https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2026

Lecture 16: OS Security III

Instructor: **Nikos Triandopoulos**

March 31, 2026

BROWN

CS1660: Announcements

- Course updates
  - Project 1, project 2, HW 1, HW2, midterm
  - Project 3 is due April 3
  - Project 4 comes out tomorrow
  - 4 weeks left

Last class

- Cryptography
- Authentication
- Web security
  - Cross-site references and risks
    - SQL injection + Cross-site scripting (XSS)
    - Database security + Buffer overflow attacks
- Operating system (OS) security
  - Access control, OS access control, file-system access control

Today

- Cryptography
- Authentication
- Web security
  - Cross-site references and risks
    - SQL injection + Cross-site scripting (XSS)
    - Database security + Buffer overflow attacks
- Operating system (OS) security
  - Access control, OS access control, file-system access control

15.1 File-system access control

Linux Vs. Windows

- Linux
  - Allow-only ACEs
  - Access to file depends on ACL of file and of all its ancestor folders
  - Start at root of file system
  - Traverse path of folders
  - Each folder must have execute (cd) permission
  - Different paths to same file not equivalent
  - File's ACL must allow requested access
- Windows
  - Allow and deny ACEs
  - By default, deny ACEs precede allow ones
  - Access to file depends only on file's ACL
  - ACLs of ancestors ignored when access is requested
  - Permissions set on a folder usually propagated to descendants (inheritance)
  - System keeps track of inherited ACE's

Linux file AC

- File Access Control for:
  - Files
  - Directories
  - Therefore…
    - \dev\ : *devices*
    - \mnt\ : *mounted file systems*
    - What else? *Sockets, pipes, symbolic links*…

Unix permissions

- Standard for all UNIXes
- Every file is owned by a user and has an associated group
- Permissions often displayed in compact 10-character notation
- To see permissions, use ls –l

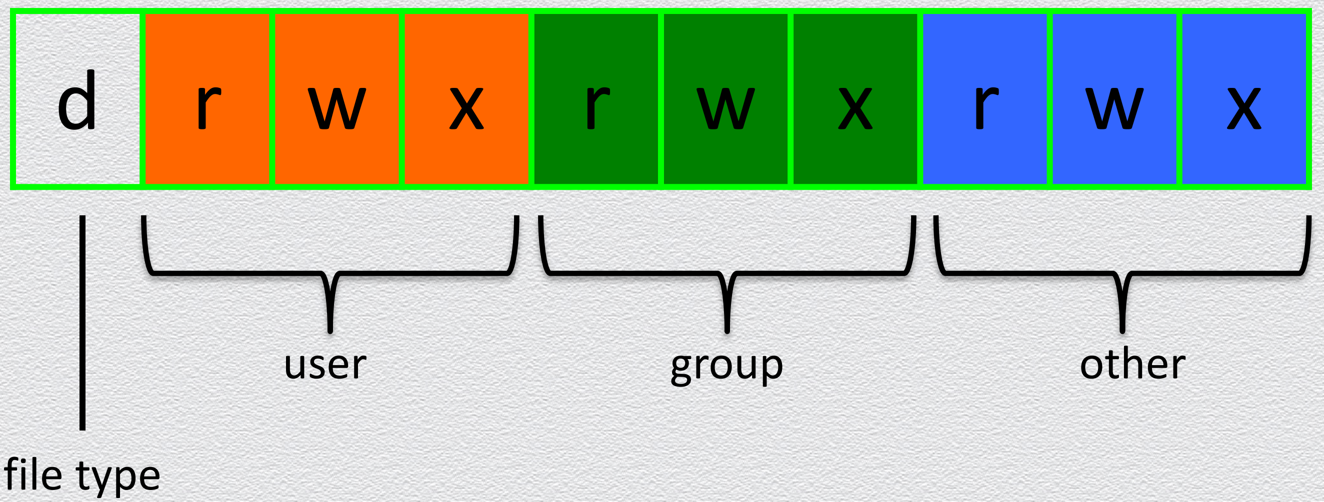
jk@sphere:~/test$ ls –l

total 0

-rw-r----- 1 jk ugrad 0 2005-10-13 07:18 file1

-rwxrwxrwx 1 jk ugrad 0 2005-10-13 07:18 file2

Unix file types and basic permissions



Permission examples (regular files)

-rw-r—r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden to others
-rwx------	read/write/execute for owner, forbidden to everyone else
-r--r--r--	read-only to everyone, including owner
-rwxrwxrwx	read/write/execute to everyone

Permissions for directories

- Permissions bits interpreted differently for directories
- *Read* bit allows listing names of files in directory, but not their properties like size and permissions
- *Write* bit allows creating and deleting files within the directory
- *Execute* bit allows entering the directory and getting properties of files in the directory
- Lines for directories in ls –l output begin with d, as below:

jk@sphere:~/test$ ls –l

Total 4

drwxr-xr-x 2 jk ugrad 4096 2005-10-13 07:37 dir1

-rw-r--r-- 1 jk ugrad   0 2005-10-13 07:18 file1

Permission examples (directories)

drwxr-xr-x	all can enter and list the directory, only owner can add/delete files
drwxrwx---	full access to owner and group, forbidden to others
drwx--x---	full access to owner, group can access known filenames in directory, forbidden to others
-rwxrwxrwx	full access to everyone

Question

Select the correct symbolic notation for a directory whose user class has full permissions, group class has read and execute permissions, and others class has only read permissions.

A. -rwxr-xr--

C. drwxr--r--

B. lr-xr-xr--

D. drwxr-xr--

Answer

Select the correct symbolic notation for a directory whose user class has full permissions, group class has read and execute permissions, and others class has only read permissions.

A. -rwxr-xr--

B. lr-xr-xr--

C. drwxr--r--

D. drwxr-xr--

Changing permissions

- Permissions are changed with chmod or through a GUI like KDE Konqueror
- Only the file owner or root can change permissions
- If a user owns a file, the user can use chgrp to set its group to any group of which the user is a member
- root can change file ownership with chown (and can optionally change group in the same command)
- chown, chmod, and chgrp can take the -R option to recur through subdirectories

Changing permissions examples

chown -R root dir1	Changes ownership of dir1 and everything within it to root
chmod g+w,o-rwx file1 file2	Adds group write permission to file1 and file2, denying all access to others
chmod -R g=rwX dir1	Adds group read/write permission to dir1 and everything within it, and group execute permission on files or directories where someone has execute permission
chgrp testgrp file1	Sets file1's group to testgrp, if the user is a member of that group
chmod u+s file1	Sets the setuid bit on file1. (Doesn't change execute bit.)

Special permission bits

- Three other permission bits exist
  - Set-user-ID (“suid” or “setuid”) bit
  - Set-group-ID (“sgid” or “setgid”) bit
  - Sticky bit

Set-user-ID

- Set-user-ID ("suid" or "setuid") bit
  - On executable files, causes the program to run as file owner regardless of who runs it
  - Ignored for everything else
  - In 10-character display, replaces the 4th character (x or -) with s (or S if not also executable)

    -rwsr-xr-x: setuid, executable by all

    -rwxr-xr-x: executable by all, but not setuid

    -rwSr--r--: setuid, but not executable - not useful

Setuid programs

- Unix processes have two user IDs:
  - real user ID: user launching the process
  - effective user ID: user whose privileges are granted to the process
- An executable file can have the set-user-ID property (setuid) enabled
- If a user A executes setuid file owned by B, then the effective user ID of the process is B and not A

Setuid programs (cont.)

- System call setuid(uid) allows a process to change its effective user ID to uid
- Some programs that access system resources are owned by root and have the setuid bit set (setuid programs)
  - e.g., passwd and su
- Writing secure setuid programs is tricky because vulnerabilities may be exploited by malicious user actions

Set-group-ID

- Set-group-ID ("sgid" or "setgid") bit
- On executable files, causes the program to run with the file's group, regardless of whether the user who runs it is in that group
- On directories, causes files created within the directory to have the same group as the directory, useful for directories shared by multiple users with different default groups
- Ignored for everything else
- In 10-character display, replaces 7th character (x or -) with s (or S if not also executable)

  -rwxr-sr-x: setgid file, executable by all

  drwxrwsr-x: setgid directory; files within will have group of directory

  -rw-r-Sr--: setgid file, but not executable - not useful

Sticky bit

- On directories, prevents users from deleting or renaming files they do not own
- Ignored for everything else
- In 10-character display, replaces 10^{th} character (x or -) with t (or T if not also executable)

drwxrwxrwt: sticky bit set, full access for everyone

drwxrwx--T: sticky bit set, full access by user/group

drwxr--r-T: sticky, full owner access, others can read *(useless)*

Symbolic link

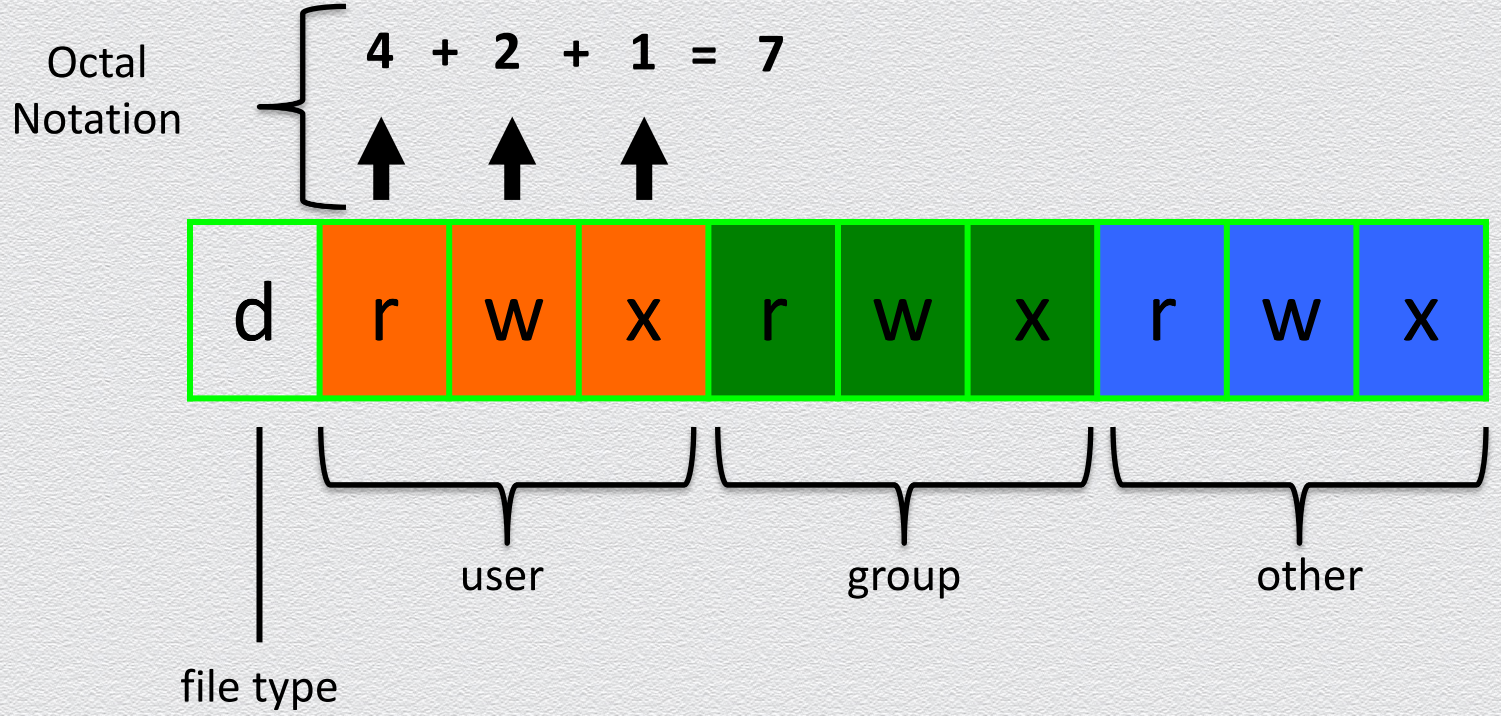
- In Unix, a symbolic link (aka symlink) is a file that points to (stores the path of) another file
- A process accessing a symbolic link is transparently redirected to accessing the destination of the symbolic link
- Symbolic links can be chained, but not to form a cycle

- ln -s really_long_directory/even_longer_file_name myfile

Octal notation

- Standard syntax is nice for simple cases, but bad for complex changes
  - Alternative is octal notation, i.e., three or four digits from 0 to 7
- Digits from left (most significant) to right(least significant):
  [special bits][user bits][group bits][other bits]
- Special bit digit =
  (4 if setuid) + (2 if setgid) + (1 if sticky)
- All other digits =
  (4 if readable) + (2 if writable) + (1 if executable)

Unix file types and octal notation



Octal notation examples

644 or 0644	read/write for owner, read-only for everyone else
775 or 0775	read/write/execute for owner and group, read/execute for others
640 or 0640	read/write for owner, read-only for group, forbidden to others
2775	same as 775, plus setgid (useful for directories)
777 or 0777	read/write/execute to everyone *(dangerous!)*
1777	same as 777, plus sticky bit

Root

- “root” account is a super-user account, like Administrator on Windows
- Multiple roots possible
- File permissions do not restrict root
- This is *dangerous*, but necessary, and OK with good practices

Becoming root

- su
  - Changes home directory, PATH, and shell to that of root, but doesn't touch most of environment and doesn't run login scripts
- sudo <command>
  - Run just one command as root
- su [-] <user>
  - Become another non-root user
  - Root does not require to enter password

The /tmp directory

- In Unix systems, directory /tmp is
  - Readable by any user
  - Writable by any user
  - Usually wiped on reboot
- Convenience
  - Place for temporary files used by applications
  - Files in /tmp are not subject to the user’s space quota
- What could go wrong?
  - Sharing of resources may lead to vulnerabilities

Limitations of Unix permissions

- Unix permissions are not perfect
  - Groups are restrictive
  - Limitations on file creation
- Linux optionally uses POSIX ACLs
  - Builds on top of traditional Unix permissions
  - Several users and groups can be named in ACLs, each with different permissions
  - Allows for finer-grained access control
- Each ACL is of the form *type*:[*name*]:*rwx*
  - Setuid, setgid, and sticky bits are outside the ACL system

Gone for 10 seconds

- You leave your desk for 10 seconds without locking your machine
- The attacker sits at your desk and types:

  % cp /bin/sh /tmp

  % chmod 4777 /tmp/sh
- The first command makes a copy of shell sh
- The second command makes sh a
- What happens next?
- The attacker can run the copy of the shell with your privileges
- For example:
  - Can read your files
  - Can change your files

Historical setuid Unix vulnerabilities: lpr

- Command lpr
  - running as root setuid
  - copied file to print, or symbolic link to it, to spool file named with 3-digit job number (e.g., print954.spool) in /tmp
  - Did not check if file already existed
  - Random sequence was predictable and repeated after 1,000 times
- How can we exploit this?
- Attack
  - A dangerous combination: setuid, /tmp, symlinks, …
  - Create new password file newpasswd
  - Print a very large file
  - lpr –s /etc/passwd
  - Print a small file 999 times
  - lpr newpasswd
  - The password file is overwritten with newpasswd

Beyond setuid and files

- Writing setuid programs is tricky
  - Easy to inadvertently create security vulnerabilities
  - Unix variants have subtle different behaviors in setuid-related calls
- Access control to files is tricky
  - A user file can be accessed by any user process
  - Shared folders and predictable file names create security vulnerabilities
- Consider alternatives
  - Manage system resources via services
  - Use databases instead of files and shared folders
  - Use RPCs (including database queries) to request access to system resources

15.1 File-system access control

setuid/setgid

Special permissions bits:

- setuid (Set User ID)
  - executable runs with privileges of <u>owner</u>, regardless of who runs it
- setuid (Set Group ID)
  - executable runs with privileges of <u>group</u>, regardless of who runs it

Unprivileged user can run program with higher privileges!
=> Powerful, but very dangerous

setuid/gid: The effects

Disclaimer

setuid/setgid is dangerous. Using it incorrectly can cause serious problems.

Just as you should never implement your own crypto,

you should not write your own setuid/setgid programs.

You are about to see why.

Background: environment variables

System variables that control how processes execute

Set up when a user logs in, as part of shell

```
# Get variables
cs1660-user@6010f6e96b02:~$ echo $TERM
xterm
cs1660-user@6010f6e96b02:~$ echo $PWD
/home/cs1660-user

# Set a variable
cs1660-user@6010f6e96b02:~$ export SOMETHING=hello
cs1660-user@6010f6e96b02:~$ echo $SOMETHING
Hello

# Show the environment
cs1660-user@6010f6e96b02:~$ env
. . .
```

Scope is per-shell: log out/open new term => different vars

Background: $PATH

Where the shell looks when you run programs

=> List separated by ":", traversed in order

```
# Get variables
cs1660-user@6010f6e96b02:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/go/bin


# which:  $PATH lookup
cs1660-user@6010f6e96b02:~$ which ls
/usr/bin/ls

cs1660-user@6010f6e96b02:~$ which go
/usr/local/go/bin/go
```

Problems

Input from user pollutes execution environment

=> Another form of code injection!

Not every command can be overridden…

Background: symbolic links

Indirection in the filesystem: path of one file can point to another

```
# Create a symlink
registrar@ceres:~$ ln –sv scripts/reg-v01.sh reg.sh
reg.sh -> scripts/reg-v01.sh


# How it looks
registrar@ceres:~$ ls –la reg.sh
lrwxrwxrwx 1 reg reg 9 Mar 12 16:40 reg.sh -> scripts/reg-v01.sh



# eg. Use it like a normal file
registrar@ceres:~$ ./reg.sh
```

Problem: anyone can create a symlink to anything!
=> Permissions checked on **access**, not at **creation**

What can go wrong?

TOCTOU: Time of check/time of use

```
# Check for access
if ! __effective_user_can_access $code_from_user; then
        echo "You don't have permission to view this file"
        exit 1
fi



# Do the access
if cmp --silent $code_expected $code_from_user; then
    echo "Override code approved!"
    add_to_course $course $user
else
    echo "Please use a valid override code"
fi
```

A race condition!

So why is setuid/gid bad?

So why is setuid/gid bad?

Up to the developer to decide what parts of the program can run with elevated privileges

=> Particularly dangerous for shell scripts

So setuid/setgid is dangerous...

setuid/setgid is dangerous...

In modern times: only for programs that <u>really</u> need it

- System programs that changing passwords/users, legacy programs
  - <u>Don't do this yourself!</u>
- Very very bad idea for shell scripts

What else can we do?

When do we need this?

In the shell: su, sudo

- Run as another user (if you have permissions)

```
user@shell:~$ su –c "command" other user
```

- Run commands as root (or another user) based on system config file (/etc/sudoers)
  - Can restrict to specific commands, environment, ...

```
user@shell:~$ sudo whoami
root
```

```
/etc/sudoers:
%wheel ALL=(ALL) NOPASSWD: ALL

. . .
```

From man page on /etc/sudoers: (aka sudoers(5))

```
ALL         CDROM = NOPASSWD: /sbin/umount /CDROM,\
            /sbin/mount -o nosuid\,nodev /dev/cd0a /CDROM
```

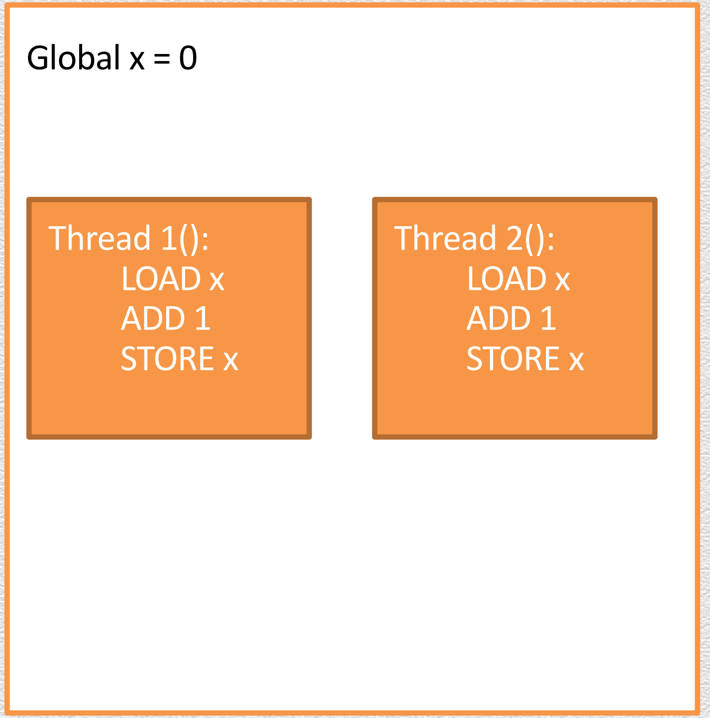
Any user may mount or unmount a CD-ROM on the machines in the CDROM Host_Alias (orion, perseus, hercules) without entering a password.

sudo has a LOT of features, see
man sudoers for details!

Race Condition

- A race condition occurs when two threads want to access the same memory

- Run Thread 1() and Thread 2()
  - Outcome is 1 or 2



Race Condition

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

- Fragment of setuid program that writes into file /tmp/X on behalf of a user who created it
- access verifies permission of real user ID
  - Transparently follows symlinks
- open verifies permission of effective user ID
  - Transparently follows symlinks
- What can go wrong?

TOCTOU Vulnerability

```
1.  if (!access("/tmp/X", W_OK)) {
        /* the real user ID has access right */
2.      f = open("/tmp/X", O_WRITE);
3.      write_to_file(f);
        }
    else {
        /* the real user ID does not have
        access right */
4.      fprintf(stderr, "Permission denied\n");
        }
```

- What can go wrong?
  - In between (1) and (2), user could replace /tmp/X with symlink to /etc/passwd
  - Not easy to accomplish (timing)
- Example of time of check to time of use (TOCTOU) vulnerability

Attempt to Fix the Race Condition

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.   int f = open("/tmp/X", O_RDWR);
4.   fstat(f, &statAfter);
5.   if (statAfter.st_ino == statBefore.st_ino) {
        /* the I-node is still the same */
6.      write_to_file(f);
      }
7.   else perror("Race Condition Attacks!");
      }
8. else fprintf(stderr, "Permission denied\n");
   }
```

- lstat and fstat access file descriptor for a path, which includes unique file ID (st_ino)
  - lstat does not traverse symlink
  - fstat accesses descriptor of open file, after symlink traversed by open
- Step (5) compares IDs of
  - file checked in (1) and
  - file opened in (3)
- Check-use-check_again approach
  - Defeats swapping in symlink between access and open
- Fails also if /tmp/X is a symlink when (2) is executed

Does the Fix Work?

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.    int f = open("/tmp/X", O_RDWR);
4.    fstat(f, &statAfter);
5.    if (statAfter.st_ino == statBefore.st_ino) {
          /* the I-node is still the same */
6.        write_to_file(f);
        }
7.    else perror("Race Condition Attacks!");
        }
8. else fprintf(stderr, "Permission denied\n");
    }
```

- New attack
  - Before (1) /tmp/X is a hard link to /etc/passwd
  - Between (1) and (2) swap in hard link to user-owned file
  - Between (2) and (3) swap in again hard link to /etc/passwd
- This passes the ID check in (5) and allows the user to write to /etc/passwd

Negative Result

- Assumptions
  - Setuid program
  - Path-based permission check for real user ID via syscall access(path, permission) that returns 0 or -1
  - No atomic check-and-open file syscall
- Theorem
  - Program is vulnerable to TOCTOU race condition
- Proof
  - Attacker can always swap good file before access and bad file after access
  - lstat/fstat do not help since they are path-based as well
- Reference
  - Drew Dean, Alan J. Hu: Fixing Races for Fun and Profit: How to Use access (2). USENIX Security Symposium, 2004.

Mitigating and Eliminating Race Conditions

- Hardness amplification
  - Force the adversary to win a large number of races instead of just one or two in order to exploit the vulnerability
  - Reduces the probability of success
  - Complex to accomplish correctly
  - Reference: Dan Tsafrir, Tomer Hertz, David Wagner, Dilma Da Silva: Portably Solving File TOCTTOU Races with Hardness Amplification. USENIX File and Storage Technologies, 2008

- Temporary privilege downgrade
  - Within same process
    - Drop to real user ID privileges via setuid(real_userid)
    - Open file
    - Restore root privileges
  - With child process
    - Fork child process with real user ID privileges to open file
  - Approach not portable across Unix variants

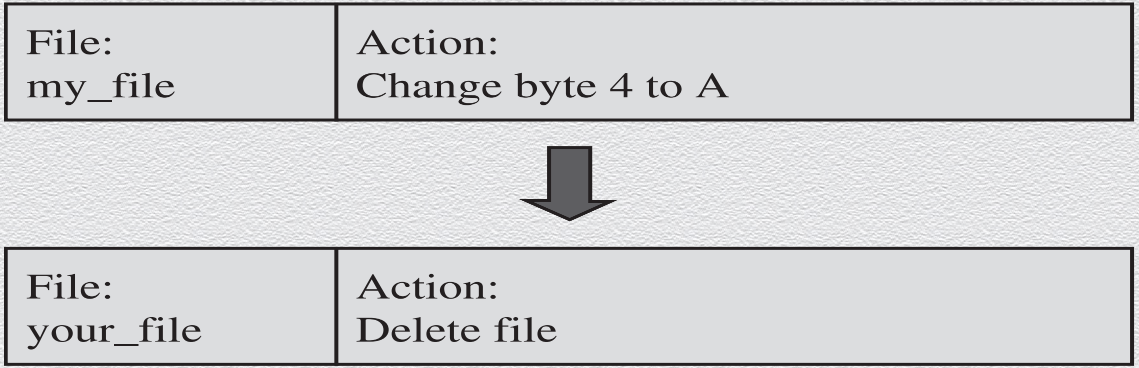
Other software security topics

Incomplete mediation

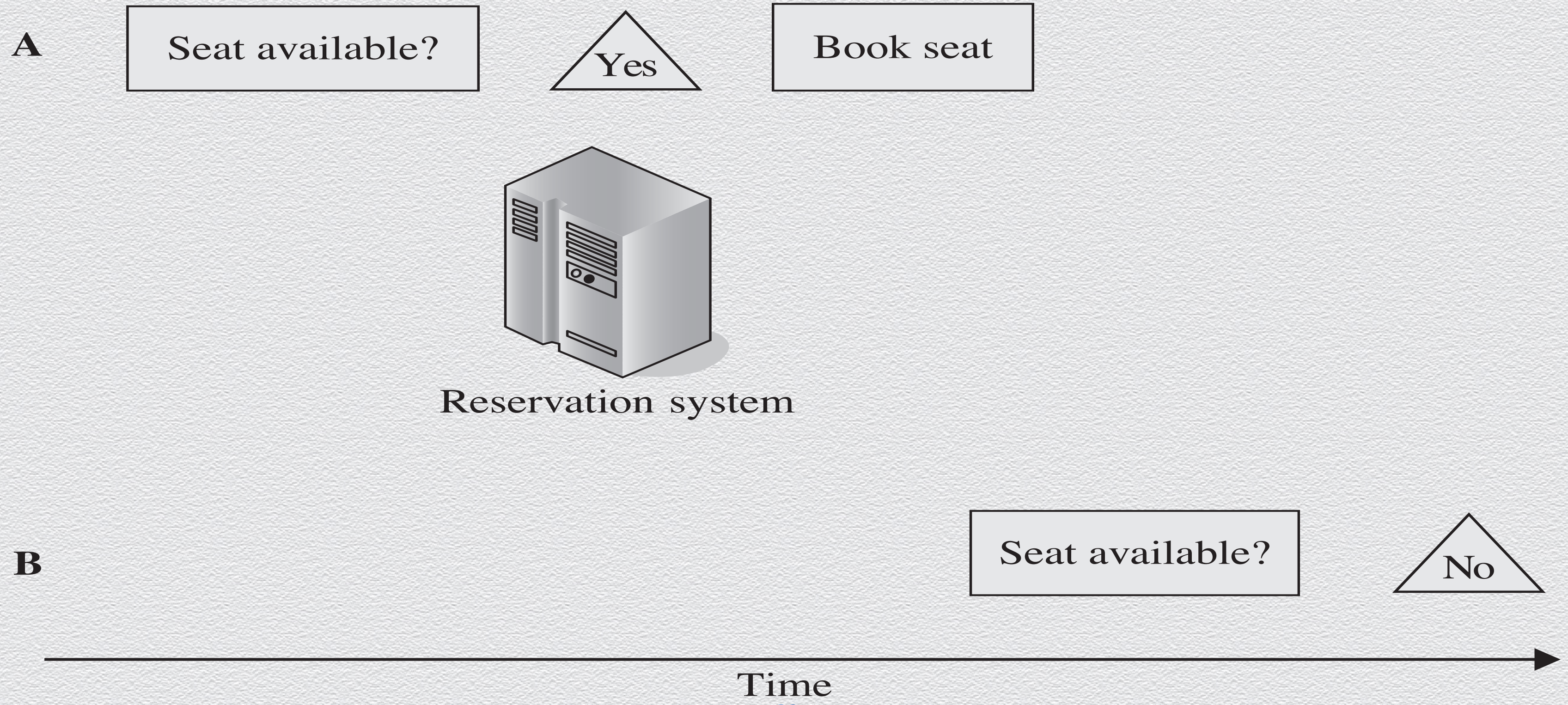
- Access control
  - what subject can perform what operation on what object
- Mediation (means checking)
  - verifying that the subject is authorized to perform the operation on an object
- Preventing incomplete mediation
  - validate all input
  - limit users’ access to sensitive data and functions
  - complete mediation using a reference monitor
    - access control that is always invoked, tamperproof and verifiable

Time-of-Check to Time-of-Use

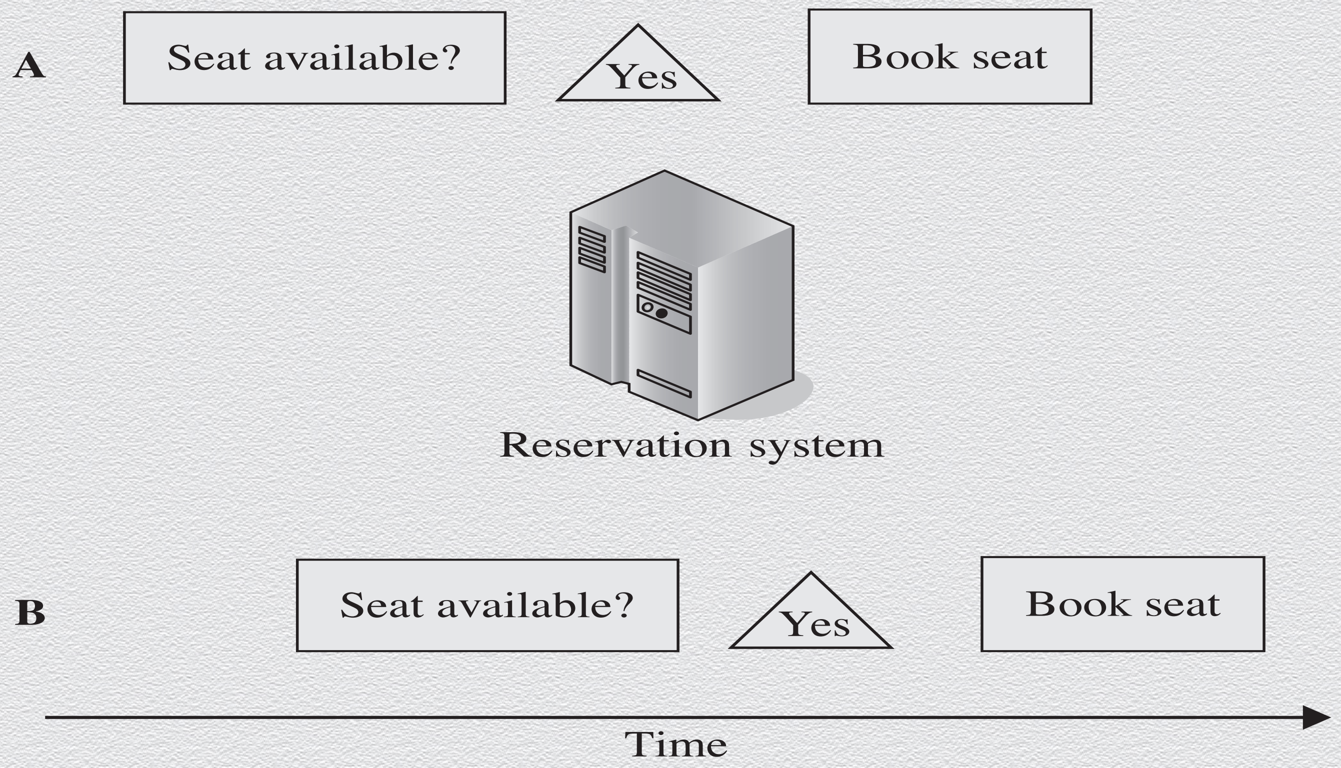
- mediation performed with a “bait and switch” in the middle
- between access check and resource use, data should remain unchanged
- exploits the details in the two processes



Race conditions



Race conditions



Other programming oversights

- Undocumented access points (backdoors)
- Off-by-one errors
- Integer overflows
- Un-terminated null-terminated string
- Parameter length, type, or number errors
- Unsafe utility libraries

Malware

Malware

- Programs planted by an agent with malicious intent
  - to cause unanticipated or undesired effects
- Virus
  - a program that can replicate itself
    - pass on malicious code to other non-malicious programs by modifying them
- Worm
  - a program that spreads copies of itself through a network
- Trojan horse
  - code that, in addition to its stated effect, has a second, nonobvious, malicious effect

Types of malware

Code Type	Characteristics
Virus	Code that causes malicious behavior and propagates copies of itself to other programs
Trojan horse	Code that contains unexpected, undocumented, additional functionality
Worm	Code that propagates copies of itself through a network; impact is usually degraded performance
Rabbit	Code that replicates itself without limit to exhaust resources
Logic bomb	Code that triggers action when a predetermined condition occurs
Time bomb	Code that triggers action when a predetermined time occurs
Dropper	Transfer agent code only to drop other malicious code, such as virus or Trojan horse
Hostile mobile code agent	Code communicated semi-autonomously by programs transmitted through the web
Script attack, JavaScript, Active code attack	Malicious code communicated in JavaScript, ActiveX, or another scripting language, downloaded as part of displaying a web page

Types of malware (cont.)

Code Type	Characteristics
RAT (remote access Trojan)	Trojan horse that, once planted, gives access from remote location
Spyware	Program that intercepts and covertly communicates data on the user or the user's activity
Bot	Semi-autonomous agent, under control of a (usually remote) controller or "herder"; not necessarily malicious
Zombie	Code or entire computer under control of a (usually remote) program
Browser hijacker	Code that changes browser settings, disallows access to certain sites, or redirects browser to others
Rootkit	Code installed in "root" or most privileged section of operating system; hard to detect
Trapdoor or backdoor	Code feature that allows unauthorized access to a machine or program; bypasses normal access control and authentication
Tool or toolkit	Program containing a set of tests for vulnerabilities; not dangerous itself, but each successful test identifies a vulnerable host that can be attacked
Scareware	Not code; false warning of malicious code attack

History of malware

Year	Name	Characteristics
1982	Elk Cloner	First virus; targets Apple II computers
1985	Brain	First virus to attack IBM PC
1988	Morris worm	Allegedly accidental infection disabled large portion of the ARPANET, precursor to today's Internet
1989	Ghostballs	First multipartite (has more than one executable piece) virus
1990	Chameleon	First polymorphic (changes form to avoid detection) virus
1995	Concept	First virus spread via Microsoft Word document macro
1998	Back Orifice	Tool allows remote execution and monitoring of infected computer
1999	Melissa	Virus spreads through email address book
2000	IloveYou	Worm propagates by email containing malicious script. Retrieves victim's address book to expand infection. Estimated 50 million computers affected.
2000	Timofonica	First virus targeting mobile phones (through SMS text messaging)
2001	Code Red	Virus propagates from 1st to 20th of month, attacks whitehouse.gov web site from 20th to 28th, rests until end of month, and restarts at beginning of next month; resides only in memory, making it undetected by file-searching antivirus products

History of malware (cont.)

Year	Name	Characteristics
2001	Code Red II	Like Code Red, but also installing code to permit remote access to compromised machines
2001	Nimda	Exploits known vulnerabilities; reported to have spread through 2 million machines in a 24-hour period
2003	Slammer worm	Attacks SQL database servers; has unintended denial-of-service impact due to massive amount of traffic it generates
2003	SoBig worm	Propagates by sending itself to all email addresses it finds; can fake From: field; can retrieve stored passwords
2004	MyDoom worm	Mass-mailing worm with remote-access capability
2004	Bagle or Beagle worm	Gathers email addresses to be used for subsequent spam mailings; SoBig, MyDoom, and Bagle seemed to enter a war to determine who could capture the most email addresses
2008	Rustock.C	Spam bot and rootkit virus
2008	Conficker	Virus believed to have infected as many as 10 million machines; has gone through five major code versions
2010	Stuxnet	Worm attacks SCADA automated processing systems; zero-day attack
2011	Duqu	Believed to be variant on Stuxnet
2013	CryptoLocker	Ransomware Trojan that encrypts victim’s data storage and demands a ransom for the decryption key

Harm from malicious code

- Harm to users and systems
  - Sending email to user contacts
  - Deleting or encrypting files
  - Modifying system information, such as the Windows registry
  - Stealing sensitive information, such as passwords
  - Attaching to critical system files
  - Hide copies of malware in multiple complementary locations
- Harm to the world
  - Some malware has been known to infect millions of systems, growing at a geometric rate
  - Infected systems often become staging areas for new infections

Transmission and propagation

- Setup and installer program
- Attached file
- Document viruses
- Autorun
- Using non-malicious programs:
  - appended viruses
  - viruses that surround a program
  - integrated viruses and replacements

Malware activation

- One-time execution (implanting)
- Boot sector viruses
- Memory-resident viruses
- Application files
- Code libraries

Virus effects

Virus Effect	How It Is Caused
Attach to executable program	• Modify file directory • Write to executable program file
Attach to data or control file	• Modify directory • Rewrite data • Append to data • Append data to self
Remain in memory	• Intercept interrupt by modifying interrupt handler address table • Load self in non-transient memory area
Infect disks	• Intercept interrupt • Intercept operating system call (to format disk, for example) • Modify system file • Modify ordinary executable program
Conceal self	• Intercept system calls that would reveal self and falsify result • Classify self as “hidden” file
Spread infection	• Infect boot sector • Infect systems program • Infect ordinary program • Infect data ordinary program reads to control its execution
Prevent deactivation	• Activate before deactivating program and block deactivation • Store copy to reinfect after deactivation

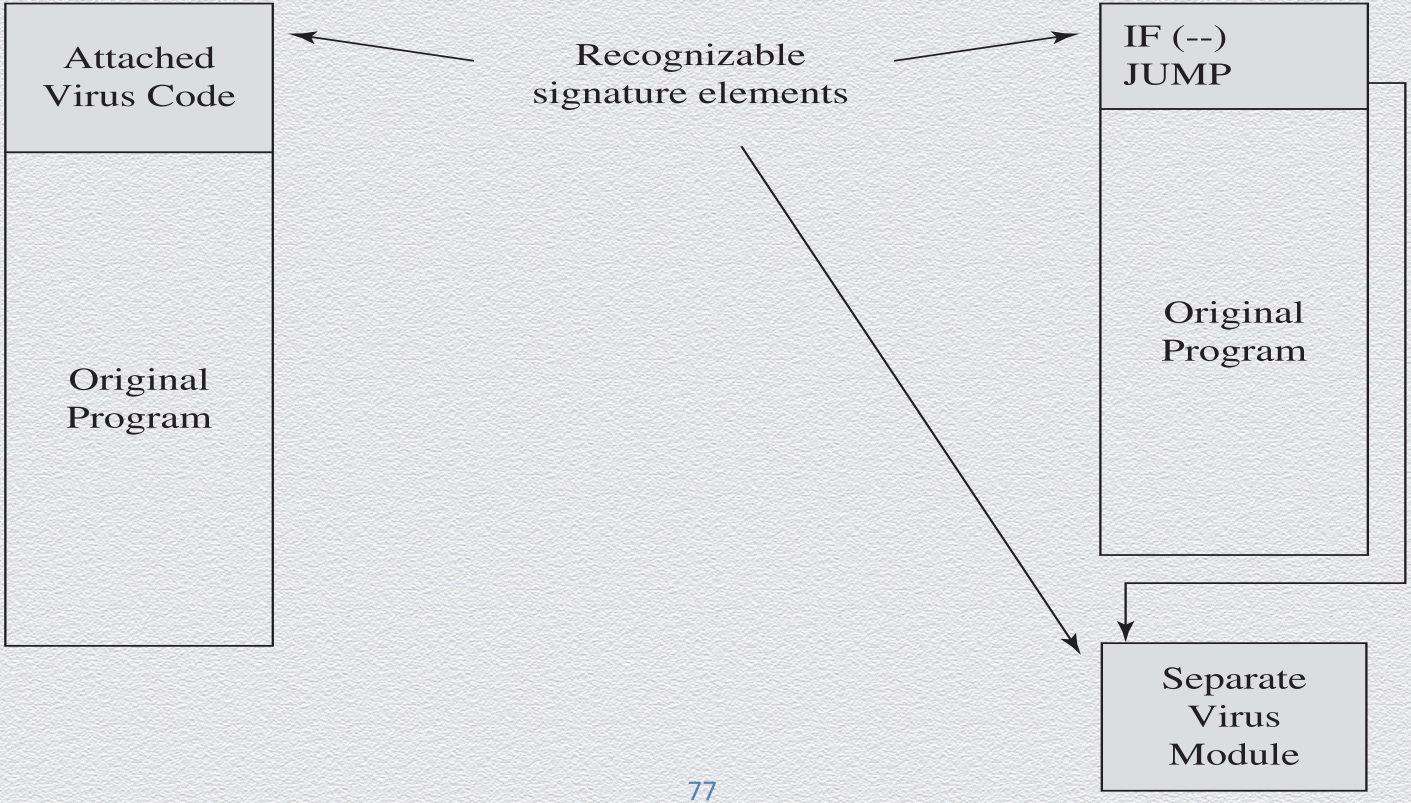
Countermeasures for users

- Use software acquired from reliable sources
- Test software in an isolated environment
- Only open attachments when you know them to be safe
- Treat every website as potentially harmful
- Create and maintain backups

Virus detection

- Virus scanners look for signs of malicious code infection using signatures in program files and memory
- Traditional virus scanners have trouble keeping up with new malware—detect about 45% of infections
- Detection mechanisms
  - Known string patterns in files or memory
  - Execution patterns
  - Storage patterns

Virus signatures



Countermeasures for developers

- Modular code: Each code module should be
  - Single-purpose
  - Small
  - Simple
  - Independent
- Encapsulation
- Information hiding
- Mutual suspicion
- Confinement
- Genetic diversity

Code testing

- Unit testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
- Regression testing
- Penetration testing

Design principles for security

- Least privilege
- Economy of mechanism
- Open design
- Complete mediation
- Permission based
- Separation of privilege
- Least common mechanism
- Ease of use

Other countermeasures

- Good
  - Proofs of program correctness—where possible
  - Defensive programming
  - Design by contract
- Bad
  - Penetrate-and-patch
  - Security by obscurity

Summary

- Buffer overflow attacks can take advantage of the fact that code and data are stored in the same memory in order to maliciously modify executing programs
- Programs can have a number of other types of vulnerabilities, including off-by-one errors, incomplete mediation, and race conditions
- Malware can have a variety of harmful effects depending on its characteristics, including resource usage, infection vector, and payload
- Developers can use a variety of techniques for writing and testing code for security