CS 140 – Summer 2008 – Handout #25: Improving OS Security

Announcements

- Project #3 grades sent out via e-mail
- Project #4: due tomorrow (Tuesday) 10 p.m.
  - remember: if anyone on your team has used 2 late days, you cannot use a late day on this assignment
- Final exam: Saturday, August 16th, 12:15–3:15 p.m.
  - right here: Terman Auditorium
  - local SCPD students: come to campus
  - remote students: exam posted on course web site @ 12:15 PDT
  - sample exams available on web site

Today: Improving OS Security

- Protection/security very hard to get right
- The protection design, mechanisms, and interface (API and user interface) provided by the OS can make it easier or harder to specify, understand, and enforce security policies.
- *Feature interaction* is powerful for both good and evil!
  - advantage: do things designers never intended
  - disadvantage: do things designers never intended!
- Some security problems (e.g. containment) difficult, if not impossible to solve (more on this later)
- But current operating systems (and other software)have largely failed at *basic* security

Case study: Protection/security in Unix

- Each process has user, group ID (uid, gid) (inherited!)
- Files, directories have permission bits

```
drwxr-xr-x 104 root wheel 8192 Aug4 04:02 /etc
```

- Devices, other objects exported through file system

  `/dev/tty1`, `/dev/disk0s1`, `/proc/xxxx`
- Non-FS object policies? restrict to `root` (uid 0) user
  - Bind any TCP/UDP port < 1024
  - Change current process's uid or gid
  - Mount/unmount file system
  - Create device nodes (/dev/xxxx) in file system
  - Change owner of file
  - Set clock; halt or reboot machine

Example: Unix `/bin/login`

- reads files from `/etc`
  - `passwd`, `group`, `shadow/master.passwd`
  - `username:hashpw:uid:gid:real name:dir:shell`
  - hashpw = one-way hash of (passwd,salt)
- runs as `root`
  - reads username and password from terminal
  - looks up in `/etc/passwd`
  - makes sure hash(passwd,salt) matches
  - if so, set user and group id, and `exec(shell)`
- Unix: hybrid of ACL (permissions) + very coarse Capability (`root` = can do anything to anything!)

Unix workaround: setuid

- Some legitimate actions require more privs than UID
  - how should users change their passwords?
  - stored in `/etc/passwd` and `/etc/shadow`
- Solution: setuid/setgid programs
  - run with privileges of file's owner or group
  - *real* and *effective* UID/GID
  - real: actual user; effective: for access checks
  - e.g. `/usr/bin/passwd` – changes user password, `/bin/su` – acquire new UID with correct password
- Have to be very careful writing setuid code
  - attackers can run setuid programs at any time
  - attacker controls many aspects of program's environment

Unix: potential setuid problems

- setuid shell scripts?
  - not if attacker controls $PATH!!
  - `/tmp/evil/bin:/bin:/usr/bin`
  - don't put "." in your path either
- setuid programs with shared libraries?
  - not if attacker controls $LD_LIBRARY_PATH
  - `/tmp/evil/lib:/lib:/usr/lib`
- setuid programs with debuggers?
  - not if attacker can attach to program, change code/data
  - `gdb /bin/su`? `ptrace()` ... see next slide
- result: all kinds of bad interactions that you need to find!
  - did you find them all? probably not...

setuid: more permissions issues

- When can process A send a signal to process B?
  - Allow if sender and receiver have same effective UID.
  - But, need ability to kill processes you launch even if setuid.
  - So, allow real UIDs to match as well.
  - Can also send SIGCONT w/o UID match if in same session.
- Debugger system call `ptrace()`
  - Lets one process modify another's memory.
  - Setuid gives a program more privilege than invoking user!
  - So, don't let process ptrace more privileged process.
  - Disable setuid if ptraced target calls exec()
  - Exception: root can ptrace anyone.
- Have we found all of the potential issues?

A sample non-setuid security hole

- Even without root or setuid, attackers can trick root-owned processes into doing things...
- Example: want to clear unused files in /tmp/
- Every night, automatically run this command as root:

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

- find identifies files not accessed in 3 days
  - executes `rm`, replacing `{}` with file name
- `rm -f --` *`path`* deletes file *`path`*
  - note: "`--`" here prevents *`path`* from being parsed as option
- What's wrong here?

An attack

find/rm	**attacker**
	`creat("/tmp/etc/passwd")`
`readddir("/tmp")->"etc"`	
`lstat("/tmp/etc")->ISDIR`	
`readddir("/tmp")->"passwd"`	
	`rename("/tmp/etc/"->"/tmp/x")`
	`symlink("/etc","/tmp/etc")`
`unlink("/tmp/etc/passwd")`	

- **Time-of-check-to-time-of-use (TOCTTOU) bug**
  - find checks that `/tmp/etc` is not symlink
  - but meaning of file name changes before it is used!

xterm command

- Provides terminal window in X Window System
- Used to run with setuid root privileges
  - required access to pseudo-terminal (`/dev/ptyxx`) device
  - required root privs to change ownership of pty to user
  - also wrote protected `utmp/wtmp` files to record users
- Had feature to log terminal session to a file

```
if (access(logfile, W_OK) <  0)
       return ERROR;
fd = open(logfile, O_CREAT|O_WRONLY|O_TRUNC,0666)
```

- `access` call avoids dangerous security hole
  - Does permission check with real, not effective UID.
  - Wrong: another TOCTTOU bug.

An attack

xterm	**attacker**
	`creat("/tmp/X")`
`access("/tmp/X") -> OK`	
	`unlink("/tmp/X")`
	`symlink("/tmp/X"->"/etc/passwd")`
`open("/tmp/X")`	

- Attacker changes `/tmp/X` between check and use
  - xterm unwittingly overwrites `/etc/passwd`
  - Time-of-check-to-time-of-use (TOCTTOU) bug
- OpenBSD `man` page: "CAVEATS: `access()` is a potential security hole and should never be used."

ssh configuration files

- `ssh` 1.2.12 – secure login program, runs as root
  - Needs to bind TCP port under 1024 (privileged operation)
  - Needs to read local host private key (for host authentication)
- Also needs to read and write files owned by user
  - Read configuration file `~/.ssh/config`
  - Record server keys in `~/.ssh/known_hosts`
- Author wanted to avoid TOCTTOU bugs:
  - First binds socket and reads root-owned secret key file, then drops all privileges before accessing user files.
  - Idea: avoid using any user-controlled arguments/files until you have no more privileges than the user.
- Was it secure? well...

Problem: ptrace() bug

- Dropping privilege allowed users to "debug" `ssh`
  - Depends on OS, but at the time several had `ptrace()` implementations which made `ssh` vulnerable.
- Once in debugger
  - Could use privileged port to connect to anywhere.
  - Could read secret key from memory.
  - Could overwrite local name to get privileges of other user.
- The fix: restructure into 3 processes!
  - Perhaps overkill, but really wanted to avoid problems.

more ptrace() woes: /bin/su in Linux (2.4.9)

- Some programs acquire then release privileges
  - e.g. `su user` is setuid, becomes user if password correct
- Consider the following:
  - A and B unprivileged processes owned by attacker
  - A ptraces B
  - A executes "`su user`" to its own identity
    - (remains attached via `ptrace()` to B past `exec()`!)
  - While `su` is superuser, B execs `su root`
    - (A remains attached, since A is superuser!)
  - A types password, gets shell, and is attached to su root
  - A can now manipulate `su root`'s memory (e.g. change bit of password check, or change PC) to get root shell.

Unix security design issues

- Feature interaction + root (unlimited) access = disaster!
- Many OS security policies subjective and not objective
  - When can you signal/debug process? Rebind network port?
  - Arbitrary/incoherent rules for non-file operations.
  - File operations also confusing (chown/chmod? hard/sym links?)
- Correct code is much harder to write than incorrect
  - Delete file without traversing symbolic link.
  - Read `ssh` configuration file (requires 3 processes?)
  - Write mailbox owned by user in dir owned by root/mail.
- Don't just blame the application writers
  - Must also blame the interfaces they program to.

Unix: complicated mechanisms lead to insecure policies

- Too many root or setuid programs

```
ps -u root
find / - perm 4000 -print | mail root
```

- Too many privileges for normal programs
  - Network programs run with *your* privileges
  - e.g. web browser: can read/write any of your files!
- Difficult to provide isolation within programs
  - web browser: no session isolation
  - word processor: no document isolation
  - mail reader: no message isolation
  - result: bad data -> application corrupted, can do anything you can do!
  - servers: worse, since more privs & unpredictable usage

Another security problem [Hardy]

- Setting: TYMSHARE multi-user system (not Unix!)
- Wanted fortran compiler to keep statistics
  - Modified compiler `/sysx/fort` to record stats in `/sysx/stat`
  - Gave compiler "home files license" – allows writing to anything in `/sysx` (kind of like Unix setuid)
- What's wrong here?

"The Confused Deputy"

- Attacker could overwrite any files in `/sysx`
  - System billing records in `/sysx/bill` got wiped
  - Probably command like `fort -o /sysx/bill file.f`
- Is this a compiler bug?
  - Original implementors did not anticipate extra rights.
  - Can't blame them for unchecked output file.
- Compiler is a "Confused Deputy"

??

  - Inherits privileges from invoking user (e.g. read `file.f`)
  - Also inherits home files license
  - Which master is it serving on any given system call?
  - OS doesn't know if it just sees `open("/sysx/bill")`
- Hardy: motivation for (objective) Capabilities.

Capabilities

- Slicing matrix along rows yields capabilities
  - e.g. for each process, store a list of objects it can access
  - Process explicitly invokes particular capabilities
- Can help avoid the confused deputy problem
  - e.g. give compiler output file and capability to write it
  - (think of passing file descriptor rather than file name)
  - -> so compiler uses no *ambient authority* to write file
- Every operation/access requires a capability check
  - read/write/execute a file?
  - open up a communication channel? map a page?
  - need capability for that action on that object
  - many capabilities, but possibly easier to track!

Capability approaches

- Hardware-enforced (tagged architectures)
  - each data item (e.g. word) augmented with label
  - Hardware enforces capability (e.g. r/w/x)
  - Similar to virtual memory/caches, with more state
- Kernel-enforced (KeyKOS, SElinux)
  - Kernel (or security kernel) maintains capability database.
  - Only permit operation if process has capability.
- Self-authenticating (Amoeba, Mach)
  - Every access must be accompanied by capability
  - e.g. encrypted secret value to give access to object
  - Good for distributed systems
  - Hard to control propagation

Limitations of Capabilities

- IPC performance is a losing battle with CPU makers
  - CPUs/MMUs generally not optimized for context switches
  - Capability systems usually involve many IPCs
- Capability programming model never took off
  - Requires changes throughout application software.
  - Mazières: Call capabilities “file descriptors” or “Java pointers” and people will use them!
  - Difficult for people to understand and use correctly
    - supported in Linux, but slow adoption...
  - “Capabilities are an OS concept of the future and always will be.”

Problem: Buggy, misbehaving applications and users!

- Even if OS is OK, what about user applications?
- Web browser: insecure plug-ins, JavaScript, cross-site scripting, phishing... and your bank login info.
- E-mail, IM clients, widgets, Facebook apps, “cool screensavers”
- Any server app (Web server, PHP, Java...)
- Even if apps are OK, what about users?
  - Stanford: payroll information accidentally e-mailed out 
    - response: block port 25 outgoing (break all e-mail programs!)
  - Bad permissions, incorrect policies, etc.
  - What about malicious users with privileges?

DAC vs. MAC

- Most people familiar with Discretionary Access Control
  - Unix permission bits are an example
  - Might set file `private` so only group `friends` can read it.
- Discretionary means anyone with access can propagate information
  - `mail spies@enemy.gov < secret.plans` 
- Mandatory Access Control
  - Security administrator can restrict propagation.
  - Abbreviated MAC
    - (not Media Access Controller, Message Authentication Code, Macintosh, etc.)
  - Attempt to solve (or mitigate) Containment problem, i.e. making sure secret information doesn’t propagate

Secrecy properties and rules for MAC

- Want to restrict flow of classified information
- Make sure that classified information only flows through classified channels, to classified users, etc.
- How? Give subjects (e.g. users/processes) and objects (e.g. files, pages) security levels/classifications
- All communication (at least in OS) has to obey MAC secrecy/classification rules:
  - Don’t allow unclassified users to read classified information: “no read-up”
  - Don’t allow classified users to write unclassified information: “no write-down”

Straw man MAC implementation

- Take an ordinary Unix system
- Put labels on all files and directories to track levels
- Each user U has a security clearance *level(U)*
- Create *current-level* and track dynamically
  - When U logs in, set *current-level* to lowest security level
  - **Read(file) // no read-up**
    - **if (level(file) > level(U)) kill(program);**
    - **else current-level = max(current-level, level(file));**
  - **Write(file) // no write-down**
    - **if (level(file) < current-level) kill(program);**
- Handle other objects (pipes, network, etc.) similarly
- Is this secure? (even just considering files/processes?)

No: Covert Channels

- System rife with *storage channels*
  - Low current-level process executes another program
  - New program reads sensitive file, gets high current level
  - High program exploits covert channels to pass data to low
- e.g. High program inherits file descriptor
  - Can pass 4 bytes of information to low program in file offset
- Other storage channels
  - Exit value, signals, file locks, terminal escape codes
- If we eliminate storage channels, is system secure?

No: Timing Channels

- Example: CPU utilization
  - To send a 0 bit, use 100% of CPU in busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
    - (receiver looks at wall clock time, measures own speed) 
- Example: Resource exhaustion
  - High program allocates all physical memory if bit is 1
  - If low program is slow from paging, knows less memory available
- More examples: disk head position, processor cache/TLB pollution, etc.

Reducing covert channels

- Observation: Covert channels come from sharing
  - No shared resources -> no covert channels
    - Mordac, preventer of information services
  - Extreme example: just use two computers!
- Problem: Sharing needed
  - e.g. read unclassified data when preparing classified
- Approach: Strict partitioning of resources
  - Strictly partition and schedule resources between levels
  - Occasionally reapportion resources based on usage
  - Do so infrequently to bound leaked information
  - In general, only hope to bound covert channel data rate
  - Approach still not so good if many security levels possible

Declassification

- Sometimes need to prepare unclassified report from classified data
- Declassification happens outside of system
  - Present file to security officer for downgrade
- Job of declassification often non-trivial
- Examples:
  - Microsoft word saves a lot of undo information!
  - Might be secret stuff you cut out of document
  - Blacked-out text still stored in PDF?
  - Did you miss anything? How to check?

Another security issue: Integrity

- Want to make sure data are not modified insecurely
  - e.g. text editor gets trojaned, subtly modifies files, might mess up `/etc/passwd` (or secret battle plan)
- Observation: Integrity is the converse of secrecy
  - In secrecy, want to avoid writing less secret files.
  - In integrity, want to avoid writing higher-integrity files.
- Implement integrity hierarchy parallel to secrecy
  - Now security level includes (secrecy, integrity)
  - Only trusted users can operate at high integrity level.
  - MAC system checks integrity level as well as secrecy.
  - If you read less authentic data, your current integrity level gets lowered, and you can no longer write high-integrity files.

Example: LOMAC (BSD, Linux, Vista?)

- "Low-watermark access control"
  - idea: get regular (non-military) systems/users to adopt MAC
- Emphasize integrity rather than secrecy
  - most important goal for many users/settings
  - e.g. don't allow viruses to corrupt your files
  - don't worry as much about covert channels
- Two integrity levels for subjects, objects
  - 2: High-integrity (system software, console, trusted terminals)
  - 1: Low integrity (network devices, untrusted terminals, etc.)
- Example: worm compromises your web server
  - Worm comes from network -> web server at low integrity
  - Won't be able to corrupt system files

Self-revocation problem

- Self-revocation
  - Unix apps don't know if they'll observe low-integrity data
  - An application can taint itself unexpectedly, revoking its own permission to access an object it created
  - e.g. something like `ps | grep user`
  - `ps` (or shell) creates pipe, forks, execs, reads tainted data...
  - `ps` becomes tainted, can no longer write to pipe/grep!
- Solution: don't consider pipes to be real objects
  - Pipe groups processes together in job
  - Any processes tied to job when they read or write pipe
  - Read of tainted data lowers integrity of both `ps` and `grep`
- Similar idea applies to other IPC (e.g. shared memory)

Related approach: Taint propagation, IFC

- Information Flow Control
  - e.g. track flow of "tainted" information through system
- Taint propagation: if you read it, you become tainted, as does anything you create
  - e.g. web server/browser: reads network – > tainted!
- Can be implemented at multiple levels
  - hardware (e.g. tags on memory, registers)
  - OS: labels on all OS objects
  - language (e.g. Java): labels on objects (inc. data, functions)
- Conservative: tainted = corrupted? assume possible!
- Multiple categories of taint
  - restrict propagation of multiple classes of information!

Untainting

- System is useless if everything is tainted
  - Need to make use of tainted data somehow
  - Like declassification – have to be very careful whom you allow to untaint something
  - Isolate into small, trusted un-tainting agent
- Example: web server
  - reads http requests from network -> tainted!
  - produces tainted report of usage, statistics
- Untainting tainted output
  - small (trusted, hopefully correct) program reads server report, producing (untainted) result which goes into (untainted) system log file

Tainting for privacy/secrecy

- All of your files are "tainted" as private to you
- You run your web browser
  - tries to read your configuration file -> tainted!
  - can't write network (not tainted as private to you)
  - oops.....
- Actually, this is good (?) behavior!
  - You should explicitly untaint your web preferences.
  - And assume that they're unsafe and will be exposed to the world!
- OS can help you understand where your information is going
  - but configuring it correctly can be a real pain.

Summary: Improving OS security

- Unix/ACL approach: hard to write secure software
  - Security checks in applications -> TOCTTOU bugs.
  - Ambient authority + feature interaction = confused deputy.
  - Difficulty of isolation in application software.
- Capabilities try to solve "confused deputy" problem
- MAC tries to address secrecy/privacy, integrity
- Configuration of capabilities, MAC is hard to get right
- Unix, Windows moving towards integrity tracking
  - e.g. signing of driver files, labels/capabilities for system files
- IFC, taint propagation
  - assumes software is unreliable; tracks info flow
  - addresses integrity, secrecy by labeling subjects, objects

Current OS (stopgap?) approaches

- Firewalls
  - try to restrict malicious use of your network
  - incoming, outgoing (e.g. Zonealarm: better privs)

- Antivirus software
  - try to **detect/stop** known threats and **suspicious activity**
- Sandboxing (e.g. Vista UAC, FreeBSD `jail()`)
  - only let web browser read/write files in a certain directory
  - restrict network connections and other activities
  - **finer-grained privileges**, **easier isolation** (hopefully)

- Mitigating buffer overflows (and associated attacks)
  - NX bit, ASLR, overflow detection schemes, Java/GC, dynamic string libraries (!) etc. – basically, fixing the C runtime!

Advice: plan for security failures

- Security is a form of reliability: things are going to go wrong, and you should have a plan for recovery
- Quickly discover and stop bad behavior
  - monitoring: detect and stop unusual activities
  - logging: track break-ins, figure out how to fix/prosecute
- Restore system to good (?) state
  - current: “malware removal tools”
    - no guarantee that things are actually OK
    - sort of like fsck -> different state, but correct?
      - malware is worse, because it *tries* to hide
  - better: checkpoints/full backups (inc. firmware!)
    - restore to known (hopefully non-corrupted) state
    - still no guarantee – when did first break-in occur?

What about encryption?

- Encryption makes it harder to read data unless you have a (hopefully secret and hard to guess) key for it
  - your little brother can’t read your diary, or your WiFi traffic
- Public-key encryption makes it easier to verify that a message has been *written* by someone with a secret key
  - SSL/TLS: verify that it’s amazon.com and not hackers.net!
  - but... what if certificates sent/granted via insecure network?
- Problems: key distribution/secrecy, certificate revocation
  - Verisign: bogus MS certificate; “Click Yes to Continue”
- Encryption doesn’t solve the OS security problems we’ve discussed today
  - Commonly used, but has limitations; see CS155, CS244B