Software security& secure programming

Java
An example of a secured architecture

Master M2 CyberSecurity & Master MoSiG

Academic Year 2016-2017

- Applets : remote/mobile code execution
- Native application : execution on a server (JSP, Entreprise Java Beans, ...)

- Security needs: potentially dangerous execution modes ...

Mobile Code

- Host security: against unreliable code
  - confidentiality: code does not allow information leakage
  - Integrity: host data and code should not be corrupted

- Code security: against unreliable host [not in Java]
  - more difficult; need to check information flow at the host level ...

JVM security architecture

- The code is not executed directly on the hardware but via a software layer (byte code)
  - Byte-code verifier : sanity check on the byte-code
  - Class loader : manage the dynamic code loading and access rights of a class
  - Access controller: manage the access rights at runtime

Ex: for the applets

- Need to forbid:
  - Read, right, destroy the client files
  - List the directory content, find a file
  - Redefine the classes of a client package
  - Create a new class loader
  - Create external threads

Java : software isolation (1)

- Type safety: compiler and runtime execution ensure that data are processed w.r.t their source-level type
  - Strong static typing
  - Type coercions and type assignments cheked at runtime
- Memory safety: compiler and runtime execution ensure that memory accesses always refer to ``correct'' objects
  - No direct memory accesses
  - Runtime vérification of array bounds
  - Object initialisation
  - No explicit memory de-allocation (garbage collection)

Java : software isolation (2)

- Control flow safety : compiler and runtime execution forbid arbitrary jumps into the code
  - Structured control-flow : methods can be accessed only through their entry points !

=> Access to OS methods restricted to the virtual machines

=> Method access control is therefore enough ...

Type preservation

- When a program executes without error, then the « runtime types » of expressions match their « declaration types »:

$$e : t \;\text{ and }\; eval(e, mem) = v$$
$$\Rightarrow v : t$$

This is not true in C !

(e.g., “char *p” means that p points to an int)

Mobile code and security

- Mobility :
  - Java source code is compiled into a (standard) byte-code intermediate format
  - Mobile code (e.g., applets) is byte-code ...

- Security :
  - Check the byte-code integrity and the language security properties (type safety, etc.)

Compilation step

```
class  myClass {
  main(){
    ...
  }
}
```

Source file

`myClass.java`

compiler

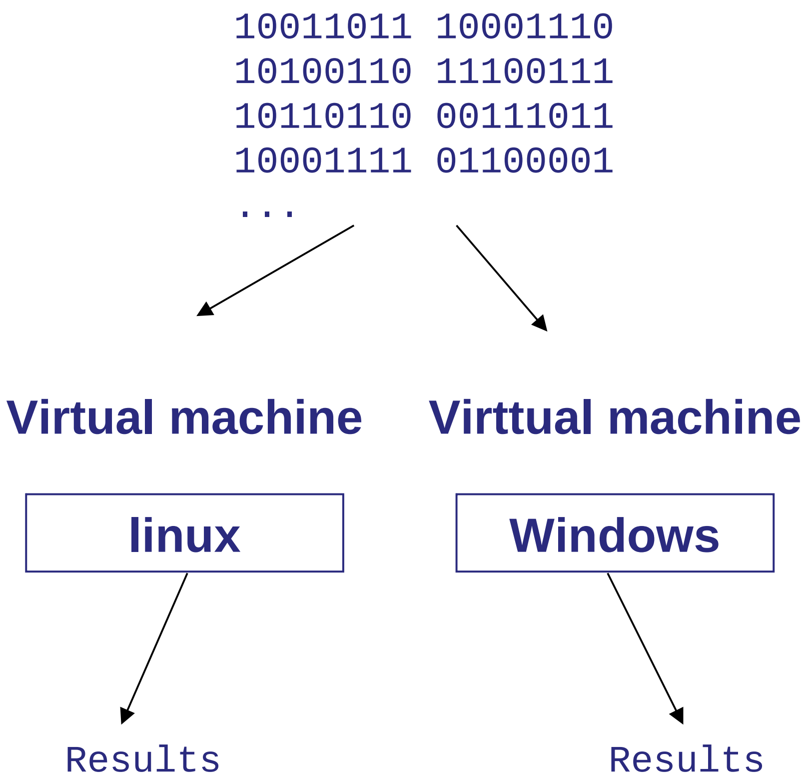
`javac myClass.java`

```
10011011 10001110
10100110 11100111
10110110 00111011
10001111 01100001
...
```

Bytecode file

`myClass.class`

Execution step



Bytecode file
`myClass.class`

`java myClass`

Execution results

How to trust the code ?

- myclass.java : verified by the compiler ...
  myclass.class : might be corrupted (integrity ?)

  - Re-compile from source (if available !)
  - Certified compilation : code + certificate
  - Check the byte-code
  - Proof-Carrying Code : code + proof

Byte-Code verifier

Verifications (1)

=> file integrity

- Check the .class structure:
  - Magic number, attributs and attribut sizes (constant zones …)

- Check the likelihood
  - Class structure, class hierarchy, fields and methods references
  - No jumps into the middle of an applet/method
  - Access verification (private, public, protected)

- Control-flow verification

Verifications (2)

- Instructions use well-typed arguments
- No illicit type conversions
- Correctness of memory and register accesses
- No stack overflow within a method call

  (+ runtime verifications)
- Variables and objects are initialized before use

Byte code

- Instructions are typed !
- (*i* for int, *a for* address, *s* for short, …)

- Instructions operate on a typed stack and on typed registers

example

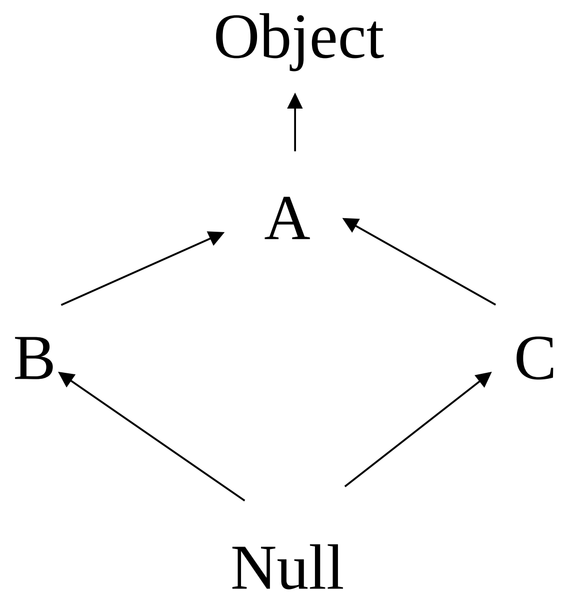
- iconst 2
  - Push the integer constant 2 on the stack
- istore r0
  - pop the integer value on top of the stack into register R0
- aload r0
  - Push an address from register R0 to the top of the stack

$\Rightarrow$ The sequence :

iconst 2   istore r0   aload r0

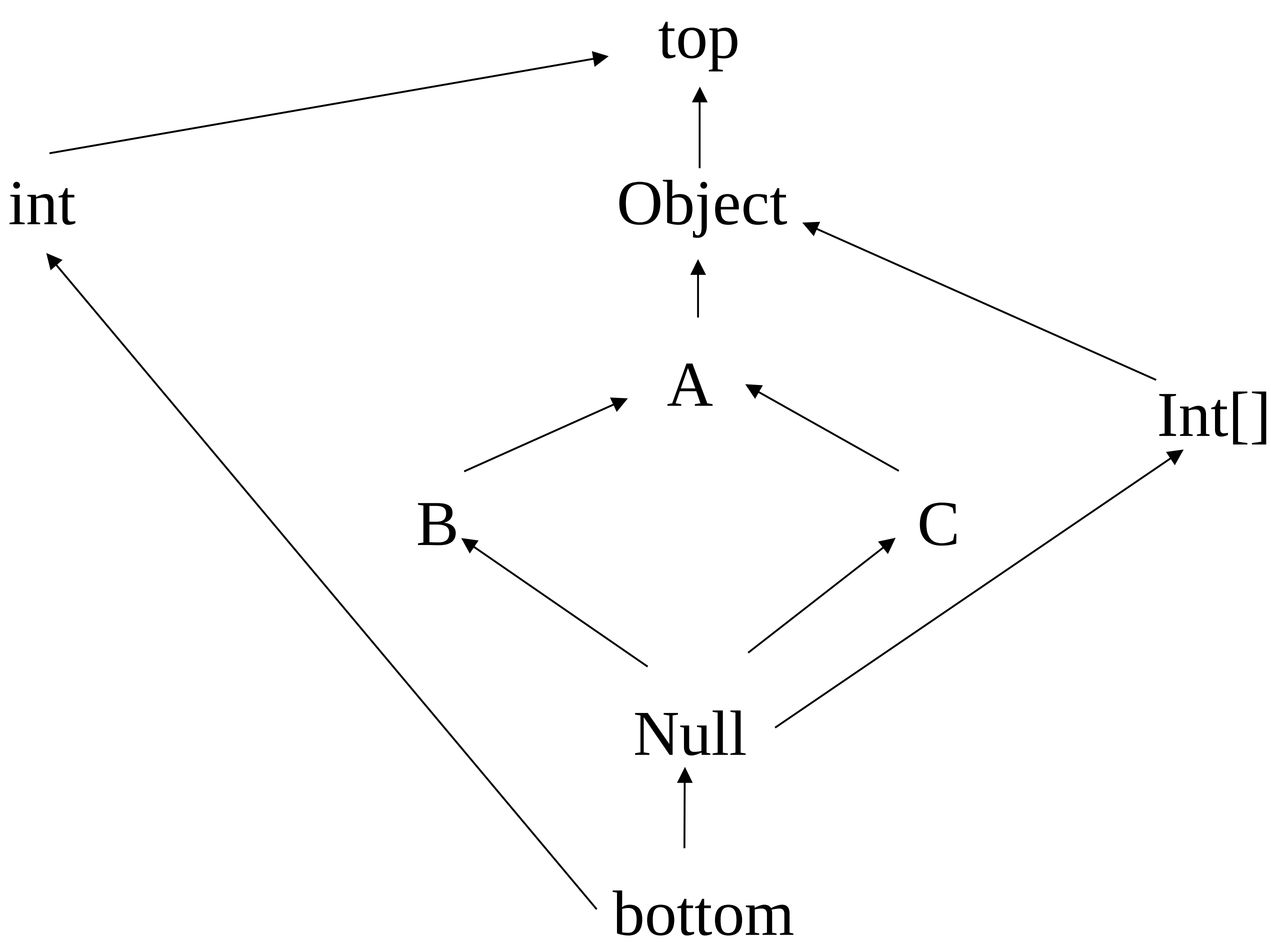
produces a type error !

Class hierarchy



class A { } ; class B extends A { } ; class C extends A { } ;

Lattice



Partial Order

- a partial order ≤ is a reflexive, anti-symetric and reflexive relation

- the least upper bound (lub) of a an element x is the smallest element l such that x ≤ l

- the greatest lower bound (glb) of a an element x is the largest element g such that g ≤ l

Lattice

(A, ≤) is a lattice if

1. ≤ is a partial order on set A

2. A has a maximal element ($\top$) and a minimal element ($\perp$)

3. Each subset of A has a glb and a lub on A

The lattice of Java types

- Lattice of types :
  - Top ($\top$) : undefined type (undefined value)
  - Bottom ($\perp$): void type (no value)

- Types are inferred from the byte code

  using the least upper bound to unify distinct types

  ...

Conditions on the instructions

- Current state: < type stack , register types>
- iadd : $<int.int.S, R> \rightarrow <int.S, R>$
- iload Rn : $<S, R> \rightarrow <int.S, R>$ with type(Rn)=int
- aload Rn : $<S, R> \rightarrow <type(Rn).S, R>$ with type(Rn) $\leq$ Object
- astore Rn : $<t.S, R> \rightarrow <S, R'>$ with type(Rn):=t
  t $\leq$ Object

algorithm

Type inference from the byte code

- Propagate the types along sequential execution paths
- Merge points : take the lub ...
- Iterate until stability (in case of loops)
- If result type is bottom ($\perp$) or top ($\top$) then error

Example

ro : int, r1 : A, r2 : B, r3 : $\top$, r4 : $\top$	stack=<>
iload r0	stack=<int>
ifeq	
iconst 42	stack = < int int >
istore r4	r4 : int stack = < int >
aload r1	stack = <A int>
else	
aload r2	stack = <B int>
end	r4 : $\top$ stack = <A int>
iload r4	error : r4 not of type int

Implementation

- Costly in computation time
- Costly in memory space :

  3 x (stack size + number of regs) x numbers of branch.

- Embedded code (javacard) :
  - External verification (certificate)
  - Proof-Carrying-Code : external verification + code annotations + on-line verification from the annotations

Class loader

Several executable components

- Main java application : executed on a dedicated JVM
- Java applet : dynamically loaded by a JVM executing other programs (e.g., other applets).
- EJB (Entreprise Java Beans)
- Tomcat : servlets and JSP loading

Dynamic class loading (at runtime) =

critical security issue

Context

- Class loading policicy
  - Lazy : do not re-load a class already loaded
- What about right accesses ?
  - Protection domain (confidence level)
- Name binding at runtime (to avoid « overtaking »)
  - No redefinition of the host classes
  - What about pre-loaded classes ?

Overtaking a class

```
 { class C1 ;
        void m () … }

paint ()   { o1 : C1 ; … o1.m() ; … }

Pre-loaded class :
{ class C1 ;
        void m ()
             { unsecure code }
  … }
```

Protection domains

Set of Java class and objects, characterized by :

- Physical origin of the byte code (http:/ …)
- A certificate
- The user (JDK 1.4)

=> define a set of permissions (rights to execute some operations on sensible resources)

- JDK 1.0 : 2 domains :
  - Local classes : all rights allowed
  - Loaded classes : very restricted permissions

- From JDK 1.2 : more general domain definition and fine-grained security policies

Class loader (1)

control when and how an application may load new classes at runtime (avoid the Java environment to be modified by malicious code).

- Check if the class has been already loaded or not
- Otherwise load the class
- Read and build the code
- Link edition
- Call the verifier

Class loader (2)

- A specific objects inherits from the class loader
- Name reference inside the JVM
  - A class is referenced by its name and the name of its class loader
  - Possibly multiple instances (e.g., in case of dynamic updates)
- JDK 1.2 : class loader hierarchy

Accessing a class

- A class may access only the classe loaded by its own class loader, or by a class loader higher in the hierarchy

=> no access between classes loaded by independent class loaders

Example

Local file system

file:///path/

1001 ... **C1.class**

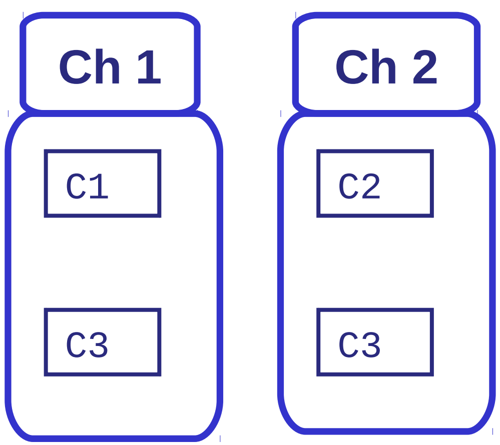
1001 ... **C3.class**

Remote file system

http://www.mysite.com/

1001 ... **C2.class**

1001 ... **C3.class**



Virtual machine

Name : C3
Loader : Ch1
URL : file:///path/C3.class

Name : C3
Loader : Ch2
URL : http://www.mysite.com/C3.class

Hierarchy (1)

Primordial class loader (bootstrap in C)

- Load the system code. Ex : file management classes, based on native OS security mechanisms
- Load the classes of the CLASSPATH
- Root of the class loader hierarchy
- No need to call the byte-code verifier
- Classes considered as secure

Hierarchy (2)

1. Primordial class loader
2. Java.lang.Classloader
3. Java.security.ClassLoader :
4. Secured class loaders (with delegations)
5. Java.net.URL.ClassLoader :
   – Standard applications

- AppletClassLoader
  – applets

Secure Class Loader

Each loader loads a class if it is possible, otherwise it delegates this task to its father (in the class loader hierarchy)

=> system-level classes are loaded by the so-called *primordial class loader*

Objet class loader

- loadClass : request for a class loading
- findLoadedClass : is there a given class already loaded ?
- findClass : look for a class to be (actually) loaded
- getParent : acces to the class loader hierarchy
- resolveClass : set up a class (verification and binding)

class loaders themselves are also concerned by security rules ...
(Java.security.securityPermission)

Summary

- Lazy class loading ("on demand")
- Delegation mechanism
- Creating a new class loader is a critical operation (hence access controled)
- To avoid the pre-loading of (insecure) methods, browsers use one class loader per applet (isolation)

Access Control

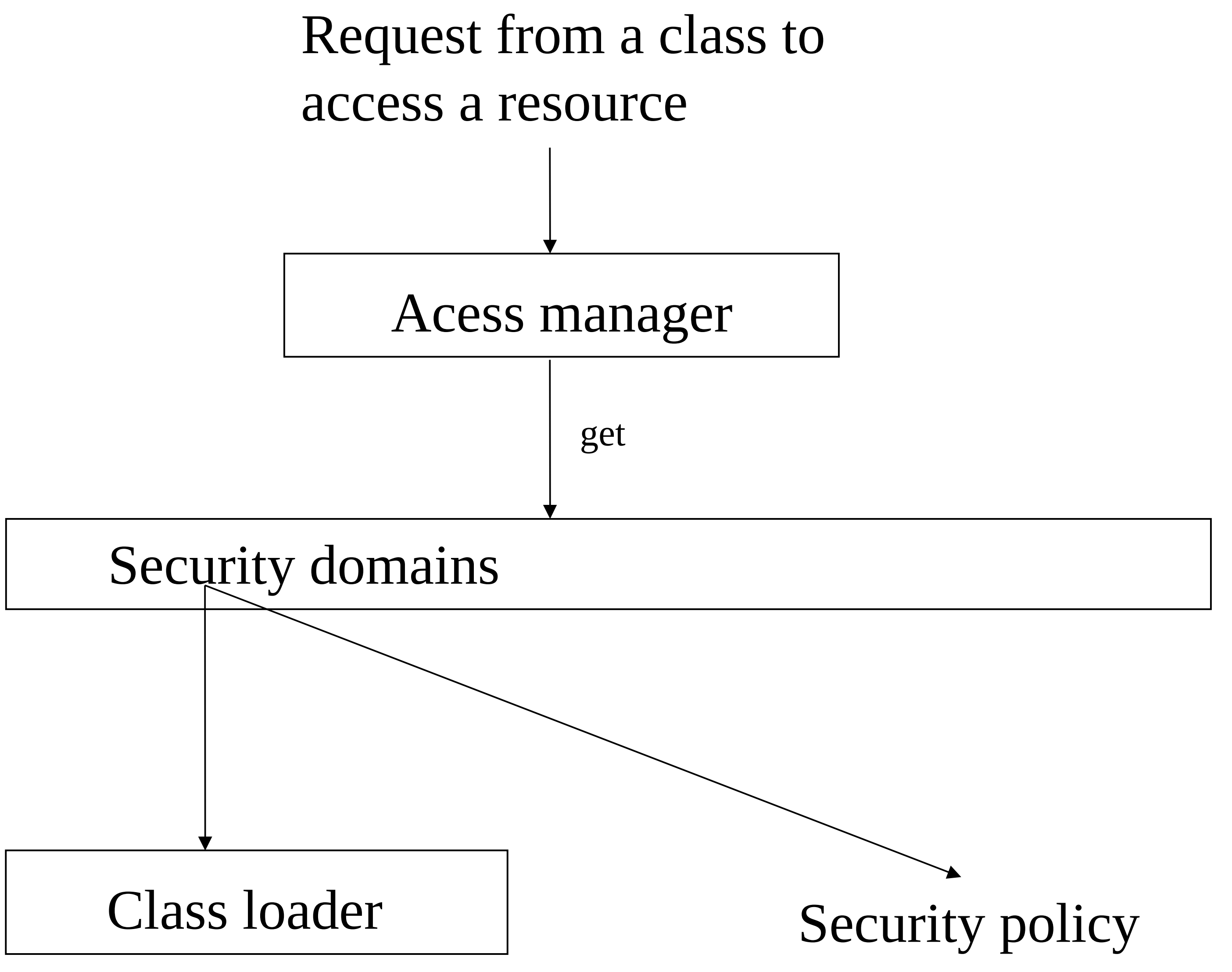
- Permission declarations
- Acces rights
- Examples

Security domains

- Static rights with respect to code origin
- Dynamic rights with respect to a policy evolving at runtime (the policy chosen for a class is the one when this class is loaded).

Java keeps a link between the objects and their permission domains

  - The class loader associates a CodeSource object to a class in terms of an URL and a set of certificates. getPermissions() returns the acces rights associated to a CodeSource object.


Request from a class to
access a resource
Acess manager
get
Security domains
Class loader
Security policy

Permission representation

- permission : a resource + set of operations allowed on this resource

- Access rights are represented by an object inherited from the class Permission.
  - A set of predefined sub-classes of the Permisison class
  - Possibility to define its owns right access to an object

=> creation of a Permission object, call to the method checkPermission of the class AccessController.

Permission class

- Java.util.PropertyPermission :
  – read/write access to JVM properties (Os name, …)
- Java.lang.RuntimePermission :
  – use of runtime functions as exit() and exec()
- Java.io.filePermission
  – Read/write/execute access to file/directories
- Java.net.SocketPermission
  – Controls use of network sockets
- Java.lang.reflect.reflectPermission
  – Control use of reflection to do class inspection
- Java.security.securityPermission
  – Control access to security methods (class loader …)
- …

Examples

- FilePermission perm1 = new FilePermission (’’ path/file ’’, ’’ read ’’);

  …

  AccessController.chekPermission(perm1) ;

  …

=> read access to the files

⇒ return AccessControlException in case of error

⇒ Read/write access to the files of /tmp

- p1 = new java.io.FilePermission (’’/tmp/-’’, ’’ read, write ’’) ;

=> read acces to the system property of user.name

- p1 = new java.util.PropertyPermission (’’user.name’’, ’’ read ’’) ;

Policy declaration

Read-only access for the user, resource and user classes being in distinct code bases :

```
grant codebase "file:///code/resource" {
        permission java.lang.RuntimePermission "resource.*";
};
grant codebase "file:///code/user" {
        permission java.lang.RuntimePermission "resource.read";
};
```

=> all local access , read only for the user code.

Policy definition

- The access control manager needs to be provided at compile time (default is « no control ») :

  Java –Djava.security.manager –Djava.security.policy=policy.txt –jar main.jar

- An application may define its own access policy
  - Lib/security/java.policy : default policy
- And its own security manager :

```
public void test()
{   SecurityManager s = System.getSecurityManager() ;
    if (s != null)
      {  String name = ‘‘SecureMethodCall.test’’ ;
           s.checkPermission(new ExecMethodPermission(name)) ; }
     System.out.println(‘’ test method executed’’) ; }
  }
```

Implemantation

- General principle
- Retrieving the right accesses
- Temporary priviledge extension

General Principle

- Definition of sensitive operations (defined in the standard APIs and extendables)

- A method should check beforehand if it is allowed (or not) to execute a sensitive operation

Flow based access control

- Only the execution of methods (read, write, connect …) depend on access rights. Acessing an object is not a protected operation.

- E.g. : accessing a system property can be controlled, but not assigning a system property ...

Main principle

- A sensitive operation can be applied to a given ressource via a method call only if **all** methods of the execution stack have a right access on this resource (intersection of rights).

- A method can temporarily gain or lost priviledges

- The access controler cheks the access rights

Example

```
Package main ;
Public class Main

{ public static void main(String[] args)
        {Foo.test() ; }
 }

Package main ;
Import java.io.File ;
Public class Foo
{
  public static void test()
        { File f = new File(‘ ‘/tmp/my-test-file ’’);
        f.delete();
          }
 }
```

Example of calling context

Stack :

Main.main

Foo.test

File.delete

OK by default

Rejected by java –Djava.security.manager –jar main.jar

Priviledge passing mechanism

- Allow to temporary suspend the context (priviledged mode)
- Only the access rights of the current class are taken into account
- Ex : calling a system method from a user method may require some specific accesses (ex. font files)

do_Privileged

- The callee may give temporary rights to the caller

(e.g. reading/writting a resource)

  - Introduce a wrapper between the resource and the user to check the access rights. The wrapper owns evry rights on the resource.
  - The user access the resource through the wrapper
  - The wrapper temporarily allows the user to access the ressource (even if this later do not own the appropriate rights)l.

Solution (1)

```
// code-base 'file:///code/resource'  owns all the rights on the resource
   public class ResourceWrapper
   { /..

        public void write()
          { if(writeIsBad) // filter out the bad stuff.
               return;
           AccessController.doPrivileged(new PrivilegedAction()
          { public Object run() { resource.write(); return null; }
             }) ;
         }
    }
```

Solution (2)

⇒ We can use a context to limit the rights in privileged mode

```
// code-base 'file:///code/resource'
public class ResourceWrapper {  ...
public void write() {
    Permissions subset = new Permissions();
    subset.add(new RuntimePermission("resource.write"));
    AccessControlContext context = new AccessControlContext
      ({ new ProtectionDomain(null,subset) });
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() { resource.write(); return null; } }, context); }
```

The wrapper explicitly specifies that write access only are given to the caller.

Algorithm for checkPermission

m calls m-1 … calls m1

```
i= m ;
While (i > 0)
{ if (caller i's domain does not have the permission)
      throw AccessControlException
  else if (caller is marked as privileged) return ;
  i= i-1 ;}
```

Lazy implementation : at each check the stack is entirely scanned

Eager implementation : each activation block is updated with the corresponding permissions

Limitations (1)

- *http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/securite-et-langage-java.html*

- Programming :
  - Security mechanisms difficult to use/understand by developers
  - Some weaknesses : integer overflow (wrap-around), serialisation (attacks by modifying data encoding classes), reflexion

Limitations (2)

- Virtual machine :
  - Hard to implement (+ Just in time compilation) … and to validate !
  - No obvious links with the OS access control mechanism
  - Possibility to use non Java code (JNI)

- Standard libraries:
  - Contain vulnerabilities ...
  - Library SUN : 1 900 000 LoC (3/4 Java, ¼ C et C++),
    HotSpot 450 000 C++ LoC)
  - Contains some critical functionnalities

Limitations (3)
Memory management

- No memory leaks, but no fine-grained dynamic memory management ; no obvious way to « erase » (confidential) data

- Only local variables and public fields can be erases

(not the case for non mutable objects, constant strings, …)

- Generational garbage collector : data copies

Exploit Example

- Exploit description (april 2003)

http://assiste.com.free.fr/p/parasites/bytverify_exploit.html

ByteVerify Exploit exploits a vulnerability of byte-code verifier of the Microsoft JVM implementation

Exploit description

- Declare a new parameter "PermissionDataSet" with a field "setFullyTrusted" defined as "TRUE".
- Allow to create its own "PermissionSet" parameter
- Defines   "PermissionSet" authorizations by creating its own "URLClassLoader", derived from the "VerifierBug.class".
- Load "Beyond.class" using "URLClassLoader" from "Blackbox.class".
- Get unrestricted rights on the local machine by calling method ".assertPermission" of the "PolicyEngine" class within "Beyond.class".
- …

Other examples of exploits

CVE-2008-5353

use doPrivileged and deserialisation

http://blog.cr0.org/2009/05/write-once-own-everyone.html

CVE 2012-1723 :

exploits a vulnerability of the HotSpot byte-code verifier (before Java 7.4) weaken type verifications, overcome the sandboxing to load malicious classes

http://schierlm.users.sourceforge.net/CVE-2012-1723.html

Some related links

- The CERT

https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java

- ORACLE secure codibg rules: http://www.oracle.com/technetwork/java/seccodeguide-139067.html

- JAVASEC :

http://www.ssi.gouv.fr/IMG/pdf/JavaSec-Recommandations.pdf