Unix Security

Privileges and Files

Adrian Coleșa

Universitatea Tehnică din Cluj-Napoca
Computer Science Department

October 28, 2015

The purpose of this lecture

1. presents basic concepts related to files and directories in UNIX-like OSes
2. discuss specific code vulnerabilities that can be introduced when operating on files and directories
   - executing code with too much privileges
   - accessing files in untrusted places in file system
   - race conditions

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Fundamental characteristics

- simple
- composed of simple and clear modules that interact each other
- a common simple interaction interface (file-like)

Outline

Users

- multiuser
- every user has an unique ID (UID)
- each UID could be associated different privileges
- user and system accounts
- UID 0 is for the superuser, named “*root*”

Groups

- define a set of related users
- each group is identified by a unique ID (GID)
- GIDs are also associated different privileges
- types
  - primary (login) group
  - supplemental (secondary) groups

Configuration Files and Associated Directories

- users are defined in "`/etc/passwd`"
  - readable by any user
  - text file: a line-based database recording basic user details
  - line format:

```
bob:x:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

- password related information is typically stored in "`/etc/shadow`"
  - readable only by root
- group related information is stored in "`/etc/group`"
- each user has a home directory
- each user has a default shell

Outline

Files

- storage abstract concept
- a general file-like interface (almost everything is a file)

Directories

- file-space organization abstract concept
- hierarchical structure of the file-space (tree or graph)
  - paths
- filesystem hierarchical standard
  - "`/etc`", "`/bin`", "`/sbin`", "`/home`". "`/var`", "`/tmp`" ...
- single hierarchy build on mounting (integrating) more file systems
  - mount point
  - each mounted FS has a corresponding kernel driver
- physical and virtual (pseudo) FSs

Properties

- every file and directory belong to a user (UID) and a group (GID)
- every file and directory is associated a set of permissions

Outline

File IDs

- ownership information: UID and GID
- set when file is created
  - owner UID = effective UID of the processes creating the file
  - two schemes used for GID
    - BSD: set GID to the GID of the file's parent directory (directory inheritance)
    - SysV/Linux: set GID to effective GID of the creating process (BSD style could also be used via mount options or specific directory permissions)
- could be changed by *chown()*, *lchown()*, *fchown()*
- normally only root can change ownership

File IDs (cont.)

- a file's owner could change file's group only to another group he belongs to

File Permissions

- 12 bits in four groups of three bits: special, owner, group, other
- basic file permissions: read (r), write (w), execute (x)
  - read: open, read
  - write: open, write
  - eXecute: execve
- special flags: SUID, SGID, sticky (or tacky)
- specified with octal numbers like: 0644
- system looks only at the most specific set of permissions (!)
  - example: 0606

File Permissions (cont.)

- could be changed with *chmod*, *fchmod*
- only owner and root can change permissions

Directory Permissions

- the same like for files, but different interpretation
- read: open/opendir,readdir
- write: creat, mmkdir, link, unlink, rmdir, rename
- eXecute (search, traverse): chdir
- SUID: no meaning
- SGID: directory GID inheritance (BSD semantic)
- sticky: restrict renaming and deletion to the owner of each file in the directory (see "`/tmp`")
- *umask* affects *mkdir*

Umask

- 9-bit mask used when (only) at file /directory creation
- specified by octal numbers like for permissions: 0022
- each process can set its own umask
- inherited by a process from its parent
- masked permissions could be changed explicitly with *chmod*

Privilege Management with File Operations

- privilege checks done when processes perform operations on the file system
- privilege checks consider
  - file's UID and GID
  - file's permission bitmask
  - process's effective UID, GID, supplemental groups

Outline

File Descriptors

- uniform file-like interface
- each process: file descriptor table (FDT)
- entries refer to data structures (open files) created by *open*
- a file descriptor is a index in FDT
- it is returned by *open*
- standard file descriptors: 0 (STDIN), 1 (STDOUT), 2 (STDERR)
- released by *close*
- file descriptors are duplicated (to the child) when a new process is created

File Descriptors (cont.)

- a process could specify which file descriptors to be closed when loads a new code (“close-on-exec”)

Inodes

- i-node = information node
- a data structure on the disk containing attributes (metadata) of each file
- fields: type, permissions, owner, size, pointers to data blocks etc.
- loaded in memory when accessing the file
- unique i-node numbers
- the kernel translates from pathnames to inodes

Directories

- a collection of *directory entries*
- directory entry fields: filename and inode number
- getting from a filepath to the file directory involves accessing inodes of all directories in the path

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Symbolic Links

- allows users to create a file that points to another file or directory
- created with *symlink()* system call
- they are special files
- could be used to refer to files on another partition (FS)
- their reference become invalid when the referred file is removed or renamed
- when created, the referred path could not exists!
- some syscalls follow symlinks automatically, while others operate on them

Symbolic Links (cont.)

- symlink-aware syscalls
  - unlink
  - lstat
  - lchown
  - readlink
  - rename

Hard Links

- allow creation of multiple paths to the same file
- created with *link()* syscall
- they refer the same inode
- each inode keeps the number of hard links to that inode (file)
- an file (inode) is physically removed only when hdlink number reaches 0
- so, when a filepath (hdlink) to a file is removed, the other remains valid
- limitation: cannot make a hardlink to a file on another FS

Hard Links (cont.)

- limitation: hardlink to directories is reserved for root only (or not even)
- see hardlink number for a directory (empty and non empty)

Outline

Processes

- program = executable file
- process = instance of a running program
- process components
  - (virtual) memory address space
  - process ID (PID)
  - possibly more threads
- each process runs with the privileges of its effective user (*effective UID*)
- there is also a *real UID*

Processes (cont.)

- processes belonging to root have absolute access to any resource
- special privileges: *set-user-id (SUID)* and *set-group-id (SETGID)*

Associated User IDs of a Process

- user IDs
  - real UID
  - saved SUID
  - effective UID
- normally a process is allowed to switch its effective UID between real UID and saved SUID
- processes with effective UID 0 have complete access to the system
- when a process runs a new program
  - real UID remains the same

Associated User IDs of a Process (cont.)

- effective UID remains the same unless SUID not set for the new program
- saved SUID is replaced with the effective UID of the new process

Associated User IDs of a Process

- group IDs
  - real GID
  - saved SGID
  - effective GID
  - supplemental GIDs
- processes with effective GID 0 normally does not have absolute control of the system

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Non-root SUID ad SGID Programs

- process's effective permissions are determined by
    - its effective UID
    - its effective GID
    - its supplemental GIDs
- start with their their effective UID equal to their saved SUID
- allow normal users to access resources of other users
- example: *wall* (broadcast messages to all users)

```
$ ls -l `which wall`
-rwxr-sr-x 1 root tty 18976 Jun 18 04:21 /usr/bin/wall
```

SUID Root Programs

- most SUID programs are SUID root, i.e. they belong to the root
- they run as root when started
- examples
  - *ping*

```
$ ls -l `which ping`
-rwsr-xr-x 1 root root 35712 Nov  8  2011 /bin/ping
```

  - *passwd*

```
$ ls -l `which passwd`
-rwsr-xr-x 1 root root 42824 Sep 13  2012 /usr/bin/passwd
```

Daemons and Their Children

- long-running processes that provide system services
- usually started automatically at boot time
- often run as root to be able to perform privileged operations
- often run other programs to handle required tasks, and their children are usually also started with root privileges
- could temporarily assume other users' identity to perform certain actions in a safer manner
  - as long as the program leaves its saved SUID and real UID set to 0, it can regain its root privileges later

Daemons and Their Children (cont.)

- example: *login*

```
$ ls -l `which login`
-rwxr-xr-x 1 root root 44928 Sep 13  2012 /bin/login
```

- after the user is logged on, the login switches all its UIDs to that user's ID

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The *seteuid()* Function

- syntax

```
int seteuid(uid_t euid);
```

- changes the effective UID
- programs use it to temporarily change their privileges
- two contexts
  - root process: can set effective UID to any needed value
  - non-root process: can only toggle the effective UID between the saved SUID and the real UID
- example - initial: user *admin* (1000) runs a SUID program of *bin* (1)

The *seteuid()* Function (cont.)

```
// real UID = 1000
// saved SUID = 1
// effective UID = 1

seteuid(1000);

// real UID = 1000
// saved SUID = 1
// effective UID = 1000

seteuid(1);

// real UID = 1000
// saved SUID = 1
// effective UID = 1
```

- on Linux glibc 2.0
  - if root and change effective UID to an UID different by real UID

The *seteuid()* Function (cont.)

- $\Rightarrow$ saved SUID is changed along with the effective UID

The *setuid()* Function)

- syntax

```
int setuid(uid_t uid);
```

- change effective UID and also both real UID and saved SUID
- mainly used for permanently assuming the role of a user, usually for the purpose of dropping privileges
- for root processes $\Rightarrow$ set all three UIDs to specified value
- for non-root processes $\rightarrow$ differences on different UNIX variants

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The *setuid()* Function) (cont.)

- some (Linux, Solaris, OpenBSD) makes it behave like *seteuid()*
- others (FreeBSD and NetBSD) makes it work similar like for superuser programs
- example (Linux): user *admin* (UID=1000) run a SUID program of user *bin* (UID=1)

The *setuid()* Function) (cont.)

```
// real UID = 1000 (admin)
// saved SUID = 1 (bin)
// effective UID = 1 (bin)

setuid(1000);

// real UID = 1000 (admin)
// saved SUID = 1 (bin)
// effective UID = 1000 (admin)

setuid(1);

// real UID = 1000 (admin)
// saved SUID = 1 (bin)
// effective UID = 1 (bin)
```

- not recommended for permanently dropping privileges for non-root processes, as its behavior depends on the UNIX variant

The *setresuid()* Function

- syntax

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
```

- used to set explicitly all three UIDs
- if “-1” is given as argument the current value of the corresponding UID is used
- root processes can set them to any desired value
- non-root processes can set any ID to the current value of any of the three current UIDs

The *setreuid()* Function

- syntax

```
int setreuid(uid_t ruid, uid_t euid);
```

- used to set real and effective UIDs
- if "-1" is given as argument the current value of the corresponding UID is used
- root processes: set them to any desired value
- non-root processes: behavior is OS dependent, but it can typically change
  - real UID to effective UID
  - effective UID to real UID, effective UID or saved SUID

The *setreuid()* Function (cont.)

- saved SUID tried to be updated, if new effective UID different by new real UID
- useful in the following situation:
  - a program is managing two UIDs as its real UID and saved SUID, none being of root
  - the program wants to drop one set of privileges
  - *setresuid()* not available
  - solution: `setreuid(getuid(), getuid());`

Group ID Functions

- *setegid*: toggle effective GID between saved SGID and real GID
- *setgid*: change effective GID, and possibly also saved SGID and real GID
- *setresgid*: change all GIDs
- *setregid*: change real and effective GIDs
- *setgroups*: set supplemental groups (requires effective UID = 0)
- *initgroups*: a convenient alternative to setgroups (requires effective UID = 0)

Group ID Functions (cont.)

- warning
  - effective UID = 0 $\Rightarrow$ root processes *rightarrow* the group functions have a special behavior
  - effective GID = 0 $\Rightarrow$ non-root processes

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Description

- context: programs running with elevated privileges
- mistake
  - performs potentially dangerous actions on behalf of an unprivileged user without first dropping privileges
  - takes no precautions before interacting with the FS
- result: expose important system files
- advice: a SUID/SGID program should drop (temporarily) its privileges when performing an action normally not allowed to the real UID

Example

- the SUID root program `XF86_SVGS` (from XFree86)

```
$ id
uid=1000(test) gid=1000(test) groups=1000(test)
$ ls -l /etc/shadow
-rw------- 1 root root .... /etc/shadow
$ cd /usr/X11R6/bin
$ ./XF86_SVGA -config /etc/shadow
Unrecognized option: root:qEXaUxSeQ45la:10171:-1:-1:-1:-1:-1:-1
...
```

- the program reads any file not checking if the real UID has permissions on that file
- the SUID was not needed for reading the config file, but for making display configuration changes

Libraries

- usage of third-party libraries
- shared libraries are often the source of potential vulnerabilities
- user do not have details about the internals of library functions, they only know its API
- example: login class capability database (see `www.osvdb.org/displayvuln.php?osvdb_id=6073`)
    - *openssh* and *login* called functions in *libutil* to read entries from login database, without dropping their privileges

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Template

- regular code template

```
// set up special socket
setup_socket();

// drop privileges
setuid(getuid());

// non-privileged part
start_procloop();
```

- common idiom for permanently drop privileges:

```
setuid(getuid());
```

- warning: in some situations it is not enough

Group Privileges

- programs with both SUID and SGID
- mistake: program forgets to also drop group privileges, beside the UIDs
- correct way and *order*

```
setgid(getgid());
setuid(getuid());
```

- if the order is changed $\Rightarrow$ wrong order
  - the saved SUID could still remain of the privileged user
  - a possible attacker's executed code could perform a `setegid(0);` or `setregid(-1, 0);` to recover group privileges

Supplemental Group Privileges

- not leaving supplemental groups of the privileged user to the non-privileged user the program drops its privileges to
- example: vulnerability of rsync (CVE-2002-0080)

```
if (root) {
  setgid(normal_uid);
  setuid(normal_gid);
}
```

- correct privilege dropping code template

```
if (root) {
  setgroups(0, NULL);
  setgid(normal_uid);
  setuid(normal_gid);
}
```

Non-Root Elevated Privileges

- *problem*: running the below as non-root is system dependent

```
setgid(getgid());
setuid(getuid());
```

- example: a SUID program belonging to a non-root user
- both `setuid()` and `setgid()` change only the effective IDs, not the saved IDs
- attackers exploiting a program vulnerability could regain the relinquished privileges
- *solution*: use the `setresgid()` / `setregid()` and `setresuid()` / `setreuid()` functions

Mixing Temporary and Permanent Privilege Relinquishment

- specific to applications that switch back and forth between privileged and non-privileged users, eventually dropping privileges at all, if possible
- subtle errors could occur from the `setuid()` usage
- example

Mixing Temporary and Permanent Privilege Relinquishment (cont.)

```
#define STARTPRIV seteuid(0);
#define ENDPRIV seteuid(getuid());   // effective UID <- real UID
void main_loop()
{
    uid_t realuid = getuid();

    // do not need privileges
    ENDPRIV
    dp_unprivileged_action();
    ...
    STARTPRIV
    do_privileged_action();
    ENDPRIV
    ...
    // drop privileges permanently
    // !!! not root, so saved SUID not changed
    setuid(realuid);
    ...
}
```

Mixing Temporary and Permanent Privilege Relinquishment (cont.)

- when dropping privileges permanently the process effective UID is not 0, but *realuid*, so saved SUID remains unchanged (0)
- an attacker could call (from the possibly compromised program) `seteuid(0);` to regain root privileges

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using the Wrong Idiom

- dropping privileges permanently ASAP is the safest option for a setuid application
- but use the correct idiom
- `seteuid(getuid());`
  - good for temporarily dropping
  - bad for permanently dropping
- good idiom for permanently dropping: `setuid(getuid());`

Dropping Group Privileges

- problem: drop group privileges in the wrong order
- example of vulnerable code

```
seteuid(pw->pw_uid);
setegid(pw->pw_gid);
```

- `setegid(pw->pw_gid);` fails
- see details at `http://security.freebsd.org/advisories/FreeBSD-SA-01:11.inetd.asc`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using More Than One Account

- specific to programs needing to use more than one user account
- example: see details at `www.unixpapa.com/incnote/setuid.html`

Using More Than One Account (cont.)

- wrong implementation

```
// become user1
seteuid(user1);
process_log1();

// become user2
seteuid(user2);
process_log1();

// become root
seteuid(0);
```

Using More Than One Account (cont.)

- correct implementation

```
// become user1
seteuid(user1);
process_log1();

// become root
seteuid(0);

// become user2
seteuid(user2);
process_log1();

// become root
seteuid(0);
```

Outline

Permanent Dropping of Privileges

- root processes: rules and order
  1. when dropping privileges effective must be UID = 0
  2. supplemental must be cleared (using *setgroups* with effective UID = 0)
  3. all three GIDs must be dropped to an unprivileged GID
     - right: `setgid(getgid());`
     - wrong: `setegid(getgid());`
  4. all three UIDs must be dropped to an unprivileged UID
     - right: `setuid(getuid());`
     - wrong: `seteuid(getuid());`
- non-root processes: rules and order
  1. cannot modify groups with `setgroups()`

Permanent Dropping of Privileges (cont.)

2. for dropping GIDs use
   - `setresgid(getgid(), getgid(), getgid())`, instead of `setgid(getgid())`
   - older systems: `setgid(getgid()` two times
3. for dropping UIDs use
   - use `setresuid(getuid(), getuid(), getuid())`, instead of `setuid(getuid())`
   - older systems: `setuid(getuid()` two times

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Temporary Dropping of Privileges

- make sure code drops any relevant group permissions as well as supplemental group permissions
- make sure the code drops group permissions before user permissions
- make sure code restore privileges before attempting to drop privileges again, either temporarily or permanently

Outline

Privilege Limitations in Traditional UNIX

- all-or-nothing privilege model in UNIX
- root has unrestricted access
- example
  - ping needs root privileges to create a raw socket
  - if exploited before dropping its privileges the program has access to all system's resources
- any program requiring special privileges essentially puts the entire system's security in danger

Linux File System IDs

- each process also maintains file system UIDs: FSUID and FSGID
- they address a potentially security problem with signals
  - the unprivileged user could send signals (to attack) the privileged process, temporarily having dropped its privileges to that of the unprivileged user
- aimed to be used for all file system accesses
- they are kept synchronized with the effective UID/GID
- used by program that temporarily want to access FS with a normal user's privileges

Linux File System IDs (cont.)

- could be changed with `setfsuid()` and `setfsgid()`
- warning: deprecated in Linux

BSD securelevels

- intended to protect the kernel from root uses a system-wide kernel value “securelevel” to help decide what actions system users are allowed to perform
- there are four security levels
  - “-1”: permanently insecure mode (always run system in level 0)
  - “0”: insecure mode - immutable and applend-only flags may be turned off
  - “1”: secure mode - immutable and applend-only flags cannot be turned off, `/dev/mem` and `/dev/kmem` cannot be opened for writing

BSD securelevels (cont.)

- "2": highly secure mode - also disks may not be opened for writing whether mounted or not
- any superuser process could increase the security level, but only *init* could lower it
- problem: does not allow fine tuning

Capabilities

- specific to Linux
- defines a set of administrative tasks (capabilities) that can be granted to or restricted from a process running with elevated privileges
- some examples
  - `CAP_CHOWN`
  - `CAP_SETUID/CAP_SETGID`
  - `CAP_NET_RAW`
  - `CAP_NET_BIND_SERVICES`
  - `CAP_SYS_MODULES`

Capabilities (cont.)

- usage example: ping process could be given only the `CAP_NET_RAW` capability
- could be applied to both processes and files
- a process has three masks of capabilities
  - permitted set: capabilities that could be enabled
  - effective set: capabilities that are enabled
  - inheritable set: capabilities inherited by a child
- file capability sets
  - permitted (forced): capabilities that are permitted in addition to those process might normally inherit

Capabilities (cont.)

- inheritable (allowed): capabilities that are allowed to be added to the process permitted sets when the executable is run
- effective: a bit indicating if added capabilities in the permitted set should automatically be transferred to the effective set, when a new process image is loaded

Outline

Outline

Opening Flags and Permissions

- syntax of *open()*

```
int open (char *pathname, int flags, mode_t mask);
```

- permissions
  - check that reasonable permissions are set at file creation
  - take into account umask
- forgetting `O_EXCL`
  - open can be used for both opening an existing and creating a new file (`O_CREAT`)
  - take care not to open an existing file instead of creating a new one
  - usage of `O_EXCL` restricts creating a file if it already exists

Opening Flags and Permissions (cont.)

- vulnerable example

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

  - if the file exists, it is just opened: an attacker could previously create a sym-link (named like the file) to sensitive system files
  - if the file exists, creation permissions are ignored: an attacker could previously create the file with more relaxed permissions to gain access to the file

Unprivileged Owner

- privileged program temporarily drops its privileges to create a file
- the non-privileged owner could read/change file's permissions and contents
- example of possibly vulnerable code

```
drop_privs();

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

regain_privs();
```

Outline

Directory Safety

- file permissions are not sufficient to protect a file
- parent directory permissions must also be considered
- example: a read-only file cannot be read, but can be deleted/renamed (or created previously), if parent directory is writable
- sticky bit only reduces the attack area, but not eliminate it completely
- if the parent directory is owned by the attacker, he can change directory permissions
- **recommendation**: for a file to be safe,

Directory Safety (cont.)

- create it in a safe manner
- all directories in its path must be safe

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Review Concepts About Pathname

- a sequence of one or more directory components separated by a special character (e.g. '/')
- pathname and filename are used interchangeable in practice
- two types: absolute and relative
- special entries in each directory: "." (current directory) and ".." (parent directory)
- in the root directory ".." points to itself
- more consecutive path separators are reduced (considered) to one

Review Concepts About Pathname (cont.)

-
  - “`/.///./../usr/..///../usr////./bin///.//file`” $\Rightarrow$
  - “`/usr/bin/file`”
- to get a file, execution (search) permission is needed for each directory in its path

Pathname Tricks

- many privileged applications construct pathnames dynamically, often incorporating user-controlled inputs
- sanity checks are needed on such paths
- example of a vulnerable code

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

- an attacker could provide a path like "`/usr/lib/safefiles/../../../../../etc/shadow`"

Embedded NUL

- NUL character terminates a pathname, as a pathname is just a string in C
- when higher-level languages (e.g. Java, PHP, Perl) interact with the FS, they mostkly use conted strings, not using “NUL-terminated” semantic

Dangerous Places

- user-supplied locations / file names
- new files and directories
- temporary and public directories
- files controlled by other users

Outline

System Configuration Files

- files in "`/etc`"
- authentication databases. like: "`/etc/passwd`", "`/etc/shadow`", "`/etc/master.passwd`" etc.
- host equivalency: "`/etc/hosts.equiv`", "`.rhosts`", "`.shosts`",
- "`/etc/ld.preload.so`"
- "`/etc/nologin`", "`/etc/hosts.allow`"

Personal User Files

- shell histories: ".history", ".bash_history"
- shell login and logout scripts: ".profile", ".bashrc", ".login"
- mail spools

Program Configuration Files and Data

- web-related files: "`.htpasswd`", source code of scripts
- ssh configuration files: "`sshd_config`", "`.shosts`", "`authorized_keys`"
- temporary files

Other files

- log files: “`/var/log`”
- program files and libraries
- kernel and boot files
- device files
    - virtual device drivers, like “`/dev/zero`”, “`/dev/random`”
    - raw memory devices: “`/dev/mem`”, “`/dev/kmem`”
    - hardware device drivers
    - terminal devices
- named pipes
- *proc* file system

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Symbolic Link Attacks

- can be used to make privileged programs opening sensitive files
- example 1 of vulnerable code

```
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(f);
      close(f);
    }
}
```

Symbolic Link Attacks (cont.)

- attacker could create a symlink to an important system file

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

- example 2 of vulnerable code

```
$ export ATTACKED_PRG_VAR="
+ +
"
$ ln -s /root/.rhosts core      # create a symlink to a config file
$ ./run_and_crash_attacked_prg    # core file overwrites the .rhosts file
$ rsh 127.0.0.1 -l root /bin/sh -i
```

- attacker force a core dump that will contain special characters “+ +” ⇒ allows anyone to log in as root remotely

File Creation and Symlinks

- dangerous context

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

open("/home/john/newfile", O_CREAT|O_RDWR, 0640); // this creates file "/tmp/nonexisten
```

- privileged programs could be tricked into creating new files anywhere on the FS
- protection strategy
  - use "`O_EXCL`" in *open* (exclusive and not follows symlinks)
  - use "`O_NOFOLLOW`" (non-portable) in *open*
- accidental creation (e.g. by using *fopen* with write access)

Attacking Sysmlink Aware Syscalls

- syscalls aware of symlink files act only for the last component of a filepath
- $\Rightarrow$ all file component excepting the last are followed, if symlinks
- example

```
$ ln -s /tmp/ /home/john
$ echo "test" > /home/john/newfile
$ unlink /home/john/newfile          # this will remove "/tmp/newfile"
```

Hard Links Attacks

- if user is allowed to create hard links to special files in the system, he could keep access on them even after they are “removed”
- similarly: preventing a program to remove a file
- is attacker creates in a sticky directory a hdlink to another users’s file, the attacker cannot remove the hdlink!
- in practice, on actual UNIX systems (at least Linux) creating hardlink is very restricted
  - e.g. creating a hard link to a root’s file is not allowed

Sensitive Files

- context: when privileged programs open existing files and modify their contents or change ownership or permissions
- vulnerable code assumed to be run in a SUID program; *userbuf* assumed to be given by the user

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

- attack example (attacker is jim)

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

Sensitive Files (cont.)

- vulnerable code

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

- attack

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

Circumventing Symbolic Link Prevention

- *lstat* could be used to detect and analyze a symlink
- *lstat* cannot distinct between different hard links to the same file
- example: code vulnerable to hdlinks, supposed to be run in a privileged program

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

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

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

Outline

Context

- an application interacting with the FS could be attacked through race conditions method, if it is suspended to an inopportune moment
- example of a vulnerable code

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

// ... moment of inopportunity

// safe to open file
fd = open("/tmp/userfile", O_RDONLY);
```

Time of Check To Time of Use (TOCTOU

- does not necessarily correspond only to FS manipulation
- situation: the state of a resource could be changed between the time its state is checked and time it is actually used
- could seem improbable such a situation to happen, still it could occur or could be induced
  - actions that slow down the system, like network-intensive flood of data, or heavy use of FS
  - send job control signal to the application to stop and start it constantly in a tight loop

Time of Check To Time of Use (TOCTOU (cont.)

- monitor application execution (e.g. file's time stamps, system call trace etc.)

The stat() Syntax and Functionality

- syntax

```
int stat(const char *pathname, struct stat *buf);
```

- return information from inode
- *lstat* acts on symbolic link, not following it ⇒ could be used to avoid symlink attacks
- checking for the number of hard links could help avoiding hard link attacks
- vulnerability: change of the file after the *stat* check

Race Condition Avoidance Attempt

- try inverting the order of actions: instead of “check and use” make “use (i.e. open) and check”
- example (still vulnerable!)

```
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");
```

Race Condition Avoidance Attempt (cont.)

- attack
  1. create a symlnik file to a sensitive file ⇒ the application opens the sensitive file
  2. before *lstat* call, remove/rename the symlink and create instead a regular file ⇒ the check passes (still, the sensitive file is to be used!)
- note: deleting an opened file works even for regular files ("*UNIX syntax*")

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

File Race Redux

- problems with system calls using path names, involving a path resolution each time they are called
- example: vulnerable code (could investigate different files)

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

- code audit: any time you see multiple successive system calls using the same pathname, evaluate what happens if the path is manipulated between different syscalls
- protection strategy: use system calls that use file descriptors ⇒ the use and check are linked together

File Race Redux (cont.)

- example: secured code (both *fstat* access the same inode)

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

- protected even to file removal or rename, due the “Unix remove semantic” (inode is not released until file is closed)

Permission Races

- problem: if an application creates temporarily a file with improper permissions (e.g. public access)
- if an attacker could open the file in the exposing period, he keeps access to the file even if the application corrects permissions (restricts them) later
- example: code vulnerable by exposing for a time improper permissions on the file (depends on the *umask* value)

```
if (!(fp = fopen(fname, "w+")))  // calls open(fname, ..., 0666) !!!
    die("fopen");

fd = fileno(fd);
if (fchmod(fd, 0600) < 0)  // avoid TOCTOU attackes, by using the fd
die("fchmod");
```

Ownership Races

- context
  - a file created with the effective privileges of a non-privileged user,
  - file's ownership later changed to that of the privileged user
- race condition
  - the non-privileged user (attacker) could access file between file creation and ownership change

Ownership Races (cont.)

- example of code vulnerable to race conditions

```
drop_privs();

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

regain_privs();

// take ownership
if (fchmod(fd, geteuid(), getegid()) < 0)
  die("fchmod");
```

Directory Races

- take care of to situations when an application traverse user-controlled file system hierarchy
- problem: infinitely recursive symbolic links (cycles)
  - kernel detect cycles when making path resolution
  - problems when application traverse by itself a sub-tree
- symbolically linked directories could not be reflected in shell commands
- system calls (i.e. getcwd) reflect it, though
- example

Directory Races (cont.)

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

Directory Symlinks for Exploiting unlink()

- consider the effect of malicious users manipulating directories that are one or two levels higher than a process' working space
- example: vulnerable code in some implementation of the "*at*" command, when called to remove a job (scheduled task)

Directory Symlinks for Exploiting unlink() (cont.)

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

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

unlink("JOBNAME");
```

- first attack vector

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

- that would delete another file than a job, but though only if it belongs to the calling user

Directory Symlinks for Exploiting unlink() (cont.)

- BUT: there is a race condition between checking the file and its removal, which can be exploited
  - between the two moments: remove the user's regular file and replace it by a symbolic link to a sensitive file

```
$ mkdir /tmp/bob
$ touch /tmp/bob/shadow
$ at -r ../../../../..//tmp/bob/shadow
$ rm -fr /tmp/bob/
$ ln -s /etc /tmp/bob    # if race conditions are met,
                         # /etc/shadow will be deleted !!!
```

- remember that *unlink()* does not follow symbolic links for last element in a path

Moving Directories Underneath a Program

- vulnerable program run

```
rm -fr /tmp/a    # removing /tmp/a/b/c
```

- syscall trace of the program run

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

Moving Directories Underneath a Program (cont.)

- attack vector
  - act before first “`chdir("..");`”
  - move “c” directory underneath “`/tmp`”
  - when go one level upwards, the “*rm*” programs will be in “`/tmp`”, going upper layers than supposed to removing the entire FS!

Outline

Unique File Creation With *mktemp()*

- takes a user template for a filename and fills it out so that it represents a unique, unused filename
- the template contains XXX characters as placeholders for random data
- can be easily predicted because is based on the process ID plus a simple patter
- example: vulnerable code

Unique File Creation With *mktemp()* (cont.)

```
char temp[1024];

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

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

- vulnerability: race condition between calls of *mktemp* and *open*
- example: about some version of GCC
  - gcc makes use of *mktemp* to create a common pattern for all its temporal files in */tmp* (the first one ends in ".i")

Unique File Creation With *mktemp()* (cont.)

- an attacker monitor for the occurrence of an “.i” file and than creates symbolic links for other file types, like: “.o”, “.s”
- if the root compiles something, it can be tricked into overwriting a sensitive file

- NOTE: *mktemp* almost always indicates a potential race condition

Unique File Creation With *tmpnam()* and *tempnam()*

- similar to *mktemp()* as functionality
- also suffer the same security problem: race conditions
- example: vulnerable code in *xpdf-0.90*

```
tmpnam(tmpFileName);
if (!(f = fopen(tmpFileName, "wb")))
  die("fopen");
```

Unique File Creation With *mkstemp()*

- much safer than *mktemp*
- finds a unique filename, creates the file and opens it, returning the corresponding file descriptor
- example: wrong way to use the *mkstemp()* function (reusing the `g_mytmpfile` after file creation)

Unique File Creation With *mkstemp()* (cont.)

```
char g_mytmpfile[1024];

void init_prog()
{
  strcpy(g_mytmpfile, "/tmp/tmpXXXXXXX");

  if ((fd = mkstemp(g_mytmpfile)) < 0)
    die("mkstemp");

  initialize_tmpfile(fd);
  close(fd);

}

void main_loop()
{
  if ((fd = fopen(g_mytmpfile, "rw")) == NULL)
    die("open");
}
```

Unique File Creation With *tmpfile()* and *mkdtemp()*

- *tmpfile* similar to *mkstemp*, but just creates a stream handler, instead of a file descriptor
- *mkdtemp* for creating temporary directories

File Reuse

- applications sometimes need using temporary files that already exists in a temporary directory
- such files could have a unique know filename or their names could be passed along between processes
- opening such files safety is difficult
  - make sure they are not links to sensitive files (others than the original file)
  - using *lstat* to prevent symlinks introduce race condition vulnerabilities
  - using *open* and *fstat* a symbolic link would be followed

File Reuse (cont.)

- even if the last component in a filepath is avoided, had it be a symlink, the other components will not be, if symlinks also
- hard link counter could also be made 1, if an attacker manipulated a hard link opened by the application is removed after *open* and before checking with *fstat*

Cryogenic Sleep Attack

- code sample: a reasonable safe idiom for opening a potentially existing file in a public directory

```
if (lstat(fname, &st1) >= 0) {
  if (!S_ISREG(st1.st_mode) || (st1.st_nlink > 1) )
    die("vulnerable");

  fd = open(fname, O_RDWR);
  if (fd < 0 || fstat(fd, &st2) < 0)
    die("open or fstat");

  if ((st1.st_ino != st2.st_ino) || (st1.st_dev != st2.st_dev) ||
      (st2.st_nlink > 1))
    die("check");
} else {
  fd = open(fname, O_RDWR | O_CREAT | O_EXCL, FMODE);
  if (fd < 0)
    die("creat");
}
```

Cryogenic Sleep Attack (cont.)

- the code is fairly robust
- still the device and inode check could be bypassed
  - after file was checked not to be a symbolic link and before *open* is called the attacker send the application `SIGSTOP` signal, suspending it
  - he get the device and inode numbers of the checked file and after that remove it
  - wait for the system to create a sensitive file with those numbers (try forcing that by running some other SUID programs)

Cryogenic Sleep Attack (cont.)

- after such a file is created, a symbolic link, having the same name like the removed file is created to point to the sensitive file
- the attacked application is resumed and it will open the file, noting the same identity, but following (*fstat*) the symbolic link

Outline

Opening a File

- done witg *fopen()*

```
FILE* fopen(char *path, char *mode);
```

- same problems like open regarding validation of path
- mode: “r”, “r+”, “w”, “w+”, “a”, “a+”
- care must be taken not to create a file accidentally
- permission rights at creation, by default “0666” (also influenced by *umask*, which is inherited from parent)
- *freopen()*: reopen a previously opened file stream – same vulnerabilities like *fopen()*

Opening a File (cont.)

- *fdopen()*: create a FILE structure for a preexisting socket descriptor – not vulnerable

Reading From a File

- *fread()* function

```
int fread(void *buf, size_t size, size_t count, FILE *fp);
```

- could be exposed to integer overflow due to multiplication of *size* and *count*
- *fgets* function

```
char* fgets(char *buf, size_t size, FILE *fp);
```

- example vulnerability due to not checking return value (NUL-terminated string is done only at success)

Reading From a File (cont.)

```
int read_email(FILE *fp)
{
    char user[1024], domain[1024];
    char buf[1024];
    int length;

    fgets(buf, sizeof(buf), fp);
    ptr = strchr(buf, '@');

    if (!ptr)
    return -1;

    *ptr++ = '\0';

    strcpy(user, buf);
    strcpy(domain, ptr);
}
```

- *fscanf()* function (like *scanf()*)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Writing Into a File

- *fprint()* - same problems like *printf()*
- inconsistencies in how the file should be formatted (if user can insert delimiters the application did not check for)
- example: vulnerable code, when one of the fields contains delimiters ':' and/or '
n'

Writing Into a File (cont.)

```
int update_info(FILE *fp, struct passwd *pw)
{
    if (fprintf(fp, "%s:%s:%lu:%lu:%s:%s:%s\n",
                pw->pw_name, pw->pw_passwd,
                pw->pw_uid, pw->pw_gid,
                pw->pw_gecos, pw->pw_dir,
                pw->pw_shell)
    return -1
    return 0;
}
```

Closing a File

- *fclose()* function

```
int fclose(FILE *stream);
```

- failure to close a file could result in a file descriptor leak
- the function also releases the FILE structure, so closing the same stream twice could result in memory corruption

Bibliography

1. “The Art of Software Security Assessments”, chapter 9, “UNIX 1. Privileges and Files”, pp. 459 – 559
2. “Setuid Demystified”, https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf