

Extensible Security Architectures for Java

Dan Wallach Dirk Balfanz Drew Dean
Edward Felten

Summarized by Wen-Ke Chen

1 Introduction

The needs for portability and performance make software protection advantageous and desirable compared to hardware protection in securing the execution of mobile code. While memory protection via either software fault isolation or proof-carrying code or type-safe languages can defeat any unauthorized access of data and/or execution of code, it is only one part of a security solution; The security of system services must not be undermined when executing untrusted code. This paper presented and compared three strategies for implementing an extensible security architecture for Java to provide service security.

2 Security in Java

In Java, memory protection is achieved by make Java bytecode type-safe. However, in the old security model, a simple “sandbox” is employed to prevent untrusted code from using sensitive system services: local code is completely trusted and given full access to system resources, while code downloaded from the Internet is completely untrusted and refused to access any sensitive system resource. This is accomplished by requiring all potentially dangerous methods in the system be implemented to call a centralized SecurityManager to check if the requested action is allowed. The SecurityManager distinguishes trusted code from those untrusted by simply checking its ClassLoader. In practice, this dichotomous model proved insufficient in practice and obviously too inflexible.

3 Approaches

3.1 Common Underpinnings

A reference monitor can be interposed between untrusted code and the code to be protected to check and make the decision whether a call to be allowed. The

inflexibility of the old Java security model results from its too coarse division of code into trusted and untrusted and of access rights of system resources into all and nothing. Fine divisions enable more flexible security policy but also require a way to identify principals and access rights on a finer-grain scale. Digital signatures are employed in the three approaches described below to identify principals and a policy engine can be consulted to enforce finer-grain protection: which code has what permissions to access which targets. Also, complexity in users' making security-relevant decisions can be alleviated by moving the work to their system administrators.

3.2 First Approach: Capabilities

A capability is an unforgeable token to get access to a controlled system resource. In Java, a capability can simply be a pointer to an object used to represent the resource. Having a capability means having some rights to access the resource represented by the capability. Capabilities can be explicitly given or obtained as a result of calling other capabilities. To use capabilities as a protection mechanism requires an exclusive interface to system resources be defined and interposed. For example, rather than using the `File` class, or the public constructor of `FileInputStream`, a program must use a file system capability to get access to the file system, which obviously requires the current Java class libraries be re-designed and re-implemented to make `File` and the like hidden from programs.

3.3 Second Approach: Extended Stack Introspection

The implementation of the old Java dichotomous security model has been extended by Netscape and Microsoft (and now by JavaSoft) to provide further flexibility in securing system services. A centralized policy engineer defines the following three primitives for a fine-grain protection of a resource:

- `enablePrivilege(target)` must first be called to consult to see whether the principal of the caller has the permission when code wishes to use the protected resource representing by *target*;
- `disablePrivilege(target)` should be called to discard the enabled privilege after accessing the resource; and
- `checkPrivilege(target)` must be called by the system before actual accessing the protected resource.

In implementation, `enablePrivilege()` makes an annotation on the current stack frame if the principal has the permission to access the *target* resource or

throws an exception if the opposite is true; `disablePrivilege()` nullifies the annotation; and `checkPrivilege()` inspects the stack to make sure the principal is allowed to access the protected resource: stack frames are searched in sequence from the newest to the oldest, access is granted after finding a stack frame annotated with an appropriate enabled privilege, or disallowed after finding a stack frame with an annotation by the policy engine forbidding access of the resource. By this way, enabled privileges are guaranteed to apply only to the thread that created them, to be discarded automatically when the method that created them exits, and to be protected from being exploited by untrusted callees. Note that Netscape denies permission while Microsoft allows it when the search reaches the end of the stack uneventfully.

3.4 Third Approach: Name Space Management

With name space management which is enabled by Java's dynamic linking mechanism, a given security policy is enforced by controlling how names in a program are resolved into runtime classes: the class can be entirely removed from the name space if access to the resource represented by the class should be denied, or its name can be mapped to a different class compatible with the original to restrict the access of the resource. These mappings are encoded in *configurations* defined for principals. In implementation, the Java `ClassLoader` is modified, or more precisely, each `AppletClassLoader` is replaced with a `PrincipalClassLoader`, which, when resolving a class reference, can

- throw an exception if the calling principal should not see the class; or
- return the class in question if the calling principal has full access to it; or
- return a subclass as specified in the configuration for the calling principal.

Note that, as in the first approach, to achieve complete protection, many classes in Java API should be replaced.

4 Analysis

The three approaches are evaluated against 10 criteria as follow:

1. *Economy of mechanism.* Designs should be as small and simple as possible to make it easier to inspect and trust. Name space management is possibly the simplest, and extended stack introspection may be the most complicated.
2. *Fail-safe defaults.* Access should be denied by default unless explicitly granted. A fully capability-based has the best fail-safe behavior, and the other two have similiar fail-safe behavior.

3. *Complete mediation.* Every access should be checked. A capability system usually has *confinement* problem, while name space management can potentially have good confinement properties, and stack introspection has excellent confinement.
4. *Least privilege.* Only grant the minimum set of privileges necessary to fulfill a task. Name space management cannot revoke a granted privilege, stack introspection can limit the lifetime of a privilege, and capabilities have the most desirable properties,
5. *Least common mechanism.* Share as little data as possible. All the three approaches are equally dissatisfactory.
6. *Accountability.* The principal using a privilege should be recorded in order to be able to trace the responsible. With name space management, information about principals is not generally available at run time, while in the stack introspection system, every call to enable a privilege can be logged, and a capability system can remember the principle to which a capability is granted.
7. *Psychological acceptability.* Users shouldnot be unduely burdened. All three systems can present the same interface to a user.
8. *Performance.* How the design constrains system performance. Assuming the JVM uses a JIT compiler, stack introspection has the highest runtime cost, while name space management and unmodified capability systems only pay similar runtime cost to implement interposition layers.
9. *Compatibility.* How many changes must be made to the existing. Both name space management and stack introspection can implemented without affecting existing applets, but a capability system would completely break compatibility with existing Java APIs.
10. *Remote calls.* Can it be extended cleanly to remote calls? Capabilities can extend quite naturally across a network, name space management is comparable to the directory services offered by most RPC systems, and stack introspection can become very complex.

5 Conclusion

Software-base protection has advantages in both performance and portability when securing the execution of mobile code, and besides memory safety, system services must also be secured. Three approaches to securing system servies are described and compared against some criteria, revealing each of them has its own strengths and weaknesses, so the best solution is to combine elements of all techniques.