USMA：Share Kernel Code with Me

Yong Liu, Xiaodong Wang, Jun Yao
360 Alpha Lab

Abstract

The gadget is a mainstream attack method in the Linux kernel. It requires attackers to find available gadgets based on the vulnerability. This is a time-consuming task, and sometimes it is impossible to find a suitable gadget. In addition, the CFI mitigation (such as the PAC on ARM) is merged into the kernel, and it limits this method.

This paper discloses a new cross-platform general attack method called USMA. It allows the process to map the kernel memory and modify the kernel code. With it, we can break the CFI mitigations and get ROOT privilege.

Contents 1

1 Introduction 4
2 Double Free In Packet 4
3 The ROP Exploitation 6
4 USMA 9
5 Conclusion 10

1 Introduction

The gadget is a mainstream attack method in the Linux kernel. It requires attackers to find available gadgets based on the vulnerability. This is a time-consuming task, and sometimes it is impossible to find a suitable gadget. In addition, the CFI mitigation (such as the PAC on ARM) is merged into the kernel, and it limits this method.

This paper discloses a new attack method called USMA, which is short for **U**ser **S**pace **M**apping **A**ttack. It allows the process to map the kernel memory and modify the kernel code. With it, we can break the CFI mitigations and get ROOT privilege.

The contributions of the paper can be summarized as follows:

1. We disclose a 0day vulnerability that affects Linux kernel 4.14 version and later.
2. We used two different methods to achieve local privilege escalation.
3. We public a cross-platform and universal attack method.

The rest of this paper is organized as follows: In chapter 2, we describe the 0day vulnerability in packet (the code based on v5.11.20). In chapter 3, we introduce how to use gadgets to get ROOT privilege. In chapter 4, we show how to use the USMA to exploit. In addition, we explain why the USMA is a cross-platform and universal attack method. We conclude our paper in chapter 5.

2 Double Free in Packet

The vulnerability is in the packet socket module, which is used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer [1]. In the packet socket, user can set the data buffer through packet_set_ring():

```
/net/packet/af_packet.c

4292 static int packet_set_ring(sk, req_u, closing, tx_ring)
4294 {
4317       if (req->tp_block_nr) {
4362              order = get_order(req->tp_block_size);
4363              pg_vec = alloc_pg_vec(req, order);
4366              switch (po->tp_version) {
4367              case TPACKET_V3:
4369                     if (!tx_ring) {
4370                            init_prb_bdqc(po, rb, pg_vec, req_u);
4371                     }
4390              }
4391       }
4414              if (closing || atomic_read(&po->mapped) == 0) {
4417                     swap(rb->pg_vec, pg_vec);
4418                     if (po->tp_version <= TPACKET_V2)
4419                            swap(rb->rx_owner_map, rx_owner_map);
4435              }
4450 out_free_pg_vec:
4451       bitmap_free(rx_owner_map);
4452       if (pg_vec)
4453              free_pg_vec(pg_vec, order, req->tp_block_nr);
4456 }
```

It allocates the buffer according to the parameters passed by the user (line 4363). If the version equals TPACKET_V3, the init_prb_bdqc() is called (line 4370). And the packet_ring_buffer.prb_bdqc.pkgdq holds a reference of the pg_vec (line 584):

```
/net/packet/af_packet.c

573 static void init_prb_bdqc(po, rb, pg_vec, req_u)
577 {
578       struct tpacket_kbdq_core *p1 = GET_PBDQC_FROM_RB(rb);
579       struct tpacket_block_desc *pbd;
583       p1->knxt_seq_num = 1;
584       p1->pkbdq = pg_vec;
603       prb_init_ft_ops(p1, req_u);
604       prb_setup_retire_blk_timer(po);
605       prb_open_block(p1, pbd);
606 }
```

If the tpacket_req.tp_block_nr equals 0, there is no pg_vec allocated, and the old one is freed (line 4453). However, the packet_ring_buffer.prb_bdqc.pkgdq still references the freed pg_vec. If we

now change the socket version to TPACKET_V2 and set the buffer, the freed pg_vec is freed again (line 4451). Because the kernel confuses the rx_owner_map and the prb_bdqc.pkgdq (line 74 and line 18):

```
/net/packet/internal.h

59 struct packet_ring_buffer {
60    struct pgv *pg_vec;
73    union {
74          unsigned long *rx_owner_map;
75          struct tpacket_kbdq_core prb_bdqc;
76    };
77 };

17 struct tpacket_kbdq_core {
18    struct pgv *pkbdq;
19    unsigned int feature_req_word;
20    unsigned int hdrlen;
21    unsigned char reset_pending_on_curr_blk;
22    unsigned char delete_blk_timer;
52    struct timer_list retire_blk_timer;
53 };
```

3 The ROP Exploitation

This exploit is divided into two steps:

1. Leak the information of kernel address.
2. Modify the credential of the process to get ROOT privilege.

Both of them need to trigger the vulnerability separately. By choosing different victim objects, we can achieve the above goals.

3.1 Information Expose

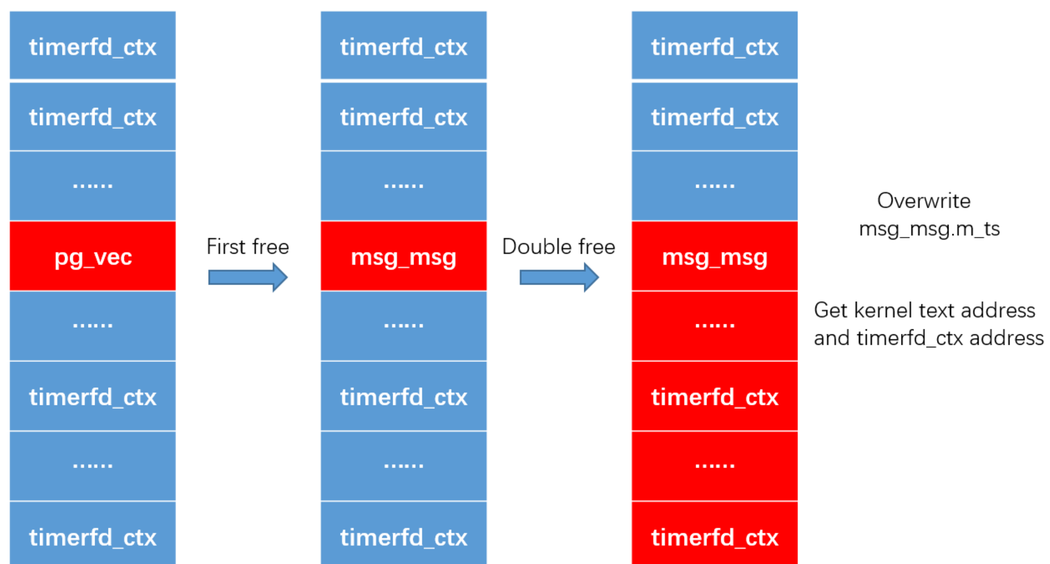
We choose the msg_msg as the victim object for the following reasons:

1. It has the m_ts field, which is used to describe the size of the buffer.
2. The process can read the contents of the buffer.

By modifying the m_ts field of msg_msg, we can read the contents of the heap out of bounds (line 128). This process occurs in the copy_msg():

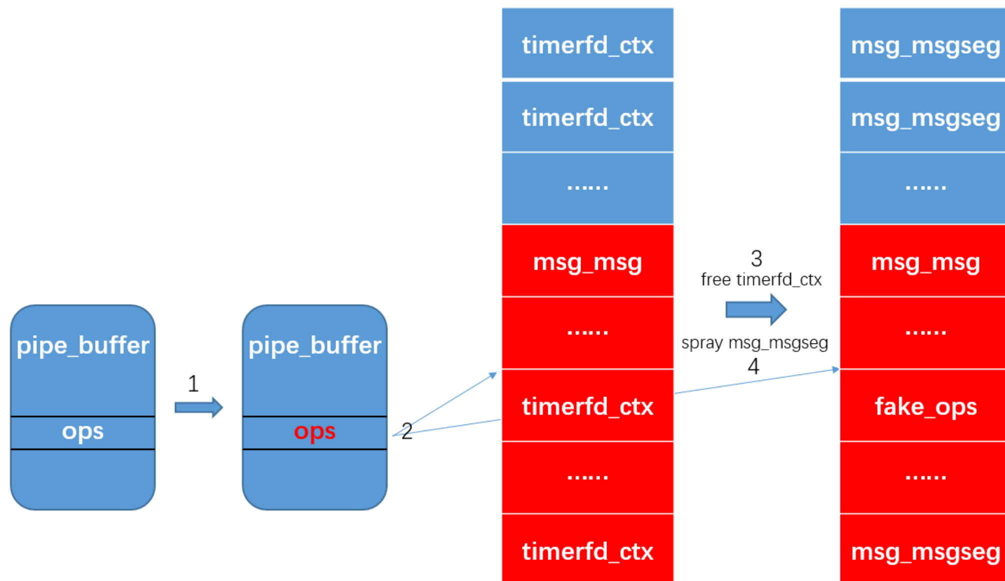
```
/ipc/msgutil.c
    118 struct msg_msg *copy_msg(src, dst)
    119 {
    121             size_t len = src->m_ts;
    127             alen = min(len, DATALEN_MSG);
    128             memcpy(dst + 1, src + 1, alen);
    129
    130             for (dst_pseg = dst->next, src_pseg = src->next;
    131                     src_pseg != NULL;
    132                     dst_pseg = dst_pseg->next, src_pseg = src_pseg->next) {
    133
    134                     len -= alen;
    135                     alen = min(len, DATALEN_SEG);
    136                     memcpy(dst_pseg + 1, src_pseg + 1, alen);
    137             }
    142             return dst;
    143 }
```

If the timerfd_ctx object locates after the buffer, the heap address and kernel symbol address can be leaked. The layout of heaps is shown in the following:



3.2 ROP

The roadmap of privilege escalation is shown as following:



The whole process is divided into four steps:

1. We select the pipe_buffer as a victim object, because it includes a pointer of the pipe_buf_operations. And we can hijack the control flow by modifying the function pointer in pipe_buf_operations.
2. We use the msg_msgseg object to spray the pipe_buffer after it is freed. And the ops pointer can be controlled by us.
3. We need to construct a fake pipe_buf_operations object and make the ops point to it. As the address of the SLAB used by timerfd_ctx is leaked, we prefer to construct the fake pipe_buf_operations on it. So, we free the timerfd_ctx object.
4. We use the msg_msgseg to spray the target SLAB.

Now we can hijack the control flow of the kernel and force the kernel to run the following gadgets [2]:

```
push rsi; jmp qword ptr [rsi + 0x39];
pop rsp; pop r15; ret;
add rsp, 0xd0; ret;
pop rdi; ret; // 0
prepare_kernel_cred;
pop rcx; ret; // 0
test ecx, ecx; jne 0xd8ab5b; ret;
mov rdi, rax; jne 0x798d21; xor eax, eax; ret;
commit_creds;
mov rsp, rbp; pop rbp; ret;
```

The gadgets change the address of stack and modify the credential of the process by calling commit_creds(prepare_kernel_cred(0)).

4 The USMA

In order to speed up the data transmission between the process and kernel, the packet maps the buffer to the process address space. Then the process can directly read and write the buffer. This is done in the packet_mmap():

```
/net/packet/af_packet.c

4458 static int packet_mmap(file, sock, vma)
4460 {
4491     for (rb = &po->rx_ring; rb <= &po->tx_ring; rb++) {
4495         for (i = 0; i < rb->pg_vec_len; i++) {
4496             struct page *page;
4497             void *kaddr = rb->pg_vec[i].buffer;
4500             for (pg_num = 0; pg_num < rb->pg_vec_pages; pg_num++) {
4501                 page = pgv_to_page(kaddr);
4502                 err = vm_insert_page(vma, start, page);
4505                 start += PAGE_SIZE;
4506                 kaddr += PAGE_SIZE;
4507             }
4508         }
4509     }
4517     return err;
4518 }
```

The packet_mmap() converts the kernel address in pg_vec into the page (line 4501), and maps the page to the process (line 4502). If we can modify the kernel address in pg_vec, then the kernel text can be mapped to the process. As the mapping attribute is RW, the process can modify the kernel text.

We use the ret2dir [3] to tamper with the pg_vec, and make the kernel map the memory of setresuid() to the process. Then we can modify its logic to achieve privilege escalation (line 659).

```
/kernel/sys.c

631 long __sys_setresuid(uid_t ruid, uid_t euid, uid_t suid)
632 {
659     if (!ns_capable_setid(old->user_ns, CAP_SETUID)) {
660         if (ruid != (uid_t) -1 && !uid_eq(kruid, old->uid) &&
661             !uid_eq(kruid, old->euid) && !uid_eq(kruid, old->suid))
662             goto error;
663         if (euid != (uid_t) -1 && !uid_eq(keuid, old->uid) &&
664             !uid_eq(keuid, old->euid) && !uid_eq(keuid, old->suid))
665             goto error;
666         if (suid != (uid_t) -1 && !uid_eq(ksuid, old->uid) &&
667             !uid_eq(ksuid, old->euid) && !uid_eq(ksuid, old->suid))
668             goto error;
669     }
694 }
```

In fact, the pg_vec can be used to convert the vulnerabilities on the heap (the UAF, Double Free, the OOB) into read-write primitives. Because the pg_vec object can be used to occupy the heaps with various sizes:

```
/net/packet/af_packet.c

4267 static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
4268 {
4269     unsigned int block_nr = req->tp_block_nr;
4270     struct pgv *pg_vec;

4273     pg_vec = kcalloc(block_nr, sizeof(struct pgv), …);
4290 }
```

As the block_nr is controlled by us (line 4269). We can occupy the certain size of heap.

5 Conclusion

This article first discloses a 0day vulnerability, and describes how to use two different methods to achieve local privilege escalation. The second method called USMA is a universal, cross-platform attack method that can convert most of the heap problems (the UAF, Double Free, and the OOB) into read and write primitives to achieve privilege escalation.

References

[1]. https://man7.org/linux/man-pages/man7/packet.7.html
[2]. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html
[3]. https://cs.brown.edu/~vpk/papers/ret2dir.sec14.pdf