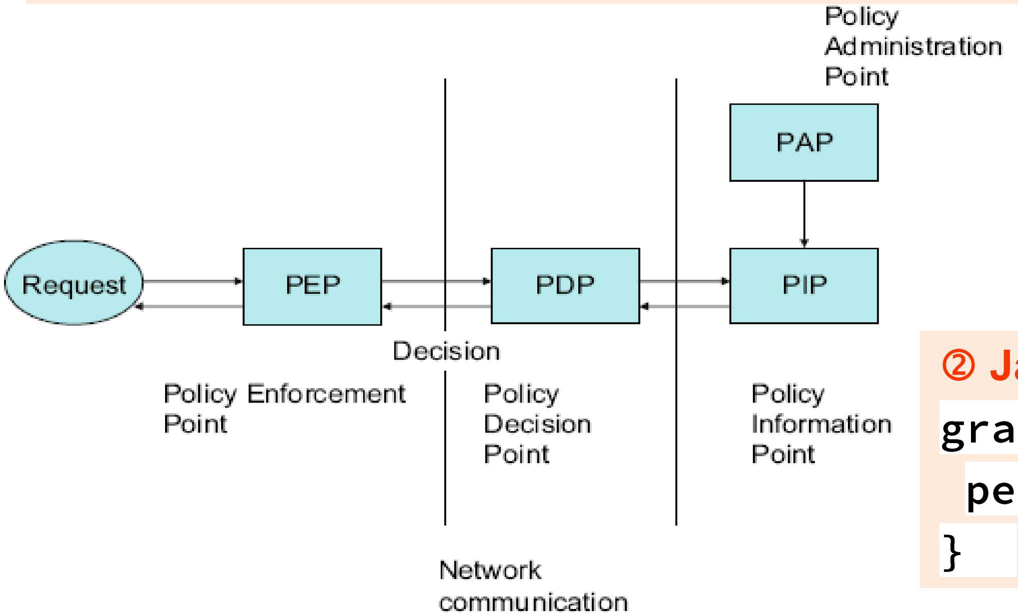
Fachhochschule Nordwestschweiz
Hochschule für Technik

Java idiosyncrasies and Java patterns that break robustness

Carlo U. Nicola, IMVS FHNW
With extracts from slides/publications of :
Andreas Sterbenz and Charlie Lai, Sun Microsystems.

Security Model: `OpenXML` and Java

④ Java PEP:
`SecurityManager, Permission, AccessController, AccessControlContext, ProtectionDomain` management.



① Java PAP → File: `java.security`

② Java PIP → File: `java.policy`

```
grant codeBase …{
 permission.java.security.AllPermissions;
}
```

③ Java PDP:

```
if (sm != null){
 context = sm.getSecurityContext();
 Filepermission p = new Fileprmission(filename, “read”);
 sm.checkPermission(p,context);
}
```

Visibility modifiers in Java

Visibility of **interfaces** and **classes** can be:

- $\rightarrow$ `public`: open to all and sundry: All parts of the application can use it
- $\rightarrow$ no modifier: open only within a package: Can be used by name only in the same package
- $\rightarrow$ special problem $\rightarrow$ inner classes

Visibility of **fields** and **methods** of a class can be:

- $\rightarrow$ `public` : All parts of the application can use it
- $\rightarrow$ `protected` : Can be used only in the same package and from subclasses in other packages
- $\rightarrow$ no modifier : Can be used only in the same package (default)
- $\rightarrow$ `private` : Can be used only in the same class

What is the purpose of visibility modifiers?

1. Information hiding: The visibility of a class (interface, field, method) should be as restricted as possible. A good mantra: Keep all the member variables private and let only those methods as public that are important for the API.
2. Compartmentalization of programs' components: Clear divide between the things you should know (`public`) and the things you can safely ignore (`private`).

Does visibility of program components affect robustness? Yes, because:

You may think that a secret may be embedded in an object in its private field. It is not a very smart idea, but it is done often.

You may think that untrusted code can be prevented from executing sensitive code by simply placing it in a package-local class. Here too, it is not a very smart idea, but it is done often.

An example of the way visibility affects robustness

```
public class Auction {
  public List bids;
  public Auction() {
    this.bids = new ArrayList();
}
public void bid(int amount) {
  if (!this.bids.isEmpty()) {
    int lastBid = ((Integer)this.bids.get(this.bids.size() - 1)).intValue();
    if (lastBid >= amount) {
      throw new RuntimeException("Bids must be higher than previous ones");
          }
  }
  this.bids.add(new Integer(amount));
}
public int getHighestBid() {
  if (this.bids.isEmpty()) {return 0;}
    return ((Integer) this.bids.get(this.bids.size() - 1)).intValue();
  }
}
```

Harmless and “robust” if a client does the following:

```
…
auction.bid(newBid);        …
auction.getHighestBid();    …
```

But a big problem if a client tries the following:

```
…
auction.bids.add(newBid);    …
auction.bids.get(i);         …
```

Changing visibility of fields when sub classing

Java allows increasing (but not decreasing) the visibility of a member. The situations where this is desirable are very rare and in any case dangerous because in so doing, one breaks the API contract.

```
public class A {
        protected void m() {…}
        …
}

public class B extends A {
        public void m() {…}
        …
}
```

← B increase visibility of `m()`

Why protected and default visibility?

In principle `public` and `private` are sufficient to warrant a clean API definition. So, what can `protected` and `default` visibility add to the program's robustness?

Not much, because:

→ A malicious client can have a class that claims to have the same package;

→ A malicious client can have a class that extends a trusted class and accesses *protected* fields.

But it would be nice to have a simple method to prohibit *packages from being added* to an application. Unfortunately, Java does not offer any easy and robust way to do this.

Ways to protect Java packages from getting joined

Method 1: Sealed JAR archives

Can add a simple entry to the manifest of a JAR file, saying that packages (all or some) in this JAR cannot be joined (i.e. sealing a package means that all classes defined in that package must be found in the same JAR file):

```
Name: ch/fhnw/securepackage/
Sealed: true
```

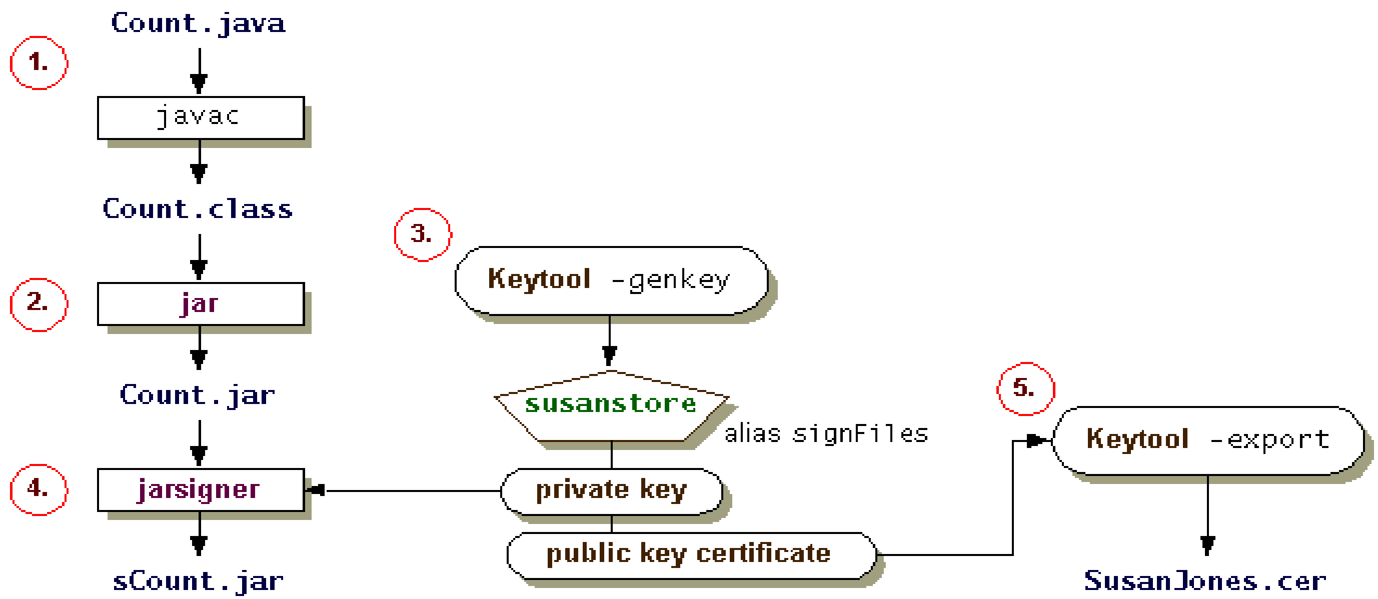
The JAR file should contain the complete package if this package is sealed. You seal a package in a JAR file by adding the sealed header in the manifest, which has the general form:

```
Name: myCompany/myPackage/ Sealed: true
```

The value `myCompany/myPackage/` is the name of the package to seal.
Note that the package name must end with a "/".
Sealing process is independent of the `SecurityManager`.

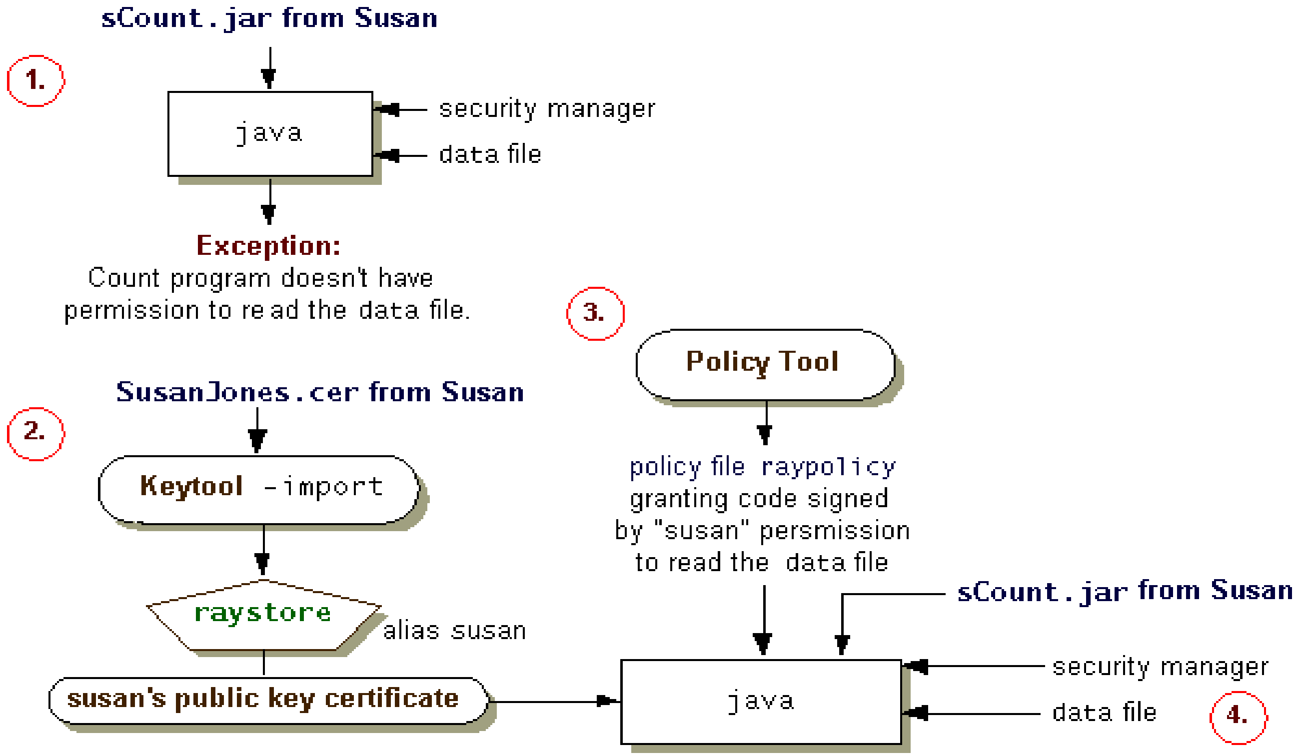
→ Does not protect much if the JAR is given to clients: you can always manipulate the manifest file in `*.jar`!

Method 2: Signing JAR archives (sending point)




```
jarsigner -keystore susanstore -signedjar sCount.jar Count.jar signFiles
keytool -export -keystore susanstore -alias signFiles -file
    SusanJones.cer
```

Method 2 Signing JAR archives (receiving end)




```
keytool -import -alias susan -file SusanJones.cer -keystore raystore
```

Verify fingerprint (optional); and add to `raypolicy` file (PIP):

```
grant CodeBase "file:./….." SignedBy "Susan" {
    permission java.io.FilePermission "C:\\TestData\\*", "read";
}
java -Djava.security.manager -Djava.security.policy=raypolicy -cp sCount.jar Count
    C:\TestData\data
```

Method 3: Join packages via security policy (PAP)

Can insert in the `java.security` file:

`package.definition=com.ibneco.securepackage`

Need additional assignments of permissions to join this package to classes that are allowed to.

A big gotcha (2013): no class loaders from Sun support this at present! (See remark in `java.security` below)

```
#
# List of comma-separated packages that start with or equal this string
# will cause a security exception to be thrown when# passed to
# checkPackageDefinition unless the
# corresponding RuntimePermission ("defineClassInPackage."+package) has
# been granted.
#
# by default, none of the class loaders supplied with the JDK call
# checkPackageDefinition.
#
```

Java inner classes

Why does Java use inner classes?

Why inner classes in Java?

1. Classes defined as members of other classes.
2. Inner classes are allowed to access private members of the enclosing class and vice versa.
3. For each instance of the outer class there is a corresponding instance of the inner class.
4. Useful especially for defining in-line implementations of simple interfaces.

```
class A {
  private int a;
  class B {
    private int b;
    private void f() {
    b = a*2;
    }
  }

  public g() {
    B bObj = new B();
    bObj.f();
    bObj.b = 2;
  }
}
```

B accesses a private field of A

A accesses a private method of B

Inner class are not understood by JVM (1)

Unfortunately, no notion of inner classes exists during run-time.

Java compilers just transform inner classes into top-level classes:

- $\rightarrow$ But JVM prohibits access to private members from outside the class!
- $\rightarrow$ So the compiler provides access to private fields accessed by inner (outer) classes via package-local methods

Inner class are not understood by JVM (2)

```
class A {
  private int m;
  private class B {
    private int x;
    void f() {
      x = m;
    }
  }
  public void g() {
    B ob = new B();
    ob.f();
  }
}
```

After compilation:

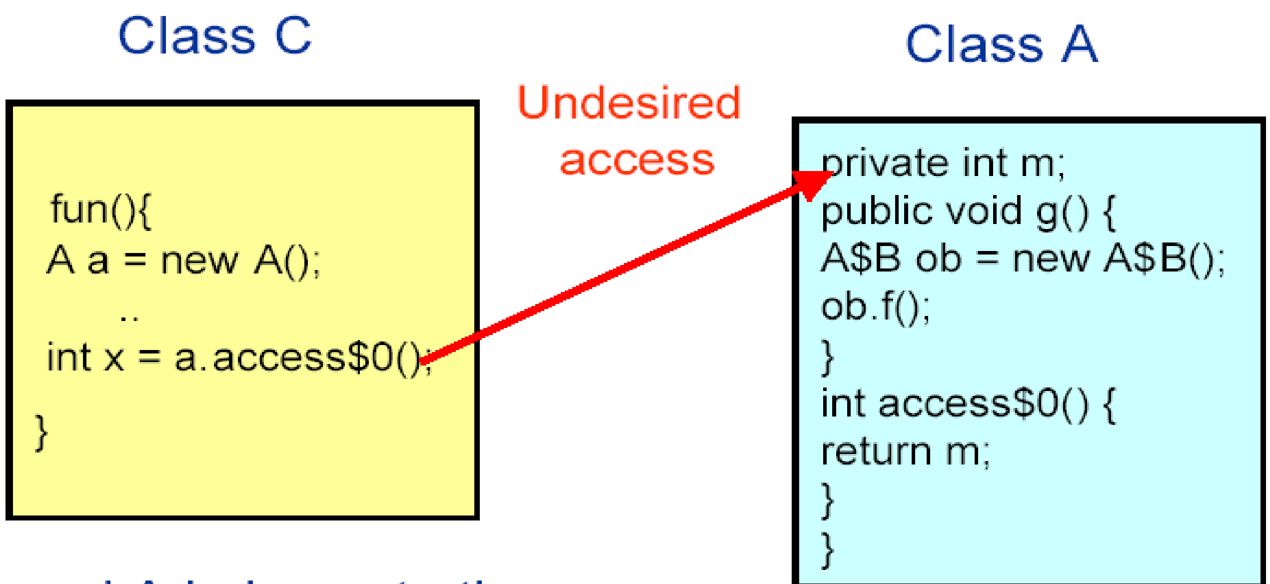
```
class A {
  private int m;
  public void g() {
    A$B ob = new A$B();
    ob.f();
  }
  int access$0(){
    return m;
  }
}
```

```
class A$B {
  A this$0;
  private int x;
  void f() {
    x = this$0.access$0();
  }
}
```

1. `access$0()` of class A has package level visibility.
2. The class **`A$B`** also has package level visibility

Why are inner class a security risk?

1. The private data members of classes get exposed through the access functions
2. Other classes belonging to the same package can call the access functions and tamper the **private data** member



- Class C and A belongs to the same package

William Pough's proposal

Goal: Only to the inner classes may access the private members of the enclosing class.

A *secret key* is shared between all the classes that need access to each others private data members:

$\rightarrow$ Class *B* wants to access a class *A* private member *m*

$\rightarrow$ Class B invokes *A*'s access function

$\rightarrow$ Class *B* passes its shared *secret key* to *A*'s access function

$\rightarrow$ Class *A* verifies whether class *B*'s *secret key* and class *A*'s *secret key* are the same object:

- if yes, give access to its private variable m
- otherwise, throw a security exception

The new implementation is built on top of the current implementation:

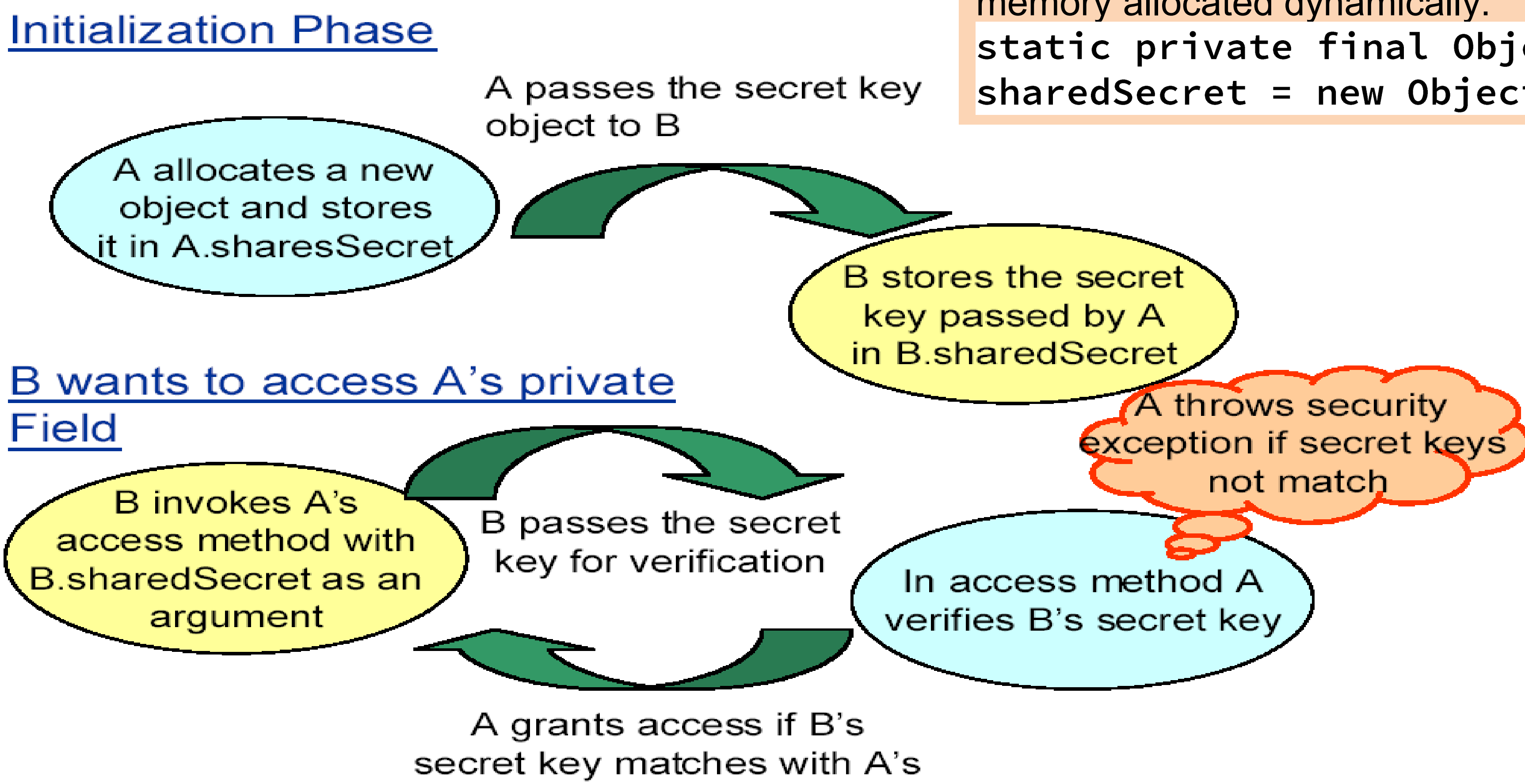
$\rightarrow$ only class files are rewritten

$\rightarrow$ No need to change the JVM

Pough's solution: idea

The secret key value is a pointer to memory allocated dynamically:

```
static private final Object
sharedSecret = new Object();
```



Pough's solution: implementation

```
class A {
  static private final Object sharedSecret = new Object();
  static { A$B.receiveSecretForA(sharedSecret); }
  private int x;
  int access$1(Object secretForA) {
    if (secretForA != sharedSecret) throw new SecurityException();
      return x;
  }
}
class A$B {
  private A this$0;
  static private Object sharedSecret;
  static void receiveSecretForA(Object secretKey) {
    if (sharedSecret != null) throw new VerifyError();
      sharedSecret = secretKey;
  }
 ... invoke this$0.access$1(sharedSecret)...
}
```

Overhead of Plough's solution

For each class allowing/needing access, one needs a single static field.
For each set of objects needing mutual access only one object is created.

a) All initializations are done in a `static` initializer.
b) One additional argument in each `access$` method
c) Few additional instructions are executed for each access call to:
    - $\rightarrow$ pass the extra argument
    - $\rightarrow$ verify the secret key

Patterns to avoid in robust programs

Antipattern: a definition

[Brown *et al.,*1998]:

An **Antipattern** is a solution to a common programming problem where the negative consequences of the solution exceed the benefits of the repetitive pattern.

Common Java Antipatterns

1. Assuming objects are immutable
2. Basing security checks on untrusted sources
3. False inheritance relationship
4. Ignoring changes to super classes
5. Neglecting to validate inputs
6. Misusing public static variables
7. Believing a constructor exception destroys the object (up to Java1.6)
8. TOC2TOU (*Time Of Check To Time Of Use*)

Antipattern 1

An example from JDK 1.1 distribution:

```
package java.lang;
  public class Class {
  private Object[] signers;
  public Object[] getSigners() {
    return signers;
  }
}
```

Note: `Class.getSigners()` is actually implemented as a native method, but the behaviour is equivalent to the above.

See: `http://java.sun.com/security/getSigners.html`

Antipattern 1: Misuse

```
package java.lang;
  public class Class {
    private Object[] signers;
    public Object[] getSigners() {
      return signers;
   }
}
```

An attacker can manipulate the `signers` array so:

```
Object[] signers = this.getClass().getSigners();
signers[0] = <new signer>;
```

Antipattern 1: Problems

We quickly forget that mutable input and output objects can be modified by the caller application.

The consequences are not always harmless:

1. Modifications in general may change the original expected behaviour of applications (breach of contract $\rightarrow$ attack on robustness).
2. But modifications to sensitive security state are even worst: they may result in elevated privileges for attacker.

In our example this means that altering the signers of a class can give the class access to unauthorised resources.

Antipattern 1: Solutions

Make a copy of mutable **output** parameters:

```
public Object[] getSigners() {
    // signers contains immutable type X509Certificate.
    // shallow copy of array is OK.
    return signers.clone();
}
```

Make a copy of mutable **input** parameters:

```
public MyClass(Date start, boolean[] flags) {
      this.start = new Date(start.getTime());
      this.flags = flags.clone();
}
```

Perform deep cloning on arrays if necessary, because `clone()` on an array produces a shallow copy of it.

Antipattern 2

An example from the Java SDK 1.5 distribution:

```
public RandomAccessFile openFile(final java.io.File f){

  askUserPermission(f.getPath());
  ...
  return (RandomAccessFile) AccessController.doPrivileged(){
    public Object run(){
      return new RandomAccessFile(f.getPath());
    }
  }
}
```

Antipattern 2: Misuse

An attacker can pass a subclass of **`java.io.File`** that overrides **`getPath()`**:

```
public RandomAccessFile openFile(final java.io.File f) {
  askUserPermission(f.getPath());
  ...
  return new RandomAccessFile(f.getPath());
  ...
}

public class BadFile extends java.io.File {
  private int count;
  public String getPath(){
    return (++count == 1) ? "/tmp/foo" : "/etc/passwd";
  }
}
```

Antipattern 2 : Problems

Do not forget that security checks can be always fooled if they are based on information that attackers can control!

→ It is naive to assume that input types defined in the Java core libraries (like `java.io.File`) are secure and can be trusted
  → Such libraries are the second gate to break programs' robustness!

→ Always remember that non-final classes/methods can be sub classed.

→ Always remember that mutable types can be modified.

Antipattern 2 : Solutions

1. Don’t assume inputs are immutable: make defensive copies of non final or mutable inputs and perform checks when using copies.

```
public RandomAccessFile openFile(File f) {
  final File copy = f.clone();
  askUserPermission(copy.getPath());
  ...
  return new RandomAccessFile(copy.getPath());
}
```

   But this solution is unfortunately wrong: the method **`clone()`** copies the attacker’s subclass!

2. A more correct solution is: make a copy of the **original** file. This however doesn't guarantee that the file is exactly the original :

```
java.io.File copy = new java.io.File(f.getPath());
```

Antipattern 2 : A correct solution

A correct (albeit very restrictive) solution: **never** invoke **doPrivileged()** with caller-provided inputs :

```
import java.io.*;
import java.security.*;

private static final String FILE = "/myfile";

public RandomAccessFile openFile() {
  return(RandomAccessFile) AccessController.doPrivileged(
                                   new PrivilegedAction() {
    public Object run() {
        return new RandomAccessFile(FILE);
        // checked by SecurityManager
     }
  });
}
```

Antipattern 3

An classical Java blunder: class **`Stack`** inherits from class **`Vector`**

```
class Stack<E> extends Vector<E>
```

The **`Vector<E>`** functionality should (if necessary!) only be added via composition. The inheritance relationship breaks the expected behaviour of a stack. Namely only **`pop()`** and **`push()`** can manipulate it.

Antipattern 3: Misuse

A user can legally write such code:

```
Stack<String> stack = new Stack<String>();
stack.push(“1”);
stack.push(“2”);
stack.insertElementAt(“sqeeze me in!”, 1);
while (!stack.isEmpty()){
   System.out.println(stack.pop();
}
```

Antipattern 4

An example from JDK 1.2 distribution:

```
java.util.Hashtable
        ↑ extends
java.util.Properties
        ↑ extends
java.security.Provider
```

```
put(key, val)
remove(key)

put(key, val) // security check
remove(key)   // security check
```

Antipattern 4

Extension of basis class **`java.util.Hashtable`**:

```
java.util.Hashtable
      ↑ extends
java.util.Properties
      ↑ extends
java.security.Provider
```

```
put(key, val)
remove(key)
Set<Map.Entry<K,V>> entrySet()
```

```
put(key, val) // security check
remove(key)   // security check
```

Aside: Is a **`PropertyList`** a **`Hashtable`** ? See Antipattern 3.

Antipattern 4: Misuse

The attacker bypasses **`remove()`** method and uses the inherited **`entrySet()`** method to delete properties:

```
java.util.Hashtable
        ↑
        extends
        |
java.util.Properties
        ↑
        extends
        |
java.security.Provider
```

```
put(key, val)
remove(key)
entrySet()  //supports removal!
```

```
put(key, val) // security check
remove(key)   // security check
```

Antipattern 4: Problem

1. Subclasses cannot guarantee encapsulation:
   $\rightarrow$ Super class may modify behaviour of methods that have not been overridden;
   $\rightarrow$ Super class may add new methods.

2. Security checks enforced in subclasses can be bypassed:
   $\rightarrow$ The `Provider.remove` security check is bypassed if the attacker calls the newly inherited `entrySet()` method to perform removal.

Antipattern 4: Solution

1. Avoid inappropriate sub-classing:

   → Subclass only when the inheritance model is well-specified and well-understood.

2. Monitor changes to super classes:

   → Check for behavioral changes in existing inherited methods and override them if it is necessary for your application;

   → Check all new methods and **override** them if it is necessary.

```
java.security.Provider put(key, value)// security check
                       remove(key)    // security check
                       Set entrySet() // immutable set
```

Antipattern 5

Example from JDK 1.4 distribution:

```
package sun.net.www.protocol.http;

public class HttpURLConnection extends java.net.HttpURLConnection {
  /**
   * Set header on HTTP request
   */
  public void setRequestProperty(String key, String value) {
    // no input validation on key and value
  }
}
```

Antipattern 5: Misuse

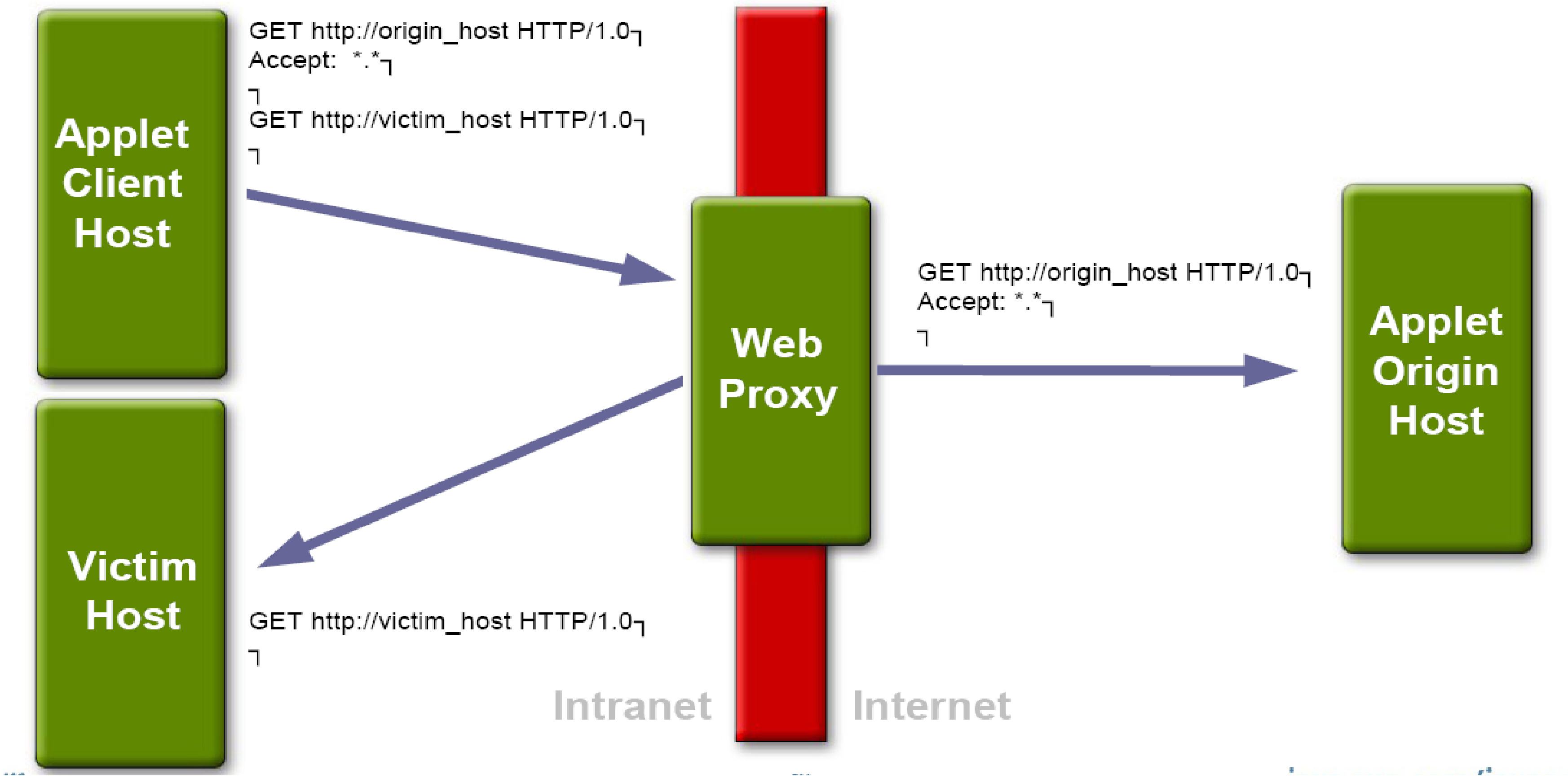
Attacker crafts HTTP headers with embedded requests that bypass security:

```
package sun.net.www.protocol.http;
public class HttpURLConnection extends java.net.URLConnection {
  public void setRequestProperty(String key, String value) {
    // no input validation on key and value
  }
}
            ...

urlConn.setRequestProperty
      ("Accept",
      "*.*\r\n\r\nGET http://victim_host HTTP/1.0\r\n\r\n");
```

Antipattern 5: Attack

Embedded request that bypasses the security check:



Antipattern 5: Problems

1. Malicious minds can craft any inputs with out-of-bounds values or escape characters.
2. It affects all code that processes requests or delegates them to sub-components:
   - $\rightarrow$ Implements network protocols
   - $\rightarrow$ Constructs SQL requests
   - $\rightarrow$ Calls shell scripts
3. Additional issues when calling **native** (JNI) methods
   - $\rightarrow$ No automatic array bounds checks

Antipattern 5: Solution

1. Validate all inputs:
   - → Check for escape characters;
   - → Check for out-of-bounds values;
   - → Check for malformed requests;
   - → Regular expression API can help validate String inputs (`java.util.regex` or C++ boost regex)
2. Pass only validated inputs to subcomponents:
   - → Wrap native methods in Java language wrapper to validate inputs;
   - → Make all native methods private.

Antipattern 6

Example from JDK 1.4.2 distribution:

```
package org.apache.xpath.compiler;

public class FunctionTable {
   public static FuncLoader m_functions;
}
```

`xpath` is a query language for XML documents. Example see appendix.

Antipattern 6: Misuse

An attacker can replace the function table in the following way:

```
package org.apache.xpath.compiler;

public class FunctionTable {
  public static FuncLoader m_functions;
}

FunctionTable.m_functions = <new_table>;
```

Antipattern 6 : Problems

1. Sensitive static state can be modified by untrusted code:
   → Replacing the function table gives attackers access to the `XPathContext` used to evaluate `XPath` expressions.
2. Static variables are **global across a Java runtime environment**:
   → Can be used as a communication channel (a.k.a. covert channel) between different application domains (e.g. by code loaded with different class loaders).

Antipattern 6 : Solutions

1. Reduce the scope of static fields:

   → `private static FuncLoader m_functions;`

2. Treat public static fields primarily as constants:

   → Consider using **enum** types

   → Make public static fields **final**

```
public class MyClass {
 public static final int LEFT = 1;
 public static final int RIGHT = 2;
}
```

3. Define access methods for mutable static state variable:

→ Add appropriate security checks as the example shows:

```
public class MyClass {
  private static byte[] data;
  public static byte[] getData() {
    return data.clone();
  }
  public static void setData(byte[] b) {
    securityManagerCheck();
    data = b.clone();
  }}
```

Antipattern 6 : `securityManagerCheck()`

An implementation of `securityCheckManagerCheck()` could be:

```
private void securityManagerCheck(){

  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(...);
  }

}
```

Antipattern 7

Example From JDK 1.0.2 distribution:

```
package java.lang;

public class ClassLoader {
  public ClassLoader() {
    // permission needed to create class loader
    securityManagerCheck();
    init();
  }
}
```

What happens if an exception is thrown during object construction?

Antipattern 7: Misuse (only Java $\leq$ 6)

An attacker overrides **`finalize`** to get a partially initialized **`ClassLoader`** instance:

```
package java.lang;
public class ClassLoader {
  public ClassLoader() {
   securityManagerCheck();
   init();
  }
}
```

```
public class MyCL extends ClassLoader{
  static ClassLoader cl;
  protected void finalize() {
    cl = this;
  }
  public static void main(String[] s){
    try {
      new MyCL()
    } catch (Exception e) {;}
    System.gc();
    System.runFinalization();
    System.out.println(cl);
  }
}
```

Antipattern 7: Problems

1. Throwing an exception from a constructor does not prevent a partially initialized instance from being acquired:

   $\rightarrow$ Attacker can override finalize method to obtain the object.

2. Constructors that call into outside code often naively propagate exceptions:

   $\rightarrow$ Enables the same attack as if the constructor directly threw the exception.

3. Good news: Java 1.7 destroys automatically objects, whose constructor has thrown an exception.

Antipattern 7: Solutions

1. Make the class `final` if possible.
2. If the `finalize` method can be overridden, ensure that partially initialized instances are unusable:

   → Do not set fields until all checks have completed

   → Use an initialized flag

```
public class ClassLoader {
  private boolean initialized = false;
  ClassLoader() {
    securityManagerCheck();
    init();
    initialized = true; // check flag in all relevant methods
  }
}
```

Antipattern 8: TOC2TOU

1. TOC2TOU: Time Of Check To Time Of Use. A typical race condition in security context.
2. One checks the permission at time t_0 and then one uses the resources without any check any more.

```
public class Foo {
  private boolean initialized = false;
  Foo() {
    BarPermission perm = new BarPermission();
    securityManagerCheck();
    init();
    initialized = true; // check flag in all relevant methods
  }
  readWithPermissionBar(){ // no check };
  writeWithPermissionBar(){ //no check };
}
```

Antipattern 8: Misuse

1. After the check phase Oscar (the cracker) can use the `read(write)WithPermissionBar()` methods without fear to generate a security exception.

2. This is typical of all Unix systems: the user's permissions are controlled only at the opening of the file and never checked again when the user manipulates them.

Oscar strategy:

1. **Create a file the user can read**
2. **Start the program**
3. **Change the file to a symlink pointing to a file that the user shouldn't be able to read**

```
...
fd = open(file, O_RDONLY); /* do something with fd...*/
```

The solution is obvious: always check the user's permissions while executing a privileged operation.

Twelve (very conservative) guidelines for writing safer Java

1. Do not depend on implicit initialization.
2. Limit access to entities.
3. Make everything final.
4. Do not depend on package scope.
5. Do not use inner classes.
6. Avoid signing your code. Code that isn't signed will run without special privileges: i.e. less likely to do damage!
7. If you must sign, put all signed code in one jar archive.
8. Make classes uncloneable. Java's object cloning mechanism allows an attacker to build new instances of the classes you define, without using any of your constructors.
9. Make classes unserializeable.
10. Make classes undeserializeable.
11. Do not compare classes by name.
12. Do not store secrets.

Appendix

XML books' search without xpath

```
ArrayList result = new ArrayList();
NodeList books = doc.getElementsByTagName("book");
for (int i = 0; i < books.getLength(); i++) {
    Element book = (Element) books.item(i);
    NodeList authors = book.getElementsByTagName("author");
    boolean stephenson = false;
    for (int j = 0; j < authors.getLength(); j++) {
        Element author = (Element) authors.item(j);
        NodeList children = author.getChildNodes();
        StringBuffer sb = new StringBuffer();
        for (int k = 0; k < children.getLength(); k++) {
            Node child = children.item(k);
            // really should to do this recursively
            if (child.getNodeType() == Node.TEXT_NODE) {
                sb.append(child.getNodeValue());
            } }
        if (sb.toString().equals("Neal Stephenson")) {
            stephenson = true;
            break;}
   }

    if (stephenson) {
        NodeList titles = book.getElementsByTagName("title");
        for (int j = 0; j < titles.getLength(); j++) {
            result.add(titles.item(j));}}}
```

DOM code to find all the title of books authored by Neal Stephenson in a XML-file

XML books' search with xpath

```
public class XPathExample {

  public static void main(String[] args)
   throws ParserConfigurationException, SAXException,
          IOException, XPathExpressionException {

    DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
    domFactory.setNamespaceAware(true); // never forget this!
    DocumentBuilder builder = domFactory.newDocumentBuilder();
    Document doc = builder.parse("books.xml");

    XPathFactory factory = XPathFactory.newInstance();
    XPath xpath = factory.newXPath();
    XPathExpression expr
     = xpath.compile("//book[author='Neal Stephenson']/title/text()");

    Object result = expr.evaluate(doc, XPathConstants.NODESET);
    NodeList nodes = (NodeList) result;
    for (int i = 0; i < nodes.getLength(); i++) {
        System.out.println(nodes.item(i).getNodeValue());
    }
  }
}
```