6.858 Lecture 6
Capabilities and other Protection Mechanisms

What's the problem the authors of "confused deputy" encountered?

- Their system had a Fortran compiler, /sysx/fort (in Unix filename syntax)
- They wanted the Fortran compiler to record usage statistics, but where?
    - Created a special statistics file, /sysx/stat.
    - Gave /sysx/fort "home files license" (kind-of like setuid w.r.t. /sysx)
- What goes wrong?
    - User can invoke the compiler asking it to write output to /sysx/stat.
        - e.g. /sysx/fort /my/code.f -o /sysx/stat
    - Compiler opens supplied path name, and succeeds, because of its license.
    - User alone couldn't have written to that /sysx/stat file.
- Why isn't the /sysx/fort thing just a bug in the compiler?
    - Could, in principle, solve this by adding checks all over the place.
    - Problem: need to add checks virtually everywhere files are opened.
    - Perfectly correct code becomes buggy once it's part of a setuid binary.
- So what's the "confused deputy"?
    - The compiler is running on behalf of two principals:
        - the user principal (to open user's files)
        - the compiler principal (to open compiler's files)
    - Not clear what principal's privileges should be used at any given time.

Can we solve this confused deputy problem in Unix?

- Suppose gcc wants to keep statistics in /etc/gcc.stats
- Could have a special setuid program that only writes to that file
    - Not so convenient: can't just open the file like any other.
- What if we make gcc setuid to some non-root user (owner of stats file)?
    - Hard to access user's original files.
- What if gcc is setuid-root? (Bad idea, but let's figure out why..)
    - Lots of potential for buffer overflows leading to root access.
    - Need to instrument every place where gcc might open a file.
- What check should we perform when gcc is opening a file?
    - If it's an "internal" file (e.g. /etc/gcc.stats), maybe no check.
    - If it's a user-supplied file, need to make sure user can access it.
    - Can look at the permissions for the file in question.
    - Need to also check permissions on directories leading up to this file.
- Potential problem: race conditions.
    - What if the file changes between the time we check it and use it?
    - Common vulnerability: attacker replaces legit file with symlink
    - Symlink could point to, say, /etc/gcc.stats, or /etc/passwd, or ...
    - Known as "time-of-check to time-of-use" bugs (TOCTTOU).

Several possible ways of thinking of this problem:

1. Ambient authority: privileges that are automatically used by process are the problem here. No privileges should ever be used automatically. Name of an object should be also the privileges for accessing it.
2. Complex permission checks: hard for privileged app to replicate. With simpler checks, privileged apps might be able to correctly check if another user should have access to some object.

What are examples of ambient authority?

- Unix UIDs, GIDs.
- Firewalls (IP address vs. privileges for accessing it)
- HTTP cookies (e.g. going to a URL like http://gmail.com)

How does naming an object through a capability help?

- Pass file descriptor instead of passing a file name.
- No way to pass a valid FD unless caller was authorized to open that file.

Could we use file descriptors to solve our problem with a setuid gcc?

- Sort-of: could make the compiler only accept files via FD passing.
- Or, could create a setuid helper that opens the /etc/gcc.stats file, passes an open file descriptor back to our compiler process.
- Then, can continue using this open file much like any other file.
- How to ensure only gcc can run this helper?
    - Make gcc setgid to some special group.
    - Make the helper only executable to that special group.
    - Make sure that group has no other privileges given to it.

What problem are the Capsicum authors trying to solve with capabilities?

- Reducing privileges of untrustworthy code in various applications.
- Overall plan:
    - Break up an application into smaller components.
    - Reduce privileges of components that are most vulnerable to attack.
    - Carefully design interfaces so one component can't compromise another.
- Why is this difficult?
    - Hard to reduce privileges of code ("sandbox") in traditional Unix system.
    - Hard to give sandboxed code some limited access (to files, network, etc).

What sorts of applications might use sandboxing?

- OKWS.
- Programs that deal with network input:
    - Put input handling code into sandbox.
- Programs that manipulate data in complex ways:
    - (gzip, Chromium, media codecs, browser plugins, ...)
    - Put complex (& likely buggy) part into sandbox.
- How about arbitrary programs downloaded from the Internet?
    - Slightly different problem: need to isolate unmodified application code.

  - One option: programmer writes their application to run inside sandbox.
    - Works in some cases: Javascript, Java, Native Client, ...
    - Need to standardize on an environment for sandboxed code.
  - Another option: impose new security policy on existing code.
    - Probably need to preserve all APIs that programmer was using.
    - Need to impose checks on existing APIs, in that case.
    - Unclear what the policy should be for accessing files, network, etc.
- Applications that want to avoid being tricked into misusing privileges?
  - Suppose two Unix users, Alice and Bob, are working on some project.
  - Both are in some group G, and project dir allows access by that group.
  - Let's say Alice emails someone a file from the project directory.
  - Risk: Bob could replace the file with a symlink to Alice's private file.
  - Alice's process will implicitly use Alice's ambient privileges to open.
  - Can think of this as sandboxing an individual file operation.

What sandboxing plans (mechanisms) are out there (advantages, limitations)?

- OS typically provides some kind of security mechanism ("primitive").
  - E.g., user/group IDs in Unix, as we saw in the previous lecture.
  - For today, we will look at OS-level security primitives/mechanisms.
  - Often a good match when you care about protecting resources the OS manages.
  - E.g., files, processes, coarse-grained memory, network interfaces, etc.
- Many OS-level sandboxing mechanisms work at the level of processes.
  - Works well for an entire process that can be isolated as a unit.
  - Can require re-architecting application to create processes for isolation.
- Other techniques can provide finer-grained isolation (e.g., threads in proc).
  - Language-level isolation (e.g., Javascript).
  - Binary instrumentation (e.g., Native Client).
  - Why would we need these other sandboxing techniques?
    - Easier to control access to non-OS / finer-grained objects.
    - Or perhaps can sandbox in an OS-independent way.
  - OS-level isolation often used in conjunction with finer-grained isolation.
    - Finer-grained isolation is often hard to get right (Javascript, NaCl).
    - E.g., Native Client uses both a fine-grained sandbox + OS-level sandbox.
  - Will look at these in more detail in later lectures.

Plan 0: Virtualize everything (e.g., VMs).

- Run untrustworthy code inside of a virtualized environment.
- Many examples: x86 qemu, FreeBSD jails, Linux LXC, ..
- Almost a different category of mechanism: strict isolation.
- Advantage: sandboxed code inside VM has almost no interactions with outside.
- Advantage: can sandbox unmodified code that's not expecting to be isolated.
- Advantage: some VMs can be started by arbitrary users (e.g., qemu).
- Advantage: usually composable with other isolation techniques, extra layer.

- Disadvantage: hard to allow some sharing: no shared processes, pipes, files.
- Disadvantage: virtualizing everything often makes VMs relatively heavyweight.
    - Non-trivial CPU/memory overheads for each sandbox.

Plan 1: Discretionary Access Control (DAC).

- Each object has a set of permissions (an access control list).
    - E.g., Unix files, Windows objects.
    - "Discretionary" means applications set permissions on objects (e.g., chmod).
- Each program runs with privileges of some principals.
    - E.g., Unix user/group IDs, Windows SIDs.
- When program accesses an object, check the program's privileges to decide.
    - "Ambient privilege": privileges used implicitly for each access.

```
      Name              Process privileges
        |                       |
        V                       V
     Object -> Permissions -> Allow?
```

How would you sandbox a program on a DAC system (e.g., Unix)?

- Must allocate a new principal (user ID):
    - Otherwise, existing principal's privileges will be used implicitly!
- Prevent process from reading/writing other files:
    - Change permissions on every file system-wide?
        - Cumbersome, impractical, requires root.
    - Even then, new program can create important world-writable file.
    - Alternative: chroot (again, have to be root).
- Allow process to read/write a certain file:
    - Set permissions on that file appropriately, if possible.
    - Link/move file into the chroot directory for the sandbox?
- Prevent process from accessing the network:
    - No real answer for this in Unix.
    - Maybe configure firewall? But not really process-specific.
- Allow process to access particular network connection:
    - See above, no great plan for this in Unix.
- Control what processes a sandbox can kill / debug / etc:
    - Can run under the same UID, but that may be too many privileges.
    - That UID might also have other privileges...

Problem: only root can create new principals, on most DAC systems.

- E.g., Unix, Windows.

Problem: some objects might not have a clear configurable access control list.

- Unix: processes, network…

Problem: permissions on files might not map to policy you want for sandbox.

- Can sort-of work around using chroot for files, but awkward.

Related problem: performing some operations with a subset of privileges.

- Recall example with Alice emailing a file out of shared group directory.
- "Confused deputy problem": program is a "deputy" for multiple principals.
- One solution: check if group permissions allow access (manual, error-prone).
    - Alternative solution: explicitly specify privileges for each operation.
        - Capabilities can help: capability (e.g., fd) combines object + privileges.
        - Some Unix features incompat. w/ pure capability design (symlinks by name).

Plan 2: Mandatory Access Control (MAC).

- In DAC, security policy is set by applications themselves (chmod, etc).
- MAC tries to help users / administrators specify policies for applications.
    - "Mandatory" in the sense that applications can't change this policy.
    - Traditional MAC systems try to enforce military classified levels.
    - E.g., ensure top-secret programs can't reveal classified information.

```
 Name    Operation + caller process
   |               |
   V               V
Object --------> Allow?
                   ^
                   |
Policy ------------+
```

- Note: many systems have aspects of both DAC + MAC in them.
    - E.g., Unix user IDs are "DAC", but one can argue firewalls are "MAC".
    - Doesn't really matter -- good to know the extreme points in design space

Windows Mandatory Integrity Control (MIC) / LOMAC in FreeBSD.

- Keeps track of an "integrity level" for each process.
- Files have a minimum integrity level associated with them.
- Process cannot write to files above its integrity level.
- IE in Windows Vista runs as low integrity, cannot overwrite system files.
- FreeBSD LOMAC also tracks data read by processes.
    - (Similar to many information-flow-based systems.)
    - When process reads low-integrity data, it becomes low integrity too.
    - Transitive, prevents adversary from indirectly tampering with files.
- Not immediately useful for sandboxing: only a fixed number of levels.

SElinux.

- Idea: system administrator specifies a system-wide security policy.
- Policy file specifies whether each operation should be allowed or denied.
- To help decide whether to allow/deny, files labeled with "types".

    - (Yet another integer value, stored in inode along w/ uid, gid, ..)

Mac OS X sandbox ("Seatbelt") and Linux seccomp_filter.

- Application specifies policy for whether to allow/deny each syscall.
    - (Written in LISP for MacOSX's mechanism, or in BPF for Linux's.)
- Can be difficult to determine security impact of syscall based on args.
    - What does a pathname refer to? Symlinks, hard links, race conditions… (Although MacOSX's sandbox provides a bit more information.)
- Advantage: any user can sandbox an arbitrary piece of code, finally!
- Limitation: programmer must separately write the policy + application code.
- Limitation: some operations can only be filtered at coarse granularity.
    - E.g., POSIX shm in MacOSX's filter language, according to Capsicum paper.
- Limitation: policy language might be awkware to use, stateless, etc.
    - E.g., what if app should have exactly one connection to some server?

- Note: seccomp_filter is quite different from regular/old seccomp, and the Capsicum paper talks about the regular/old seccomp.]

Is it a good idea to separate policy from application code?

- Depends on overall goal.
- Potentially good if user/admin wants to look at or change policy.
- Problematic if app developer needs to maintain both code and policy.
- For app developers, might help clarify policy.
- Less-centralized "MAC" systems (Seatbelt, seccomp) provide a compromise.

Plan 3: Capabilities (Capsicum).
Different plan for access control: capabilities.

- If process has a handle for some object ("capability"), can access it.

```
    Capability --> Object
```

- No separate question of privileges, access control lists, policies, etc.
- E.g.: file descriptors on Unix are a capability for a file.
    - Program can't make up a file descriptor it didn't legitimately get. (Why not?)
    - Once file is open, can access it; checks happened at open time.
    - Can pass open files to other processes.
    - FDs also help solve "time-of-check to time-of-use" (TOCTTOU) bugs.
- Capabilities are usually ephemeral: not part of on-disk inode.
    - Whatever starts the program needs to re-create capabilities each time.

Global namespaces.

- Why are these guys so fascinated with eliminating global namespaces?
- Global namespaces require some access control story (e.g., ambient privs).

- Hard to control sandbox's access to objects in global namespaces.

Kernel changes.

- Just to double-check: why do we need kernel changes?
    - Can we implement everything in a library (and LD_PRELOAD it)?
- Represent more things as file descriptors: processes (pdfork).
    - Good idea in general.
- Capability mode: once process enters cap mode, cannot leave (+all children).
- In capability mode, can only use file descriptors -- no global namespaces.
    - Cannot open files by full path name: no need for chroot as in OKWS.
    - Can still open files by relative path name, given fd for dir (openat).
- Cannot use ".." in path names or in symlinks: why not?
    - In principle, ".." might be fine, as long as ".." doesn't go too far.
    - Hard to enforce correctly.
    - Hypothetical design:
        - Prohibit looking up ".." at the root capability.
        - No more ".." than non-".." components in path name, ignoring ".".
- Assume a process has capability C1 for /foo.
- Race condition, in a single process with 2 threads:

```
T1: mkdir(C1, "a/b/c")
T1: C2 = openat(C1, "a")
T1: C3 = openat(C2, "b/c/../..")   ## should return a cap
for /foo/a
Let openat() run until it's about to look up the first ".."
T2: renameat(C1, "a/b/c", C1, "d")
T1: Look up the first "..", which goes to "/foo"
Look up the second "..", which goes to "/"
```

- Do Unix permissions still apply?
    - Yes --can't access all files in dir just because you have a cap for dir.
    - But intent is that sandbox shouldn't rely on Unix permissions.
- For file descriptors, add a wrapper object that stores allowed operations.
- Where does the kernel check capabilities?
    - One function in kernel looks up fd numbers -- modified it to check caps.
    - Also modified namei function, which looks up path names.
    - Good practice: look for narrow interfaces, otherwise easy to miss checks.

libcapsicum.

- Why do application developers need this library?
- Biggest functionality: starting a new process in a sandbox.

fd lists.

- Mostly a convenient way to pass lots of file descriptors to child process.
- Name file descriptors by string instead of hard-coding an fd number.

cap_enter() vs lch_start().

- What are the advantages of sandboxing using exec instead of cap_enter?
- Leftover data in memory: e.g., private keys in OpenSSL/OpenSSH.

- Leftover file descriptors that application forgot to close.
- Figure 7 in paper: tcpdump had privileges on stdin, stdout, stderr.
- Figure 10 in paper: dhclient had a raw socket, syslogd pipe, lease file.

Advantages: any process can create a new sandbox.
- (Even a sandbox can create a sandbox.)

Advantages: fine-grained control of access to resources (if they map to FDs).
- Files, network sockets, processes.

Disadvantage: weak story for keeping track of access to persistent files.
Disadvantage: prohibits global namespaces, requires writing code differently.

Alternative capability designs: pure capability-based OS (KeyKOS, etc).
- Kernel only provides a message-passing service.
- Message-passing channels (very much like file descriptors) are capabilities.
- Every application has to be written in a capability style.
- Capsicum claims to be more pragmatic: some applications need not be changed.

Linux capabilities: solving a different problem.
- Trying to partition root's privileges into finer-grained privileges.
- Represented by various capabilities: CAP_KILL, CAP_SETUID, CAP_SYS_CHROOT…
- Process can run with a specific capability instead of all of root's privs.
- Ref: capabilities(7), http://linux.die.net/man/7/capabilities

Using Capsicum in applications.
- Plan: ensure sandboxed process doesn't use path names or other global NSes.
  - For every directory it might need access to, open FD ahead of time.
  - To open files, use openat() starting from one of these directory FDs.
  - .. programs that open lots of files all over the place may be cumbersome.
- tcpdump.
  - 2-line version: just cap_enter() after opening all FDs.
  - Used procstat to look at resulting capabilities.
  - 8-line version: also restrict stdin/stdout/stderr.
  - Why? E.g., avoid reading stderr log, changing terminal settings…
- dhclient.
  - Already privilege-separated, using Capsicum to reinforce sandbox (2 lines).
- gzip.
  - Fork/exec sandboxed child process, feed it data using RPC over pipes.
  - Non-trivial changes, mostly to marshal/unmarshal data for RPC: 409 LoC.
  - Interesting bug: forgot to propagate compression level at first.
- Chromium.
  - Already privilege-separated on other platforms (but not on FreeBSD).
  - ~100 LoC to wrap file descriptors for sandboxed processes.
- OKWS.

    - What are the various answers to the homework question?

Does Capsicum achieve its goals?

- How hard/easy is it to use?
    - Using Capsicum in an application almost always requires app changes.
        - (Many applications tend to open files by pathname, etc.)
        - One exception: Unix pipeline apps (filters) that just operate on FDs.
    - Easier for streaming applications that process data via FDs.
    - Other sandboxing requires similar changes (e.g., dhclient, Chromium).
    - For existing applications, lazy initialization seems to be a problem.
        - No general-purpose solution -- either change code or initialize early.
    - Suggested plan: sandbox and see what breaks.
        - Might be subtle: gzip compression level bug.
- What are the security guarantees it provides?
    - Guarantees provided to app developers: sandbox can operate only on open FDs.
    - Implications depend on how app developer partitions application, FDs.
    - User/admin doesn't get any direct guarantees from Capsicum.
    - Guarantees assume no bugs in FreeBSD kernel (lots of code), and that the Capsicum developers caught all ways to access a resource not via FDs.
- What are the performance overheads? (CPU, memory)
    - Minor overheads for accessing a file descriptor.
    - Setting up a sandbox using fork/exec takes O(1msec), non-trivial.
    - Privilege separation can require RPC / message-passing, perhaps noticeable.
- Adoption?
    - In FreeBSD's kernel now, enabled by default (as of FreeBSD 10).
    - A handful of applications have been modified to use Capsicum: dhclient, tcpdump, and a few more since the paper was written (Ref: http://www.cl.cam.ac.uk/research/security/capsicum/freebsd.html)
    - Casper daemon to help applications perform non-capability operations.
        - E.g., DNS lookups, look up entries in /etc/passwd, etc.
        - http://people.freebsd.org/~pjd/pubs/Capsicum_and_Casper.pdf
    - There's a port of Capsicum to Linux (but not in upstream kernel repo).

What applications wouldn't be a good fit for Capsicum?

- Apps that need to control access to non-kernel-managed objects.
    - E.g.: X server state, DBus, HTTP origins in a web browser, etc.
    - E.g.: a database server that needs to ensure DB file is in correct format.
    - Capsicum treats pipe to a user-level server (e.g., X server) as one cap.
- Apps that need to connect to specific TCP/UDP addresses/ports from sandbox.
    - Capsicum works by only allowing operations on existing open FDs.
    - Need some other mechanism to control what FDs can be opened.

  - Possible solution: helper program can run outside of capability mode, open TCP/UDP sockets for sandboxed programs based on policy.

References:

- http://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf
- http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/prctl/seccomp_filter.txt;hb=HEAD
- http://en.wikipedia.org/wiki/Mandatory_Integrity_Control