PL Mechanisms for Security Engineering

Dan Wallach

Rice University

Previous talk

- Broad architectural ideas
  - Separation of privilege
  - Least privilege
  - Narrow interfaces
  - Software engineering processes

Today: using Java (or C#) mechanisms

Type safety is your friend

- **C / C++ problems:**
  - Buffer overflows
  - Cross-site scripting
  - Denial of service
  - File corruption
  - Format string vulnerabilities
  - Improper bounds checking
  - Insecure access control
  - Integer overflows
  - Memory corruption
  - Out-of-bounds array access
  - Privilege escalations
  - SQL injection
- **Java / C# problems:**
  -
  - Cross-site scripting
  - Denial of service
  -
  -
  -
  - Insecure access control
  -
  -
  -
  - Privilege escalations
  - SQL injection

Cool Java features

- No need to free / delete memory
  - Garbage collector does all the hard work
  - Memory leaks still possible
- Access modifiers (public, private, …) are enforced
- Arrays are bounds checked
- Easy to have extensibility (e.g., applets) with security controls

Similar features in other safe languages (C#)

Capability-based design

- Having a Java object reference ➔
  Permission to invoke methods on it

- Least privilege via capabilities
  - Control access to dangerous primitives
  - Pass different capabilities to different modules

- Comparable to using “factory patterns”

Classic Java style …

```
InputStream is = new FileInputStream("foo");
```

Capability style …

```
FileSystemCapability fsc;
InputStream is = fsc.openFile("foo");
```

- How do you get capability instance?
- What about the static constructor?

Getting the capability instance

- Need to change the initial interface

```
interface RunnableCapability {
    public void run(FileSystemCapability fsc,
                    NetworkCapability nc,
                    …);
}
```

- Pass capabilities or store them in static vars

Capability-style: why bother?

- Embrace and extend! Wrappers can *delegate* to internal capabilities.

```
public class SingleUseFSCapability extends FileSystemCapability {
    private FileSystemCapability fsc;
    private boolean valid = true;

    public SingleUseFSCapability(FileSystemCapability fsc) {
      this.fsc = fsc;
    }

    public InputStream openFile(String name) {
      if(!valid) throw new SecurityException(…);
      valid = false;
      return fsc.openFile(name);
    }

    public invalidate() { fsc = null; }
```

Capability uses

- File system
  - Restricted subdirectories / visibility
  - Restricted file sizes
- Networks
  - Restricted connection destination
  - Restricted bandwidth
  - Transparent SSL
- User operations
- Database operations

Banning the static constructors

Simple *grep* rules on source code

- No imports of java.io.*
- Hand-audit of any reflection calls

In a “pure” capability system

- No “public static” variables
- Only operations available from capabilities
- Capabilities represent your privileges
- No other security checks necessary
- Classic issue: capability leakage

Embraces the principle of least privilege

Distributed capabilities

- Assign each capability a 128-bit random number (it's "name")
  - Warning: use cryptographically strong RNG
- Mapping from names to capabilities
- Web cookies, session IDs, etc.
  - Login module creates per-user capabilities
  - Web page generator easily restricted
  - Expiration, other features, easy to do

Duff's Law (redux)

E Programming Language

- Somewhat like Python in syntax
- Runs on the JVM

www.erights.org

- Excellent discussion of other issues with capability-style software engineering

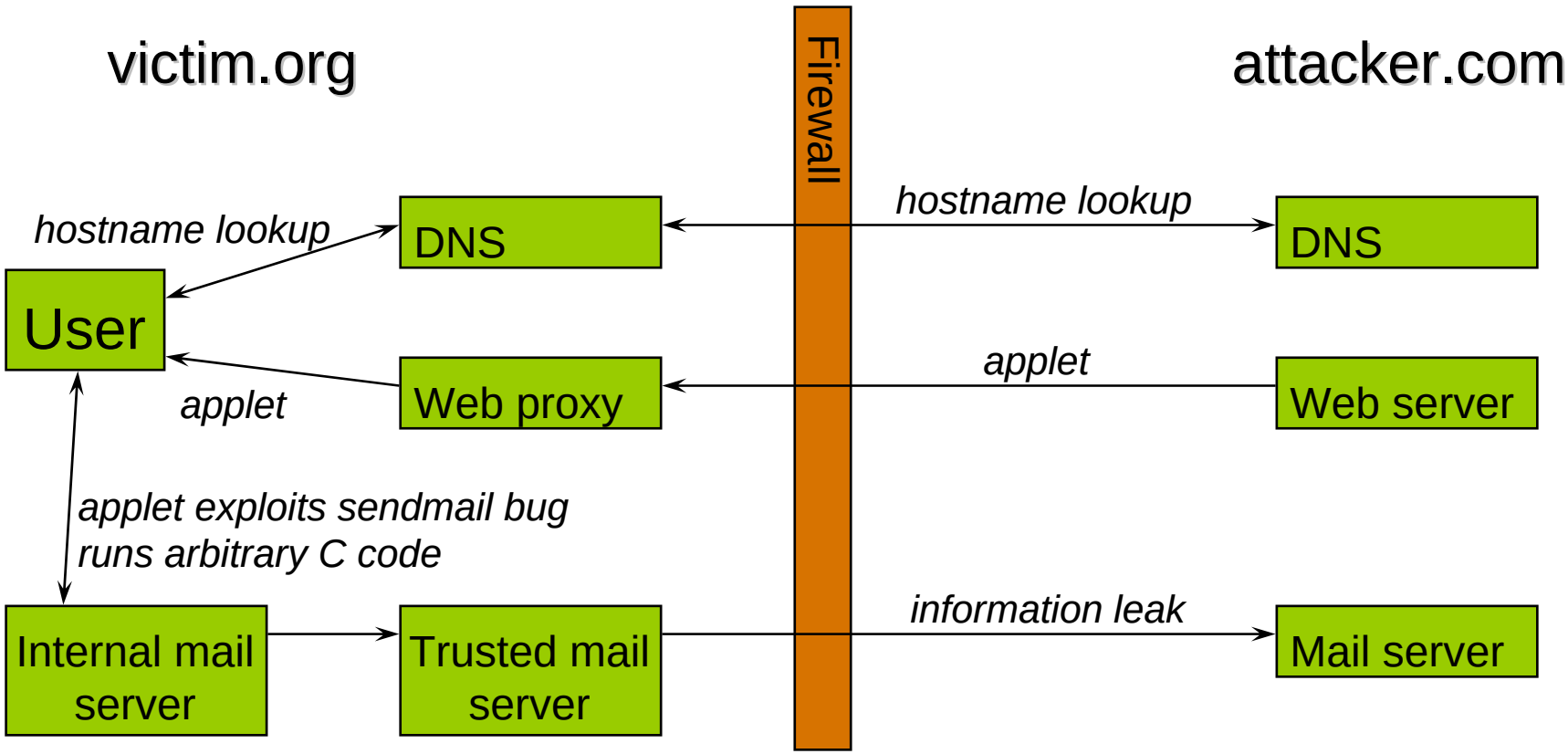
Java capability limits

- No way to control CPU or memory use
  - You could restrict use of *new* operations
- No way to kill an errant computation
  - You could invalidate its capabilities
- Legacy code
  - Need other mechanisms

Netscape 2.0 insecurity

- Java trusts DNS
  - Internet hosts can have multiple IP addresses
  - Java host equality test is too lenient
- With a hacked DNS server
  - Two-way channel to any machine on the Internet
  - Applets can connect to machines behind a firewall
    - Exploit numerous Unix and Windows bugs
    - Talk to internal Web and mail servers

Netscape DNS attack



The DNS attack allows connections to *any* machine behind the firewall.

Joint work with Dean and Felten (1996)

Solutions possible?

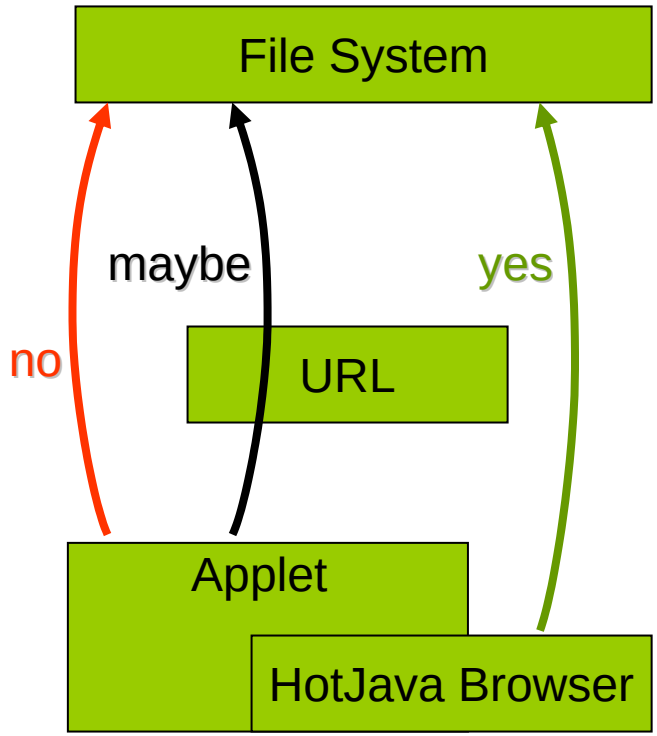
Capability-style: Applets shouldn't have access to the "real" java.net classes.

(But the Applet APIs were already frozen.)

Actual solution: Try to figure out "who" is calling and behave differently.

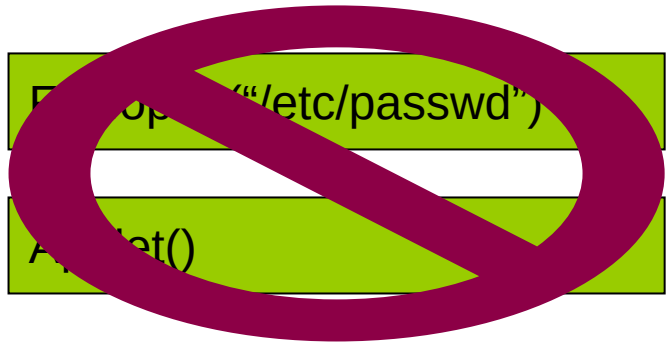
Related problem: file access

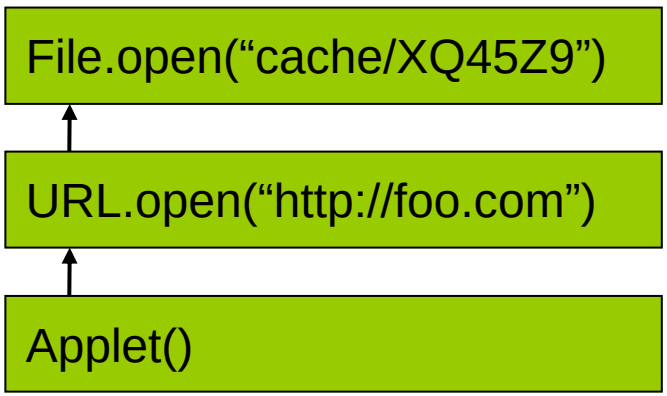
- *Some* parts of Java need the file system!
  - URL file cache
  - Class dynamic loader
- Secure services
  - Use dangerous primitives
  - Export safe interfaces
  - How to decide if an operation should be allowed?

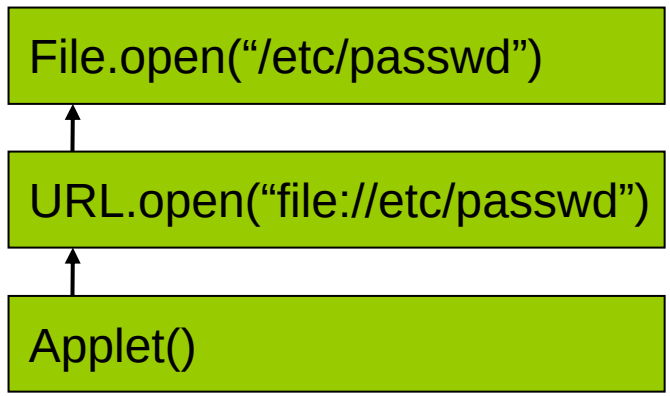


Handling the “maybe” cases

- Dangerous actions should be forbidden unless explicitly allowed
  - principle of least privilege
  - fail-safe

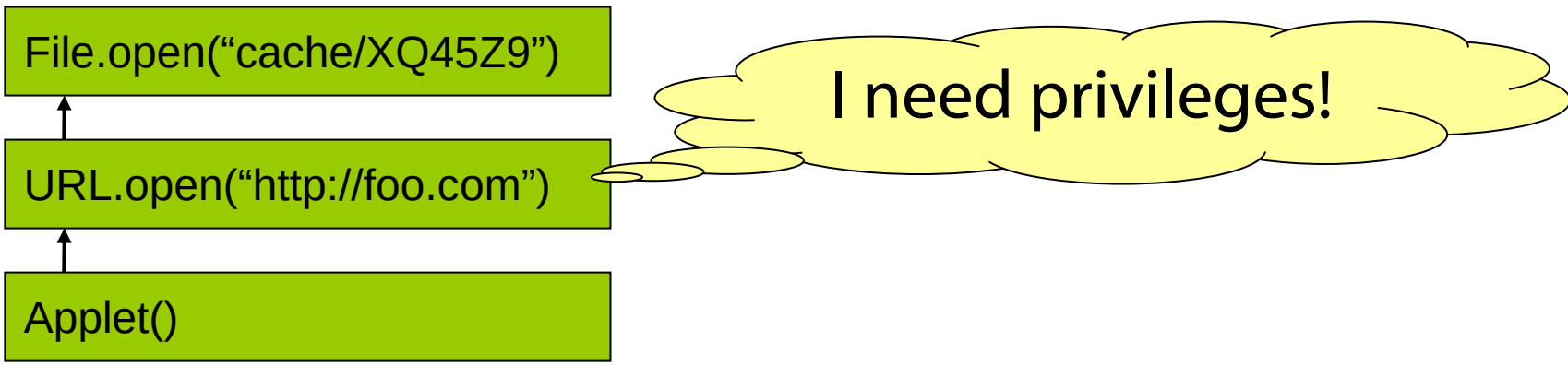






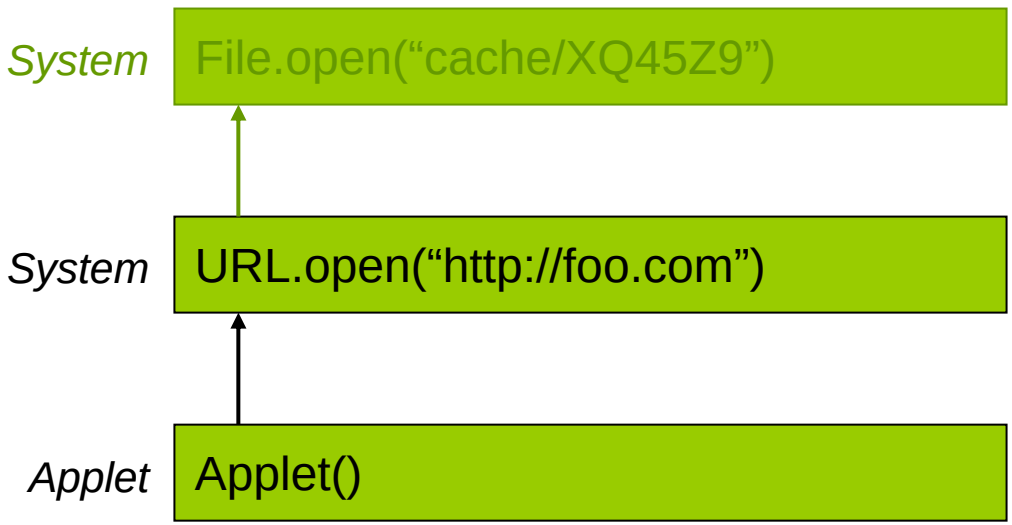
Solution: Stack inspection

- Code *must* explicitly authorize a dangerous action
  - A method *enables its privileges*
    - Privileges revert when the method returns
- Standard Java / JavaScript / C# feature
  - Invented at Netscape



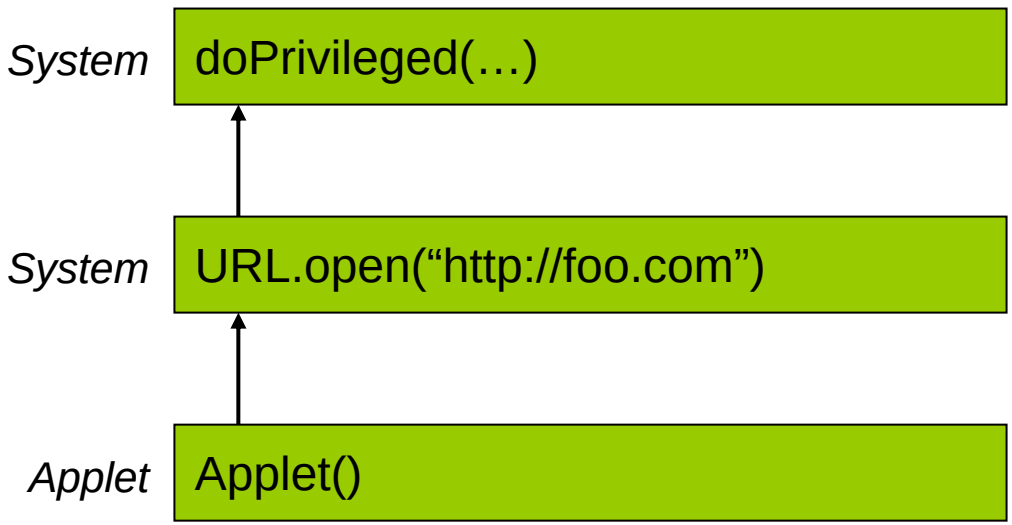
How stack inspection works

What if the URL code wants to use a file cache?



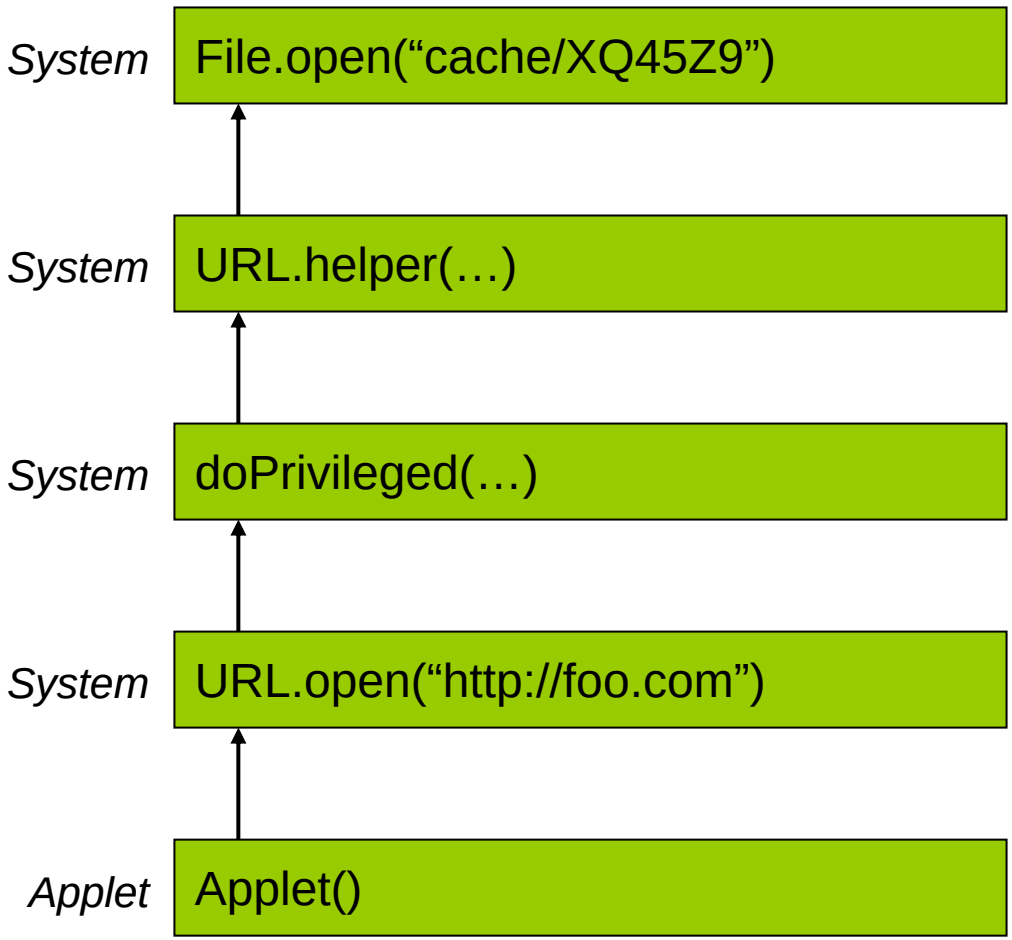
How stack inspection works

First, enable privileges...



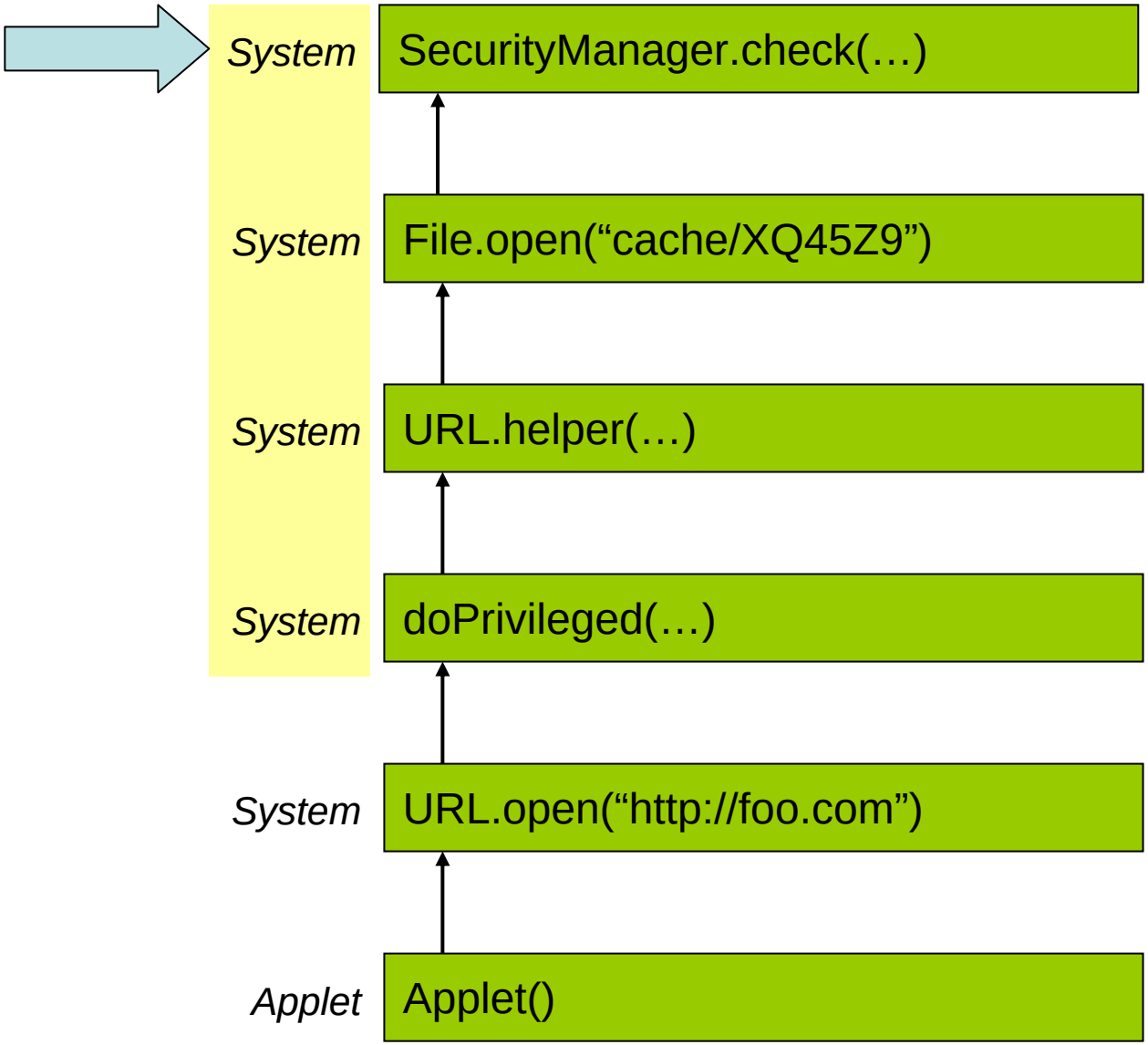
How stack inspection works

… which calls into whatever you want (typically an anonymous inner class)



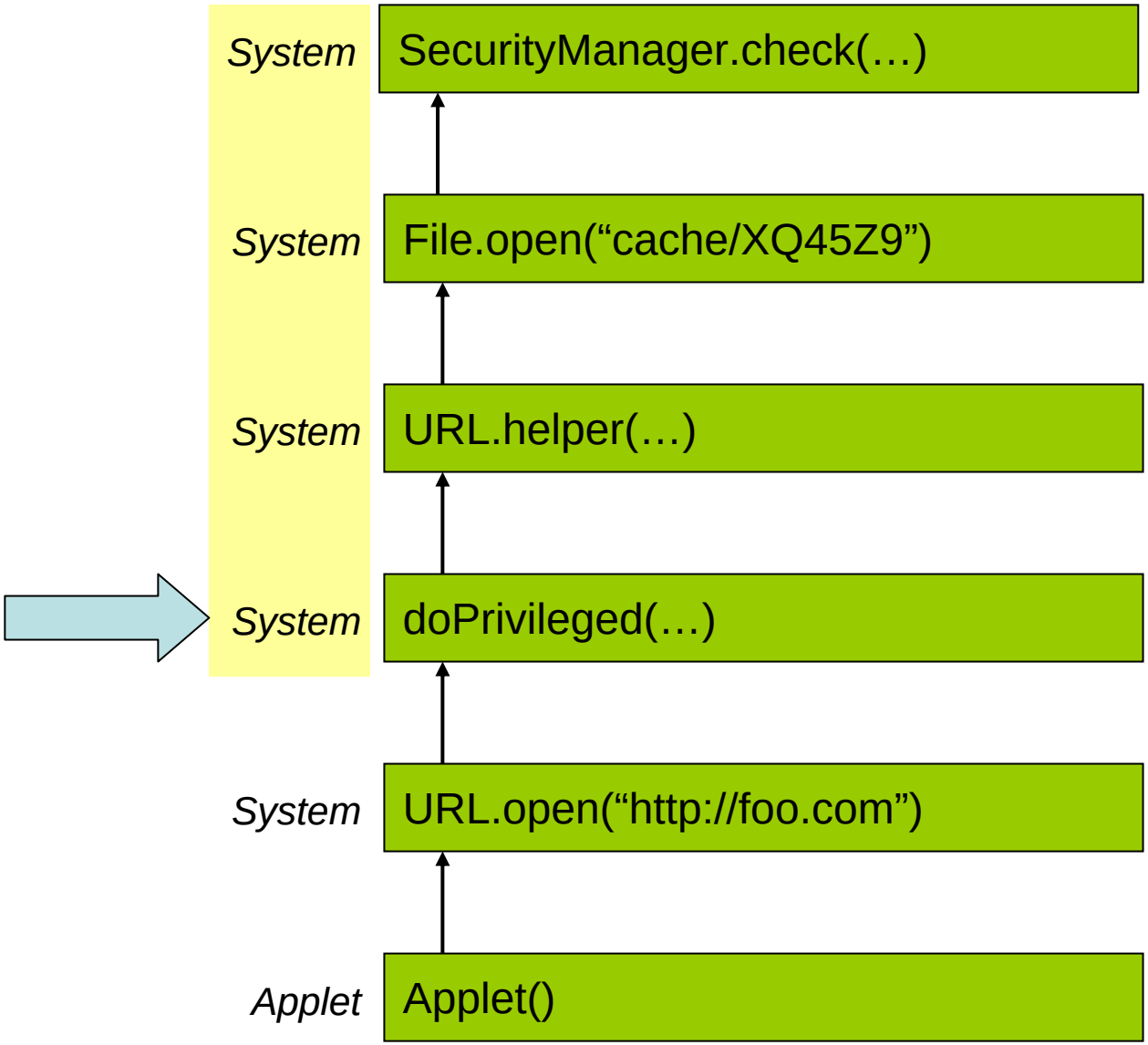
How stack inspection works

… then searches for the doPrivileged() frame



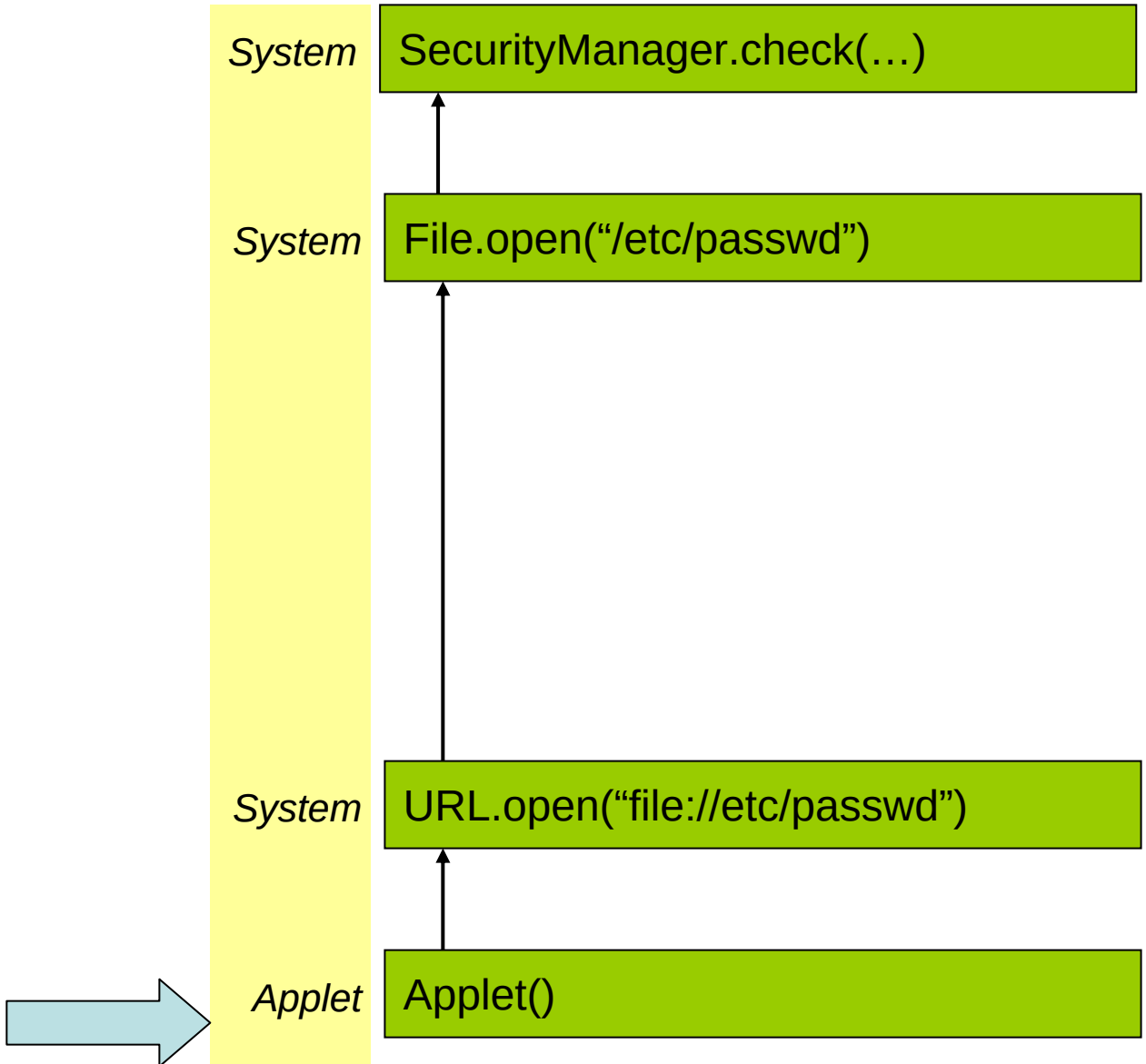
How stack inspection works

… if they're all privileged, then the operation is allowed



How stack inspection works

… if not, then is the applet privileged?



Using stack inspection yourself

- Code signing / secure class loader can assign principals to each class
- Elaborate policy specification language
- Reduces size of trusted computing base
  - You must still audit all doPrivileged() calls
  - Preferably placed close to where priv needed
- Typically mixed with capability-style
  - Stack inspection check to open a file
  - InputStream object acts as a capability
  - Significant performance improvement

Other desired features

- Memory management / limits
- CPU scheduling / priorities
- Termination

Java “isolates”

JSR-121 “Application Isolation API Spec

- Tech details: see Czajkowski et al. “MVM”
  - Java isolate == Unix process
  - Java link == Unix pipe
- Monitoring, termination, security managers
- No guarantees about scheduling, memory
- javax.isolate package (not yet part of J2SE)
  - MIDP 2.0 (J2ME) has early implementation

Java resource consumption

JSR-284 “Resource Consumption Mgmt. API”

- Control heap memory, CPU, etc.

Currently “in progress”, draft API available

- See also, JSR-278 “Resource Mgmt API for Java ME”

What you can do today

Separate JVMs in separate processes

- Consumes more memory
  - Limits on max transactions
- Slower startup time
  - Maybe hide latency by pre-starting JVMs
- Excellent fault isolation
  - Standard OS tools

Similar issues: CGI vs. FastCGI

Research on Java termination

- Wish to terminate Java task running on the system
  - Without destabilizing the system
  - Without the task ignoring the kill signal
  - With minimal changes to the task

- *Task*: Coherent set of related classes from the same source

Relevant publications

- Soft termination
  - ACM Transactions on Information and Systems Security (2002)
- Transactional rollback
  - International Conference on Dependable Systems and Networks (2002)
- GC-based memory accounting
  - IEEE Security and Privacy (2003)

Joint work with Algis Rudys, David Price, and John Clements

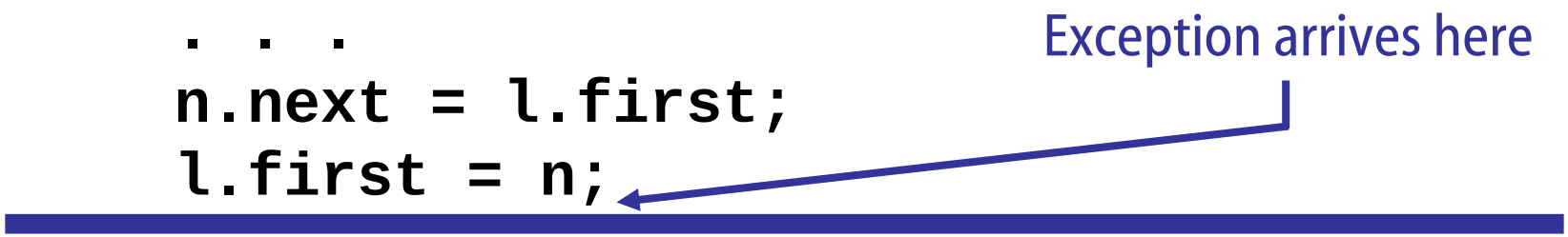
Termination possibilities

- Naïve termination
  - Blindly throw an asynchronous exception
  - Deschedule a thread (**Thread.stop()** in Java)
- "Hard termination"
  - Same as Unix processes or Java isolates

Naïve termination

- User code can simply catch the exception
- Might destabilize the system
  - Violating system invariants

```
system_list_insert(node n) {
   . . .
   n.next = l.first;
   l.first = n;
   l.counter++;
   . . .
}
```



Naïve termination: Which thread?

- Task threads can spawn other threads
- System threads calling into the task can be hijacked
  - **Object.finalize()** to hijack GC thread

☞ Don't kill threads, disable tasks

Soft termination: Our goals

- Portability
  - Run without modifying language runtime
- Reasonable performance
  - Don’t interfere with the optimizer
- Well-defined semantics
  - Effect of termination signal is clear
  - Preserve system code invariants

Soft termination: Design

- Inspiration: “Safe points”
  - Used in language runtime systems for years
- Wherever the code acts to extend its running time, check termination flag first
- Code-to-code transformation
  - User code instrumented to perform check
  - User code terminates itself
- System code is not rewritten, only user code

The basic case

- An infinite loop

```
foo() {
   . . .

   foo();
}
```

Insert termination checks

- Rewrite code to perform a termination check before each function call

```
foo() {
  . . .
  if (termination_flag)
    throw exception;
  foo();
}
```

Fun implementation issues

- Blocking calls
  - Some calls not guaranteed to return
- Weird Java control flows
  - Switch statements
  - Exception handlers
- System code with state
  - Avoid breaking invariants on system state
- Optimizations
  - Don’t do checks if you can prove termination would happen anyway

Java blocking functions

- Java blocking functions are:
  - I/O functions
  - Java synchronization functions
- Cannot directly apply model
  - Assumes we have “equivalent” non-blocking functions

Java blocking calls: Solution

- All Java blocking calls are **`native`**
  - Easy to find by searching Java API source
- Can simulate non-blocking with **`Thread.interrupt()`**
  - Causes all blocking calls to throw Java exception
  - Part of blocking function APIs – callers required to handle
    - Expected behavior

Which threads to interrupt?

- Wrapper registers the current thread
  - Maps current thread to task on behalf of which it is blocking
  - If task is terminated, thread is interrupted

```
wrapper_bar() {
  register_blocking( // Uses stack inspection
    Thread.currentThread());
  blocking_bar();
  unregister_blocking(
    Thread.currentThread());
}
```

Implementation

- 2732 lines of Java source
- Uses JOIE, IBM CFParse classfile manipulation libraries
  - Used bytecode transformation exclusively

- *Prototype available for Tomcat Servlets*

  `www.cs.rice.edu/~arudys/software/`

Further extensions: rollback

- It would be nice to restart a task after you kill it
- Inspiration: transactional database rollback/recovery
- Making backup copies for undo is expensive!
  - 6*x* – 24*x* slowdown on benchmarks
- Optimizations help, but overhead still unacceptable (may require JVM changes)
- Details: see our paper (DSN 2002)

Memory overuse / abuse

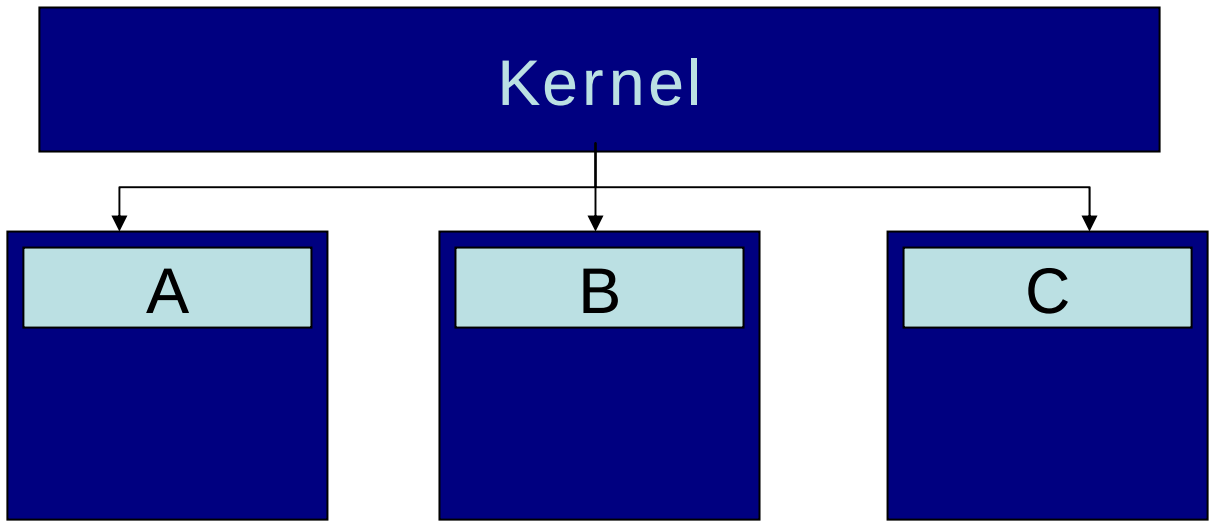
- Possible to hold live significant memory
- Attacker or broken task can:
  - At worst: crash the VM
  - At best: cause thrashing and poor performance
- We want to manage memory better

Stopping memory overuse

- Can solve this problem if we can:
  - Measure memory usage
  - Identify policy violators
  - Constrain/terminate them
- Our work: measuring memory usage
  - without giving up sharing benefits of a language-based system

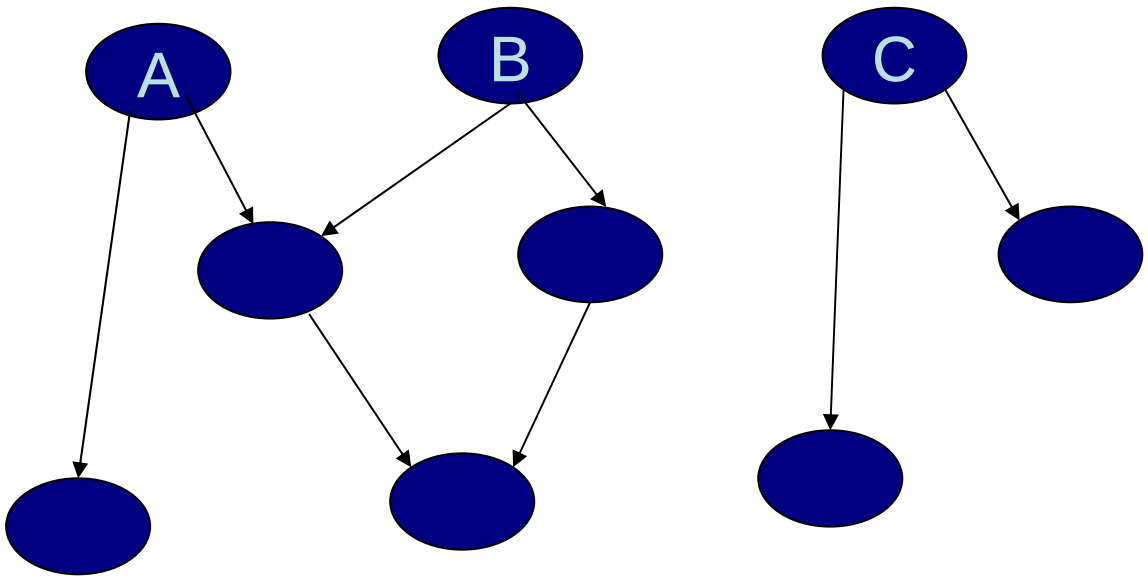
Measuring memory

- How can we determine how much memory a task is using?
- Very easy in a traditional OS model
  - Just measure allocated heap



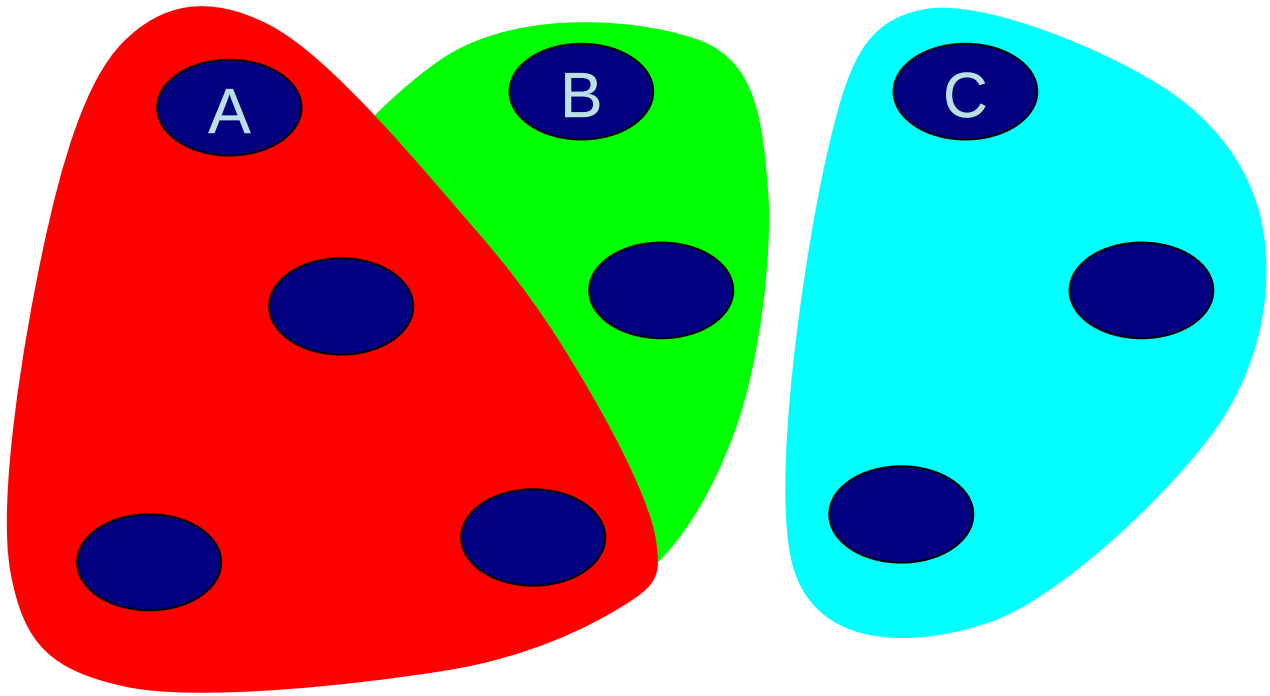
Harder with a shared heap

- No clean boundaries
- Memory shared across tasks



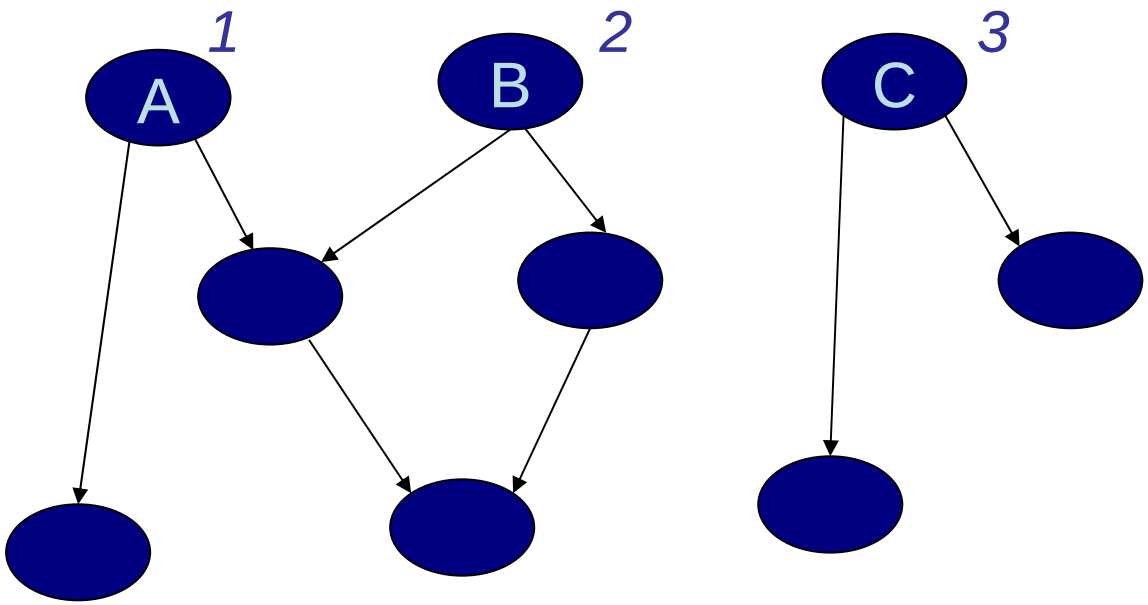
Idea: graph traversal

Defining memory usage: Traverse the graph created by heap memory and measure what we find reachable



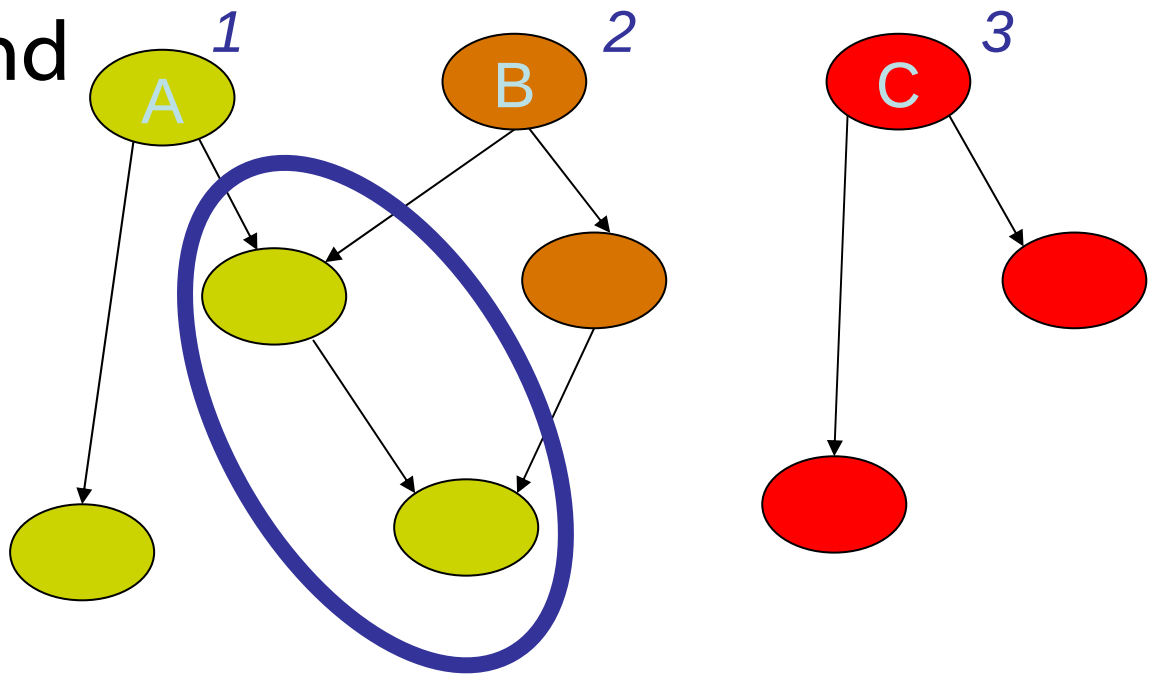
Use the garbage collector

- Start with roots for first task
- Find and count reachable memory
- Repeat for all tasks
- Don’t double-count



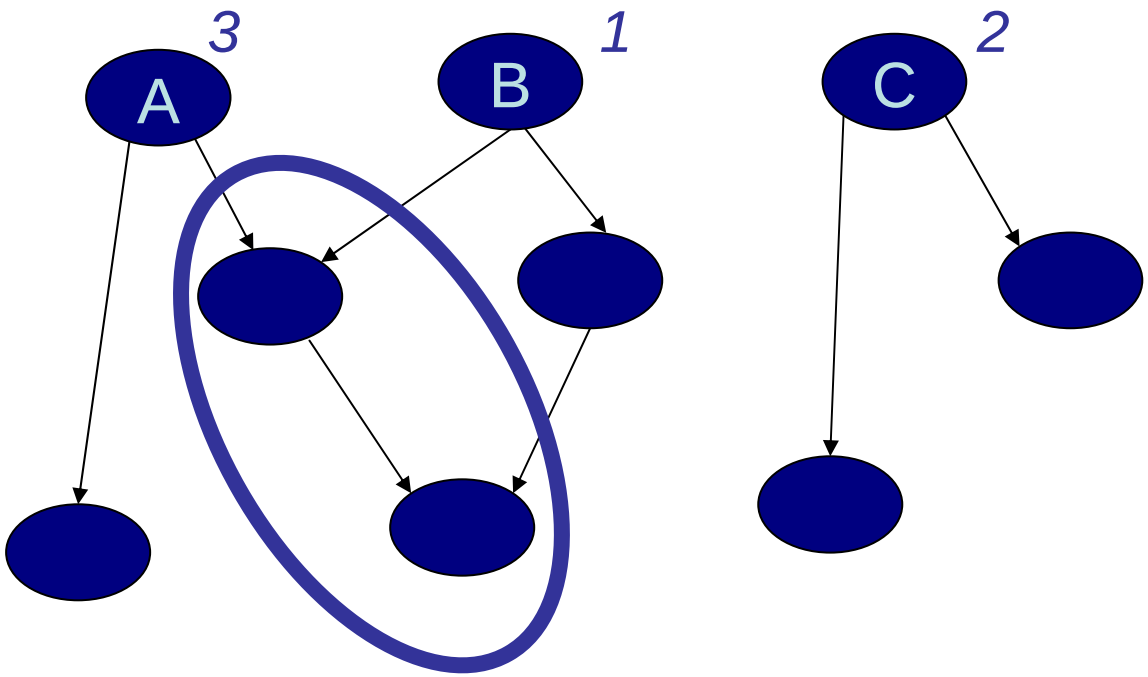
Shared memory

- Memory could be held live by multiple tasks
- Charged to the first such task scanned
- First task: upper bound, last task: lower bound



Finding shared memory

- Reorder the processing of tasks on each GC
- Update upper/lower bounds
- We get a statistical picture of each task's sharing with others



Advantages of GC approach

- Measurements happen when we want them most
  - When memory pressure goes up, so does measurement frequency
  - Can force measurement whenever desired at cost of additional collection
- Rotation of roots gives upper/lower bounds on shared data

Advantages of GC approach

- Measurement is transparent to tasks
- General approach: works in any language runtime with a suitable garbage collector
- Implemented as a tweak of already-existing behavior

Policy implications

- When do we decide that a task is misbehaving?
  - System-specific decision
- Consuming lots of unshared memory
- Holding too much memory live
- Sudden increase in usage
  - May serve as warning, pay more attention to this task in the future

Implementation

- We implemented our system in Java
  - IBM Jikes™ Research Virtual Machine (RVM), version 2.1.0
  - 1000-line patch covering 2 garbage collectors
- Defined tasks on a ClassLoader basis
  - Other definitions possible
- Performance overhead generally < 5%
  - Free ride on the garbage collector
- Details: see our paper (Oakland 2003)

Summary

- Secure services / least privilege
  - Capability-style or stack inspection
- Termination
  - OS processes, isolates, or soft termination
- Memory management
  - OS processes, GC modifications