Advanced Topics in Operating Systems, CS262
Prof. Eric A. Brewer

Lecture Notes, 10/21/98

Secure Environment
Java Security

I. Secure Environment

Approach: intercept all kernel crossings to create a secure environment

The problem:

- how to run untrusted code
- example: helper applications for netscape
- goal: safety not correctness (the program can be broken, but it can't hurt anything as a result)

Why mobile code?

- full-control over local UI
- reduced traffic due to local decisions
- between network protocols (complete end-to-end control)

Orthogonal security:

- Key idea: guard only the entrances and exits
- security must be orthogonal to functionality
  - security can't have bugs (bugs are security holes)
  - application code *will* have bugs
  - therefore, we must keep them separate (orthogonal)
- traditional view: security must be integrated
  - this conflicts with orthogonality
  - better version: design should be integrated, code should not

Approach:

- use Solaris debug facility to intercept all kernel calls (that matter)
- check arguments, remove some calls
- recursive contain child processes
- use config file to control variable access such as file system and network

Base functionality:

- process tracing facility allows interception of all kernel calls, turn on/off per call
- not perfect, but good enough: e.g. can't always abort a system call, which means we must abort the whole application
- can modify args and return values

Alternatives:

- make each app secure: unrealistic and untrusted authors
- firewall: can keep all or nothing out
- java (discussed below): use JVM to provide security -- keep language secure rather than just kernel calls.

Policy modules:

- control access explicitly
- access denied by default
- try to reuse policy modules to avoid configuration for each helper application

Example for helper apps:

- forked children recursively traced
- signals only self and children
- limit environment variables
- app placed in a subtree -- full access within, but can't get out
- checks only on open, since read & write can only use open'd file descriptors
- network access only to an X proxy (nested X system in a window; this prevents communication via X to other apps -- it's like you are the only X app running)
- see samples from the paper

Apps:

- ghostview, mpeg_play, xdvi, xv, xanim
- recent additions: java and sendmail (!)
- no real performance impact

II. Extensible Security Architectures for Java

IPC is too expensive to use for protecting separate subsystems... We need a way to make cross domain calls fast

- Software Fault Isolation: add code to check to do bounds checking on every read and write. "play in your own sandbox only". Wild writes can still corrupt the program, but only its own code/data.
- Type-based checking (java): ensure that "bad" pointers can't be created. (Implies: must GC (!), must do array bounds checking, must prevent casts (except to superclass,

or to subclass with dynamic check), no pointer arithmetic.) Assuming all of this, then multiple subsystems can exist in one address space, with no interference other than that enabled by the explicit creation of cross-domain pointers.

New challenges: now that we have more flexibility, how to we implement a reasonable security policy that is safe, extensible and flexible?

Three approaches:

- capabilities: every pointer is a capability (hard to revoke)
- stack introspection: enable/disable privileges a frame and its children, but disable when calling an untrusted class
- name space management: change the class loader to use special versions of system libraries that limit and check access. Prevent any other way of using the real libraries by taking them out of the namespace. Kind of like capabilities, but completely not revocable.

Notes on language-based safety: can be done statically or dynamically

- static => fast, but limited expressive power, since you can't do casting (even if safe).
- dynamic => requires run-time checks, but need not prevent all things that could be unsafe (since you can check them explicitly at the last minute)
- However, static prevents run-time errors and exceptions and thus has advantages for unsophisticated users and for program reliability.

Stack introspection:

- privileges can't be global -- they only apply to one thread (so make them part of the thread state)
- calling into untrusted code needs to disable privileges automatically (can't trust the callee to do it!).
- need to automatically disable privileges when leaving the frame that enabled them -- this prevents a common but serious error: forgetting to disable privileges on ALL paths out of the procedure
- Q: why is it bad to allow trusted code to access a resource if there is an untrusted ancestor on its call stack? A: because the untrusted code manipulate the trusted code in bad ways. For example, the "file open" code is trusted, but an untrusted procedure could ask it to open the password file.

Up-front target requests: it is a good idea to test all the privilege you might need at the start of the program!

- Else, you get bad run-time errors when you try to complete an operation without permission (like saving your work!)