Extensible Security Architectures for Java

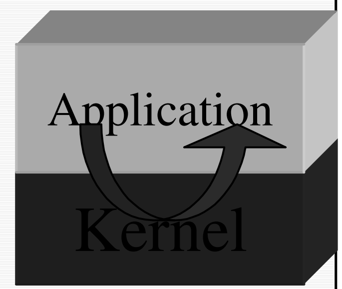
Authors: Dan S.Wallch, Dirk Balfanz
Presented by Moonjoo Kim

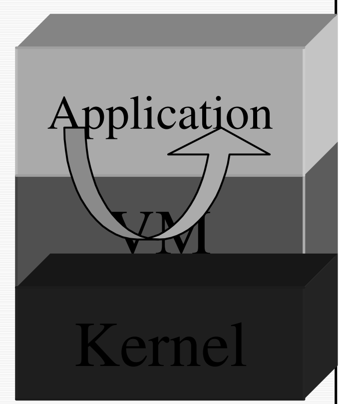
Outline of presentation

- Introduction
- Security in Java
- 3 Approaches
  - Capabilities
  - Extended Stack Introspection
  - Type Hiding-Analysis
- Evaluation of above 3 approaches
  - Criteria
- Conclusion

Introduction

- Mobile code systems must have stronger security
- Web browsers rely on software mechanisms for protection
- Advantage of software protection over hardware protection
  - Portability : Platform independent security mechanism
  - Performance : Cheap cross-domain call
- Secure services with software protection
  - Not only memory protection, but more!





Security in Java

- PL mechanisms to enforce memory safety
- "Sandbox" model for sensitive system service in Sun's JVM
  - Local code is completely trusted.
  - Remote code is completely untrusted.
- Sun's JVM implementation to make above checks
  - `ClassLoader`
  - Stack introspection
    - Reference to the class in every frame on the call stack.
  - All the potentially dangerous methods were designed to call a centralized `SecurityManager`
  - Too inflexible

3 Approaches

- Providing fine-grain security policy
  - Distinguishing between "different" sources of programs
  - Provide appropriate policies for each of them.
  - All presume the presence of digital signatures
- 3 Approaches
  - Capabilities
  - Extended Stack Introspection
  - Type Hiding

Capabilities

- A capability is an unforgeable pointer to a controlled system resource.
- In Java, a capability is simply a reference to an object
- Java capabilities would implement interposition by providing the exclusive interface to system resources
  - Rather than using the File class, a program would be required to use a file system capability acquired on startup or through a broker
- Java run-time library would require significant changes

```
// In this example, any code wishing to open a file must first
// obtain an object which implements FileSystem
interface FileSystem{public FileInputStream getInputStream(String
path);}

// This is the primitive class for accessing the filesystem.
// Note that the constructor is not public -- this restricts creation
of these
// objects to other code in the same package
public class FS implements FilsSystem{
        FS(){}
        public FileInputStream getInputStream(String path){
                return internalOpen(path);    }
        private native FileInputStream internalOpen(path);
}

// This class allows anyone holding a FileSystem to export a subset of
// that filesystem.  Calls to getInputStream() are prepended with the
// desired filesystem root, then passed on to the internal FileSystem
capability
public class SubFS implements FileSystem{
        private FileSystem fs;
        private String rootPath;
        public SubFS(String rootPath, FileSystem fs){
                this.rootpath = rootPath;
                this.fs = fs;}
        public FileInputStream getInputStream(String path){
```

Extended Stack Introspection

- To support multiple principals with different privileges
- Target is defined as (prin,string-name)
  - Ex> (System,"networking")
- Policy engines
  - given a principal requesting access for a target, say yes,no, or blank
- `enablePrivilege(target)`
  - when code wishes to use the protected resource, call it. It will consult policy engine and privilege is written in caller's stack frame if permitted.
- `checkPrivilege(target)`
  - It searches the call stack for an enabled privilege to the given target.
- JVM should be modified for extended stack introspection

Example: Network Policy

•Default Java policy is to allow network connections to be created only to the host from which the applet originated

•Implementation of default java network policy

Principal	Call Stack	Privilege operations
user	User()	
System	Safe_net_conn()	+networking
System	net_connect()	
System	checkPrivilege()	

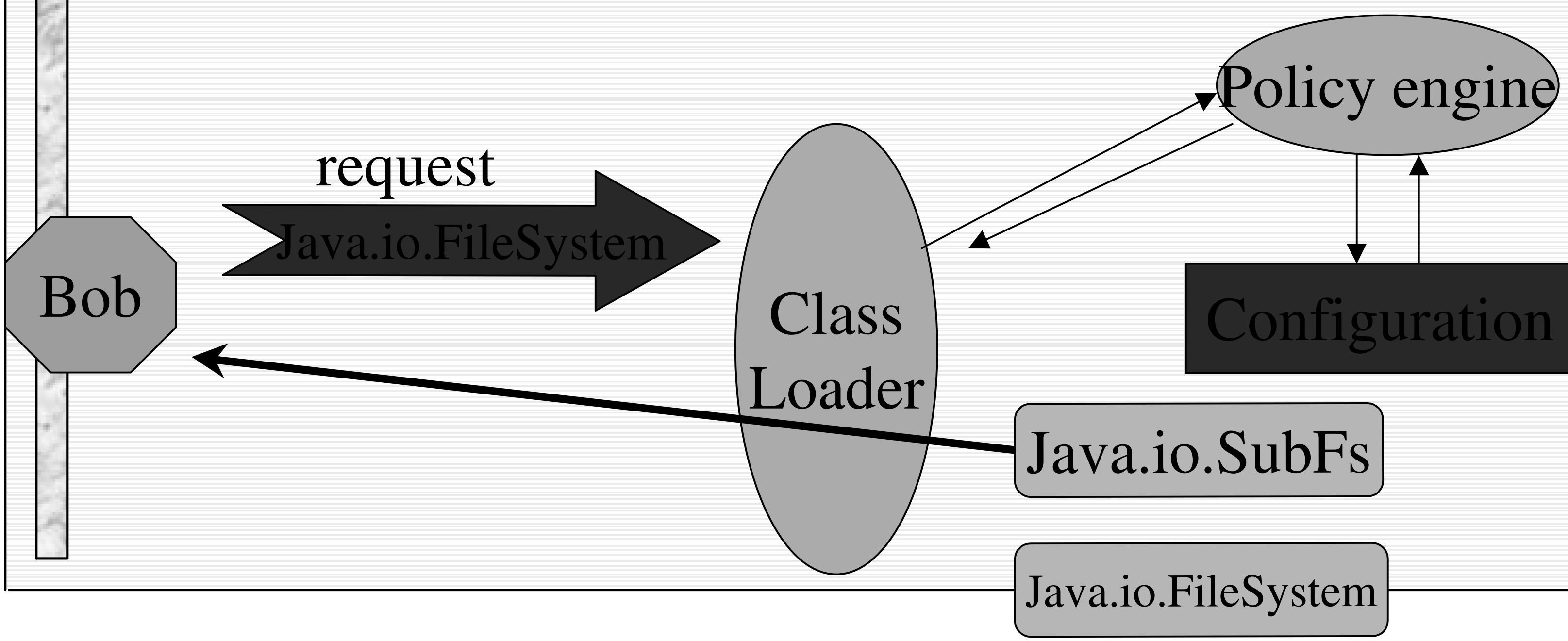
Type Hiding

- Java uses run-time dynamic linking mechanism.
  - We enforce a given security policy by controlling how names in a program are resolved into run-time entities.
- Policy engine determines which configuration is used for the code's name space, as a function of its principal.
  - Configuration, which is a mapping from class names to implementations
- Java `ClassLoader` should be modified
- `ClassLoader` is used to provide mapping from class-name to class-implementation

Example

- Configuration

Original name	Alice	Bob
Java.net.Socket	Java.net.MySocket	Java.net.Socket
Java.io.FileSystem		Java.io.SubFS



Criteria

- Economy of mechanism
- Complete mediation
- Least privilege
- Performance
- Compatibility
- Fail-safe defaults
- Least common mechanism
- Accountability
- Psychological acceptability
- Remote calls

Economy of mechanism

- Type hiding is the simplest
- Unmodified capability is simple, too. But fully capability-based Java requires redesign of library.
- Stack introspection requires complex change

Complete mediation

- Simple capability is problematic
- Type hiding have good confinement property
- Stack introspection have excellent confinement.

Least privilege

- Type hiding is somewhat problematic
- Stack introspection provides middle ground
- Capabilities have very good properties.

Performance

- Stack introspection has the highest runtime costs
- Type hiding and capability system don't incur run-time overhead

Compatibility

- Language based protection can be implemented without diverging much from original specification of the language.
- Type hiding and extended stack introspection provide good compatibility
- Capability requires changes of Java class API design

- Fail-Safe Defaults
- Least common mechanism
- Accountability
- Psychological acceptability
- Remote calls

Conclusion

- Software-based protection systems
- Three designs for security are suggested
  - Capability system
    - It can be implemented naturally in Java. But it requires drastic change of API design
  - Stack introspection
    - It introspection offers good compatibility. But its complexity is troubling and its implementation is very specific to JVM
  - Type hiding
    - It offers good compatibility.
- To combine these techniques could be a solution