何为ebpf

ebpf 是linux内核的一个组件，最初用于实现数据链路层的包过滤功能和解决“如何在内核沙箱中安全的运行一段程序”这一问题，后续经过各种迭代添加了大量新的监视功能和辅助函数，目前已经是linux中内核监控的强大工具

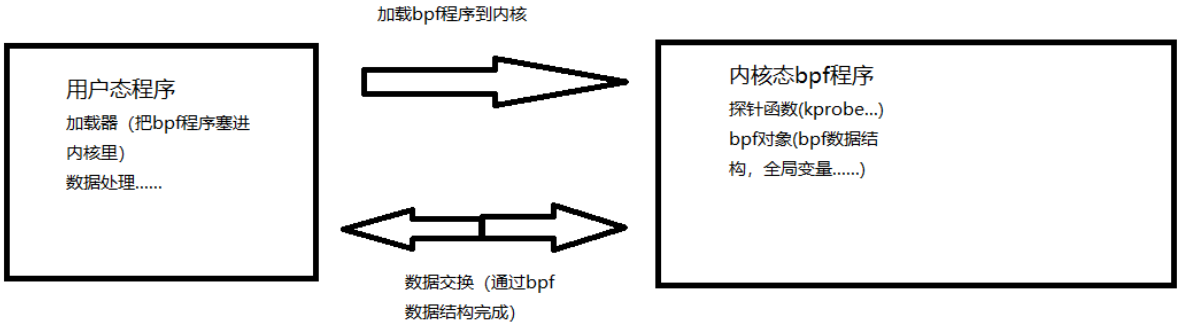
在逆向中有什么用

看雪上有位师傅说过，风控对抗要找 边界值 ，也就是敏感数据的最初来源，在java层就是安卓 binder 通信,native层则是 syscall ，打个比方对root的检测，以前就是调用 libc 的 open 去各种目录下扫有没有 su ，后来用户态 inline hook 被 frida 之类的玩烂了，就搞内联 libc ，然后搞自实现 open 和 svc 直接调用 openat ，这里的 svc 指的是arm的 svc 指令，根据调用号直接向内核发起系统调用， libc 大量和系统交互的函数，最终也是调用 svc 发起的系统调用，当水位上升到这个高度的时候用户态hook显得有点不够用，毕竟要在安全sdk大量的代码里找到一小块 svc 调用几乎是不可能的，更别说很可能会漏，而且用户态hook大多都是侵入式的，为了防止检测又要各种找，而往往这些检测又通过 svc 发起，陷入一个死循环

在这种场景下， ebpf 成为了一个完美的方案， ebpf 在所有 syscall 的入口点 sys_enter 默认就有一个监测点，可以监视绝大多数常用 syscall 触发时的参数和上下文(寄存器，pc,sp,任意读内存),甚至可以在触发的时候发送信号，更关键的是在监视系统调用这一场景下， ebpf 对用户态完全无痕，因为所有的数据采集完全在内核中完成

ebpf软件的组成结构

由于其在内核中运行的特殊性， ebpf 软件实际上可以认为由两部分组成，分别是用户态和内核态程序



用户程序负责通过 加载器 将bpf二进制文件运行在内核中，并且处理bpf程序采集的信息，这里有相当多

的加载器框架，完成的都是加载bpf程序和与bpf对象交互这两项工作
内核态程序可以认为是一个比较特殊的elf目标文件，其主要由探针程序和bpf对象组成，这些部分都要通过 `SEC` 宏将其指定在对应区段，这样内核才能正确加载这些对象以及用户态程序才能正确与这些对象交互
这个内核态elf文件在进入内核前还要经过内核的验证器的检查，确保其安全性

如何使用ebpf

`ebpf` 生态这几年发展迅速，有非常多不同语言的框架，主流的有如下几种

- python: bcc
- rust: aya
- c/c++: libbpf
- go: cilium/ebpf-go ebpf-manager(对前者的再封装)

这里 `libbpf` 是对原生加载指令的封装，而其他框架则是对 `libbpf` 的再封装， `bcc` 框架采用动态编译的方案，在手机环境中配置 `bcc` 会非常困难（缺少包管理器），其余都是静态编译方案，这里 `ebpf-go` 保留了大部分 `libbpf` 的风格

准备工作

搞一个wsl
安装go，libbpf
准备bpftool，并从手机中获取
vmlinux.h `bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h` (这是内核所有符号的定义文件，在内核态程序编写时使用)

内核态程序编写

内核态程序只能通过c语言编写，且不能使用 `libc` 或其他 `builtin` 函数，只能使用内核导出的函数或内核 `bpf` 库中的函数

探针函数

```
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include "vmlinux.h"
SEC("tp_btf/sys_enter")
int BPF_PROG(sys_enter, struct pt_regs *regs, long id) {
  // bpf_printk("pid : %d   target : %d \n", (bpf_get_current_pid_tgid() >> 32),
  //            targetPid);
  if ((bpf_get_current_pid_tgid() >> 32) != targetPid) {
    return 0;
  }
  if (!filter_syscall(id))
    return 0;
  if (!filter_MemoryRange(regs->pc))
    return 0;
  int tid = bpf_get_current_pid_tgid() & 0xFFFFFFFF;
  u64 *lastHash = bpf_map_lookup_elem(&tidStateMap, &tid);
  u64 hash = calc_syscall_hash(regs);
  bpf_printk("syscall hash: %llu ", hash);
  bpf_printk("x0: %llx x1: %llx x2: %llx x3:\n", regs->regs[0], regs->regs[1],
             regs->regs[2]);
  if (lastHash != NULL && *lastHash == hash)
    return 0;
  bpf_map_update_elem(&tidStateMap, &tid, &hash, BPF_ANY);
  bpf_send_signal(SIGSTOP);

  // prepare the stack data
  struct sysEnterData_noStack *data =
      bpf_ringbuf_reserve(&sysEnterRb, sizeof(struct sysEnterData_noStack), 0);
  if (data == NULL) {
    bpf_printk("Failed to reserve space in ring buffer\n");
    return 0;
  }
  // 这里把相关的数据读取一下
  int err = read_nostack_data(regs, id, data);
  if (err) {
    bpf_printk("err reading data\n");
    bpf_ringbuf_discard(data, 0);
    return 0;
  }
  bpf_ringbuf_submit(data, 0);
```

```
  // bpf_send_signal(SIGSTOP);
  return 0;
}
```

```c
SEC("uprobe/common_uprobe")
int common_uprobe(struct pt_regs *ctx) {
  struct uprobeCommonData *data =
      bpf_ringbuf_reserve(&uprobeRb, sizeof(struct uprobeCommonData), 0);
  if (!data) {
    bpf_printk("fail to reserve ringbuf space (uprobe common)\n");
    return 0;
  }
  u64 regCollectSettingMask = bpf_get_attach_cookie(
      ctx); // 直接拿bpf_cookie传配置掩码，这样正好解决了字符串不好作为hashmap映射的问题
  data->mask = regCollectSettingMask;
  bpf_printk("uprobe hit at 0x%llx\n", ctx->pc);
  data->pc = ctx->pc;
  data->sp = ctx->sp;
  data->tid = bpf_get_current_pid_tgid() & 0xFFFFFFFF;
  bpf_printk("uprobe pc:%llx sp:%llx tid:%d\n", data->pc, data->sp, data->tid);
  if (bpf_probe_read_kernel(data->regs, sizeof(data->regs), ctx->regs)) {
    bpf_printk("read reg fail\n");
    bpf_ringbuf_discard(data, 0);
    return 0;
  }
  bpf_printk("read reg success\n");
  u8 strCollectCnt = 0;
  if (data->mask != 0) {
    // bpf_loop(31, reg_str_read_helper, data, 0);
    for (int i = 0; i < 31 && strCollectCnt < 8;
         i++) { // 这里只采前8个设置为采集的，多余的直接抛弃
      if (data->mask & (1 << i)) {

        bpf_probe_read_user(data->buf[strCollectCnt],
                            sizeof(data->buf[strCollectCnt]),
                            (void *)(data->regs[i]));
        bpf_printk("collect str for x%d ('0x%02x,0x%02x,0x%02x')\n", i,
                   data->buf[strCollectCnt][0], data->buf[strCollectCnt][1],
                   data->buf[strCollectCnt][2]);
        strCollectCnt++;
      }
    }
  }
  bpf_send_signal(SIGSTOP);
  bpf_ringbuf_submit(data, 0);
```

```
  return 0;
}
```

maps

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
struct {
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, PAGE_SIZE * 65536);
} uprobeRb SEC(".maps"); // 传输uprobeCommon数据

struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, 0x1c3);
  __type(key, u32);
  __type(value, u8);
} targetSyscalls SEC(".maps");

struct {
  __uint(type, BPF_MAP_TYPE_HASH);
  __uint(max_entries, 1024);
  __type(key, u32);
  __type(value, u64);
} stackBaseAddrTable SEC(".maps");
```

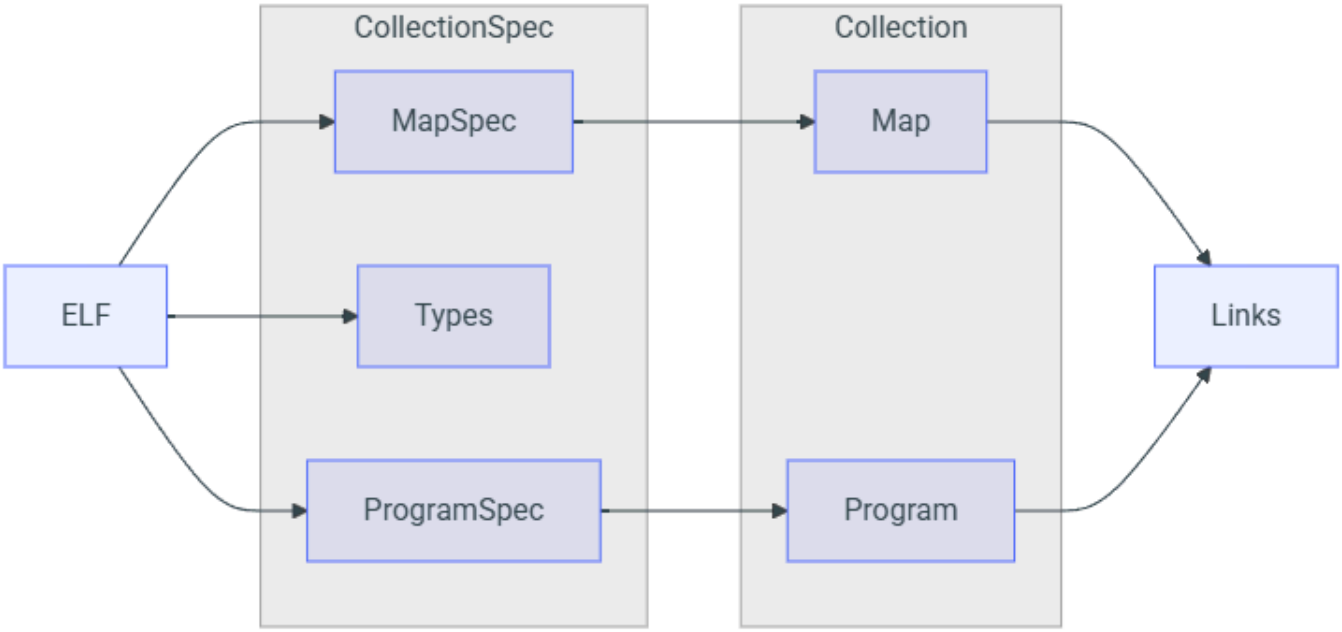
采用bpf2go将其转换为go框架

bpf2go 是一个用于直接将c源文件编译成.o二进制文件并生成对应的go语言框架代码，框架代码指的是一系列与bpf对象交互的预生成的接口以及将bpf程序中定义的类型导出其对应的go声明

```
go run github.com/cilium/ebpf/cmd/bpf2go \
-go-package bpfloader \
-target bpfel \
-cc "$(BPF_CLANG)" \
-cflags "$(BPF_CFLAGS)" \
-type sysEnterData_noStack \
-type uprobeCommonData \
Probes_ ../kernel/probes.c
```

用户态程序编写

对于 ebpf-go 这一加载器框架，bpf二进制文件被划分为下图所示的部分



Spec 指的是从elf文件中解析出的bpf对象的元数据定义， Collection 是真正加载到内核中的部分(支持选择性加载)，而 link 则是讲已经在内核中的bpf对象与用户侧程序进行连接

加载bpf对象

对于简单的程序，可以直接使用之前生成的框架中预生成的函数一次性加载所有对象

```
func LoadProbes_Objects(obj interface{}, opts *ebpf.CollectionOptions) error {
        spec, err := LoadProbes_()
        if err != nil {
                return err
        }

        return spec.LoadAndAssign(obj, opts)
}
```

对于对象数比较多（尤其是大量的缓冲区数据结构）的程序，可以选择性的加载对象，只需要先获取整个elf文件的 spec （这也有自动生成的函数替我们完成），然后定义自己的bpf对象组结构体后用 spec.LoadAndAssign 方法手动加载对象即可， ebpf-go 采用go语言结构体注释驱动的方式自动解析需要加载哪些对象

```
type sysEnterObjsType struct {
	TargetPid         *ebpf.Variable `ebpf:"targetPid"`
	SysEnter          *ebpf.Program  `ebpf:"sys_enter"`
	SysEnterRb        *ebpf.Map      `ebpf:"sysEnterRb"`
	TargetSyscalls    *ebpf.Map      `ebpf:"targetSyscalls"`
	TidStateMap       *ebpf.Map      `ebpf:"tidStateMap"`
	TargetMemoryRange *ebpf.Map      `ebpf:"targetMemoryRange"`
}


func DoCommonBpfInit() { //加载bpf变量并解锁内存限制，，初始化bpf_spec,这里开销很小，直接做就行
	if err := rlimit.RemoveMemlock(); err != nil { // remove kernel memory lock limit
		log.Fatal(err)
	}

	var err error
	spec, err = LoadProbes_()
	if err != nil {
		log.Fatalf("loading spec: %v", err)
	}
	if err := LoadProbes_Objects(BpfConst, getDebugOptForLoad()); err != nil {
		log.Fatalf("loading objects: %v", err)
	}

	debug.Debug("bpf var loaded\n")
}

func AttachTp_sysEnter() {
	var err error
	err = spec.LoadAndAssign(SysEnterObj, getDebugOptForLoad())
	if err != nil {
		log.Fatal(err)
	}
	SysEnterTp, err = link.AttachTracing(link.TracingOptions{Program: SysEnterObj.SysEnter
	if err != nil {
		log.Fatal(err)
	}
	debug.Debug("tp attached\n")
}
```

启用tracepoint

启用 tracepoint 类型的程序只需要调用 link.AttachTracing 这个方法即可，这里以 tp_btf 为例，在 TracingOptions 结构体中填充bpf程序对象和附加类型即可，高版本内核还可以添加 cookie 字段

启用uprobe

uprobe 涉及到对maps的解析，这里不多赘述，唯一需要注意的是uprobe注入的位置并不是maps文件中 最小的地址+偏移 这么简单，而是要考虑maps中的 段基址 参数，具体计算方法如下

```
func ConvertSimpleOffsetToUprobeOffset(soName string, offset uint64) uint64 {
        debug.Debug("%s EXE_BASE: %x,offset:  %x SoBaseAddr: %x, SoSegmentBase: %x\n", soName,
        return EXE_BASE + offset - SoBaseAddr[soName] + SoSegmentBase[soName]
}
```

除此之外 uprobe 只需要提供pid和目标二进制文件路径就能愉快的hook了，另外如果目标函数是导出的话也可以直接提供符号名不提供地址

```
uprobeOps := link.UprobeOptions{}
process, err := link.OpenExecutable(mapsparser.GetSoPath(setting.SoName, targetPid))
if err != nil {
        log.Fatal(err)
}
uprobeOps.PID = targetPid
uprobeOps.Cookie = uint64(setting.Reg_read_mask)<<32 | uint64(setting.Str_read_mask) // 高32位是
var res link.Link
if setting.Symbol == "" {
        uprobeOps.Address = mapsparser.ConvertSimpleOffsetToUprobeOffset(setting.SoName, setti
        res, err = process.Uprobe("", UprobeCommonObj.CommonUprobe, &uprobeOps)
} else {
        res, err = process.Uprobe(setting.Symbol, UprobeCommonObj.CommonUprobe, &uprobeOps)
}
if err != nil {
        if setting.Symbol == "" {
                setting.Symbol = "offset_0x" + strconv.FormatUint(setting.Offset, 16)
        }
        fmt.Printf("err uprobe [%s] %v\n", setting.Symbol, err)
        os.Exit(0)
}
```

连接ringbuf缓冲区接收数据

ebpf中大量数据的通信方式有传统的 perf_buffer 和现代的 ringbuf 两种，其中 perf_buffer 是每个cpu独占缓冲区，无法保证全局时序且在溢出场景下会导致旧数据被覆盖，且在内存效率上非常笨重（不忙碌的cpu也被分配了内存）， ringbuf 则是所有cpu共享的一个单一缓冲区，严格按照FIFO传递数据且在溢出时丢弃新数据而不是覆盖旧数据

在内核侧，我们只需要通过 bpf_ringbuf_reserve 预先从缓冲区中分配一块内存，然后往里面填充数据，最后用 bpf_ringbuf_submit 发送或者用 bpf_ringbuf_discard 销毁即可

```
struct {
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, PAGE_SIZE * 65536);
} sysEnterRb SEC(".maps");
```

ringbuf 也属于bpf数据结构，定义在 .maps 段， max_entries 参数的含义是缓冲区容量，单位是字节，注意这里要求页对齐

```
struct sysEnterData_noStack *data =
      bpf_ringbuf_reserve(&sysEnterRb, sizeof(struct sysEnterData_noStack), 0);
int err = read_nostack_data(regs, id, data);
if (err) {
  bpf_printk("err reading data\n");
  bpf_ringbuf_discard(data, 0);
  return 0;
}
bpf_ringbuf_submit(data, 0);
```

用户侧很简单，直接调 ringbuf.NewReader 把框架中的rb对象传进去即可

```
func LoadRb(reader **ringbuf.Reader, rb *ebpf.Map) {
          var err error
          *reader, err = ringbuf.NewReader(rb)
          if err != nil {
                    log.Fatal(err)
          }
}
```

```
var sysEnterData bpfloader.Probes_SysEnterDataNoStack
data, err := bpfloader.SysEnterRb.Read()
if err != nil {
          fmt.Printf("reading sysEnter err: %v\n", err)
          continue
}
if err := binary.Read(bytes.NewBuffer(data.RawSample), binary.LittleEndian, &sysEnterData); err
          fmt.Printf("reading sysEnterData err: %v\n", err)
          continue
}
```

这里直接调 `reader` 的 `Read` 方法就可以从rb中读二进制数据，注意这里是纯二进制数据，所以要进行序列化操作才能还原，这里之前框架中导出的bpf类型声明就有用了，可以方便的重新恢复结构体，rb的读取非常智能，每当接收到新数据其会向连接的消费者发送信号提示读取，所以也不用管数据到底怎么传的，就这么简单

编译

注意目前 `ebpf-go` 库并不认安卓是 `linux` ，所以请自行将库中 `/internal/platform/platform.go` 中的 `IsLinux` 这一项改为 `IsLinux   = runtime.GOOS == "linux" || runtime.GOOS == "android"`

```
GOOS=android \
GOARCH=arm64 \
go build
```