能够以安全方式注入到目标系统中。

2. bpftrace：bpftrace是一个动态追踪工具，可以使用类似awk语法的脚本语言对系统进行实时监测与跟踪。它利用libbpf库解析并执行由用户定义的事件处理逻辑，并支持实时查看、过滤和聚合各种类型的跟踪数据。

3. libbpf：libbpf是一个用户空间库，提供了与eBPF交互的API。它允许开发者在用户态编写和加载eBPF程序，并提供了一些辅助函数用于操作eBPF映射（maps）和事件处理。

这些工具/库为开发人员提供了丰富的资源来编写、测试和分析eBPF程序，从而更加便捷地利用eBPF技术解决各种问题。

10.2.1 bcc

`bcc`是一款开源的eBPF快速开发工具，最初使用Python作为eBPF程序的开发语言。随着社区的发展，该工具支持了C语言开发eBPF程序。你可以在其仓库地址https://github.com/iovisor/bcc找到该项目。

该仓库提供了一组用Python编写的eBPF工具集，位于tools目录下。这些工具涉及文件、进程、网络、延时、性能观测等多个应用场景，并且提供了C语言版本的eBPF工具集，位于libbpf-tools目录下。这些C语言实现版本是非常好的学习资料，适合初学者入门。

在项目README中列出了不同系统位置上运行eBPF所需环境的分布图，并介绍了tools目录下各个工具的用途。例如，要监控文件打开操作，可以执行以下命令：

```
$ sudo python3 tools/opensnoop.py
```

这将启动opensnoop脚本，监控文件的打开操作并输出相关信息。

10.2.2 bpftrace

bpftrace是一个用于记录和追踪系统方法调用的工具。它使用eBPF程序来处理网络数据包、系统调用、文件访问等场景。通过使用bpftrace，开发者可以快速验证要观测的函数是否支持使用eBPF实现。

bpftrace是一个开源工具，你可以在其仓库地址https://github.com/iovisor/bpftrace找到它。按照官方说明安装好该工具后，会提供一个名为 bpftrace 的主程序。这个主程序接受单选命令以及一个bt格式的脚本作为输入。在脚本中，你可以设置观测程序的入口和出口、参数传递等信息，非常方便。

需要注意的是，目前bpftrace只提供了观测功能，并没有提供数据修改功能。相比之下，在这一点上不如bcc和libbpf。

执行以下命令可观测所有文件打开操作：

```
$ sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm,
str(args->filename)); }'
```

当运行此命令时，将会监控所有文件打开操作并输出进程名称与文件名。

10.2.3 libbpf

libbpf是一个用于编写和运行eBPF程序的开源库。你可以在其仓库地址https://github.com/libbpf/libbpf找到它。该库提供了一组C接口函数，允许开发者使用C或Rust语言编写和运行eBPF程序。

除此之外，libbpf官方还单独提供了一些使用该库进行eBPF程序开发的示例代码，这些示例代码位于仓库地址https://github.com/libbpf/libbpf-bootstrap下的examples/c目录中。这些示例代码与 bcc的libbpf-tools目录下的样例类似，前者更注重简洁性而后者则功能更加丰富。

总体来说，bcc、bpftrace和libbpf都是非常重要的工具，用于开发与eBPF相关的工具。它们提供了丰富的功能和工具集合，并为开发人员提供便利，在进行eBPF程序的开发、调试和追踪时都能够起到很大帮助。

10.3 安卓系统集成eBPF功能

eBPF的功能实现和完善在优先支持x86_64架构上进行。对于其他系统架构如arm64等，会在后续逐步补充支持。大多数安卓手机设备采用的是arm64架构的处理器，因此与arm64版本的Linux发行版相比，eBPF在这些设备上的功能支持可能会稍有延迟，并且其程度也与arm64版本的Linux其他发行版保持一致。例如，在使用相同内核版本的Ubuntu系统和安卓系统中，它们对eBPF功能的支持基本是一致的。

需要注意的是，随着时间推移和开源社区不断努力改进，ARM体系结构上对eBPF功能的支持将不断提高，并逐渐接近x86_64架构所具备的水平。

10.3.1 不同版本内核对eBPF的影响

安卓系统的版本更新通常伴随着系统内核版本的升级。目前最新的安卓14采用了6.1版本的内核。在该版本中，默认的内核配置已经支持了一些常用功能，如kprobes、uprobes、tracepoint和raw_tracepoint，并且与同样是arm64架构下使用6.1版本内核的Ubuntu系统上eBPF功能保持一致。

然而，还有一些特性，如TRACING类型的eBPF程序：fentry, fmod_ret, kfuncs, LSM, SYSCALL, 和tp_btf等，在6.1的arm64内核中仍然不被支持。意味着在基于Ubuntu arm64架构上使用6.1内核时仍会遇到测试失败问题。这种问题在内核6.4 RC1版本的一个补丁合并中得到了修正，可以通过以下链接查看这个合并的详情：https://github.com/torvalds/linux/commit/df45da57cbd35715d590a36a12968a94508ccd1f。其中，有一个Tracing的名为**Support for "direct calls" in ftrace, which enables BPF tracing for arm64**的更新，并且这个补丁目前合并进入了安卓的主线内核中，这意味着，在不久最新版本的安卓系统中，不需要对内核做任何的补丁，就完美的支持eBPF开发与测试。安卓主线内核的更新日志可以通过链接https://android.googlesource.com/kernel/common/+log/refs/heads/android-mainline查看。

安卓12使用的内核是5.10，安卓13所采用的Linux内核是5.10与5.15。这两个版本对于上述提到的uprobes、tracepoint和raw_tracepoint功能提供了支持。然而，在对于 kprobes 的支持上有一些不足之处，只能说是部分支持。这是由于安卓的GKI 2.0引入了一些变化，导致无法成功启动像CONFIG_DYNAMIC_FTRACE这样的选项。具体细节将在下面关于内核配置注意事项的章节进行说明。

10.3.2 一些需要注意的内核配置

与安卓的eBPF相关的内核配置有以下几个：

1. CONFIG_DYNAMIC_FTRACE：如果内核开启了该选项，Ftrace框架内部的mcount函数会被实现为空函数（只包含一条ret指令）。在系统启动时，所有调用到的mcount都会被替换成无操作（nop）指令。当开启跟踪器后，所有函数入口处位置将动态替换为跳转至 ftrace_caller()的指令。这个选项是fentry内核配置中CONFIG_FPROBE的依赖项，因此可能导致fentry无法生效。

2. CONFIG_FUNCTION_TRACER: 开启该选项后，在每个函数入口处插入一个跳转指令(bl mcount)到mcount()函数中进行运行时追踪。在mcount()中会检查是否注册了函数指针ftrace_trace_function, 默认情况下注册为空函数ftrace_stub。这是用于静态方法跟踪的Ftrace内核配置选项。同样地，这个选项也是"fentry"内核配置中CONFIG_FPROBE的依赖项，并且还提供一个名为available_filter_functions的文件来供用户配置Ftrace跟踪功能。如果未开启此选项，则由于缺少此功能而导致bpftrace在Kprobe功能函数列表时失败。

3. CONFIG_FTRACE_SYSCALLS: 这是几乎所有Ubuntu发行版本中默认开启的内核配置选项，但在安卓中默认关闭。此外，在Pixel6及以上设备上启用该选项，并同时启用Kprobe相关选项，可能会导致设备性能下降。该内核配置会在tracefs的events目录下增加一个syscalls子目录，以支持对所有系统调用进行单独的跟踪和观测。这是一个非常有用的内核配置选项。

关于其他与eBPF相关的内核配置对影响，可以参考bcc提供的一个内核配置说明文档：https://github.com/iovisor/bcc/blob/master/docs/kernel_config.md。

10.3.3 为低版本系统打上eBPF补丁

eBPF的强大功能很大一部分来源于其内核辅助方法。在这里不得不提两个功能强大的方法：bpf_probe_read_user与bpf_probe_write_user，这两个接口允许eBPF读取与写入内存地址指定的数据，它们拥有内核一样的能力，却有着比内核高得多的稳定性，功能不可谓不强大。

大多数eBPF程序都有观测函数方法的参数的需求，对于整形的参数，数据来源于其上下文的寄存器。直接读取其值便可以。涉及到字符串或结构体类型的数据，则需要使用 bpf_probe_read_user方法来读取。如果该方

法在内核中功能欠缺，则会让eBPF程序无法实现其整体功能。然而，在arm64架构5.5版本之前的内核中就存在这种问题。由于arm64平台更新滞后，只在Linux主线内核5.5中引入了`bpf_probe_read_user`接口对arm64平台进行支持。具体补丁链接为：https://github.com/torvalds/linux/commit/358fdb456288d48874d44a064a82bfb0d9963fa0。该补丁内容非常繁多，修改了17个文件，包括导出bpf.h头文件接口声明、添加bpf/core.c接口实现逻辑以及更新与内存相关的接口等，共计597处修改和197处删除。

要在安卓11内核5.4上使用`bpf_probe_read_user`接口，需要对内核代码进行向前移植（backport）操作。这一操作难度可控，在相应位置按照补丁中的代码进行添加和修改即可。更低版本如4.19和4.14则更加麻烦，主要因为主线内核大版本不同造成接口变化较大。版本5的内核在多线程同步下做了很多精细工作，而这些在内核4中是没有的，所以整个向前移植会变得更加困难。我自己尝试过将补丁应用于安卓10模拟器上的4.14内核和安卓11模拟器上的5.4内核，并使其正常运行。

针对5.4内核补丁已经有网络上讨论并提供了具体解决方案。有人发布了适用于安卓5.4内核的补丁代码，并提供了完成补丁后的分支代码。当然，大部分人关心如何利用修复后的结果而不是补丁的具体内容。因此，后者更受欢迎。这里提供一个已经修改好的方案链接：https://github.com/HorseLuke/aosp_android_common_kernels/tree/android-11-5.4-bpf_probe_read_user。

编译内核采用官方的build.sh脚本。执行下面的命令，下载内核代码。

```
mkdir -p android-kernel && pushd android-kernel
repo init -u https://android.googlesource.com/kernel/manifest -b common-android11-
5.4
echo Syncing code.
repo sync -cj8
```

下载完成后，做一个内核代码替换，执行下面的命令：

```
rm -rf common
git clone https://github.com/feicong/aosp_android_common_kernels common
cd common
git checkout android-11-5.4-bpf_probe_read_user
```

最后，执行下面的命令编译生成内核。

```
BUILD_CONFIG=common-modules/virtual-device/build.config.goldfish.aarch64
SKIP_MRPROPER=1 CC=clang build/build.sh -j12
```

如果读者不关心内核与编译，可以到这里下载编译好的内核文件。https://github.com/feicong/ebpf-course/releases/tag/latest。比如，安卓模拟器5.4内核，其名字为android-arm64-common-5.4-kernelgz开头的zip文件，解压密码：qq121212。下载后，将其放到模拟器镜像目录下，替换kernel文件即可。

10.4 测试eBPF功能

安卓设备环境准备好后，需要bcc与bpftrace等工具来测试eBPF功能。这里使用的工具名叫ExtendedAndroidTools。将下载的bpftools推送到设备上。执行如下命令。

```
$ adb push bpftools /data/local/tmp/
```

执行bcc工具集需要管理员权限，执行如下命令获取root shell权限。

```
$ adb root
```

bcc工具集支持主流x86_64处理器的Linux系统。而对安卓系统的支持是有限的。主要的原因是常用的工具集使用的系统调用hook点，有可能在安卓系统上不存在。在执行命令过程中，如果出现错误，需要具体的问题具体分析，找出相应的解决方法。

打开一个adb shell，然后执行如下命令，开启文件打开监控。注意，所有的工具位于share/bcc/tools/目录下。

```
$ adb shell
# cd /data/local/tmp/bpftools
# ./python3 share/bcc/tools/opensnoop
```

如果不出意外，会有打开的文件列表输出。有一些工具会用到debugfs路径，在执行命令前需要执行如下命令，加载debugfs。

```
mount -t debugfs debugfs /sys/kernel/debug
```

有一些工具内容输出采用的Ftrace提供的tracing接口-bpf_trace_printk。这个时候，需要先打开Ftrace的日志输出开关。执行如下命令即可。

```
# echo 1 > /sys/kernel/tracing/tracing_on
```

后面，想要监控输出的内容，可以执行下面的命令。

```
# cat /sys/kernel/tracing/trace_pipe
```

接下来，测试一下bpftrace工具的使用效果。bpftrace工具位于share/bpftrace/tools/目录下。执行方法与bcc一样。如尝试执行如下命令，监控命令执行操作。

```
$ adb shell
# cd /data/local/tmp/bpftools
# ./bpftrace share/bpftrace/tools/execsnoop.bt
```

```
share/bpftrace/tools/execsnoop.bt:21-23: ERROR: tracepoints not found:
syscalls:sys_enter_exec*
```

从上面的输出可以看到，在内核没有开启CONFIG_FTRACE_SYSCALLS的情况下，是没有“tracepoint/syscalls”这个类别的，而execsnoop.bt使用这个跟踪点就会报错。解决这个问题有两种方法：

1. 将tracepoint更改为kprobe，然后调整参数名字与输出。
2. 为内核开启CONFIG_FTRACE_SYSCALLS，如果设备不支持开启，可以考虑更新开发板或模拟器环境。

执行如下命令可以监控设备的TCP网络连接。

```
# ./bpftrace share/bpftrace/tools/tcpconnect.bt
Attaching 2 probes...
Tracing tcp connections. Hit Ctrl-C to end.
TIME PID COMM SADDR SPORT DADDR DPORT
```

更多工具的使用与用法见bpftrace官方的说明文档。仓库地址是：https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md。

10.5 eBPF实现安卓App动态库调用跟踪

本小节讲解如何使用eBPF开发一个完整的功能的跟踪工具。该工具名为ndksnoop。是笔者使用bpftrace实现的安卓NDK中常见的so动态库接口的跟踪工具。

整个工具分为三部分组成。头文件申明、BEGIN初始化块、Hook函数体。下面，分别进行讲解。

10.5.1 头文件的引用

新版本的bpftrace使用BTF来确定要处理的方法的参数类型、返回值与结构体类型。在没有开启支持BTF的环境中运行的话，或者Hook的第三方库没有BTF文件，只有头文件。这时，需要将使用到的类型信息通过头文件的方式引入到.bt脚本的开头。如下所示。

```
#!/usr/bin/env bpftrace
/*
 * ndksnoop trace APK .so calls.
 *      For Android, uses bpftrace and eBPF.
 *
 * Also a basic example of bpftrace.
 *
 * USAGE: ndksnoop.bt
 *
 *
 * Copyright 2023 fei_cong@hotmail.com
 * Licensed under the Apache License, Version 2.0 (the "License")
 *
 * 09-Apr-2023  fei_cong created first version for libc.so tracing.
 */
```

```
#ifndef BPFTRACE_HAVE_BTF
#include <linux/socket.h>
#include <net/sock.h>
#else
#include <sys/socket.h>
#endif
```

最开始的部分，是.bt文件的用途与版本说明信息。说明脚本开发的目的、时间、作者、功能等。 然后，根据BPFTRACE_HAVE_BTF宏判断是否支持BTF来引入不同的头文件。这里引入的是与网络相头的socket结构体相头的申明，里面涉及到的Hook点，将在下在小节进行讲解。

在这里，除了使用#include引入头文件，还可以像C语言那样直接申明类型。如typedef、#define、struct xxx{}等。

10.5.2 传入参数的处理

有时候脚本需要使用传入参数来指定变化的参数信息。例如、ndksnoop需要支持对不同的安卓App进行过滤，这里使用到的过滤参数是App相关的uid。

安卓App在安装时，会被赋予一个不变的uid数值。可以对这个值进行过滤，来Hook指定的App。比如com.android.settings也就是设置应用，它的uid为1000，shell用户的uid为2000。想要查看一个App的uid。可以在adb shell下执行如下命令。

```
# ls -an /data/data/com.android.systemui
total 36
drwx------   4 10095 10095 4096 2023-02-03 17:47 .
drwxrwx--x 139 1000  1000  8192 2023-03-16 09:32 ..
drwxrws--x   2 10095 20095 4096 2023-02-03 17:47 cache
drwxrws--x   2 10095 20095 4096 2023-02-03 17:47 code_cache
```

ls命令的-n参数，会列出目录的uid信息。上面的命令列出的是systemui包的uid信息。对于的cache与code_cache目前行可以看出，第2列的uid值为10095。

bpftrace支持解析传入参数，以$1、$2、$N来命名。只传入一个uid，则执行如下命令传入的参数在脚本中$1的值为10095。

```
# ./bpftrace ndksnoop.bt 10095
```

BEGIN块是.bt脚本的初始化部分，可以用于对传入参数进行处理。如下所示。

```
BEGIN
{
    // # ls -an /data/data/io.github.vvb2060.mahoshojo
    if ($1 != 0) {
        @target_uid = (uint64)$1;
    } else {
```

```
        @target_uid = (uint64)10095;
    }

    printf("Tracing android ndk so functions for uid %d. Hit Ctrl-C to end.\n",
@target_uid);
}
```

脚本的$1传给了@target_uid变量，前面的@表示这是一个全局变量，临时变量使用$。 当脚本没有传入参数时，$1的值为0，这个时候，可以给它一个默认的值10095，或者其它感兴趣的App的uid。

最后，使用printf方法打印输出一行调试信息。

10.5.3 Hook方法的实现

Hook用户态的程序与动态库，使用uprobe与uretprobe来实现。 uprobe负责处理方法执行前的上下文信息，uretprobe用于处理方法执行完返回时的返回值信息，通常一些输出的字符串与缓冲区信息也在这里进行处理。

以libc.so动态库的mkdir方法为例。它的Hook逻辑实现如下：

```
// int mkdir(const char *pathname, mode_t mode);
uprobe:/apex/com.android.runtime/lib64/bionic/libc.so:mkdir /uid == @target_uid/ {
    printf("mkdir [%s, mode:%d]\n", str(arg0), arg1);
}
```

//是注释，语法与C语言一样。主要是方便理解与阅读。

uprobe关键字指定进行uprobe类型的Hook。后面跟上库名或完整的库路径。在安卓系统上，bpftrace无法找到安卓apex目录下的动态库，因此，需要手动输入完整的路径。

//是过滤器，中间的内容uid == @target_uid为过滤表达式，表明，只有当表达满足时，才执行方法体内容。这里的表达式含义是：只Hook当前执行时uid为@target_uid的方法调用。uid关键字是bpftrace 的保留字，由bpftrace程序替换表示当前执行时的程序的uid。而@target_uid则上上面初始化部分设置好的目标uid，这样就完成了过滤操作。

uprobe的参数为arg0-arg5。取参数很简单，整形直接赋值就可以了！字符串类型使用str()来读取。字节数组使用buf()来读取。更多的方法参考bpftrace文档。

代码部分只有两行！就完成了一个方法的跟踪与参数值输出，实在是太方便了。

10.5.4 特殊参数与字段的处理

有一些参数，它们传入时没有传，只有在方法执行返回时才设置内容。对于这些方法，可以使用uprobe传入时保存指针，uretprobe执行时解析。如下所示，是__system_property_get()方法的Hook代码。

```
uprobe:/apex/com.android.runtime/lib64/bionic/libc.so:__system_property_get /uid
== @target_uid/ {
    @name[tid] = str(arg0);
```

```
    @val[tid] = arg1;
}

uretprobe:/apex/com.android.runtime/lib64/bionic/libc.so:__system_property_get
/uid == @target_uid/ {
    if (sizeof(@name[tid]) > 0) {
        printf("getprop [%s:0x%x:%s], ret:%d\n", @name[tid], (int32)(@val[tid]),
str(@val[tid]), retval);
    }

    delete(@name[tid]);
    delete(@val[tid]);
}
```

`__system_property_get()`用于读取属性系统的值。传入的第一个参数为字符串类型的key，第二个参数为返回的内容。在`uprobe`中，使用`str()`读取了key的内容。而`arg1`存放的值，只保存了它的指针。在`uretprobe`中会对其进行`str()`内容读取。注意，最后需要调用`delete()`来删除这两个变量，因为它们是与`tid`相关的线程变量，执行后不删除，会让内存消耗越来越多，直到程序崩溃。

还有一类是比较复杂的结构体。比如`connect()`方法的第二个参数`struct sockaddr`，想要从这个参数中取得IP地址。可以使用如下方法。

```
// int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
uprobe:/apex/com.android.runtime/lib64/bionic/libc.so:connect /uid == @target_uid/
{
    $address = (struct sockaddr *)arg1;
    if ($address->sa_family == AF_INET) {
        $sa = (struct sockaddr_in *)$address;
        $port = $sa->sin_port;
        $addr = ntop($address->sa_family, $sa->sin_addr.s_addr);
        printf("connect [%s %d %d]\n", $addr, bswap($port), $address->sa_family);
    } else {
        $sa6 = (struct sockaddr_in6 *)$address;
        $port = $sa6->sin6_port;
        $addr6 = ntop($address->sa_family, $sa6->sin6_addr.s6_addr);
        printf("connect [%s %d %d]\n", $addr6, bswap($port), $address->sa_family);
    }
}
```

这是一种类C语言的语法，通过结构体指针强转的方式，来处理结构体中的字段信息。将字节数组的内容转换成IP地址，使用`ntop()`方法，而网络字节序的转换，使用`bswap()`方法。

10.5.5 效果展示

执行对`uid`为1000的`libc.so`方法调用跟踪。效果如下所示。

```
emulator64_arm64:/data/local/tmp/bpftools # ./bpftrace ./ndksnoop.bt 1000
WARNING: Cannot parse DWARF: libdw not available
```

```
Attaching 64 probes...
Tracing android ndk so functions for uid 1000. Hit Ctrl-C to end.
__system_property_find [net.qtaguid_enabled]
getenv [ANDROID_NO_USE_FWMARK_CLIENT]
getenv [ANDROID_NO_USE_FWMARK_CLIENT]
__system_property_find [persist.log.tag.android.hardware.vibrator-service.example]
__system_property_find [log.tag.android.hardware.vibrator-service.example]
__system_property_find [persist.log.tag]
__system_property_find [log.tag]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [persist.log.tag.AutofillManagerService]
__system_property_find [log.tag.AutofillManagerService]
__system_property_find [persist.log.tag.ActivityTaskManager]
__system_property_find [log.tag.ActivityTaskManager]
__system_property_find [debug.force_rtl]
__system_property_find [debug.force_rtl]
open [/proc/uid_procstat/set]
opendir [/proc/1041/task]
open [/proc/1041/timerslack_ns]
open [/proc/1048/timerslack_ns]
open [/proc/1050/timerslack_ns]
open [/proc/1054/timerslack_ns]
open [/proc/1056/timerslack_ns]
open [/proc/1057/timerslack_ns]
open [/proc/1058/timerslack_ns]
open [/proc/1059/timerslack_ns]
open [/proc/1060/timerslack_ns]
open [/proc/1061/timerslack_ns]
open [/proc/1063/timerslack_ns]
open [/proc/1064/timerslack_ns]
open [/proc/1066/timerslack_ns]
open [/proc/1078/timerslack_ns]
open [/proc/1079/timerslack_ns]
open [/proc/1107/timerslack_ns]
open [/proc/1225/timerslack_ns]
open [/proc/1241/timerslack_ns]
open [/proc/1282/timerslack_ns]
open [/proc/1341/timerslack_ns]
open [/proc/1361/timerslack_ns]
open [/proc/1372/timerslack_ns]
open [/proc/1374/timerslack_ns]
__system_property_find [debug.renderengine.capture_skia_ms]
open [/proc/1375/timerslack_ns]
open [/proc/1378/timerslack_ns]
open [/proc/1490/timerslack_ns]
open [/proc/1817/timerslack_ns]
open [/proc/2226/timerslack_ns]
open [/proc/2227/timerslack_ns]
open [/proc/2250/timerslack_ns]
open [/proc/2521/timerslack_ns]
open [/proc/6029/timerslack_ns]
opendir [/proc/889/task]
```

```
open [/proc/889/timerslack_ns]
open [/proc/902/timerslack_ns]
open [/proc/911/timerslack_ns]
open [/proc/913/timerslack_ns]
......
open [/proc/6256/timerslack_ns]
open [/proc/6258/timerslack_ns]
open [/proc/6260/timerslack_ns]
__system_property_find [persist.log.tag.AutofillManagerService]
__system_property_find [log.tag.AutofillManagerService]
__system_property_find [persist.log.tag.BpBinder]
__system_property_find [log.tag.BpBinder]
__system_property_find [persist.log.tag]
__system_property_find [log.tag]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [persist.log.tag.goldfish-address-space]
__system_property_find [log.tag.goldfish-address-space]
__system_property_find [persist.log.tag]
__system_property_find [log.tag]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
getenv [ART_APEX_DATA]
getenv [ANDROID_DATA]
getenv [ANDROID_DATA]
faccessat [/system_ext/priv-app/Launcher3QuickStep]
getenv [ART_APEX_DATA]
getenv [ART_APEX_DATA]
open [/system_ext/priv-app/Launcher3QuickStep/oat/arm64/Launcher3Quic]
open [/system_ext/priv-app/Launcher3QuickStep/oat/arm64/Launcher3Quic]
open [/system_ext/priv-app/Launcher3QuickStep/Launcher3QuickStep.apk]
readlink [/proc/self/fd/426 /system_ext/priv-
app/Launcher3QuickStep/Launcher3QuickStep.apk 0]
readlink [/proc/self/fd/380 /system_ext/priv-
app/Launcher3QuickStep/Launcher3QuickStep.apk 0]
open [/system_ext/priv-app/Launcher3QuickStep/Launcher3QuickStep.apk]
faccessat [/data/misc/iorapd/com.android.launcher3/31/com.android.launcher]
faccessat [/proc/1041/stat]
open [/proc/1041/stat]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
......
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
faccessat [/data/system_ce/0/snapshots]
faccessat [/data/system_ce/0/snapshots/135.proto.bak]
open [/data/system_ce/0/snapshots/135.proto.new]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
open [/data/system_ce/0/snapshots/135.jpg]
__system_property_find [debug.renderengine.capture_skia_ms]
open [/data/system_ce/0/snapshots/135_reduced.jpg]
__system_property_find [debug.renderengine.capture_skia_ms]
__system_property_find [debug.renderengine.capture_skia_ms]
```

```
^C

@target_uid: 1000

emulator64_arm64:/data/local/tmp/bpftools #
```

目前，Hook监控了libc.so共计64个接口方法。后面，可以扩展ndksnoop，实现对其它方法与其它库的方法跟踪。这种方式Hook最大的好处是输出内容中，没有多余的信息，所有的输出都是目标进程的行为捕获。缺点也是有的，那就是无法捕获直接使用系统调用方式执行的方法。

10.6 小结

本节主要介绍了eBPF相关的信息，以及在安卓系统上配置eBPF开发与运行环境的方法。最后，通过ndksnoop工具的代码示例，详细讲解了如何跟踪分析安卓系统中的动态库调用。

在学习系统定制与软件安全过程中，每个工具和技术方案都有其优势和限制。因此，在实际应用中，我们应根据具体情况结合不同方案，并充分利用各自长处、避免短板，以达到最终目标。