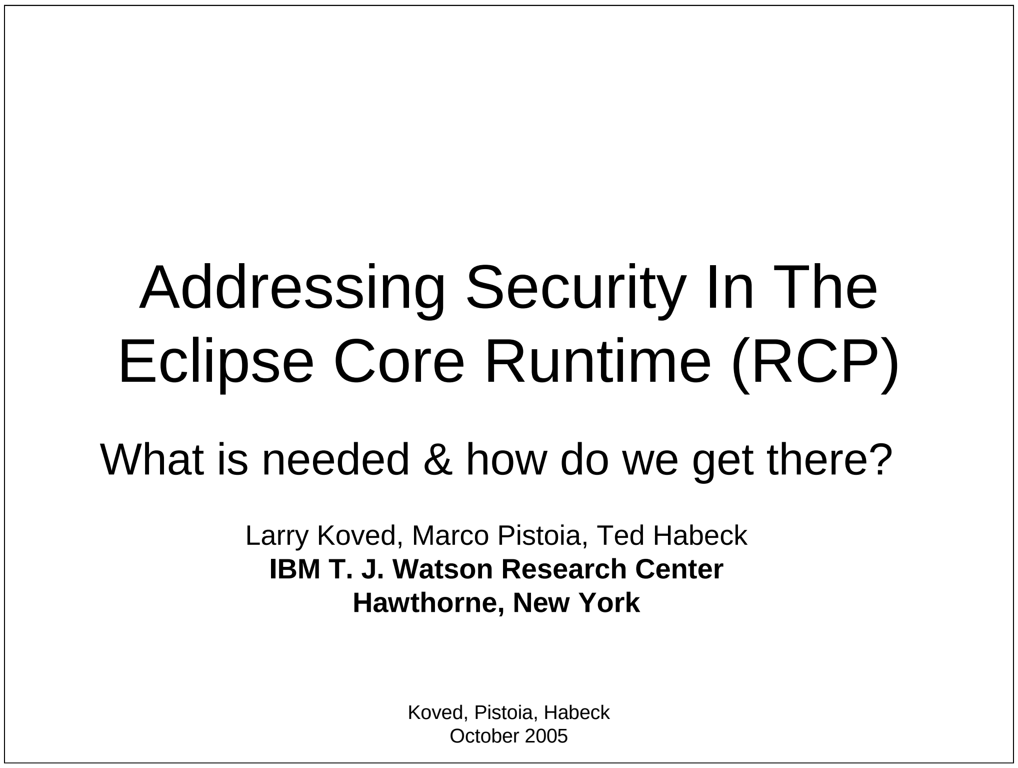


Addressing Security In The Eclipse Core Runtime (RCP)

What is needed & how do we get there?

Larry Koved, Marco Pistoia, Ted Habeck
IBM T. J. Watson Research Center
Hawthorne, New York

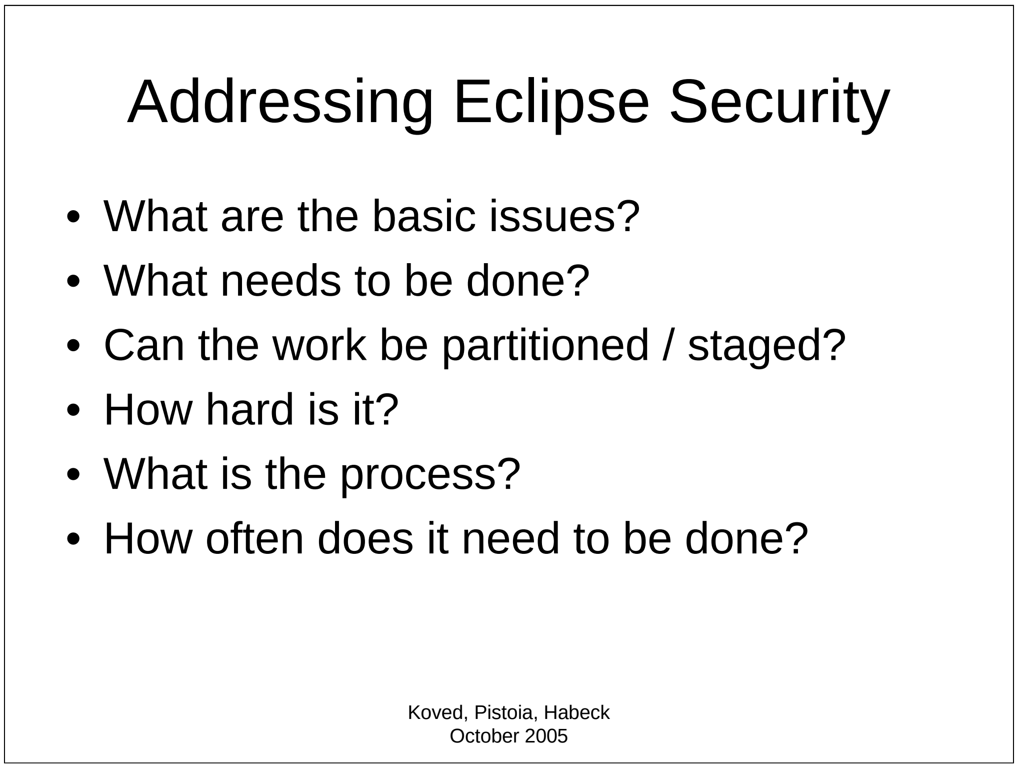
•Eclipse RCP is intended as a platform for configuring and deploying applications, expanding beyond Eclipse's origins as an IDE.

•Notably, Eclipse is now implemented on top of the OSGi framework. In particular, OSGi supports an extension of the Java 2 security model for authorization.

•Organizations, including Lotus, are interested in a platform on which applications can be securely deployed, employing a security sandbox similar to Applets in a browser.

•Some of the fundamental questions are:

- •What does it mean to add security to Eclipse?
- •What are the steps to adding security to Eclipse?



Addressing Eclipse Security

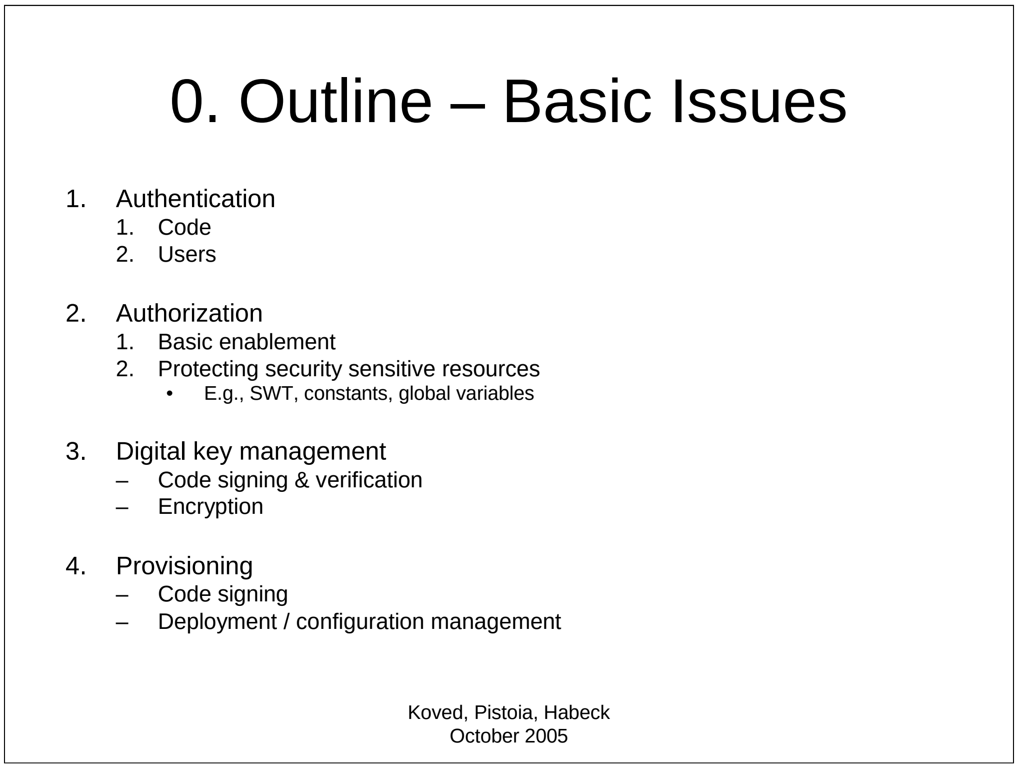
- What are the basic issues?
- What needs to be done?
- Can the work be partitioned / staged?
- How hard is it?
- What is the process?
- How often does it need to be done?

Koved, Pistoia, Habeck
October 2005

•These are commonly asked questions about security and its relationship to Eclipse.

•This is the general outline of the rest of this presentation.

•Note that we have released a tool, called Security WORkbench Develop for Java (*SWORD4J*) on *IBM alphaWorks* (http://www.alphaworks.ibm.com/tech/sword4j) to assist developers with a number of the security issues discussed in this presentation.



0. Outline – Basic Issues

1. Authentication
   1. Code
   2. Users
2. Authorization
   1. Basic enablement
   2. Protecting security sensitive resources
      - E.g., SWT, constants, global variables
3. Digital key management
   – Code signing & verification
   – Encryption
4. Provisioning
   – Code signing
   – Deployment / configuration management

Koved, Pistoia, Habeck
October 2005

While security is a very broad topic, we see that there are four basic security subtopics that are certainly relevant to security enabling Eclipse RCP. These are discussed in this presentation:

•Authentication. In Java,

•There is authentication of both code (from whom was the code received), and

•Users (who is executing the code)

•Authorization. This is the basis of the Java *security sandbox*. The same mechanisms can be exploited by other Eclipse projects, including ECF and Higgins.

•Java has basic mechanisms to perform authorization, such as the SecurityManager and AccessController. However, since the RCP code was not designed / implemented to support security, it needs to be updated to allow both trusted and untrusted code to be deployed in RCP. This presentation describes a process by which RCP can be security enabled.

•Some security sensitive resource need to be protected. This will be described in greater detail.

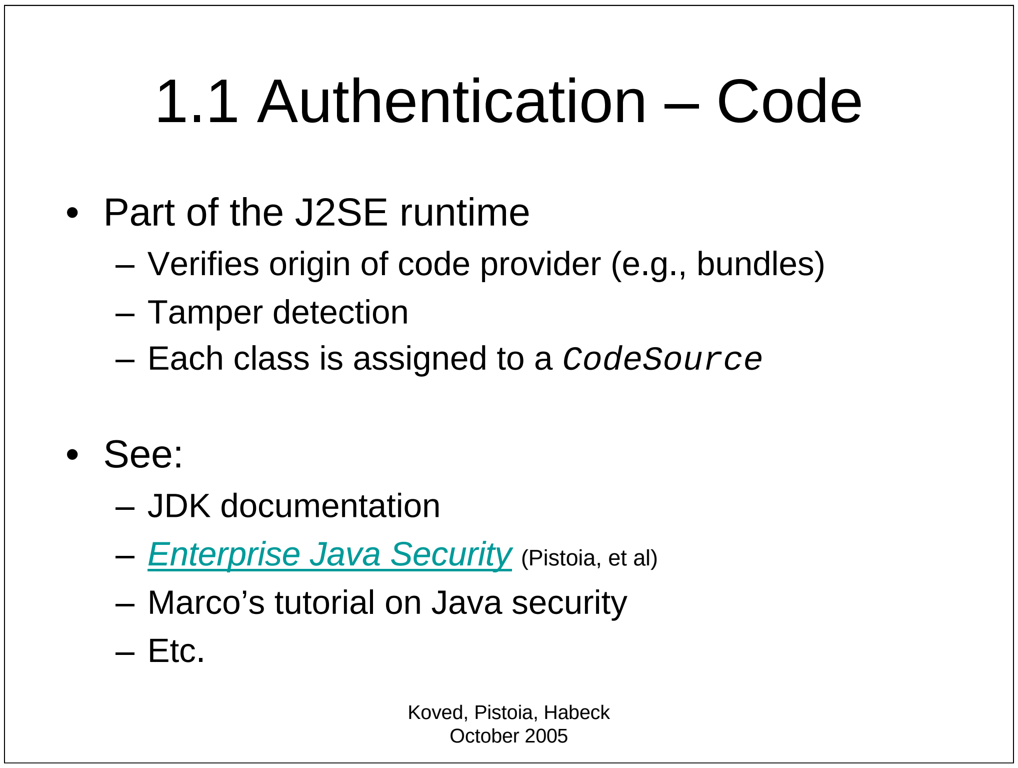
•Digital key management

•Digital keys are used in support of both code signing and encryption operations. We'll describe prototype Eclipse plug-ins that make generation, management and use of digital keys easier.

•Provisioning

•Code signing uses the digital keys to "sign" (compute a secure hash value) of the codde

•Additional work has been proposed elsewhere on top of the OSGi frameworks to address issues that are Eclipse-specific, including protection of extension points



1.1 Authentication – Code

- Part of the J2SE runtime
  - Verifies origin of code provider (e.g., bundles)
  - Tamper detection
  - Each class is assigned to a *`CodeSource`*
- See:
  - JDK documentation
  - *Enterprise Java Security* (Pistoia, et al)
  - Marco's tutorial on Java security
  - Etc.

Koved, Pistoia, Habeck
October 2005

In the first part of this presentation, we describe some basic security issues in Java-centric terms. In subsequent sections we propose what work is needed in Eclipse RCP, and a process by which a security sandbox can be enabled for RCP. Management of authenticated identities is outside the scope of this presentation, though it is a topic of active interest within Higgins, ECF and other Eclipse projects.

•Java 2 (e.g., Java 1.4.x and Java 5) provide support for verifying the origin of code based on the use of digital signatures (code signing).

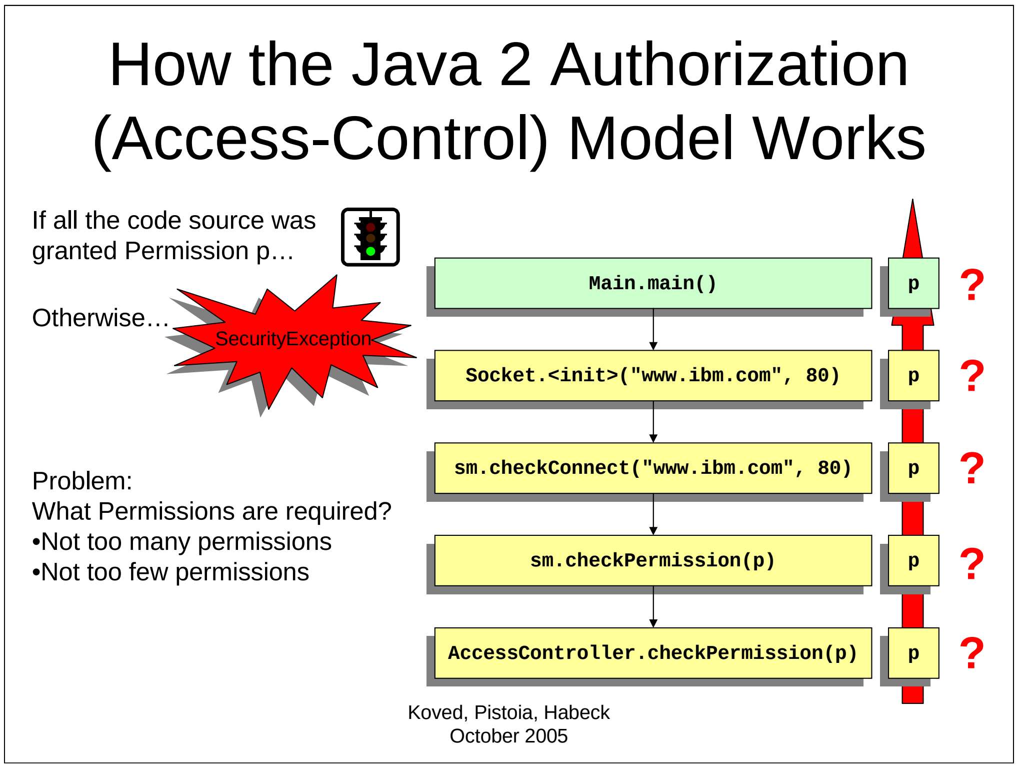
  •Detect whether there has been modification of the code after it was signed (tamper detection)

  •Once verified, code is assigned to a "CodeSource". A CodeSource is:

    •Cryptographic signature of the code (a secure hash value)

    •Location from which the code was loaded

•There are extensive descriptions of code authentication in the JDK documentation, books, and a tutorial that Marco has presented a number of times.



How the Java 2 Authorization (Access-Control) Model Works

If all the code source was granted Permission p…

Otherwise…

Problem:
What Permissions are required?
•Not too many permissions
•Not too few permissions

```
Main.main()                               p  ?
Socket.<init>("www.ibm.com", 80)          p  ?
sm.checkConnect("www.ibm.com", 80)        p  ?
sm.checkPermission(p)                     p  ?
AccessController.checkPermission(p)       p  ?
```

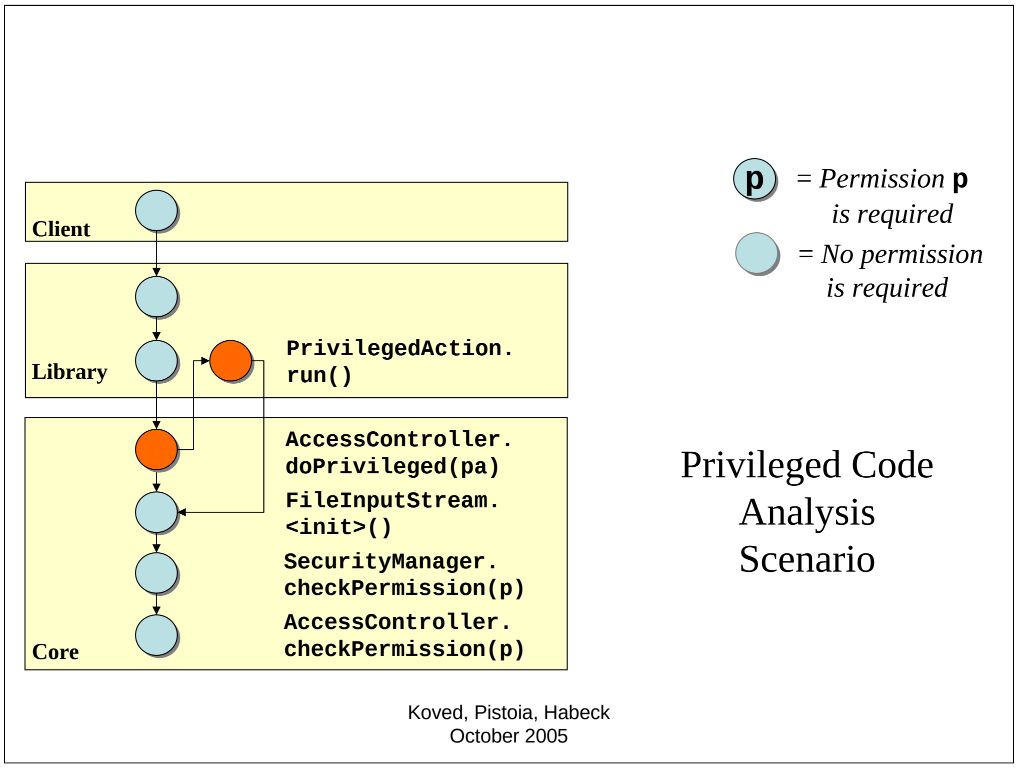
Koved, Pistoia, Habeck
October 2005

The challenge, including for RCP core runtime developers and Eclipse plug-in / OSGi bundle developers, is determining which permissions should be granted to the code in their plug-ins / bundles. Later in this presentation we will describe a tool, called SWORD4J, that assists developers. The results of the authentication are used for authorization (access control).

First, a ***really short*** tutorial on how Java 2 code authorization works.

In general, when a Java 2 authorization test is performed, all of the code on the runtime stack must be authorized for the privileged operation to succeed. If *one or more* of the methods on the runtime stack is *not* authorized, a SecurityException is thrown (and the privileged operation is *not* peformed). Authorization begins at the stack frame for the AccessController.checkPermission() method and the Java authorization logic does a *stack walk* back to the first stack frame of the Thread, checking the authorization for each stack frame's target method's Class.

Note that all of the classes/methods loaded by the *bootclasspath* ClassLoader (including the Java runtime libraries) are granted the equivalent of *AllPermission* by default, so they are automatically authorized for *all* privileged operations.



Privileged Code Analysis Scenario

Koved, Pistoia, Habeck
October 2005

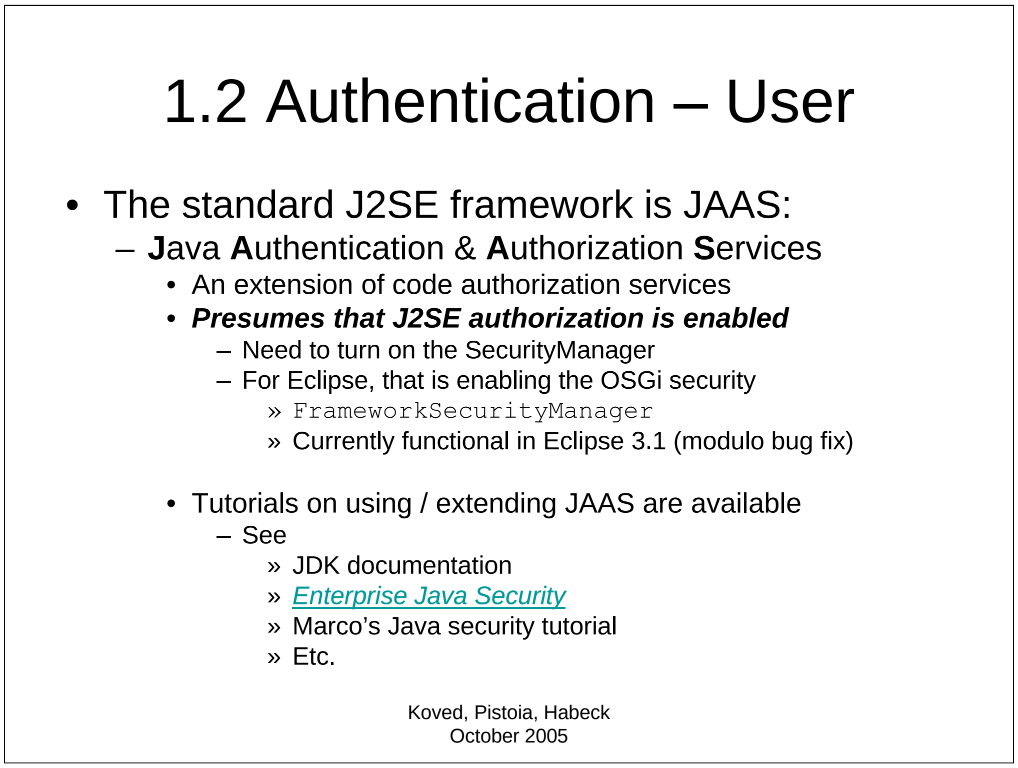
One of the most important issues that arises is whether there is a way to make libraries, such as plug-ins or bundles, “privileged”. For example, enabling them to read configuration information or writing to log files in a way that does not require the library’s client code (code calling the library code) to also be privileged.

In Java, there is a way to do this using the AccessController.doPrivileged() method. The code needing privileges is encapsulated in a *run()* method of a *PrivilegedAction* instance. Often this is done by creating an anonymous *inner class* whose *run()* method contains the privileged code. The PrivilegedAction instance is then passed to a call to AccessController.doPrivileged().

When authorization testing is done via a call to AccessController.checkPermission(), the authorization logic walks back through the stack as previously described. However, this time the authorization testing stops after it checks the stack frame where AccessController.doPrivileged() was called. This stack frame is part of the library code, but is not part of the client code.

If all of the code up through the library code is authorized for the privileged operation being checked, then the authorization succeeds without the client being authorized.

Example of Privileged Code (Using an Anonymous Inner Class)



1.2 Authentication – User

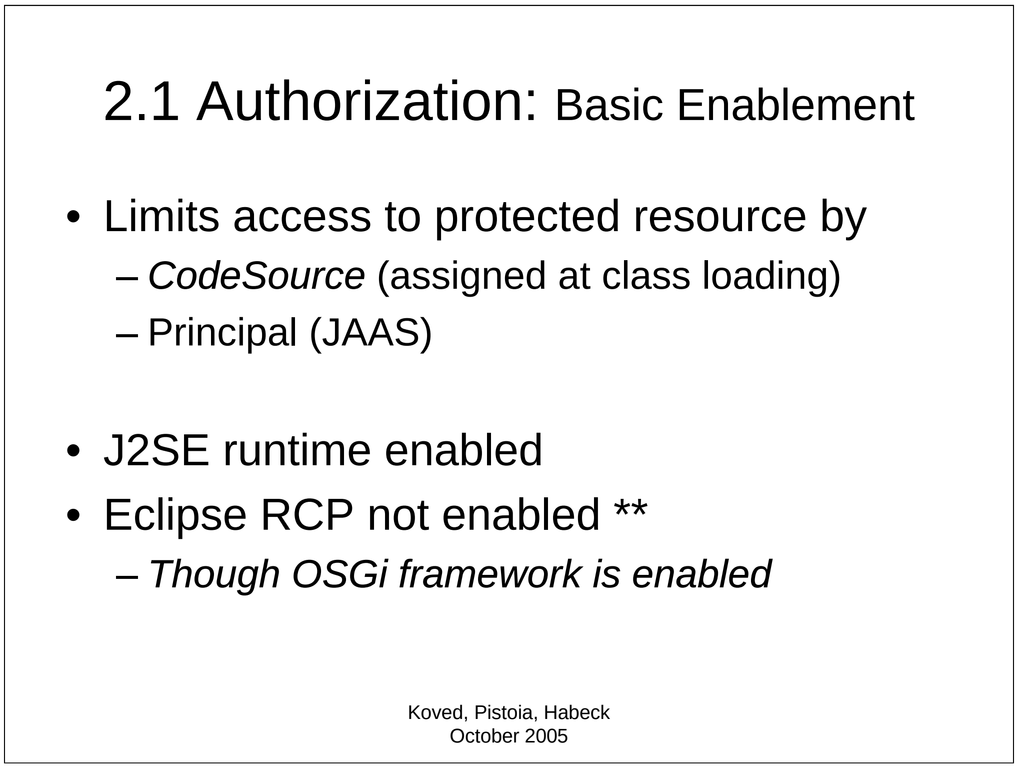
- The standard J2SE framework is JAAS:
  - **J**ava **A**uthentication & **A**uthorization **S**ervices
    - An extension of code authorization services
    - ***Presumes that J2SE authorization is enabled***
      - Need to turn on the SecurityManager
      - For Eclipse, that is enabling the OSGi security
        - » `FrameworkSecurityManager`
        - » Currently functional in Eclipse 3.1 (modulo bug fix)
    - Tutorials on using / extending JAAS are available
      - See
        - » JDK documentation
        - » *Enterprise Java Security*
        - » Marco's Java security tutorial
        - » Etc.

Koved, Pistoia, Habeck
October 2005

As opposed to the traditional Java approach of authenticating / authorizing code to perform privileged operations, it is also possible to authenticate principals (e.g., users) and associate them with threads of execution. Then authorization can also be based on privileges associated with the principals. In Java 2, the framework for authenticating users is called *JAAS* – *J*ava *A*uthentication and *A*uthorization *S*ervices.

JAAS is an extension of code authentication and authorization, and presumes that J2SE authorization (e.g., a SecurityManager has been set in the Java runtime). In Eclipse 3.1, this would be the FrameworkSecurityManager, part of the OSGi framework implementation.

There are a number of tutorials available that describe JAAS, including its authentication frameworks (a.k.a. pluggable and stackable *LoginModules*).



2.1 Authorization: Basic Enablement

- Limits access to protected resource by
  - *CodeSource* (assigned at class loading)
  - Principal (JAAS)
- J2SE runtime enabled
- Eclipse RCP not enabled **
  - *Though OSGi framework is enabled*

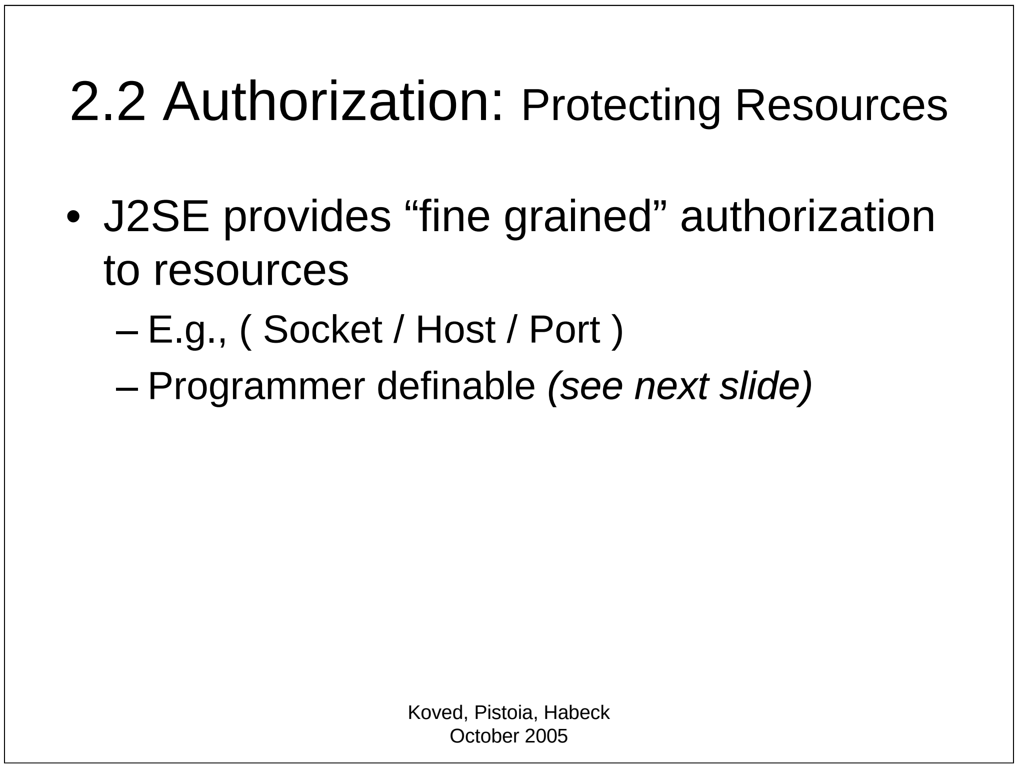
Koved, Pistoia, Habeck
October 2005

In Java, access to protected resources (e.g., data, functions) is mediated by calls to the SecurityManager. Security policies define which code and/or users are authorized access to each of the resources.

The Java 2 runtime libraries are enabled to protect resources and functions that have been previously identified as security sensitive.

** The OSGi framework extends Java 2 authorization frameworks. While Eclipse contains the OSGi framework, which includes the runtime security frameworks, Eclipse RCP itself is *not* security enabled.

** While Eclipse can run with the FrameworkSecurityManager installed in the Java runtime, there are locations in the Eclipse code that should be made privileged. Also, appropriate Java 2 Permissions need to be assigned to the bundles that comprise the Eclipse runtime. Later in this presentation we propose a process for adding the appropriate privileged code and authorizations.



2.2 Authorization: Protecting Resources

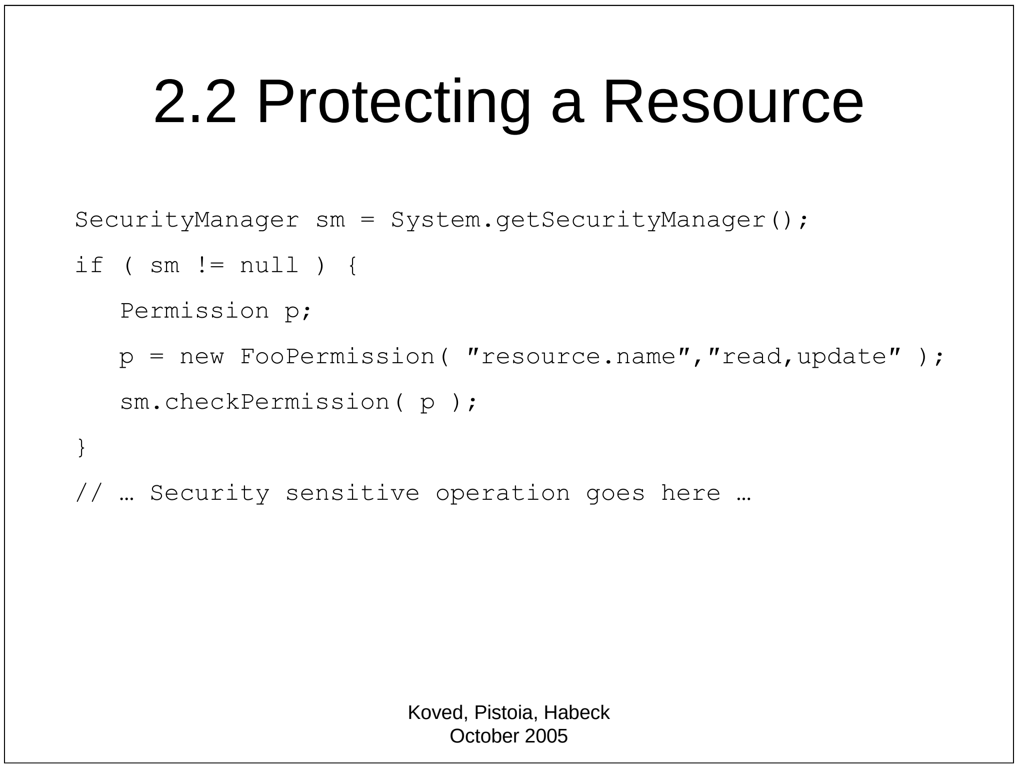
- J2SE provides “fine grained” authorization to resources
  - E.g., (Socket / Host / Port)
  - Programmer definable *(see next slide)*

Koved, Pistoia, Habeck
October 2005

Resource protection in Java can be as *fine grained* as is needed. For example, we can define authorization to access a field (e.g., defining a Socket factor implementation), or a value held in a collection (e.g., a “System property” via System.getProperty() or System.setProperty()).

Not all authorizations need to be fine grained. Examples include access to the system clipboard, printer, or some AWT functionality.

The typical authorization coding pattern is described in the next slide.

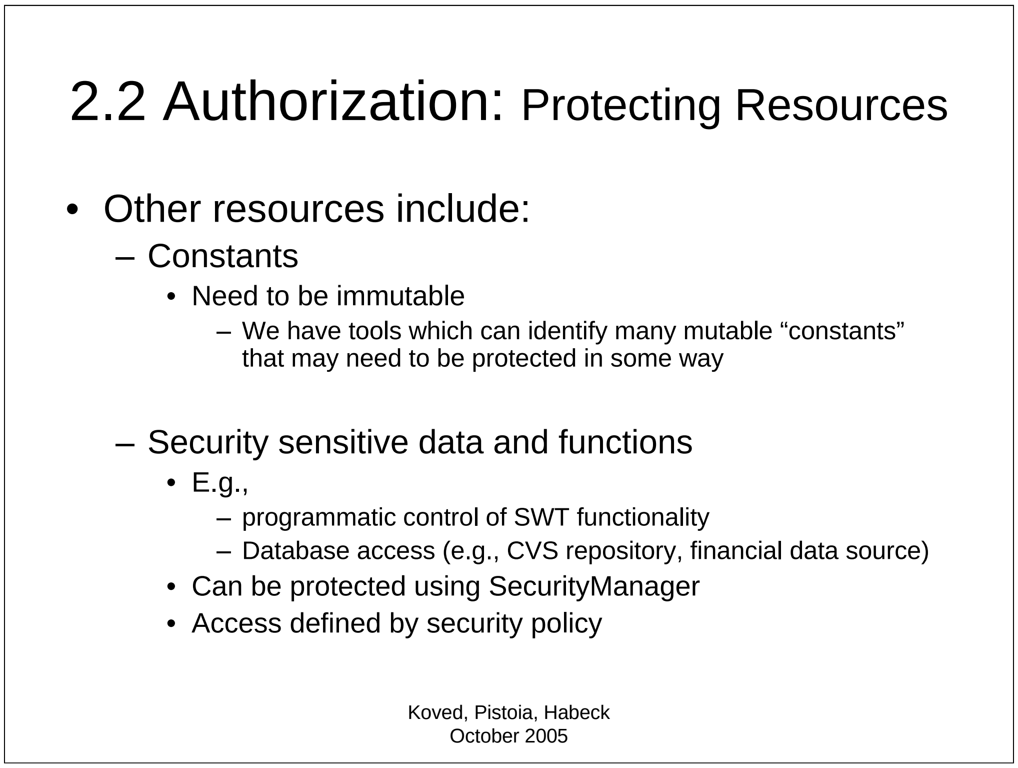


2.2 Protecting a Resource

```
SecurityManager sm = System.getSecurityManager();
if ( sm != null ) {
   Permission p;
   p = new FooPermission( "resource.name","read,update" );
   sm.checkPermission( p );
}
// … Security sensitive operation goes here …
```

Koved, Pistoia, Habeck
October 2005

1. First see if a security manager is installed,
2. A Permission object *p* is created to describe the protected resource (*resource.name*), and the operation(s) on the resource (e.g., *read, update*).
3. The SecurityManager.checkPermission() is called with *p* and the Java authorization system logic checks to see if the code/user is authorized for the protected resource / operation(s), as previously described.



2.2 Authorization: Protecting Resources

- Other resources include:
  - Constants
    - Need to be immutable
      - We have tools which can identify many mutable “constants” that may need to be protected in some way
  - Security sensitive data and functions
    - E.g.,
      - programmatic control of SWT functionality
      - Database access (e.g., CVS repository, financial data source)
    - Can be protected using SecurityManager
    - Access defined by security policy

Koved, Pistoia, Habeck
October 2005

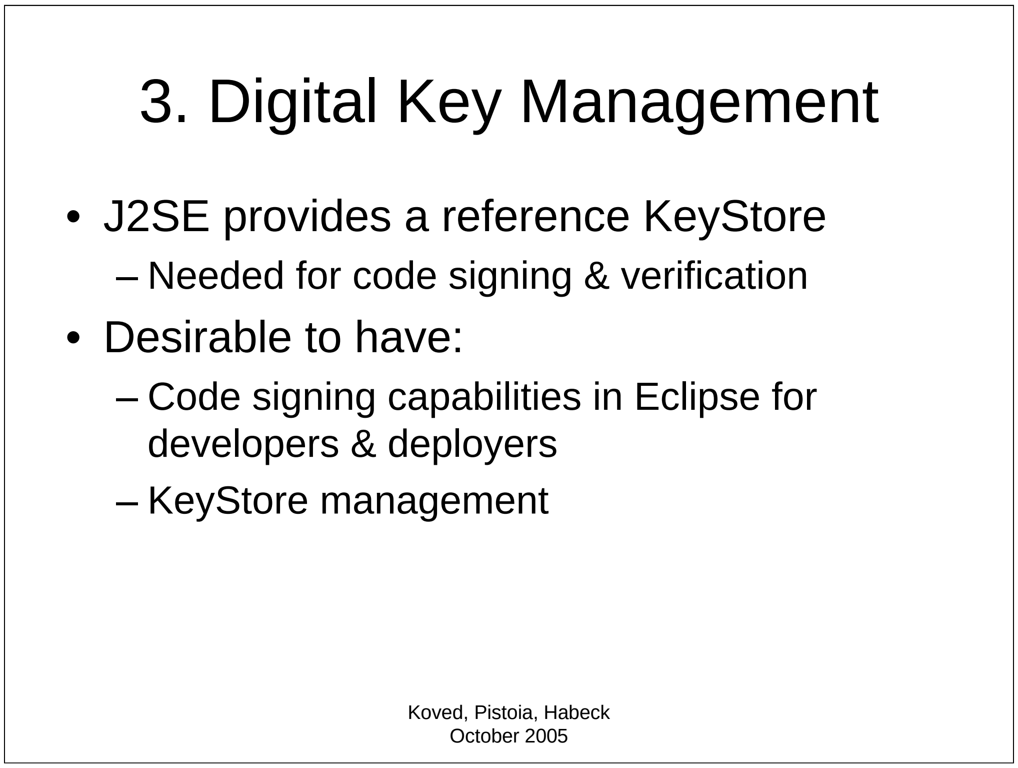
Other resources that need to be protected:

•Constants should be immutable – there should be no way for an unauthorized client to modify a shared constant.

> •We have code which can usually identify whether a constant is mutable or immutable. We hope to make that functionality available in an upcoming release of SWORD4J.
>
> •If constants can not be made immutable, they may need to be protected via authorization checks.

•Security sensitive data and functions need to be protected.

> •This includes some functionality in SWT (e.g., programmatically navigate and control the GUI. Probably done in a medium-grained authorization approach as is currently done by AWT.
>
> •Database access (e.g., CVS repository, Eclipse configuration files)
>
> •Access to these resources defined by policies (e.g, Permissions defined for their bundles).



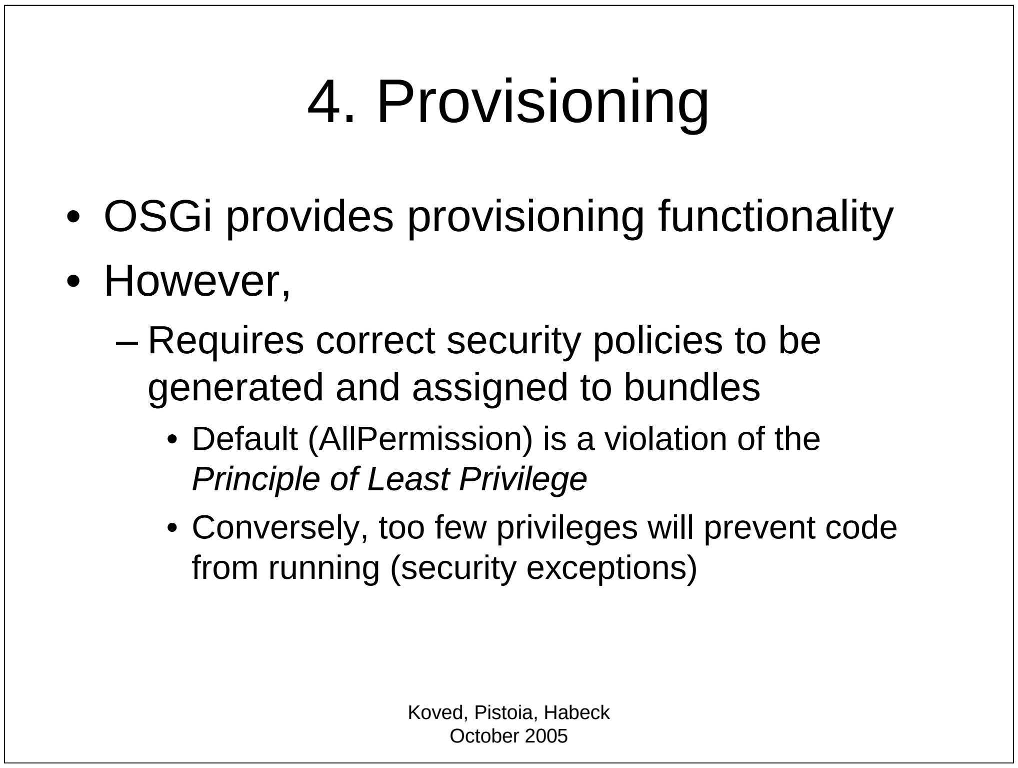
3. Digital Key Management

- J2SE provides a reference KeyStore
  - Needed for code signing & verification
- Desirable to have:
  - Code signing capabilities in Eclipse for developers & deployers
  - KeyStore management

Koved, Pistoia, Habeck
October 2005

In a security environment, such as a secure RCP, X.509 digital keys are used in a number of ways, including code signing and SSL sessions. As a practical matter, these digital keys need to be stored in a standard location / format, including utilities to manage the digital keys. In Java, this Is called a *keystore*.

By default, Eclipse does not provide any digital key management functionality. However, if deployment of security enabled Eclipse-based systems are to be developed, it would be desirable to have digital key management (including keystore) functionality built into Eclipse. ECF, Higgins, among other projects, are likely to need this sort of functionality. If nothing else, it would enable digitally signing plug-ins and bundles for testing security functionality.



4. Provisioning

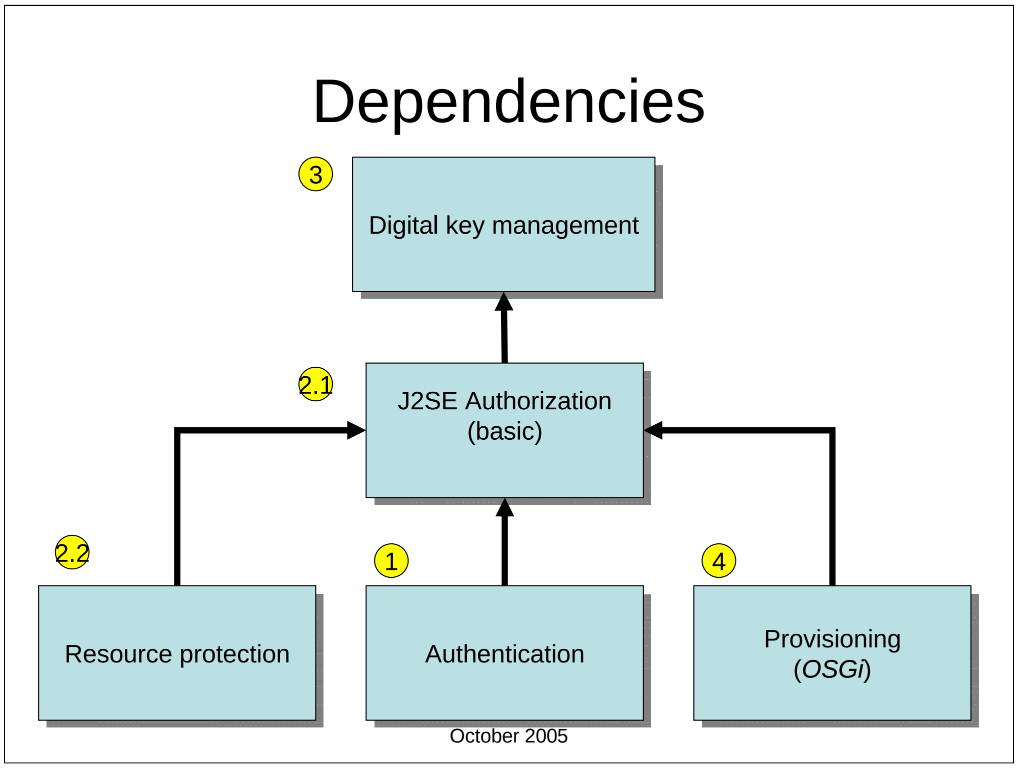
- OSGi provides provisioning functionality
- However,
  - Requires correct security policies to be generated and assigned to bundles
    - Default (AllPermission) is a violation of the *Principle of Least Privilege*
    - Conversely, too few privileges will prevent code from running (security exceptions)

Koved, Pistoia, Habeck
October 2005

Currently there is no way to easily / reliably identify the Java 2 permissions that are required for each Eclipse plug-in / OSGi bundle. The simplest solution is to give each plug-in / bundle *AllPermission*, but this is not a secure solution.

An alternative is to run test cases and observe the SecurityExceptions that are thrown and then add the appropriate Permissions to the security policy (e.g., to the appropriate OSGi bundle). However, generating sufficient test case coverage is hard and error prone.

We propose a solution to this issue later in this presentation.



Dependencies

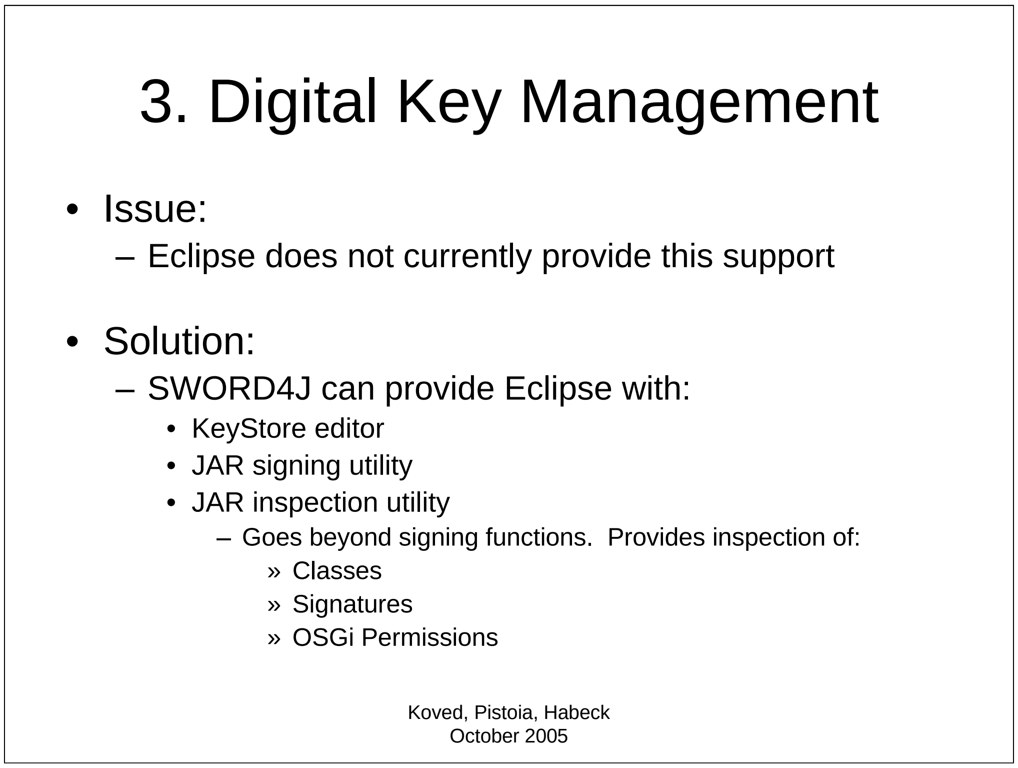
In this section of the presentation, we propose an approach to enabling security in Eclipse RCP.

Based on the five security areas previously described, we believe that the above figure represents the dependencies between the security issues outlined so far.

Digital key management is essential for any further work. As a result, we have a functioning prototype in SWORD4J.

Enabling the basic J2SE authorization framework is the next step since the remaining three areas of security functionality have authorization as a prerequisite. Substantial work has begun in this area.

Work on resource protection, authentication (including user authentication) and OSGi / Eclipse provisioning can then proceed in parallel.



3. Digital Key Management

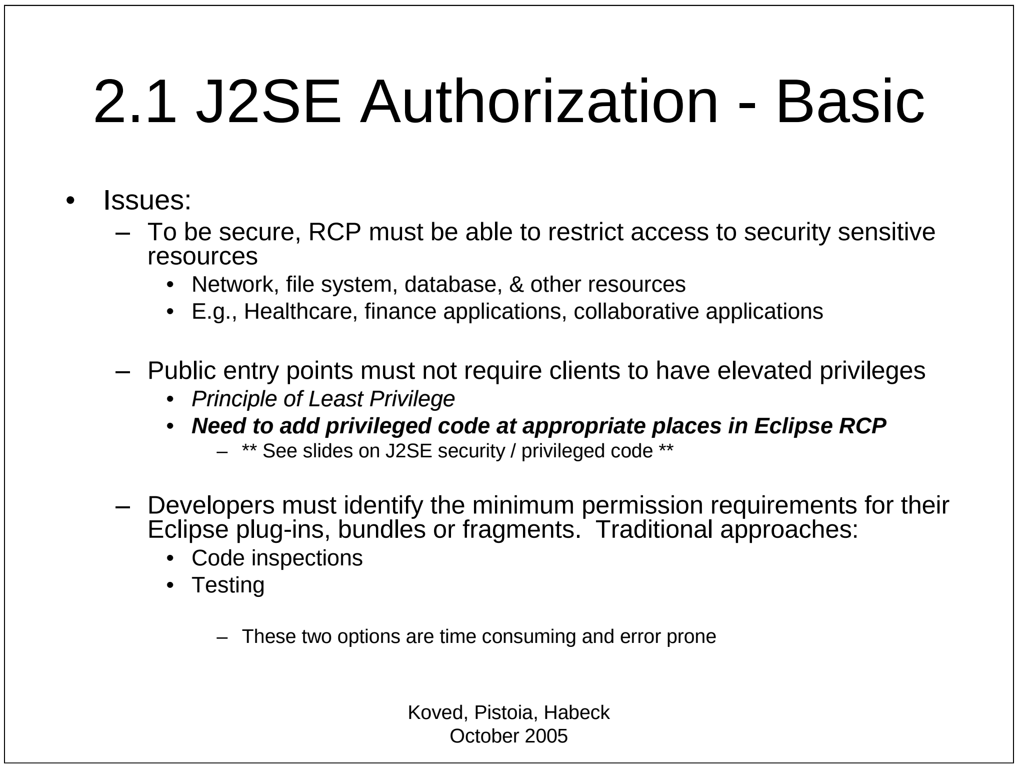
- Issue:
  - Eclipse does not currently provide this support
- Solution:
  - SWORD4J can provide Eclipse with:
    - KeyStore editor
    - JAR signing utility
    - JAR inspection utility
      - Goes beyond signing functions. Provides inspection of:
        - » Classes
        - » Signatures
        - » OSGi Permissions

Koved, Pistoia, Habeck
October 2005

Basic digital key management is need in Eclipse, especially for code signing. Code signing is useful for testing whether new security sensitive functionality is in fact working as intended. We see this functionality useful for developers and administrators who would like to verify digital signatures and inspect authorization requirements.

Note that in practice, we anticipate that code signing of code for distribution to customers will be performed by a build process outside of Eclipse (e.g., an ANT build). However, including the functionality in Eclipse makes it easier for development / testing.

The prototype we have included in SWORD4J provides the functionality outlined above. In particular, there is a JAR inspection tool to allow navigation through a JAR file. This includes inspection of the digital signatures and certificates.



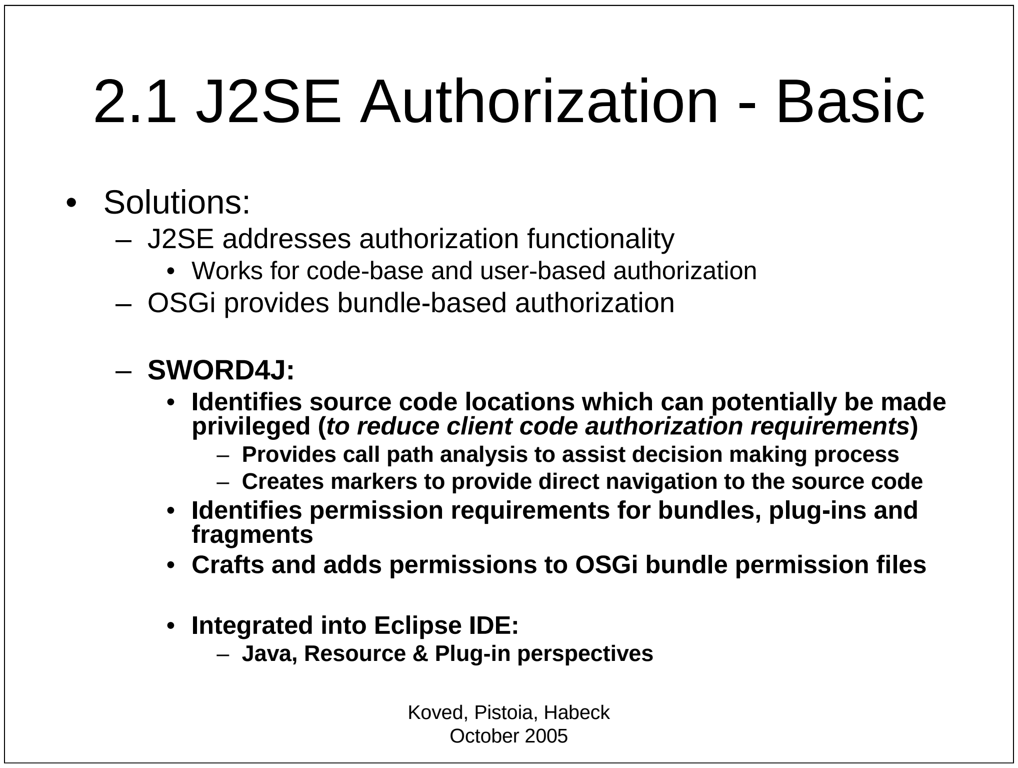
2.1 J2SE Authorization - Basic

- Issues:
  - To be secure, RCP must be able to restrict access to security sensitive resources
    - Network, file system, database, & other resources
    - E.g., Healthcare, finance applications, collaborative applications
  - Public entry points must not require clients to have elevated privileges
    - *Principle of Least Privilege*
    - ***Need to add privileged code at appropriate places in Eclipse RCP***
      - ** See slides on J2SE security / privileged code **
  - Developers must identify the minimum permission requirements for their Eclipse plug-ins, bundles or fragments. Traditional approaches:
    - Code inspections
    - Testing
      - These two options are time consuming and error prone

Koved, Pistoia, Habeck
October 2005

We observe that within Eclipse RCP there is a need to protect security sensitive resources as noted above.

Enabling of basic J2SE authorization can be time consuming. As a result, we created SWORD4J, a set of security analysis plugins, to make it easier for developers to enable and verify security properties of Eclipse plugins.



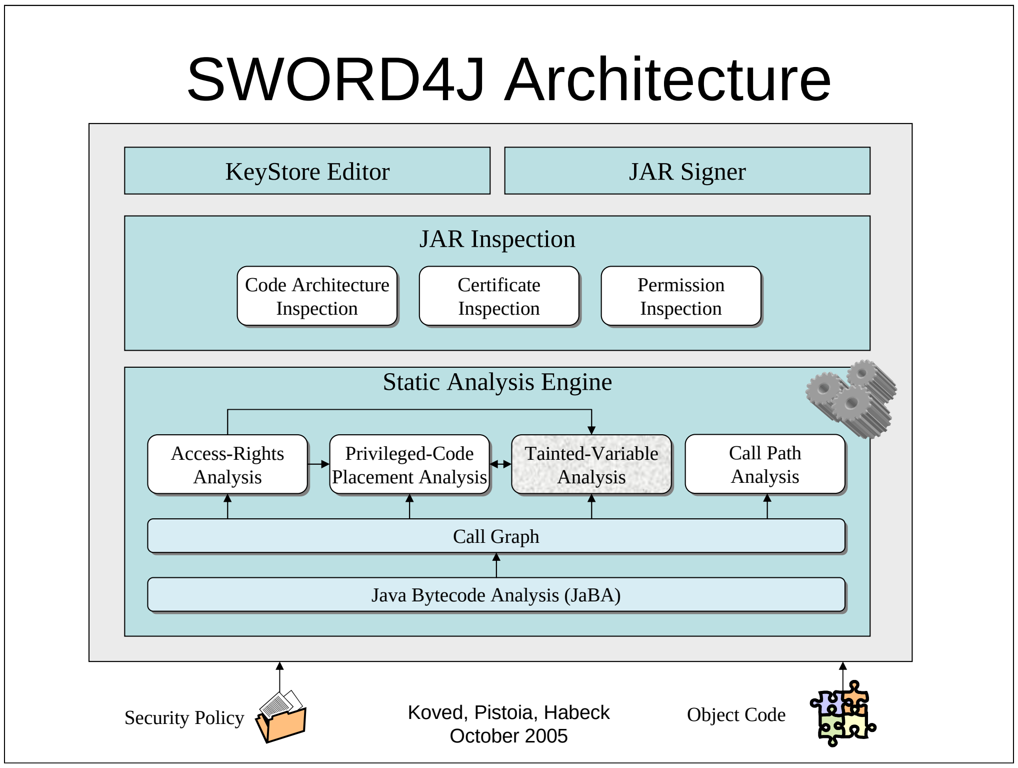
2.1 J2SE Authorization - Basic

- Solutions:
  - J2SE addresses authorization functionality
    - Works for code-base and user-based authorization
  - OSGi provides bundle-based authorization
  - **SWORD4J:**
    - **Identifies source code locations which can potentially be made privileged (*to reduce client code authorization requirements*)**
      - **Provides call path analysis to assist decision making process**
      - **Creates markers to provide direct navigation to the source code**
    - **Identifies permission requirements for bundles, plug-ins and fragments**
    - **Crafts and adds permissions to OSGi bundle permission files**
    - **Integrated into Eclipse IDE:**
      - **Java, Resource & Plug-in perspectives**

Koved, Pistoia, Habeck
October 2005

J2SE provides the base level of security authorization functionality.

SWORD4J is a set of Eclipse plugins to reduce the burden on developers for security enable their code. If necessary, we could make some of the SWORD4J functionality available outside the Eclipse environment.



SWORD4J Architecture

Koved, Pistoia, Habeck
October 2005

Architecture

•KeyStore editor – access to and manipulation of X.509 digital keys

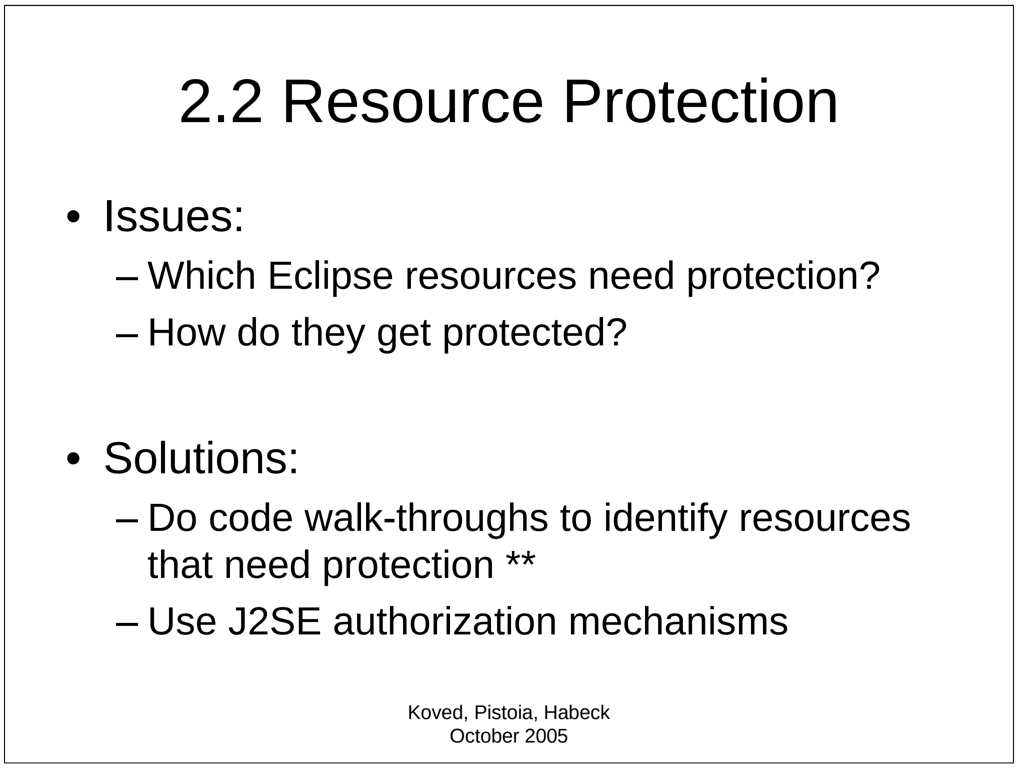
•JAR signer – enable signing of JAR files

•JAR inspector – view the

- •packages, classes, etc. in a JAR
- •digital certificates used to sign a JAR file.
  - •An updated to SWORD4J will show whether the certificate has expired,
  - •Also, whether there has been tampering with the contents since signing.
- •J2SE / OSGi permissions assigned to the OSGi bundle

•Analysis of Java code to determine various security properties. In the initial release of SWORD4J, this includes:

- •J2SE / OSGi permission requirements
- •Locations where privileged code may need to be added



2.2 Resource Protection

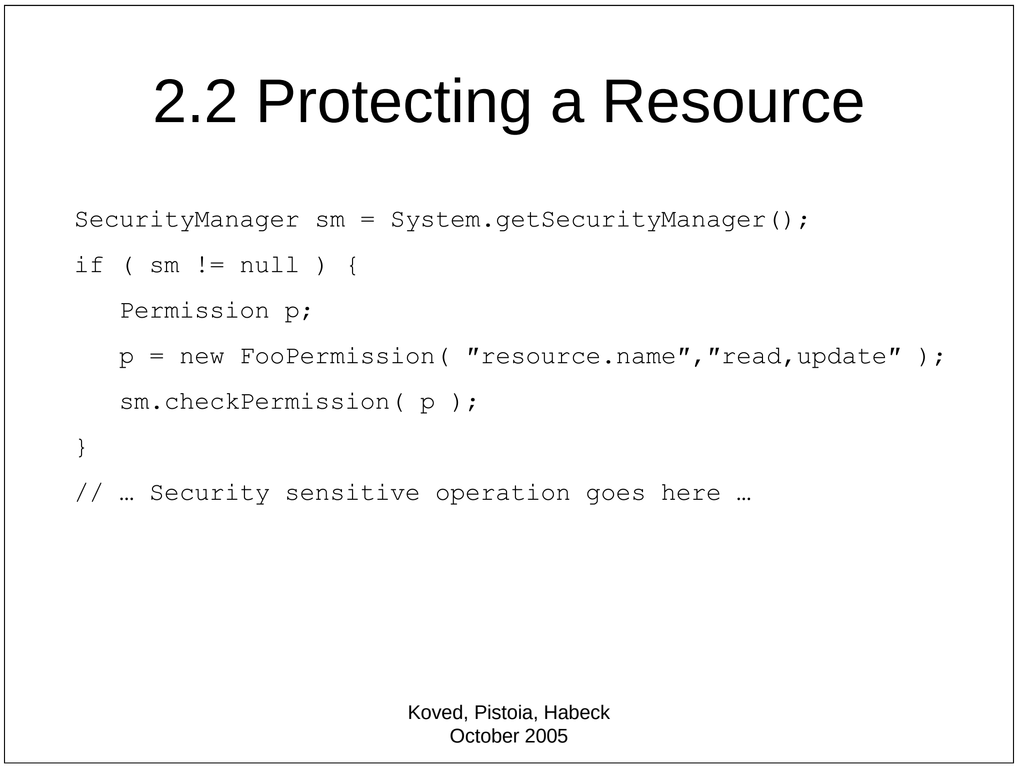
- Issues:
  - Which Eclipse resources need protection?
  - How do they get protected?
- Solutions:
  - Do code walk-throughs to identify resources that need protection **
  - Use J2SE authorization mechanisms

Koved, Pistoia, Habeck
October 2005

Some data and operations defined by Eclipse plugins have security implications. These need to be protected using the technique described on the next page.

We may be able to create a tool which will give a first order approximation resources needing protection. We hope to add that to SWORD4J in an upcoming release.

However, it will still be necessary to perform walkthroughs of the code (e.g., the API's) to identify additional security vulnerabilities.

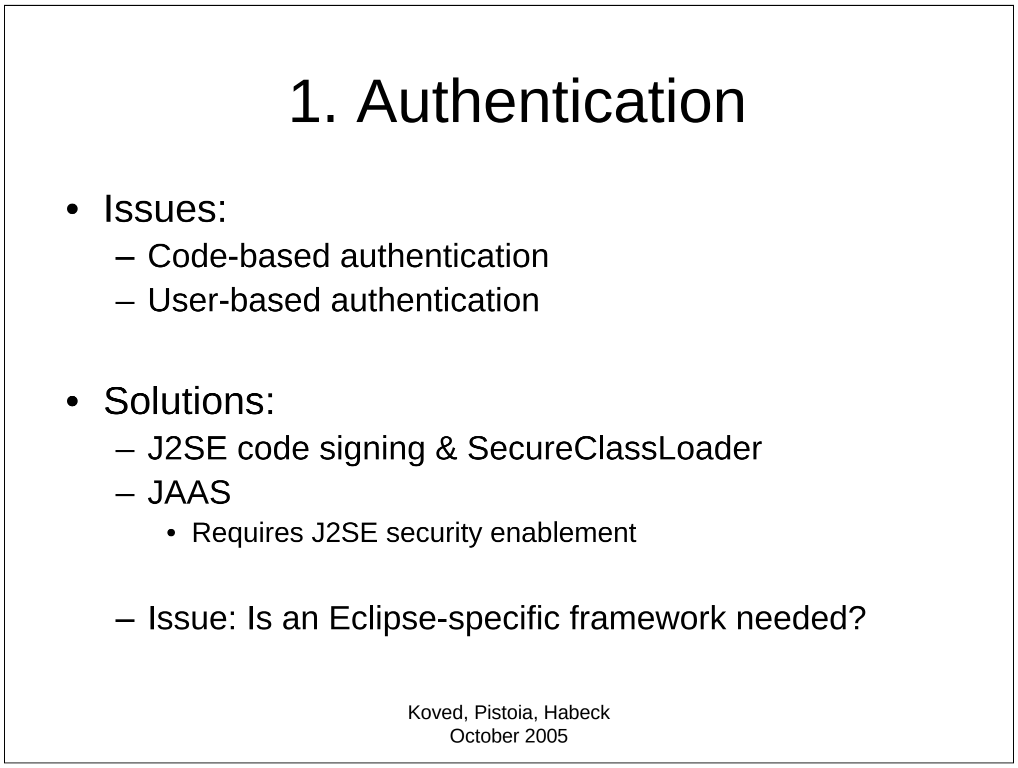


2.2 Protecting a Resource

```
SecurityManager sm = System.getSecurityManager();
if ( sm != null ) {
   Permission p;
   p = new FooPermission( "resource.name","read,update" );
   sm.checkPermission( p );
}
// … Security sensitive operation goes here …
```

Koved, Pistoia, Habeck
October 2005

1. First see if a security manager is installed,
2. A Permission object *p* is created to describe the protected resource (*resource.name*), and the operation(s) on the resource (e.g., *read, update*).
3. The SecurityManager.checkPermission() is called with *p* and the Java authorization system logic checks to see if the code/user is authorized for the protected resource / operation(s), as previously described.



1. Authentication

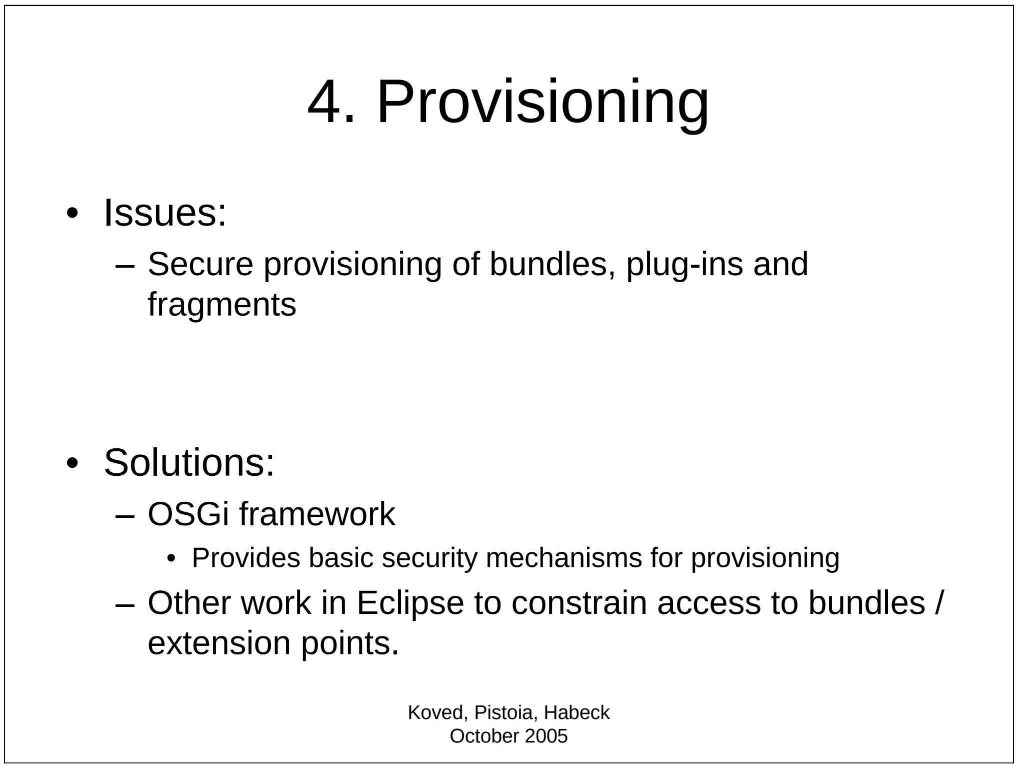
- Issues:
  - Code-based authentication
  - User-based authentication
- Solutions:
  - J2SE code signing & SecureClassLoader
  - JAAS
    - Requires J2SE security enablement
  - Issue: Is an Eclipse-specific framework needed?

Koved, Pistoia, Habeck
October 2005

There is a need for both code-based and user-based authentication. Code-based authorization for enabling security sandboxing. User authentication for access to local and remote services.

J2SE provides standard mechanisms to support both of these. These APIs have industry support.

For Eclipse projects, including ECF and Higgins, what additional identity management frameworks are needed? There is ongoing discussion with these projects, among others, on how to best satisfy these requirements.



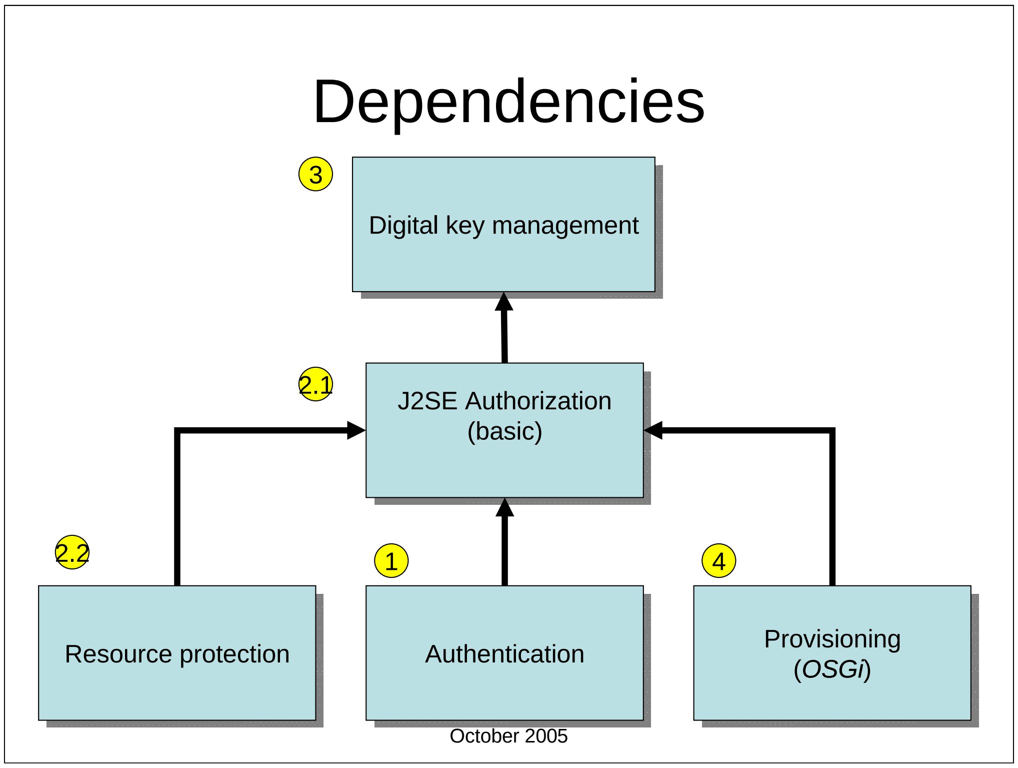
4. Provisioning

- Issues:
  - Secure provisioning of bundles, plug-ins and fragments
- Solutions:
  - OSGi framework
    - Provides basic security mechanisms for provisioning
  - Other work in Eclipse to constrain access to bundles / extension points.

This is an area for ongoing discussions and exploration on the best way to protect access to code/function, etc.

Outline – What Needs To Be Done

Koved, Pistoia, Habeck
October 2005



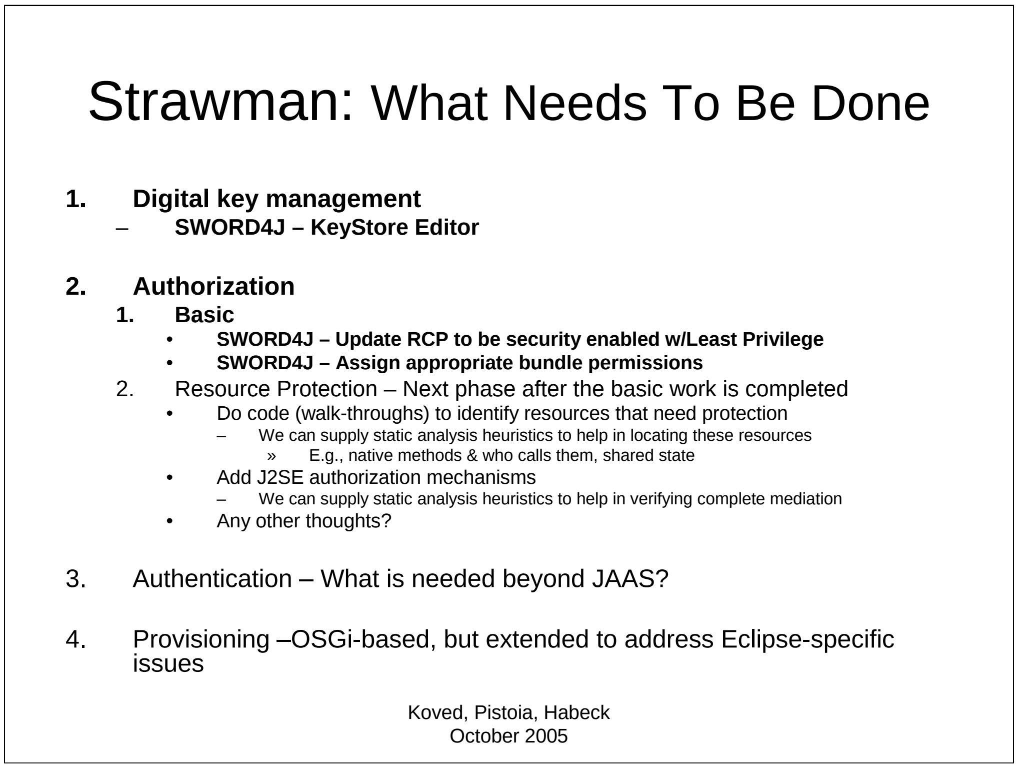
Dependencies

(Same as slide 15)

This is the order in which we propose attacking security.

Notably, some of the work on security can proceed in parallel.

The following slides will describe how we propose to proceed.



Strawman: What Needs To Be Done

1. **Digital key management**
   - **SWORD4J – KeyStore Editor**

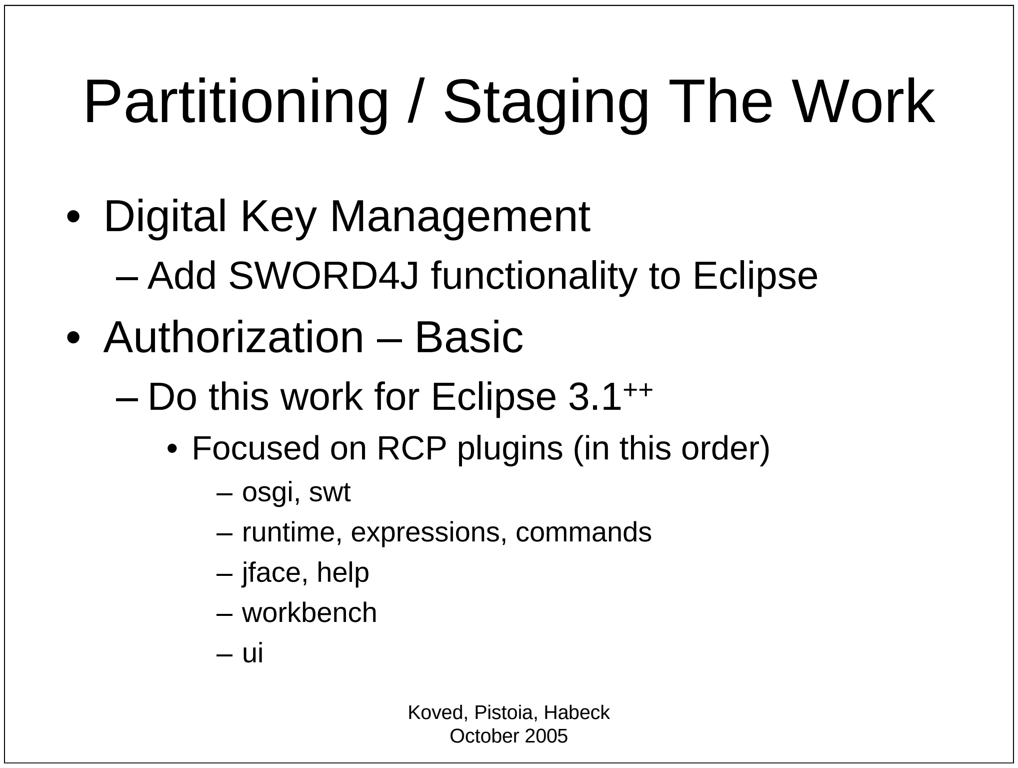
2. **Authorization**
   1. **Basic**
      - **SWORD4J – Update RCP to be security enabled w/Least Privilege**
      - **SWORD4J – Assign appropriate bundle permissions**
   2. Resource Protection – Next phase after the basic work is completed
      - Do code (walk-throughs) to identify resources that need protection
        - We can supply static analysis heuristics to help in locating these resources
          - » E.g., native methods & who calls them, shared state
      - Add J2SE authorization mechanisms
        - We can supply static analysis heuristics to help in verifying complete mediation
      - Any other thoughts?

3. Authentication – What is needed beyond JAAS?

4. Provisioning –OSGi-based, but extended to address Eclipse-specific issues

Koved, Pistoia, Habeck
October 2005

These are the four basic areas we have outlined and will be discussed in the following slides

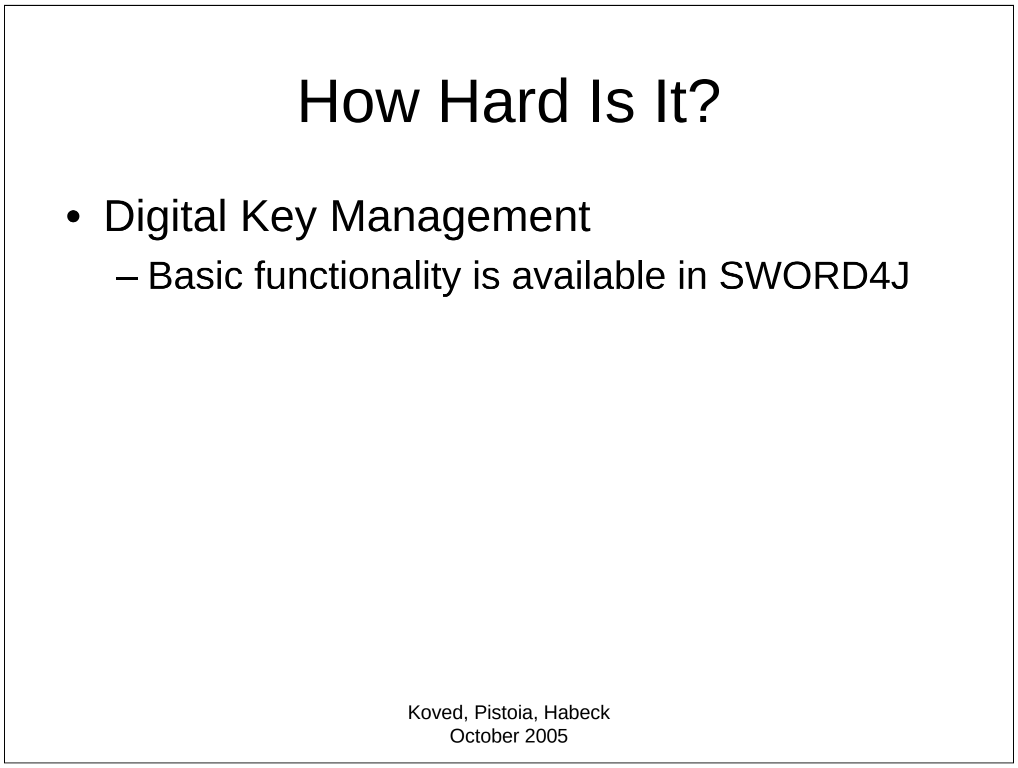


Partitioning / Staging The Work

- Digital Key Management
  - Add SWORD4J functionality to Eclipse
- Authorization – Basic
  - Do this work for Eclipse 3.1^{++}
    - Focused on RCP plugins (in this order)
      - osgi, swt
      - runtime, expressions, commands
      - jface, help
      - workbench
      - ui

Koved, Pistoia, Habeck
October 2005

We see that digital key management and basic authorization enablement are key to getting started.



How Hard Is It?

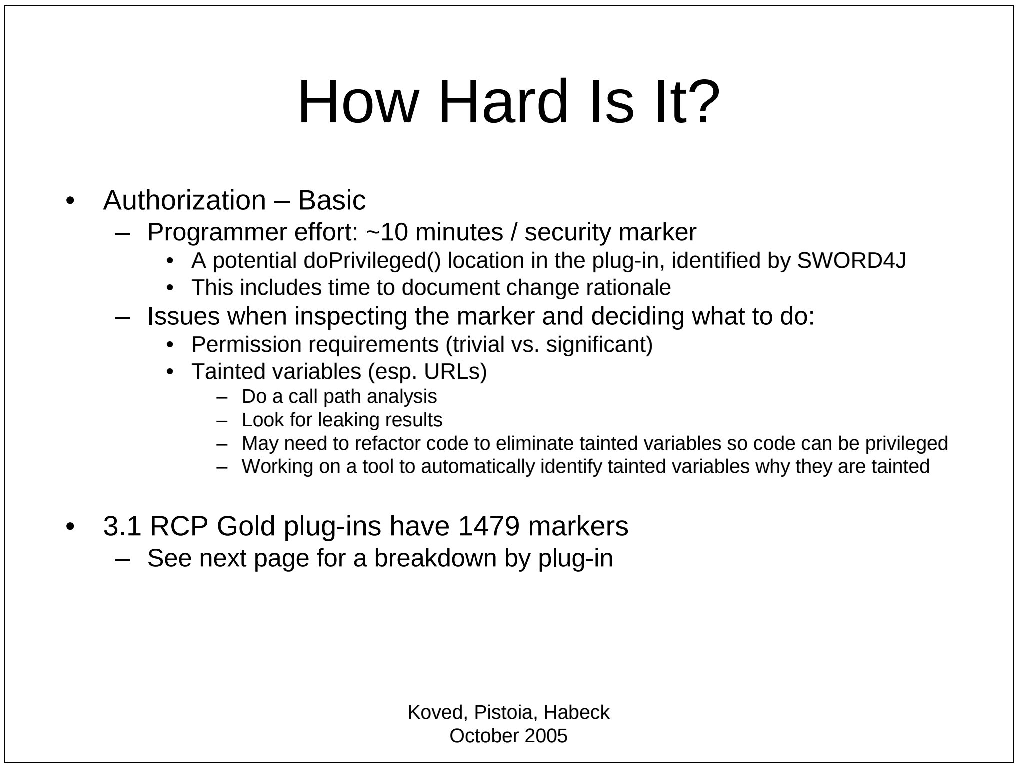
- Digital Key Management
  - Basic functionality is available in SWORD4J

Koved, Pistoia, Habeck
October 2005

We are open to suggestions on how to improve the functionality in the current prototype.

Demo of SWORD4J

Koved, Pistoia, Habeck
October 2005



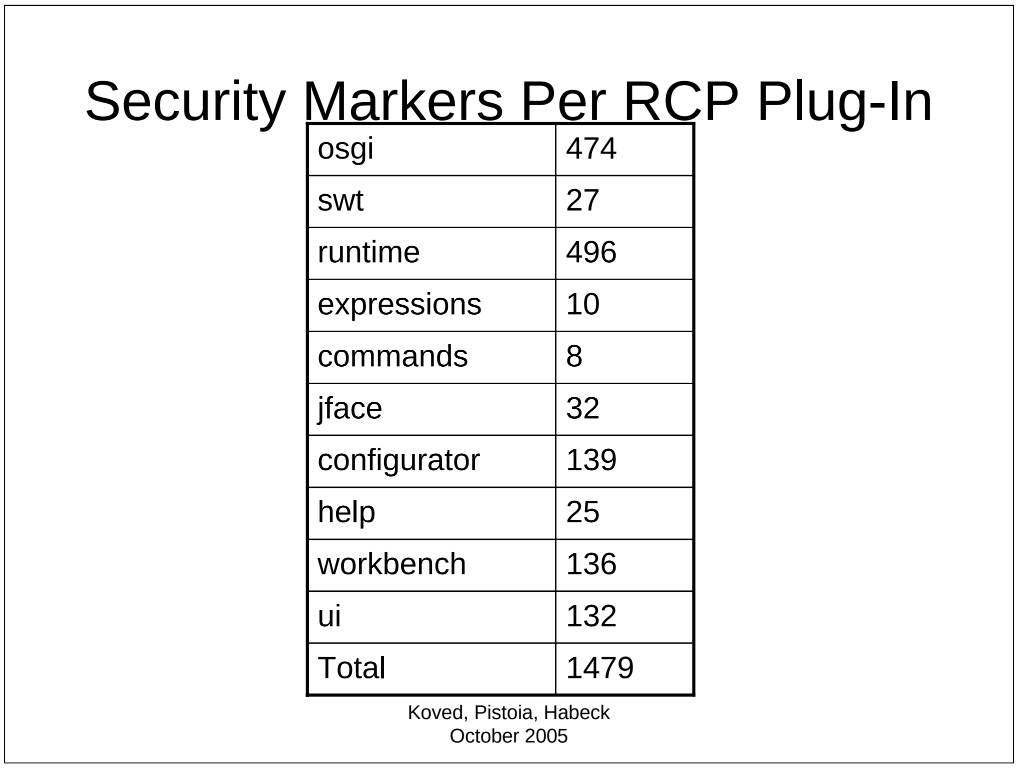
How Hard Is It?

- Authorization – Basic
  - Programmer effort: ~10 minutes / security marker
    - A potential doPrivileged() location in the plug-in, identified by SWORD4J
    - This includes time to document change rationale
  - Issues when inspecting the marker and deciding what to do:
    - Permission requirements (trivial vs. significant)
    - Tainted variables (esp. URLs)
      - Do a call path analysis
      - Look for leaking results
      - May need to refactor code to eliminate tainted variables so code can be privileged
      - Working on a tool to automatically identify tainted variables why they are tainted
- 3.1 RCP Gold plug-ins have 1479 markers
  - See next page for a breakdown by plug-in

Koved, Pistoia, Habeck
October 2005

We have focused on how to minimize the amount of time needed to security enable plugins / bundles. Using the analysis features in SWORD4J, it is possible to identify the authorization requirements (permissions) needed by each plug-in, fragment, etc.

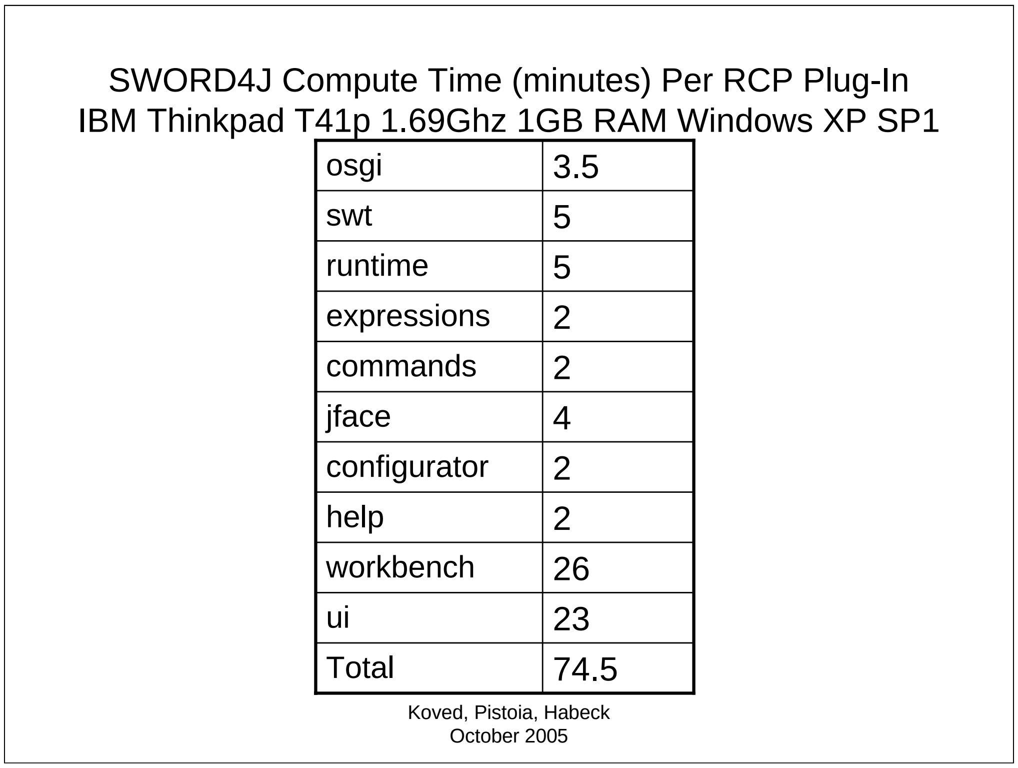
We have reduce the time to analyze each security marker associated with permissions so that it should take 10 minutes, or less, on average to decide how to address the security issues raised by the marker.



Security Markers Per RCP Plug-In

osgi	474
swt	27
runtime	496
expressions	10
commands	8
jface	32
configurator	139
help	25
workbench	136
ui	132
Total	1479

This table gives an estimate of the number of security markers per plug-in.



SWORD4J Compute Time (minutes) Per RCP Plug-In
IBM Thinkpad T41p 1.69Ghz 1GB RAM Windows XP SP1

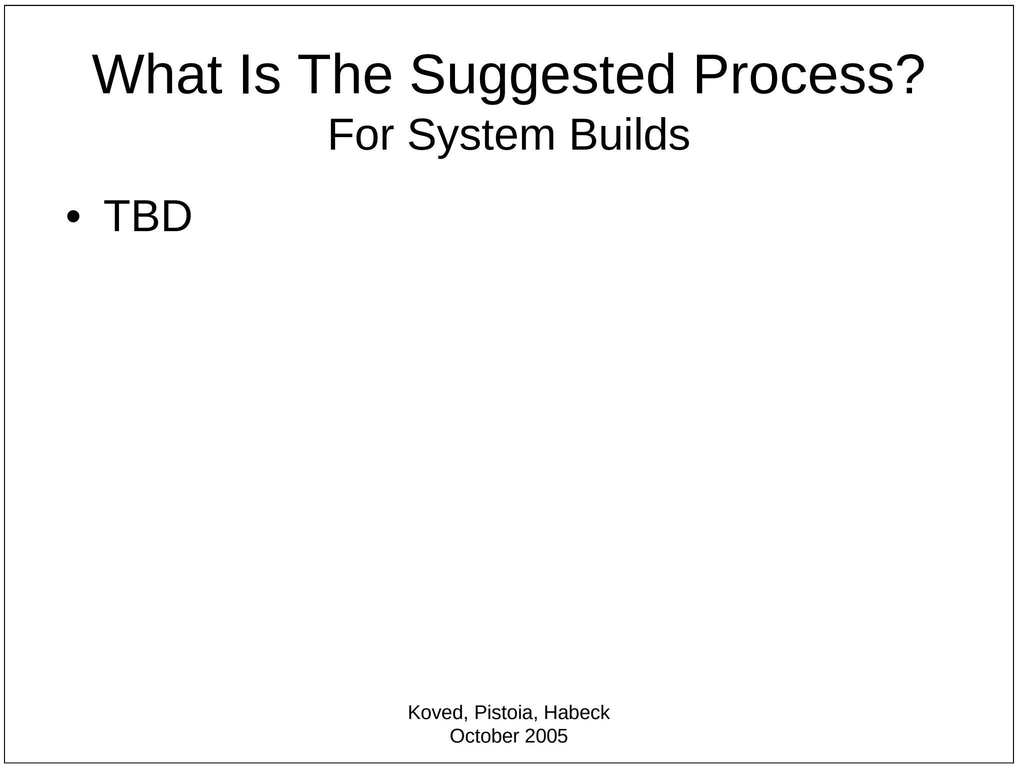
osgi	3.5
swt	5
runtime	5
expressions	2
commands	2
jface	4
configurator	2
help	2
workbench	26
ui	23
Total	74.5

Koved, Pistoia, Habeck
October 2005

This table gives a summary of the amount of processing time (in minutes) to analyze a plug-in (with its dependent plug-ins and the Java runtime) to get the security markers described on the previous page.

What Is The Suggested Process?
For Plug-In Developers

- Analyze plug-in using SWORD4J
- “Process” the generated security markers
  - As appropriate:
    - Refactor code
    - Make trusted libraries / method / code blocks
    - Grant Permissions by adding them to the bundle
    - Add annotation so security marker(s) are surpressed -- TBD
- Build plug-in / bundle & test with security turned on
- Check code into CVS



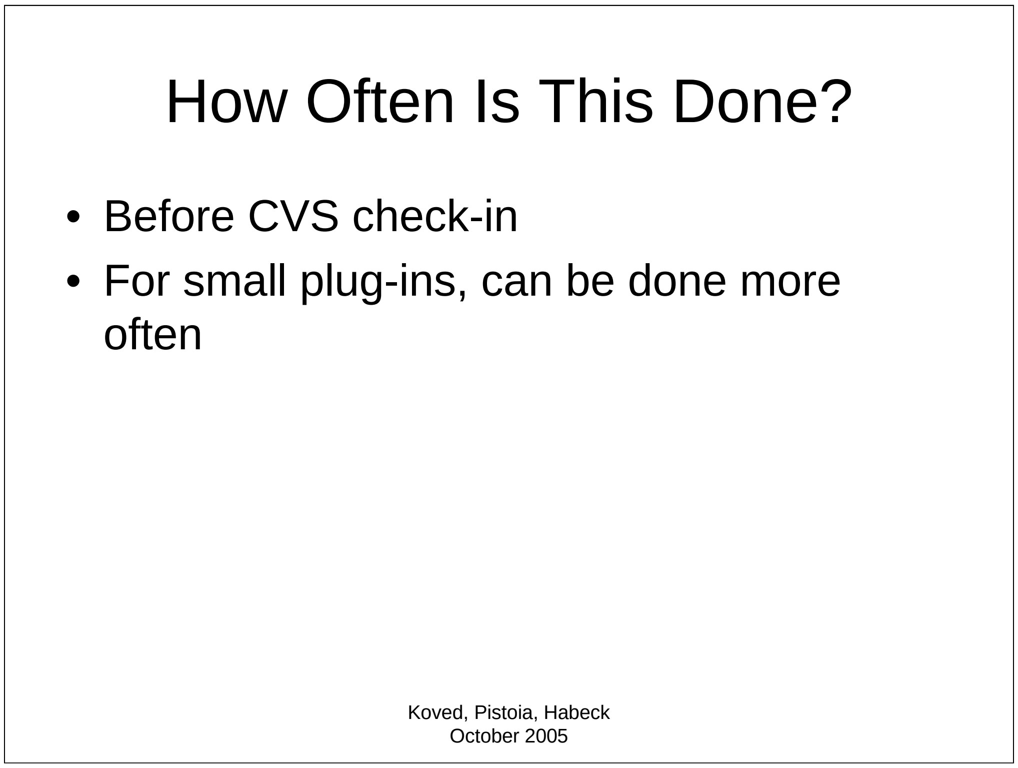
What Is The Suggested Process?

For System Builds

- TBD

Koved, Pistoia, Habeck
October 2005

We should be able to decouple the security analyses from the Eclipse GUI to generate HTML and/or text descriptions of the security results. (Before the analyses were built into Eclipse, we had run the analysis tools in batch mode).



How Often Is This Done?

- Before CVS check-in
- For small plug-ins, can be done more often

Koved, Pistoia, Habeck
October 2005

During plug-in development, we would expect people to use SWORD4J regularly during development. The frequency of running the security analyses depends on the programmer's development style. As a minimum, we would expect that SWORD4J be run prior to checking code into CVS.