Securitate Software

VI Vulnerabilități specifice sistemelor de operare

Neprotejarea datelor stocate pe disc

Obiective

- prezentarea vulnerabilităților ce rezultă din manipularea greșită a permisiunilor datelor stocate pe disc
- prezentarea mecanismelor de asignare a permisiunilor (UNIX) și vulnerabilitățiile asociate

Continut

1. Neprotejarea datelor stocate

2. Permisiunile fisierelor in Linux, vulnerabilitati
   - Permisiunile fisierelor
   - Crearea unui fișier
   - Legaturi
   - Race conditions
   - Fisiere temporare

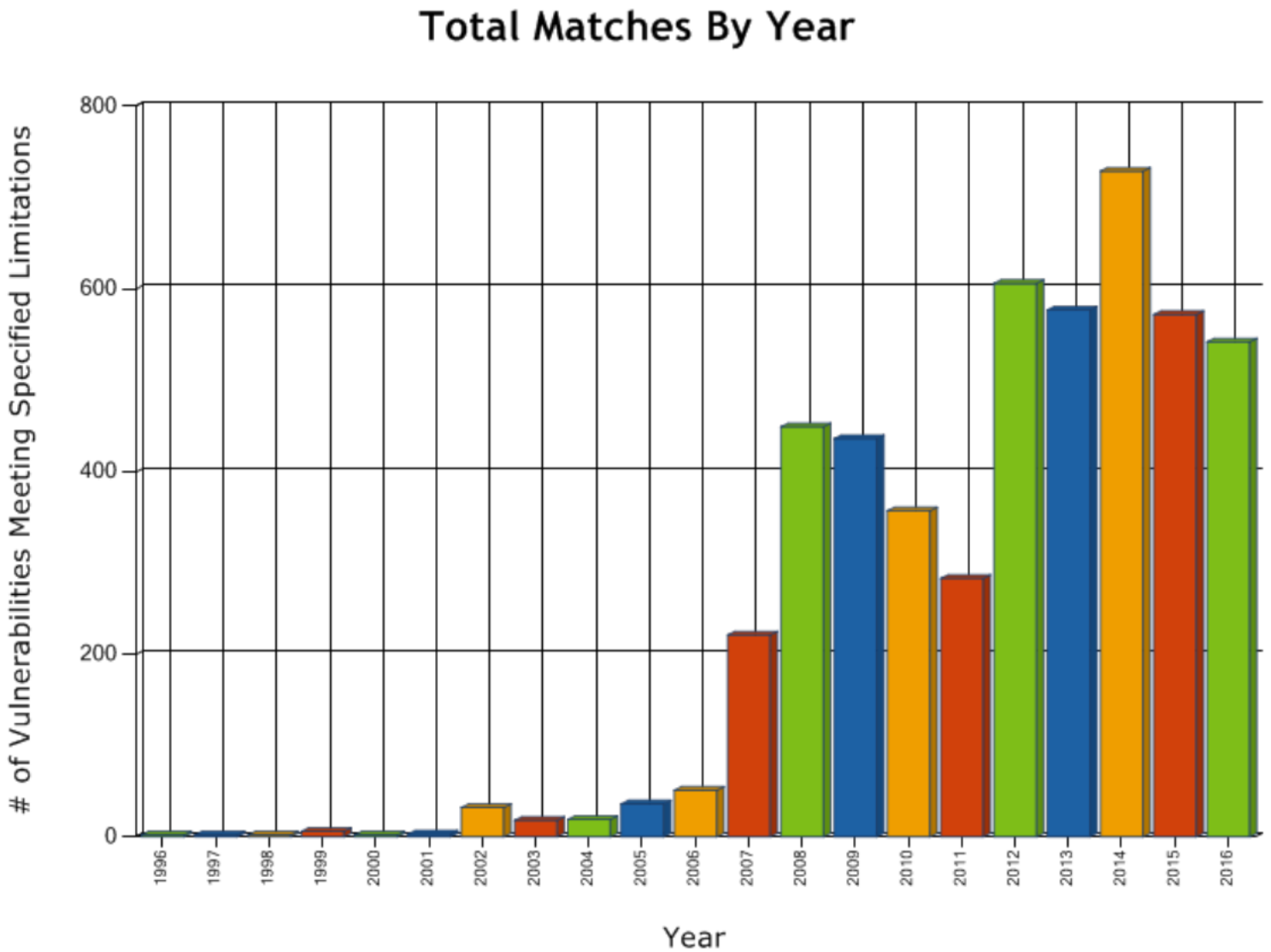
Descriere

- **neprotejarea datelor stocate pe disc** (*data at rest*)
  - cel mai rău caz: neprotejarea datelor
  - neprotejarea corectă a datelor
- două componente
  1. mecanism absent / slab pentru controlul accesului
  2. criptare slabă / inexistentă a datelor
- cazuri pentru controlul slab al accesului
  - se permite accesul la toată lumea
  - se permite accesul la utilizatorii neprivilegiați
  - acces pentru scriere pe un executabil
  - acces pentru scriere pe fișiere de configurare
  - acces pentru citire pe fișiere importante
  - utilizarea fișierelor sistem fără mecanisme de control al accesului

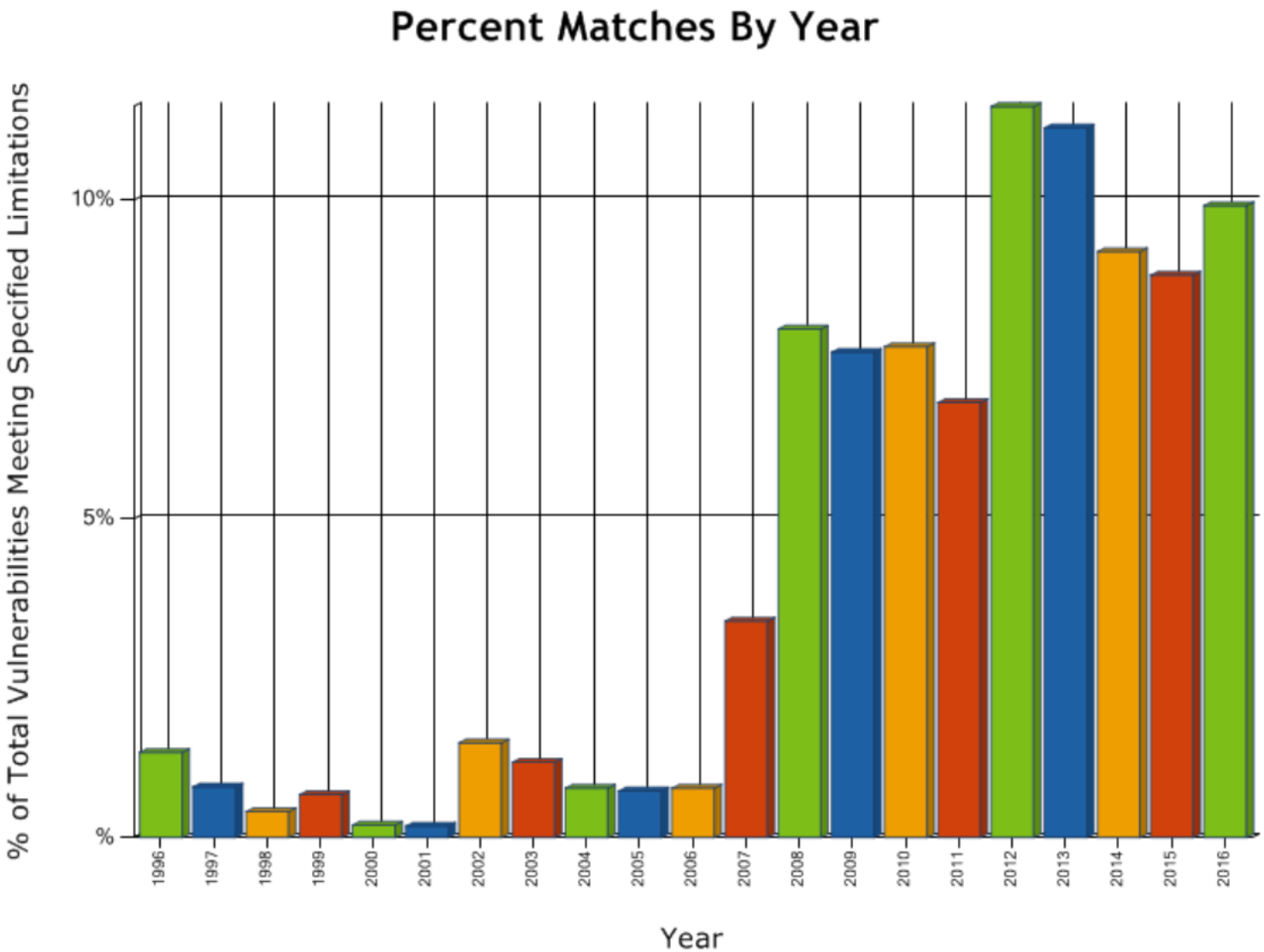
Referințe CWE

- CWE-264: "Permissions, Privileges, and Access Controls"
  - foarte generală
  - legat de managementul permisiunilor, privilegiilor și alte mecanisme de securitate necesare pentru controlul accesului
- CWE-284: "Improper Access Control"
  - nu restricționează corect accesul la resurse persoanelor neautorizate
- CWE-693: "Protection Mechanism Failure"
  - folosirea incorectă a mecanismelor de protecție
- CWE-282: "Improper Ownership Management"
- CWE-275: "permission Issues"
  - atribuirea incorectă sau manipularea greșită a permisiunilor

Relevanța - numarul vulnerabilităților legat de privilegii



Relevanța - numarul vulnerabilităților legat de privilegii (II)



Vulnerabilități

- scurgeri de informații
  - scurgerea accidentală de informații prin mesaje de eroare și alte surse
- *race conditions*
- folosirea de parole slabe
  - indiferent de calitatea criptării, parolele folosite pentru a proteja datele sunt slabe
- folosirea greșită a algoritmilor criptografici
  - folosirea algoritmilor de criptare dezvoltați de voi sau a unora ce suportă doar chei de dimensiune mici

Identificarea vulnerabilităților

- uitativă după cod care
  - configurează / controlează accesul **și**
  - asignează drepturi de acces utilizatorilor neprivilegiați
- uitativă după cod care
  - creează un obiect fară a configura controlul accesului **și**
  - creează obiectul într-un loc unde utilizatorii neprivilegiați au drepturi de scriere
- uitativă după cod care
  - scrie informații de configurare într-un loc partajat
  - scrie informații sensibile într-o zonă unde utilizatorii neprivilegiați au drepturi de citire
- uitativă după fișiere cu permisiuni slabe
  - ex. Linux: *find / -type f -perm +002*

Recomandări

- atenție la permisiuni și criptați datele corect
- trebuie să aveți grijă la fiecare bit configurat în schema de permisiuni
- aveți grijă sa nu expuneți date sau fișiere binare
- fișierele binare ar trebui să fie în directoare sistem sau zone protejate
- scrieți datele legat de un utilizator în directorul local al utilizatorului

File IDs

- informații de proprietate: *UID* și *GID*
- configurate la crearea fișierului
  - *owner UID* = *effective UID* al procesului care a creat fișierul
  - pentru *GID* sunt două abordări
    - BSD: *GID* primește valoarea *GID* directorului părinte
    - Linux: *GID* primește valoarea *effective GID* al procesului care a creat fișierul
- pot fi schimbate cu *chown()*, *lchown()*, *fchown()*
- în mod normal doar *root* poate schimba drepturile de acces
- deținătorul unui fișier poate schimba *GID* cu un alt grup de care aparține

Permisiuni asupra fișierelor

- 12 biți în grupe de câte 3 biți: *special*, *owner*, *group*, *other*
- permisiuni: citire (r), scriere (w), execuție (x)
  - citire: deschidere, citire
  - scriere: deschidere, scriere
  - execuție: execve
- special: *SUID*, *SGID*, *sticky* (sau *tacky*)
- specificat prin numere în octal: 0644
- pot fi schimbate cu *chmod* și *fchmod*
- doar *root* și proprietarul pot schimba drepturile
- situatii confuze, ex. permisiuni 0606

Permisiuni asupra directoarelor

- asemenea fișierelor dar altă interpretare
- citire: deschidere director, citire director (lista continutul directorului)
- scriere: creare (*mkdir*, *link*, *unlink*, *rmdir*)
- execuție (căutare, parcurgere): *chdir*
- *SUID*: nu are semnificație
- *SGID*: moștenire *GID* (semantica BSD)
- sticky: doar proprietarul fișierului din director are drepturi de redenumire și ștergere (ex. */tmp*)
- umask influențează *mkdir*

Umask

- masca de 9 biți este folosită la crearea fișierelor / directoarelor
- specificată de numere în octal: 022
- fiecare proces poate să-și configureze propria mască
- moștenită de un proces
- permisiunile pot fi schimbate cu *chmod*

Managementul privilegiilor în operațiile cu fișiere

- atunci când un proces lucrează cu fișiere se verifică privilegiile
- se iau în considerare
  - *UID* și *GID* ale fișierului
  - masca de permisiuni a fișierului
  - *UID*, *GID* ale procesului

Permisiuni

- sintaxa:
  *int open(char *pathname, int flags, mode_t mask)*
- permisiuni
  - verifică permisiunile la crearea fișierului
  - se ia în considerare *umask*
- dacă se uită indicatorul *O_EXCL*
  - *open* poate fi folosit pentru a deschide un fișier existent și a crea un fișier nou (*O_CREAT*)
  - trebuie avut grijă să nu se deschidă un fișier existent
  - folosirea *O_EXCL* restricționează crearea unui fișier dacă acesta există
- exemplu cod ce nu verifică existența fisierului

```
if ((fd=open("/tmp/tmpfile.out", O_RDWR|O_CREAT, 0600)) < 0)
  die("open");
```

Directoare publice

- cod vulnerabil: nu se verifică existența fișierului în directorul comun (director accesibil)

```
if ((fd = open("/tmp/tmpfile.out", O_CREAT|O_RDWR, 0600)) < 0)
  die("cannot open file");
```

- dacă fișierul există, este deschis
  - un atacator poate crea anterior o legătură simbolocă - *symlink* (named like the file) către fișiere ce conțin informații confidențiale
- dacă fișierul există, permisiunile de creare sunt ignorate
  - un atacator poate crea anterior fișierul cu permisiuni mai puțin restrictive pentru a avea acces la fișier

Proprietar neprivilegiat

- un program privilegiat își coboară privilegiile pentru a crea un fișier
- utilizatorul neprivilegiat poate citi / schimba permisiunile fișierului și conținutul acestuia
- exemplu cod posibil vulnerabil (depinde de cum e utilizat fisierul de sistem mai departe)

```
drop_privs();

if ((fd = open("/usr/account/resultfile", O_CREAT|O_RDWR, 0600)) < 0)
  die("cannot open file");

regain_privs();
```

Recomandări - siguranța directorului

- permisiunile fișierului nu sunt suficiente pentru a proteja un fișier
- permisiunile directorului părinte trebuie luate în considerare
- exemplu: un fișier cu permisiuni doar pentru citire, poate fi șters / creat dacă directorul părinte are drepturi de scriere
- bitul *sticky* reduce doar suprafața de atac
- dacă directorul părinte este deținut de atacator, acesta poate schimba permisiunile directorului
- **recomandări**
  - toate directoarele din calea fișierului trebuie să fie sigure

Concepte legat de calea unui fișier

- o secvență de unul sau mai multe directoare separate de un caracter special (ex. "/")
- de două tipuri: căi absolute și căi relative
  - "." - directorul curent
  - ".." - directorul părinte
- pentru directorul rădăcină (root) ".." indică directorul curent
- mai multe caractere separator sunt reduse la unul singur
- ex.
  "/.///./../usr/..///../usr////./bin//.//file" $\Longrightarrow$ "/usr/bin/file"
- pentru a naviga la un fișier, fiecare director din cale trebuie să aibă drepturi de execuție (căutare)

Trucuri legat de calea unui fișier

- multe aplicații privilegiate construiesc căile dinamic, adesea incorporând și date de la utilizator
- verificarea datelor primite de la utilizator este necesară
- exemplu de cod vulnerabil la parcurgeri (path traversal)

```
if (!strncmp(filename, "/usr/lib/safefiles/", 19)) {
   debug("data file is in /usr/lib/safefiles");
   process_libfile(filename, NEW_FORMAT);
} else {
   debug("invalid data file location");
   exit(1);
}
```

- un atacator ar putea furniza ca și date de intrare șirul "/usr/lib/safefiles/../../../../../etc/passwd"

Caracterul NUL incorporat

- caracterul NUL termină o cale de fișier, calea este doar un șir în C
- când limbaje de nivel înalt (ex. Java, PHP, Perl) interacționează cu sistemul de fișiere nu folosesc șiruri terminate cu NUL
- $\Longrightarrow$ vulnerabilități path truncation

Locuri periculoase

- fișiere / căi furnizate de utilizator
- fișiere / directoare noi
- directoare publice temporare
- fișiere controlate de alți utilizatori

Fișiere interesante

- fișiere de configurare sistem
  - "/etc/*", "/etc/passwd", "/etc/shadow", "/etc/hosts.equiv", ".rhosts", ".shosts", "/etc/ld.preload.so"
- fișiere personale
  - ".history", ".bash_history", ".profile", ".bashrc", ".login", mail spools
- fișiere de configurare pentru programe
  - ".htpasswd", cod sursă pentru scripturi, "sshd_config", "authorized_keys", fișiere temporare
- altele
  - "/var/log/*", kernel și boot files, executabile și biblioteci, "/dev/*", "/proc/*", named pipes

Atacuri folosind legături simbolice

- pot fi folosite pentru a forța programe privilegiate să acceseze informații sensibile
- exemplu cod vulnerabil

```
void start_processing(char *username) {
  char *homedir, tmpbuf[PATH_MAX];
  int f;
  homedir = get_user_homedir(username);
  if (homedir) {
    snprintf(tmpbuf, sizeof(tmpbuf), "%s/.optconfig", homedir);
    if ((f = open(tmpbuf, O_RDONLY)) < 0)
      die("cannot open file");
    parse_opt_file(tmpbuf);
    close(f);
  }
}
```

- un atacator poate crea o legătură simbolică la un fișier sistem important

```
$ ln -s /etc/shadow ~/.optconfig
```

Crearea fișierelor si legături simbolice

- context periculos

```
$ ln -s /tmp/nonexistent /home/john/newfile
open("/home/john/newfile", O_CREAT|O_RDWR, 0640);
// creeaza fisierul "/tmp/nonexistent"
```

- programele privilegiate pot fi pacălite în a crea noi fișiere oriunde în sistemul de fișiere
- strategie de protecție
  - folosirea "*O_EXCL*" în funcția open (exclusiv și nu urmarește legaturile)
  - folosirea "*O_NOFOLLOW*" în funcția open
- crearea accidentală (folosind *fopen* cu drept de scriere)

Atac pe apel sistem ce ține cont de legaturi simbolice

- apeluri sistem țin cont de legăturile simbolice, acționează doar pe ultima componentă din calea unui fișier
- $\Longrightarrow$ apoi urmează celelalte componente
- exemplu

```
$ ln -s /tmp/ /home/john

# creaza fisierul "/tmp/newfile"
$ echo "test" > /home/john/newfile

# sterge "/tmp/newfile"
$ unlink /home/john/newfile
```

Hard links attacks

- dacă utilizatorului i se permite creearea de legături fizice (hard links) către anumite fișiere sistem
  - $\Longrightarrow$ poate păstra accesul la ele chiar și după ce acestea sunt șterse
- similar: prevenind un program să șteargă un fișier
  - dacă atacatorul creează o legătură fizică intr-un director *sticky* către un fișier al altui utilizator
  - $\Longrightarrow$ utilizatorul nu poate șterge legătura fizică!
- în practică, pe sisteme UNIX (Linux) procesul de creeare a legăturilor fizice este foarte restrictiv
  - e.g. crearea unei legături fizice către un fișier deținut de *root* nu este permisă

Fișiere sensibile

- context
  - când programe privilegiate deschid fișiere existente și modifică conținutul acestora sau schimbă proprietarul / permisiunile fișierului
- codul vulnerabil e rulat de un program SUID; *userbuf* dat de utilizator

```
if ((fd = open("/home/jim/.conf", ORDWR)) < 0)
   die("cannot open file");
write(fd, userbuf, len);
```

- exemplu de atac (atacatorul este jim)

```
$ cd /home/jim
$ ln /etc/passwd .conf
$ run_suid_prog
$ su evil
```

Fișiere sensibile (II)

- cod vulnerabil

```
if ((fd = open("/home/jim/.conf", ORDWR)) < 0)
   die("cannot open file");
fchmod(fd, 0644);
```

- atac

```
$ cd /home/jim
$ ln /etc/shadow .conf
$ run_suid_prog
$ cat /etc/shadow
```

Ocolirea mecanismelor de interzicere a legăturilor simbolice

- *lstat* poate fi folosit pentru detecția și analiza legăturilor simbolice
- *lstat* nu face distincție între diferite legături fizice către același fișier
- exemplu: cod vulnerabil la legături fizice, se presupune că este rulat de un program privilegiat

```
if (lstat(fname, &st) != 0)
  die("cannot stat file");

if (!S_ISREG(st.st_mode))
  die("not a regular file");

fd = open(fname, O_RDONLY);
```

Race conditions - Context

- o aplicație ce interacționează cu sistemul de fișiere poate fi atacată prin metoda *race condition* - defecte de sincronizare, dacă este suspendată într-un moment inoportun
- exemplu cod vulnerabil

```
if ((res = access("/tmp/userfile", R_OK)) < 0)
   die("no access");

// ... moment inoportun

// sigur pentru deschiderea fisierului
fd = open("/tmp/userfile", O_RDONLY);
```

Time of Check To Time of Use (TOCTOU)

- starea unei resurse poate fi schimbată între
  - timpul în care starea ei este verificată (**check**) și
  - timpul când este folosită (**use**) efectiv
- nu corespunde doar cazurilor de manipulare a sistemului de fișiere
- pare improbabil, dar se poate întâmpla sau se poate produce
  - încetinește sistemul: trafic mare în rețea, utilizarea intensivă a sistemului de fișiere
  - trimiterea semnalului de control de stop și start constant într-o buclă
  - monitorizarea execuției explicației

Sintaxa și funcționalitatea stat()

- sintaxa:
  *int stat(const char *pathname, struct stat *buf)*
- întoarce informații
- *lstat* acționează asupra legăturilor simbolice, nu le urmărește
  - $\Longrightarrow$ poate fi folosită pentru a evita atacuri de legături simbolice
- verificarea numărului de legături fizice poate preveni atacurile
- vulnerabilitate: schimbarea fișierului după verificarea *stat*

Încercare de evitare a problemelor de sincronizare

- se încearcă inversarea ordinii acțiunilor: verificarea se face în timpul folosirii "*check and use*" $\rightarrow$ "*use (open) and check*"
- exemplu vulnerabil

```
if ((fd = open(fname, O_RDONLY) < 0)
  die("open");

if (lstat(fname, &st) < 0)
  die("lstat");

if (!S_ISREG(st.st_mode))
  die("not a reg file");
```

Încercare de evitare a problemelor de sincronizare (II)

- atac
  1. crearea unei legături simbolice către un fișier sensibil $\Longrightarrow$ aplicația deschide fișierul
  2. înaintea apelului *lstat*, șterge / redenumește legătura simbolica și crează un fișier $\longrightarrow$ se trece peste verificare (dar fișierul sensibil va fi folosit în continuare!)
- observație: ștergerea unui fișier deschis merge și pentru fișiere normale

File race redux

- probleme cu apeluri sistem ce folosesc căi
- exemplu: vulnerabil - poate investiga fișiere diferite

```
stat("/tmp/file", &st);
stat("/tmp/file", &st);
```

- audit: cand vedeți mai multe apeluri sistem succesive ce folosesc aceeași cale, evaluați ce se întâmplă dacă se schimbă calea între apeluri
- strategie de protecție: folosiți apeluri sistem ce folosesc descriptori $\Longrightarrow$ utilizarea și verificarea se fac împreună
- exemplu: cod sigur (ambele *fstat* accesează același nod)

```
fd = open("tmp/file", O_RDWR);
fstat(fd, &st);
fstat(fd, &st);
```

Permission Races

- problema
  - o aplicație creează fișiere temporare
  - permisiuni greșite (ex. acces public)
- atac
  - dacă un atacator poate deschide fișierul în perioada când este expus
  - menține acces la fișier chiar dacă ulterior aplicația restricționează accesul

Permission Races (II)

- cod vulnerabil: expune pentru un anumit timp fișierul (permisiuni greșite, depinde de valoarea *umask*)

```
FILE *file;

// apel open(fname, ..., 0666) !!!
if (!(file = fopen(fname, "w+")))
  die("fopen");

int fd = fileno(file);

// evita atacuri TOCTOU, se foloseste fd
if (fchmod(fd, 0600) < 0)
  die("fchmod");
```

Ownership Races

- context
  - un fișier creat cu privilegiile unui utilizator neprivilegiat
  - deținătorul acelui fișier este schimbat mai târziu la un utilizator privilegiat
- nesincronizări
  - utilizatorul neprivilegiat (atacatorul) poate accesa fișierul între momentul când a fost creat și când se schimbă deținătorul fișierului

Ownership Races (II)

- exemplu cod vulnerabil

```
drop_privs();

if ((fd = open(fname, O_RDWR | O_CREAT | O_EXCL, 0666)) < 0)
  die("open");

regain_privs();

// schimba detinatorul fisierului
if (fchmod(fd, geteuid(), getegid()) < 0)
  die("fchmod");
```

Directory Races

- context: aplicație ce parcurge sistemul de fișiere
- problema: legături simbolice infinit recursive (cicluri)
  - kernel detectează cicluri când stabilește calea
  - problemele apar când aplicația traversează mai multe fișiere
- directoarele referite simbolic nu pot fi referite în comenzi shell
- apeluri sistem (ex getcwd)
- exemplu

```
$ cd /home/jim
$ ln -s /tmp mydir
$ cd /home/jim/mydir
$ pwd
/home/jim/mydir
```

Legături simbolice pentru directoare - exploatarea unlink()

- efectul utilizatorilor rău intenționați ce manipulează directoare care sunt cu unul sau două nivele mai sus în ierarhia de directoare
- exemplu: cod vulnerabil în unele implementări ale comenzii "*at*"

```
chdir("/var/spool/cron/atjobs");

stat64(JOBNAME, &statbuf);
if (statbuf.st_uid != getuid())
exit(1);

unlink("JOBNAME");
```

- primul vector de atac

```
$ at -r ../../../../../tmp/somefile
```

- care ar șterge un alt fișier decât cel programat, dar deși numai dacă aparține utilizatorului apelant

Legături simbolice pentru directoare - exploatarea unlink() (II)

- dar: există o vulnerabilitate de sincronizare (*race condition*) între timpul în care se verifică fișierul și ștergerea acestuia, vulnerabilitate ce poate fi exploatată
  - între cele două momente: ștergerea fișierului utilizatorului și înlocuirea acestuia cu o legătură simbolică către un fișier sensibil

```
$ mkdir /tmp/bob
$ touch /tmp/bob/shadow
$ at -r ../../../../..//tmp/bob/shadow
$ rm -fr /tmp/bob/
$ ln -s /etc /tmp/bob # daca apare problema de sincronizare
    # se sterge /etc/shadow !!
```

- unlink() nu urmărește legătura simbolică pentru ultimul element din cale

Mutarea directoarelor

- program vulnerabil

```
rm -fr /tmp/a # se sterge /tmp/a/b/c
```

- ce se întâmplă

```
chdir("/tmp/a");
chdir("b");
chdir("c");
chdir("..");
rmdir("c");
chdir("..");
rmdir("b");
fchdir(3);
rmdir("/tmp/a");
```

- vector de atac
  - acționați înainte de primul "chdir("..");"
  - mutați directorul "c" într-un subdirector din "/tmp"

Crearea fișierelor unice cu mktemp()

- preia un șablon dat de utilizator pentru un fișier și îl completează astfel încât să reprezinte un fișier unic, nefolosit
- șablonul conține caracterele *XXX* pentru locul unde se v-a completa cu date
- poate fi prezis ușor deoarece este bazat pe ID procesului și un model simplu
- exemplu cod vulnerabil:

```
char temp[1024];

strcpy(temp, "/tmp/myfileXXXX");
if (!mktemp(temp))
  die("mktemp");

if ((fd = open(temp, O_CREAT | O_RDWR, 0700)))
  die("open");
```

Crearea fișierelor unice cu mktemp() (II)

- vulnerabilitate: *race condition* între apelul *mktemp* și *open*
- exemplu: unele versiuni GCC
  - gcc folosește *mktemp* pentru a crea un model comun pentru toate fișierele sale din */tmp* (primul se termina în ".i")
  - un atacator poate monitoriza apariția unui fișier ".i" și poate crea legături simbolice pentru alte tipuri de fișiere, ".o", ".s"
  - dacă *root* compilează, poate să suprascrie fișiere sensibile

Crearea fișierelor unice

- *mkstemp()* o alternativă mai sigură la *mktemp*
  - găsește un nume de fișier unic
  - creează fișierul și îl deschide
  - întoarce descriptorul de fișier
- *tmpfile* similar cu *mkstemp*
- *mkdtemp* folosit pentru a crea directoare temporare

Bibliografie

- "The Art of Software Security Assessments", chapter 9, "UNIX 1. Privileges and Files", pp. 476 – 576
- "24 Deadly Sins of Software Security", chapter 17, "Failure to Protect Stored Data".