OS Specific Vulnerabilities

Failure to Protect Stored Data

Purpose and Contents

The purpose of this lecture

- 1. presents vulnerabilities due to bad manipulation of stored data permission
- 2. presents OS-specific (Linux and Windows) permission-assignment mechanisms and re-

lated vulnerabilities

The “Failure to Protect Stored Data” Vulnerability

- Description
  - do not protect data at rest, i.e. data that is stored
    - worst case: do not protect at all
    - do not protect correctly
  - two faces / components
    - weak of missing access control mechanisms
    - lousy or missing data encryption
  - cases of weak access control mechanisms
    - allow full control access to everyone
    - allow full control access to non-privileged users
    - allow write permission on a (system) executable

CWE References

- CWE-264: “Permissions, Privileges, and Access Controls”
  - most general
  - related to the management of permissions, privileges, and other security features that are used to perform access control
- CWE-284: “Improper Access Control”
  - does not restrict or incorrectly restricts access to a resource from an unauthorized actor
- CWE-693: “Protection Mechanism Failure”
  - does not use or incorrectly uses a protection mechanism that provides sufficient defense against directed attacks against the product
- CWE-282: “Improper Ownership Management”
  - assigns the wrong ownership, or does not properly verify the ownership, of an object
- CWE-275: “Permission Issues”
  - related to improper assignment or handling of permissions

Actual Relevance

Related vulnerabilities

- information leakage
  - accidental leakage of data through application error messages and various side-channels
- race conditions
  - can lead to permission-based race condition issues that cause leakage
- the use of weak password-based systems
  - passwords used to protect the data are lousy, regardless of how good the encryption
- using the wrong cryptography
  - using your own crypto or one that supports a small key is sinful

Identify the Vulnerability

- look for code that
  - sets access controls AND
  - grants write access to low-privileged users
- look for code that
  - creates an object without setting access controls AND
  - creates the object in a place writable by low-privileged users
- look for code that
  - writes configuration information into a shared (which means weak protection) area
- look for code that
  - writes sensitive information into an area readable by low-privileged users
- look for files with weak permission
  - e.g. in Linux: find / -type f -perm +002

Redemption Steps

- do not use lousy permissions or ACLs and encrypt data correctly
- you must vouch for every bit that’s set in a permission and
- make sure you’re not exposing data and binaries to corruption or disclosure
- you should write your binaries to the computer’s system directories or a similarly protected location,
- and write user-specific data to the current user’s home directory

2 Linux File Permissions and Related Vulnerabilities

File System Permissions

- 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
- 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
- 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

File Creation

Opening flags and permissions

- syntax: 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

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

Public Directory

- vulnerable code: do not check for existence of file in a publicly accessible directory

```
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 symlink (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: do not check for existence of file in a regular user accessible directory

```
drop_privs();

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

regain_privs();
```

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

Filenames and Paths (review)

- 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 to one
- for example “/.///./../usr/..///../usr////./bin///.//file”⇒“/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 code vulnerable to 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);
}
```

- an attacker could provide a path like

"/usr/lib/safefiles/../../../../../etc/passwd"

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 mostly use counted strings, not using “NUL-terminated” semantic
- ⇒ path truncation vulnerability

Dangerous Places

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

Interesting Files

- system configuration files
  - “/etc/*”, “/etc/passwd”, “/etc/shadow”, “/etc/hosts.equiv”, “.rhosts”, “.shosts”, “/etc/ld.preload.so”
- user’s personal files
  - “.history”, “.bash_history”, “.profile”, “.bashrc”, “.login”, mail spools
- program configuration files and data
  - “.htpasswd”, source code of scripts, “sshd_config”, “authorized_keys”, temporary files
- others
  - “/var/log/*”, kernel and boot files, executable and libraries, “/dev/*”, “/proc/*”, named pipes

Links
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(tmpbuf);
    close(f);
  }
}
```

- attacker could create a symlink to an important system file

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

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/nonexistent"
```

- 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 file path
- ⇒ all file
- example

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

# this will create the file "/tmp/newfile"
$ echo "test" > /home/john/newfile

# this will remove "/tmp/newfile"
$ unlink /home/john/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
  - if attacker creates in a sticky directory a hard link to another users’ file – ⇒ the user cannot remove the hard link!
- 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 (II)

- 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 hardlinks, supposed to be run in a privileged program

```
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

- 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)

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

- 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”)

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
- 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
  - an application creates temporarily a file
  - with improper permissions (e.g. public access)
- attack
  - if an attacker could open the file in the exposing period,
  - he keeps access to the file even if the application restricts permissions later

Permission Races (II)

- example of vulnerable code: expose for a time improper permissions on the file (depends on the umask value

```
FILE *file;

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

int fd = fileno(file);

// avoid TOCTOU attackes, by using the fd
if (fchmod(fd, 0600) < 0)
    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
- 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

- context: application traverses' 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

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

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");
```

- 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!

Temporary Files

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 of vulnerable code

```
char temp[1024];

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

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

Temporary Files (II)

- 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”)
  - 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

Bibliography

1. “The Art of Software Security Assessments”, chapter 9, “UNIX 1. Privileges and Files”, pp. 459 – 559
2. “24 Deadly Sins of Software Security”, chapter 17, “Failure to Protect Stored Data”.