Java Security Model and Signing Code

Spring 99

Geoffrey Fox, Nancy McCracken,

and Mehmet Sen

Syracuse University

NPAC

111 College Place
Syracuse NY 13244 4100

3154432163

Java Security Model

- Review of Java Security Mechanisms
- Models for JDK1.0, JDK1.1, and JDK1.2
- Signing Code for 1.1 and 1.2
- Signing Code for JavaScript
- Resouces:
  - http://java.sun.com/security

The Java Security Model

Three mechanisms in Java help ensure safety:

- Language design features (bound checking on arrays, legal type conversions only, no pointer arithmetic, etc.)
- Java “sandbox” mechanism that controls what the code can do (like local file accesses) Common to both Java 1.0 and Java 1.1.
- Code signing: Programmers can use standard cryptographic algorithms to embed a “certificate into a Java class. Then, the users can precisely understand who implemented the code and signed. If one alters the code, he would not be able to sign it with the original signature. Users may decide to use or not to use the code depending on the trust of the signer.

Sandbox mechanism

- This addresses security of the client machine once an applet has been downloaded and includes processing of security mechanisms such as authentication certificates
- There are three parts of the Java Security model:
  - Byte Code Verifier: checks that the downloaded .class files obey the rules of the Java Virtual Machine
  - Class Loader: makes certain that Java classes have a security structure that prevents outside applets contaminating built in runtime.
  - Security Manager: implements overall policy which depends on particular browser and includes privileges open to applets and processing of authentication mechanisms
  - Note first two parts can have bugs; last part can have both bugs and ill advised policies!

What can applets do - I?

- Currently the rules are strict and in fact unreasonably so because these rules stop applets from doing things that we want general client programs to do. They are necessary as we must assume the worst about applet!
  – We need security so that we can identify those applets that can be given more privileges!
- Applets cannot read write delete rename files
- Applets cannot create a client directory or list a directory
- Applets cannot find out any information about a client file including whether or not it exists
- Applets can only create a connection to computer from which it was downloaded
- Applets cannot accept or listen to network connections on any client port

What can applets do - II?

- Applets can only create top level windows which carry message “untrusted window”. This helps protect one making operating system look alike windows which innocently request you type password or similar information
  - e.g. suppose I wrote an applet that generated window saying “network connection broken” and put up identical window to internet login process. Most people would type in password without thinking!
- Applets cannot obtain user’s username or home directory name.
- Applets cannot define system properties
- Applets cannot run any program on the client system using Runtime.exec().

What can applets do - III?

- Applets cannot make the interpreter exit through either System.exit() or Runtime.exit() methods.
- Applets cannot load dynamic libraries on the client using load() or loadlibrary() methods of the Runtime or System classes
- Applets cannot create or manipulate any thread that is not part of same ThreadGroup as the applet.
- Applet cannot create a ClassLoader or SecurityManager
- Applet cannot specify network control functions including ContentHandlerFactory, SocketImplFactory or URLStreamHandlerFactory
- Applet cannot define classes that are part of built in client side packages

The Byte Code Verifier

- This check ensures that code to be executed does not violate various semantic rules of the Language and its runtime
- In particular check that pointers are legal, access restrictions obeyed and types correct
- a .class file consists of some overall information including a magic number, versioning information, a constant pool, information about the class itself (name, superclass etc.), information about fields and methods with the byte codes themselves, debugging information.
- The byte codes are essentially machine language for an idealized stack based machine which are translated into true machine code
  - Note such stack based machines are not necessarily best for optimized performance. Compilers use rather different intermediate representation

Byte Code Verification

- **First one checks that .class file has correct format**
- **Checks all that can be done without looking at byte codes**
  - **this includes superclass checking, verification of constant pool, and legal names and types for field and method references, final classes cannot be subclassed or overwritten**
- **Then one looks at the byte code and checks that as executed it will give finite size stacks, correct typing for registers, fields and methods**
- **One does not download the byte codes of required classes. However one does check that the class is used consistently with its definition**
- **Some steps are for run time efficiency such as checking for some run time exceptions can be done at verification stage and removed in running code.**

Why is type checking important!

- If one has either deliberately or accidentally a “wild object pointer” that should be to a user defined on/off object but has somehow been applied to a sensitive object.
- Then turning userobject.onoff to true is uncontroversial but this applied to appletprivilege could turn on the ability to write files!
  - Note setting userobject.onoff = true is really “go to location of this object and set its start address plus so many bytes to value true”!
- Thus normal computer programs often overwrite themselves when you screw-up with a software error.
- Java applets can obviously have software bugs but such errors do not let them ever overwrite themselves or anybody else.
  - Otherwise the overwriting can radically change security
- Thus Java must guarantee types of objects precisely so operations can be stupid but never violate security.

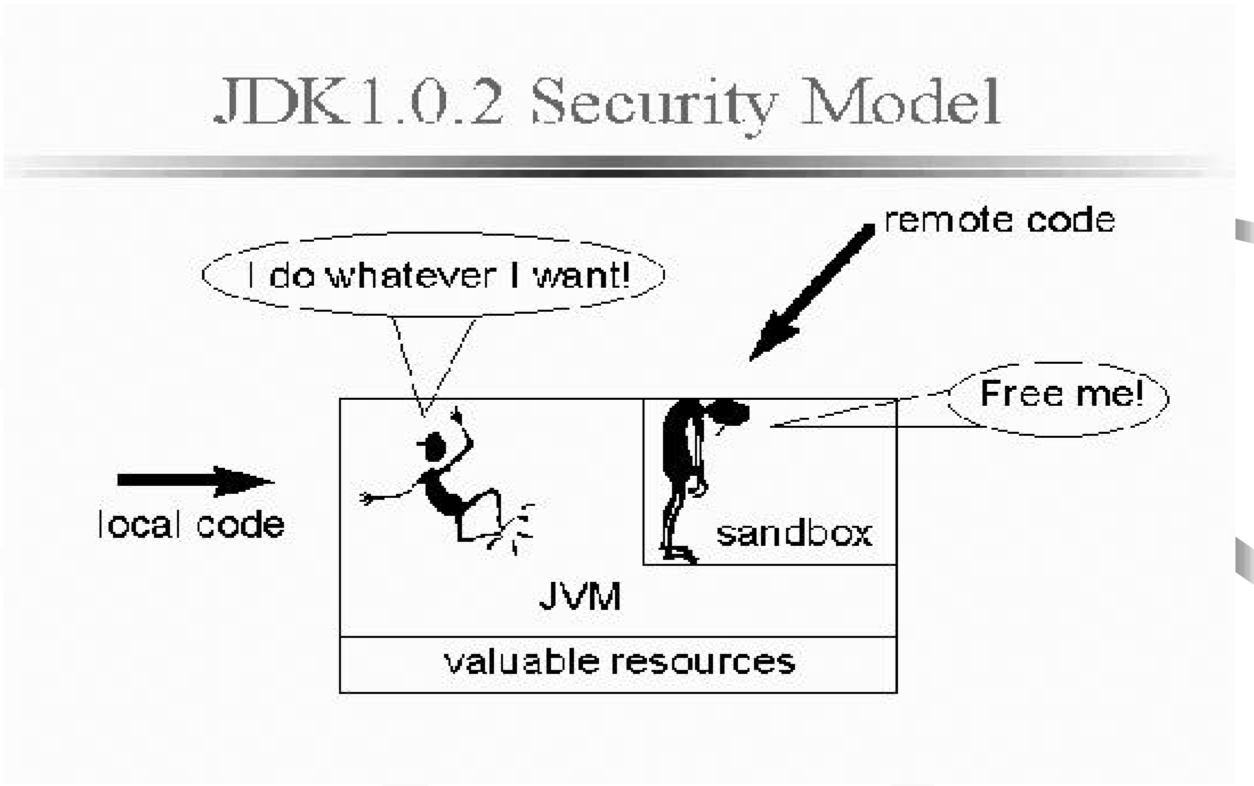
Applet Class Loader

- This second part of Java security implements a policy as to which classes can access which others!
- Java classes are divided into groups which have strict access control. There are different (class) name spaces defined by different Class Loader instances and in particular different run in different name spaces and can NOT interfere with each other.
- Classes from same source (defined by directory and host machine) are stored in same name space. An Applet can access those classes in its name space as well the built in local classes. It can access classes from other sites if it explicitly specifies a URL and the methods are public.
- Note one searches the local classes first or else one could override the built-in classes and so do things like file I/O.

Going beyond the Sandbox: History of Java Security Models

- **The Original Security Model known as sandbox model used as JDK 1.0.2 Security Model**

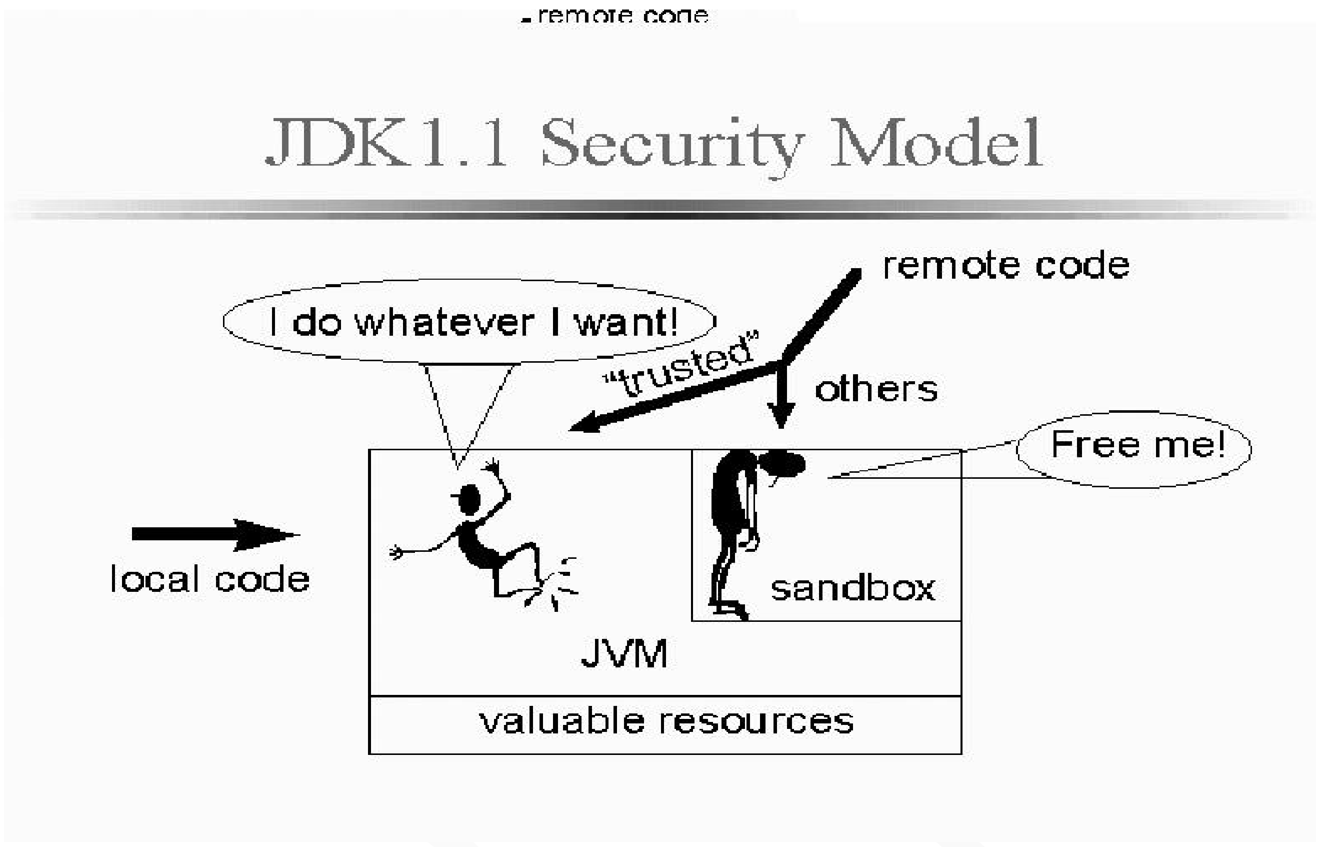
Sandbox protects access to all system resources Application developers have to write their own SecurityManager to open up the sandbox.



Going beyond the Sandbox-2

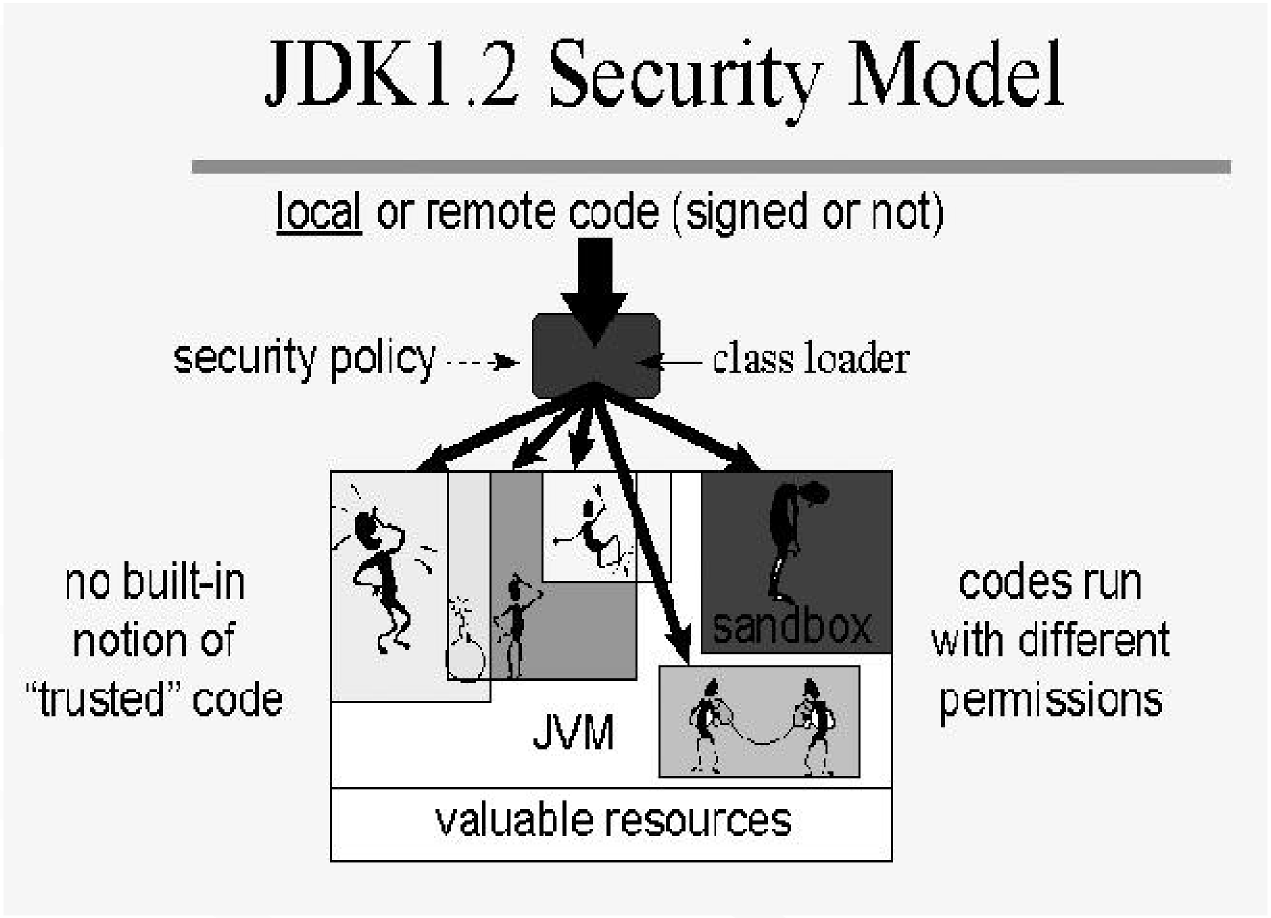
JDK1.1.x presented the signed applet concept additional to sandbox mechanism.

- Code signing provides additional levels of security when downloading remote code:
    - Authentication
    - Integrity
- Everything on CLASSPATH is trusted



Going beyond the Sandbox-3

- **Finally, JDK 1.2 introduced fine-grained access control mechanism**
- **Easily configurable security policy**
- **Easily extensible access control structure**
- **Extension of security checks to all Java programs, including applets.**


JDK1.2 Security Model
local or remote code (signed or not)
security policy
class loader
no built-in notion of "trusted" code
sandbox
JVM
codes run with different permissions
valuable resources

JDK 1.2 Security Model

Among some features the followings are included

- **The least-privilege principle by automatically intersecting the sets of permissions granted to protection domains.**
- **Underlying platform independent security features.**
- **Does not override the protection mechanisms of the underlying operating system**

The Java Cryptography Extension (JCE) extends the JDK to include APIs for encryption, key exchange, and message authentication code (MAC). Together the JCE and the cryptography aspects of the JDK provide a complete, platform-independent cryptography API.

- **JDK 1.2 Software includes**
  - **Certificate support (X.509 v1,2,3)**
  - **Support for Secure Socket Layer (SSL) v3.0**
  - **Code-signing support (keytool, jarsigner) and Policy Tool to define security policy.**

JAVA Fine-grained Access Control-1

- Essential mechanisms include the following:
- Identity:Every piece of code needs a specific identity for security decisions. Origin (URL) and signature, represented in the class java.security.CodeSource , define identity.
- Permissions: System requests to perform a particular operation on particular target are allowed based on permissions. A policy says which permissions are granted to which principals. Permissions include:
  - java.io.FilePermission for file system access, e.g.,
    ```
    f = new filePermission ("/tmp/applets.db", "read");
    ```
  - java.net.SocketPermission for network access, e.g.,
    ```
    sp= new SocketPermission(”npac.syr.edu:3768", "connect")
    ```
  - java.lang.PropertyPermission for Java properties
  - java.lang.RuntimePermission for access to runtime system resources
  - java.security.NetPermission for authentication
  - java.awt.AWTPermission for access to graphical resources such as windows

JAVA Fine-grained Access Control-2

- **Implies: All permissions must implement the implies method "a implies b" means that if one is granted permission "a", then one is also granted permission "b"**

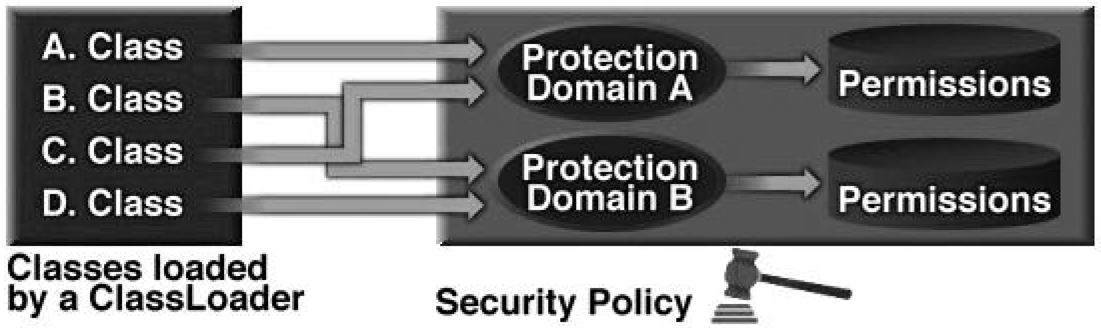
```
  Permission p1 = new FilePermission("/tmp/*", "read");
  Permission p2 = new FilePermission("/tmp/readme",
                                                   "read");
  p1.implies(p2) == true          p2.implies(p1) == false
```

Policy: is a mapping from identity to a set of access permissions granted to the code. An example policy object ;

```
grant CodeBase "http://www.npac.com/users/gcf", SignedBy "*"
{
 permission java.io.FilePermission "read,write", "/tmp/applets/*";
 permission java.net.SocketPermission "connect", "*.npac.com";
};
```

JAVA Fine-grained Access Control-3

- Policies can be defined by a user or a system administrators. It is always possible to use a particular policy for selected applications, e.g.,
  appletviewer -Djava.policy=~/gcf/policies/mypolicy1 applet.html
  java -usepolicy[:policyfile] some.local.App
- Protection Domains: A domain consists of a set of objects belonging to a principal. A protection domain is based on an identity made up on demand. Permissions are granted to protection domains and not directly to classes or objects.

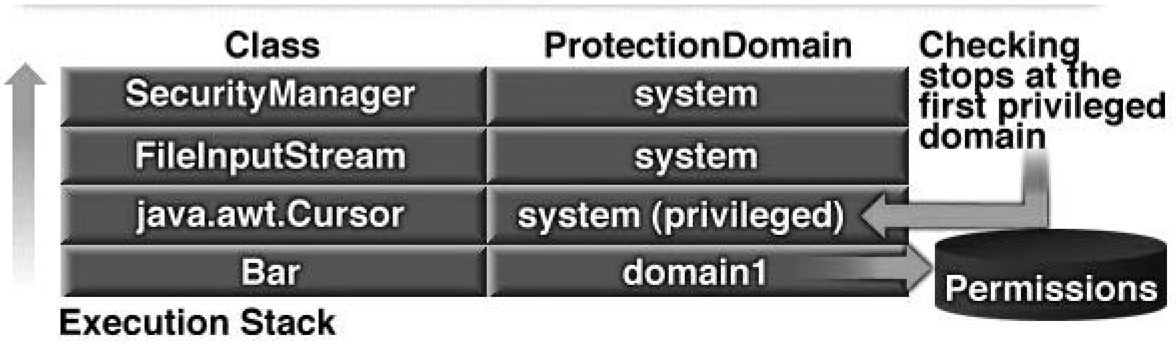


JAVA Fine-grained Access Control-4

- **Access Control: The java.security.AccessController class implements a dynamic stack inspection algorithm. The method checkPermission() provides permission grant check. Example: FilePermission p = new FilePermission("/tmp/junk", "read");**
- **AccessController.checkPermission(p);**
- **Privilege: Privileges are used to grant temporary permission to less-trusted code. Whenever a resource access is attempted, all code traversed by the execution thread up to that point must have permission for that resource access, unless some code on the thread has been marked as "privileged". That is, suppose access control checking occurs in a thread of execution that has a chain of multiple callers. When the AccessController checkPermission method is invoked by the most recent caller, it decides whether to allow or deny the requested access.**

JAVA Fine-grained Access Control-5

- The AccessController checkPermission algorithm is as follows:
  - If the code for any caller in the call chain does not have the requested permission, AccessControlException is thrown, unless the following is true - a caller whose code is granted the said permission has been marked as "privileged" and all parties subsequently called by this caller (directly or indirectly) all have the said permission.



JAVA Fine-grained Access Control-6

- Marking code as "privileged" enables a piece of trusted code to temporarily enable access to more resources than are available directly to the code that called it. This is necessary in some situations. For example, an application may not be allowed direct access to files that contain fonts, but the system utility to display a document must obtain those fonts, on behalf of the user. In order to do this, the system utility becomes privileged while obtaining the fonts.
- The Secure Class Loader, java.security.SecureClassLoader, tracks the code source and signatures of each class, and hence assigns classes to protection domains.

JAVA Fine-grained Access Control-7

- **Here is what the usage of privileged code looks like. Note the use of Java's inner classes capability:(If you are using an anonymous inner class, any local variables you access must be final to make JAVA language looks like having closures.)**

```
• somemethod() {
•         <normal code>
•     AccessController.doPrivileged (new PrivilegedAction()
•             {
•               public Object run()
•                 {   <insert dangerous code here>      return null;      }
•             });
•         <more normal code>
•           }
```

Java Security-Related Tools

- The keytool is used to create pairs of public and private keys, to import and display certificate chains, to export certificates, and to generate X.509 v1 self-signed certificates and certificate requests that can be sent to a certification authority.
- The jarsigner tool signs JAR (Java ARchive format) files and verifies the authenticity of the signature(s) of signed JAR files.
- The Policy Tool creates and modifies the policy configuration files that define your installation's security policy. The Policy Tool has a graphical user interface to define policies.

How to sign Java Code

Currently, separate vendors have the following major code-signing tools;

- Netscape's Object Signing
- Microsoft's Authenticode
- Sun's JDK 1.1 Code Signing
- Sun's Java 2 Code Signing

Though Java is designed to be portable, there are various vendor specific code signing tools that make signed Java code incompatible in others environment. Having various complexity levels , vendor dependencies for leaving sandboxes, giving different control mechanisms to user makes them independent from each other.

Signing Classes with the Netscape Object Signing Tool

- Get a signing certificate from a CA, Certificate Authority, listed at https://certs.netscape.com.
- Import the certificate to your Netscape Communicator browser.
- Use The Netscape Object Signing Tool, which is a command line program, signtool, to sign JAR files. (Note that digital signature information is transmitted in JAR files);
- Refer to the signed code as
 <APPLET CODE="signed.class" ARCHIVE="myjarfile.jar">
- Use signtool to sign the classes and create a JAR file:
```
signtool -d"path to certificate" -k"my CA" -e ".class”
                    -Z myjar.jar
```

Netscape Object Signing Tool -2

- Some useful options of signtool is illustrated as following;
  - signtool -d"path to certificates" -l
    list available certificates in your database
  - signtool -d"<path to certificate>" -v myjar.jar
    Check validity of signature on jar file whether it is tampered or not.
  - signtool -d"<path to certificate>" -w myjar.jar
    Check who signed myjar.jar
- Adding Capabilities to Signed Classes: The Netscape Capabilities library provides a class called the Privilege Manager which turns on-off privileges for incoming program requests.

Netscape Object Signing Tool -3

- For the first request, the Privilege Manager asks the Netscape user whether the privilege should be granted, showing the certificate used to sign the code requesting the privilege. Grants are valid for lifetime of the applets.
- Example privileges are UniversalFileAccess, UniversalSendMail, UniversalExitAccess, UniversalExecAccess, PrivateRegistryAccess.
  - import netscape.security.PrivilegeManager;
  - import netscape.security.ForbiddenTargetException; . . .
  - try { PrivilegeManager.enablePrivilege("UniversalFileRead");
  - ta.appendText("Read enabled!\n");
  - } catch (ForbiddenTargetException fte) {
  - ta.appendText("Read not enabled.\n");
  - ta.appendText(fte.toString());
  - }

Signing Java Applets with Microsoft's Authenticode

- Get an Authenticode Certificate, e.g., from
  digitalid.verisign.com/developer/ms_pick.htm.
- Get Microsoft Java SDK, from
  www.microsoft.com/msdownload/java/sdk/31f/SDK-JAVA.asp
- Prepare archive file , in Microsoft's CAB format,

  ```
  cabarc N test.cab *.class
  ```
- Use Microsoft Security Zones concept. By default, a security zone, assigned a security level, is a group of Web sites. The available levels are Low, Medium, High, Custom.

-
  ```
  signcode -j JavaSign.dll -jp High -spc c:\myCert.spc
           -v a:\myKey.pvk -n "My Applet Software"
           -i http://www.mywebpage.com/ myapp.cab
  ```

  where -j is to sign java code, -jp to pass parameter to JavaSign.dll, zone level, -i information page for the software that user can check.

Microsoft's Authenticode 2

- To test signature

```
chkjava test.cab
```

- To access applets from browser html tag is like

```
<APPLET CODE= "MyApplet.class">
<PARAM name="cabbase" VALUE="myapp.cab">
</APPLET>
```

- makecert -sk myKey.pvk -n "CN=Your Name" myCert.cer provides unauthorized certificate with key.
- cert2spc myCert.cer myCert.spc provides signing certificate.

Signing Code with Sun's JDK 1.1.x

- **Sun has its own set of signing tools, which have evolved along with the JDK.**
- **The JDK ships with a command-line tool called javakey. “javakey” provides management of a database of entities (people, companies, etc.) and their public/private keys and certificates.**
- **HotJava and the appletviewer program that comes with the JDK can validate JARs signed by javakey. Since the VMs in Communicator and Internet Explorer do not support javakey signing, in order to run javakey-signed applets with those browsers, users must download and install Sun's Java Plug-In.**

Signing Code with Sun's JDK 1.1.x-2

- **Some examples of using javakey**
- **javakey -cs your_name :makes a new signer**
- **javakey -ld : list the database (stored in identitydb.obj file by default)**
- **javakey -gk your_name DSA 512 : generates new key pair with modulus 512**
- **javakey -gc cert_directives.dir : generate a certificate using the directive file "cert_directives.dir"**
- **javakey -dc outfile.509 : display the certificate contents from the certificate file**

Signing Code with Sun's JDK 1.1.x-3

- javakey -gs signMyApplet.dir Animator.jar: sign the applet, i.e. Animator.jar from the directives file signMyApplet.dir
  - (Making jar files is done as following:
    - jar cvf Animator.jar AnimatorApplet.class)
  - As an example try to browse a signed applet which is at URL
    - http://java.sun.com/security/signExample/
    - See following foil for details
- javakey -h: brings the help menu for all the available options
- Note that certificates and the keys produced by javakey are all in DER format.

Browsing Signed Applets

- http://java.sun.com/security/signExample/ contains an applet signed by Duke
- To run the applet with appletviewer,
  - First get a copy of Duke's certificate and store it in a file named Duke.x509
  - Make an identity Duke in your JDK identity database in your local machine;
    - % javakey -c Duke true
  - Import Duke's certificate into your identity database,
    - % javakey -ic Duke Duke.x509
  - Run the applet signed by Duke

The Java Authentication Framework

The basic concepts are

- Principal Interface; this describes real-world entities like persons, companies etc.
- Identity class; an identity is derived from Principal Interface and has property corresponding to a public key
- Certificate class; a certificate has two properties of class Identity: one is the Identity that is being certified, and the other Identity is a guarantor, with which the principal is associated for this certificate.
- To keep identities safe from conflicts, e.g., " G. Fox" at NPAC and "G. Fox" at Sun Inc. , Java defines IdentityScopes.
- An IdentityScope may have other IdentityScopes in it. For example, Syracuse University is an IdentityScope, and it contains the NPAC IdentityScope.

The Java Authentication Framework-2

- Each Java Virtual Machine has a system IdentityScope, which keeps unique identities in its scope, and is available to all Java programs in this VM.
  - It is best to use your own identity scope and add it to the scopes contained in the system IdentityScope.
- Java 1.1 requires applets to be signed to be able to take advantage of this framework

Signing Code with Sun's Java 2

- The javakey tool from JDK 1.1 has been replaced by two tools in Java 2.
- keytool manages keys and certificates in a database. jarsigner is responsible for signing and verifying JAR files. The keystore, that contains certificate and key information, replaces the identitydb.obj from JDK 1.1.
- Java 2 does allow Certificate Authorities to sign generated certificates
- Useful command line examples are
- keytool -alias keyname -genkey : Generating a public and private key pair and self-signed certificate.
- keytool -storepasswd to change password. Note that keystore stores keys, and identity information necessary for certificates protected under a password.

Signing Code with Sun's Java 2-II

- keytool -list view the fingerprints of certificates in the keystore.
- keytool -list -v view the personal information about the issuer and owner of the certificate.
- keytool -identitydb - Import information from a JDK 1.1.x-style identity database.
- keytool -keypasswd - Assign a password to a private key in a key/certificate entry.
- keytool -printcert - Print out the information in a specified file containing a certificate.
- keytool -certreq -alias keyname -file requestfile :Generate a certificate request.

Signing Code with Sun's Java 2-III

- **keytool -import -alias newalias -trustcacerts -file response :import certificates from authorities.**
- **keytool -export -alias keyname -file mycert : Export**
- **jarsigner SignMe.jar keyname sign a jar file**
- **jarsigner -verbose SignMe.jar keyname monitor signing process**
- **jarsigner -verify Unknown.jar verify signing of a file**
- **jarsigner -verify -verbose -certs Unknown.jar More detailed check.**
- **Running a Signed Applet**
- **keytool -import -alias analias -file acert : Import to run others signed programs**

Signing Code with Sun's Java 2-IV

- Make a Simple Policy for Signed Applets
- Example   .java.policy is
- keystore ".keystore";
- grant signedBy "friend”;
- codeBase "http://www.friendly.com/~mybuddy/applets/"
-   {
-       permission java.io.FilePermission "c:\\tmp\\*", "write";
-   };

Some Comparisons of Sign Tools

- **Differences Between Netscape Object Signing and JDK javakey**
- **Netscape Object Signing only works within Communicator. JDK 1.1 signed applets can work in any browser, but with Java Plug-In for the applet to leave the sandbox.<APPLET> tag must be changed by HTMLConverter for plugin use.**
  - **JDK1.2 applets all require JRE plug-in**
- **JDK 1.1 javakey-signed applets that are trusted get complete access to the host while Netscape Object Signing prompts for specific actions.**
- **JDK 1.1 users can generate their own certificates without needing a certificate authority.**
- **IN general, 1.1 javakey is less useful than the others.**
- **JDK1.2 users have very fine-grained control over privileges.**

Some Comparisons of Sign Tools - 2

- Comparing Authenticode to Netscape Object Signing
- Netscape has finer-grained access to resources. Microsoft simply gives an access level instead of dealing with privileges, which is easy for developers.
- Microsoft's Authenticode is more simple for end-user. Netscape prompts to user to grant permission for each privilege request. User has more control but may get exhausted with unrelated security questions.

JavaScript Security Model

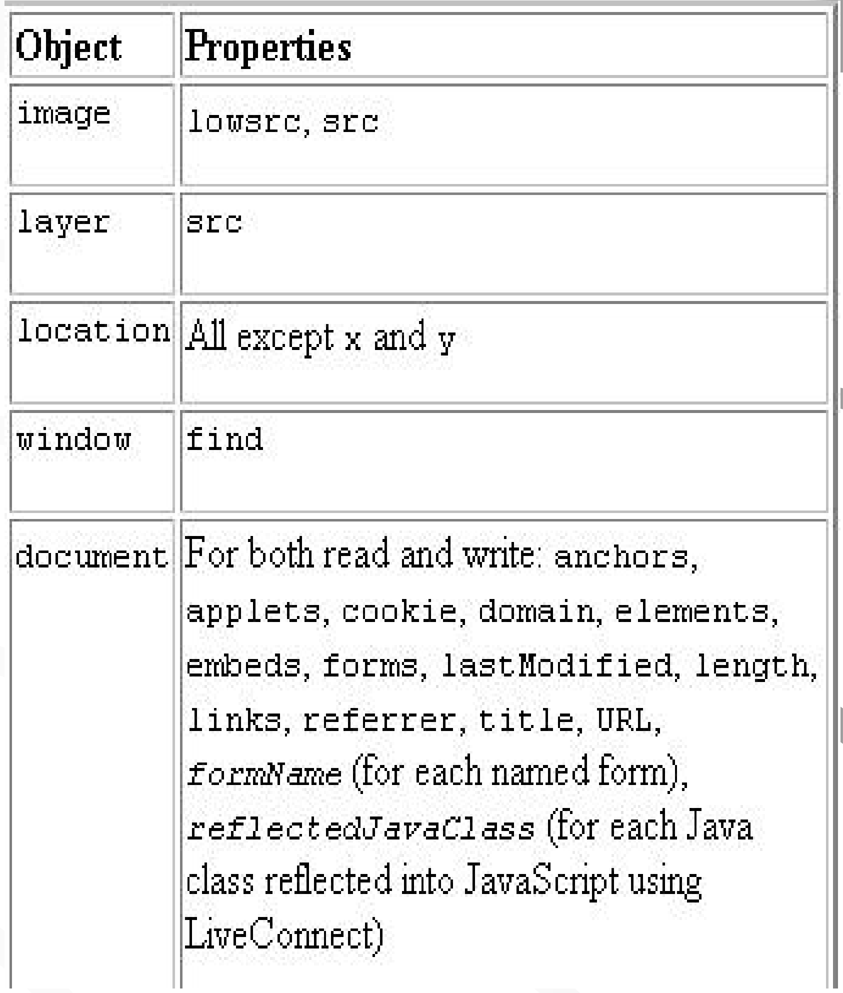
- JavaScript Security depends on the implementation of the browsers.
- There are two security policies in JavaScript:
  - Same Origin Policy: Navigator version 2.0 and later automatically prevents scripts on one server from accessing properties of documents on a different server, including user session histories, directory structures etc..
  - Signed Script Policy: The JavaScript security model for signed scripts is based upon the Java security model for signed objects. The scripts you can sign are inline scripts (those that occur within the SCRIPT tag), event handlers, JavaScript entities, and separate JavaScript files.

JavaScript Security Issues

- Signed Script Policy comes with Netscape version 4.
- In Navigator 3.0, one could use data tainting to get access to additional information.
  - This was very clumsy and almost impossible to use and in particular to make precise
- However, Navigator 4.0 replaces data tainting with the signed script policy. Because signed scripts provide greater security and greater precision than tainting, tainting has been disabled in Communicator 4.x.

Same Origin Policy

- When loading a document from one origin, a script loaded from a different origin cannot get or set certain predefined properties of certain browser and HTML objects in a window or frame.
- Origin is defined as protocol://host, where host may include optional parts of URL including :port, part of an URL.
- Any applets in the document are also subject to origin checks when calling JavaScript.
- The same origin policy is the default policy since Netscape 2.

- Properties subject to origin check
- 

Object	Properties
image	lowsrc, src
layer	src
location	All except x and y
window	find
document	For both read and write: anchors, applets, cookie, domain, elements, embeds, forms, lastModified, length, links, referrer, title, URL, *formName* (for each named form), *reflectedJavaClass* (for each Java class reflected into JavaScript using LiveConnect)

Signed Script Policy-1

- The JavaScript security model for signed scripts is based upon the Java security model for signed objects. The scripts you can sign are inline scripts (those that occur within the SCRIPT tag), event handlers, JavaScript entities, and separate JavaScript files.
- A signed script requests expanded privileges, gaining access to restricted information. It requests these privileges by using LiveConnect and the Java classes referred to as the Java Capabilities API. These classes add facilities to and refine the control provided by the standard Java SecurityManager class.
- Access control decisions are given based on who, called principal, is allowed to do what, called target, and the privileges associated with the principal.

Signed Script Policy-2

- Netscape's Page Signer tool provides signing of scripts. Page Signer associates a digital signature with the scripts on an HTML page.
- A single HTML page can have scripts signed by different principals
- The digital signature is placed in a Java Archive (JAR) file.
- JAR files may include the JavaScript source if one sign a JavaScript file with Page Signer.
  - (If you sign an inline script, event handler, or JavaScript entity, Page Signer stores only the signature and the identifier for the script in the JAR file. If you sign a JavaScript file with Page Signer, it stores the source in the JAR file as well.)

Signed Script Policy-3

- The associated principal allows the user to confirm the validity of the certificate used to sign the script. The user can ensure that the script hasn't been tampered with since it was signed. The user may grant privileges based on the validated identity of the certificate owner and validated integrity of the script.
- An simpler alternative to using the Page Signer tool to sign scripts is to serve them from a secure server. On the browser, scripts act as though they were signed with the public key of that server. There is no need to sign the individual scripts.
- SSL servers are particularly helpful if scripts are dynamically generated on the server side.

Codebase Principals-1

- As does Java, JavaScript supports codebase principals. A codebase principal is a principal derived from the origin of the script rather than from verifying a digital signature of a certificate. Since codebase principals offer weaker security, they are disabled by default in Communicator.
- To enable codebase principals, end users must add the appropriate preference to their Communicator preference file:
- user_pref("signed.applets.codebase_principal_support", true);
- If enabled, when the user accesses the script, a dialog displays similar to the one displayed with signed scripts. The difference is that this dialog asks the user to grant privileges based on the URL and doesn't provide author verification. It says that the script has not been digitally signed and may have been tampered with.

Codebase Principals-2

- If a page includes signed scripts and codebase scripts, and signed.applets.codebase_principal_support is enabled, all of the scripts on that page are treated as though they are unsigned and codebase principals apply.
- Netscape 4 always keeps track of codebase principals to use in enforcement of the same origin security policy whether codebase principals are disabled or enabled.

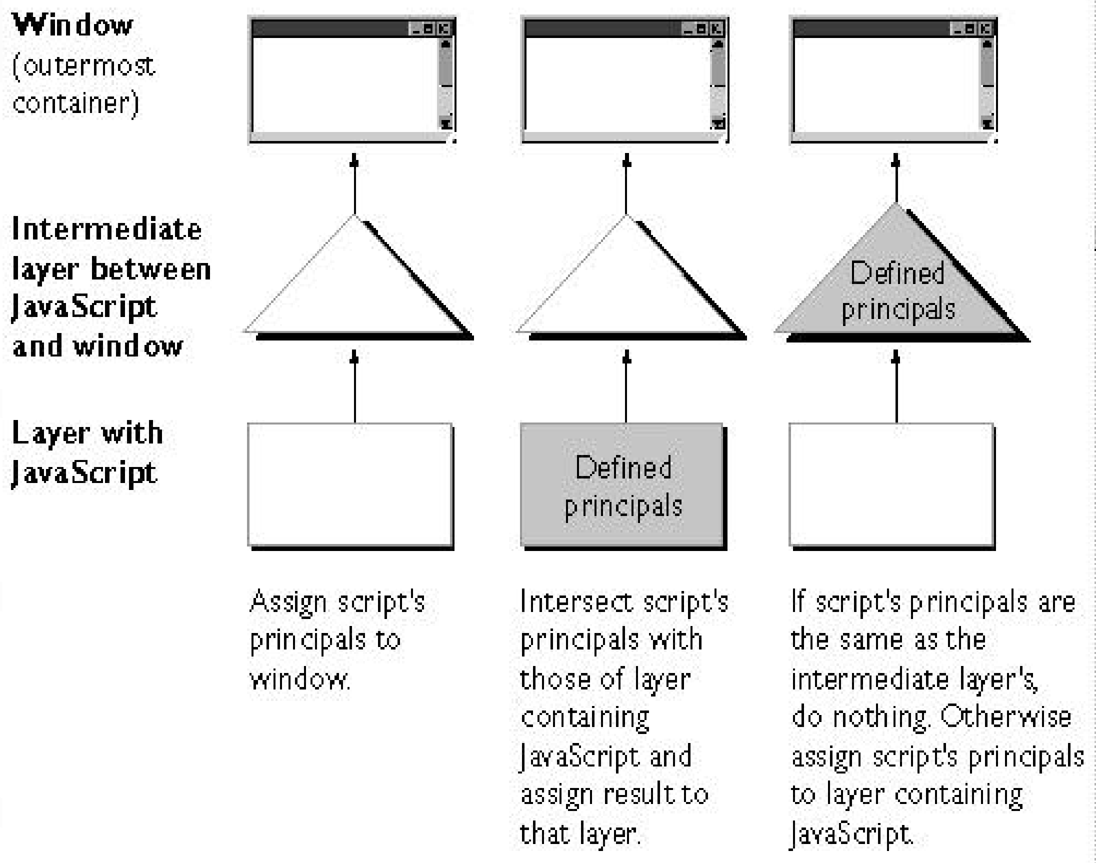
Scripts Signed by Different Principals

- Since JavaScript does not have internal protection mechanisms like Java, e.g., protected and private, and object properties including methods can be changed at runtime, simple signing of scripts is sometimes not secure enough.
- Different scripts from different principals on the same page can change each other's behaviour.
- Security of the JavaScript is ensured by the following assumption:
- Mixed scripts on an HTML page operate as if they were all signed by the intersection of the principals that signed each script.
- For example, assume principals A and B have signed one script, but only principal A signed another script. In this case, a page with both scripts acts as if it were signed by only A.

Principals of Windows and Layers

- In order to protect signed scripts from tampering, Navigator 4.0 adds a new set of checks at the container level, where a container is a window or layer.
- A script, which wants to access the properties of a signed container, should be signed by a superset of principals that signed the container.
- If some scripts in a layer are signed by different principals, the special container checks apply to the layer.

Figure 1 Assigning principals to layers.



Window (outermost container)
Intermediate layer between JavaScript and window
Layer with JavaScript
Defined principals
Defined principals
Assign script's principals to window.
Intersect script's principals with those of layer containing JavaScript and assign result to that layer.
If script's principals are the same as the intermediate layer's, do nothing. Otherwise assign script's principals to layer containing JavaScript.

Determining Container Principals

- Following structure is used (by Communicator) :
- 1. If this is the first script that has been seen on the page, assign this script's principals to be the principals for the window.
- 2. If the innermost container (the container directly including the script) has defined principals, intersect the current script's principals with the container's principals and assign the result to be the principals for the container. If the two sets of principals are not equal, intersecting the sets reduces the number of principals associated with the container. Done.
- 3. Else, find the innermost container that has defined principals. If the principals of the script are the same as the principals of that container, leave the principals as is. Else assign the current script's principals to be the principals of the container.

Identifying Signed Scripts

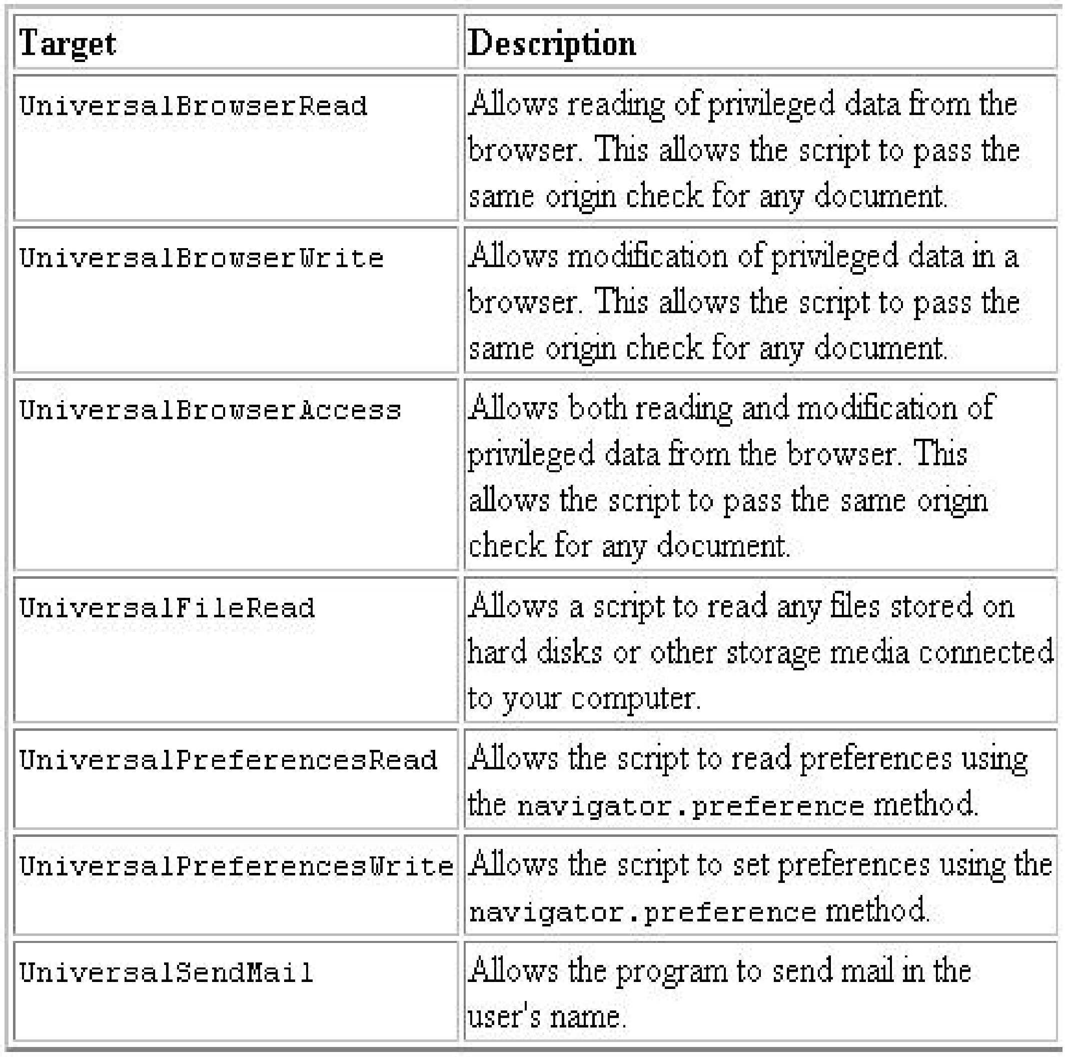
- **All signed scripts (inline script, event handler, JavaScript file, or JavaScript entity) require a SCRIPT tag's ARCHIVE attribute whose value is the name of the JAR file containing the digital signature ; <SCRIPT ARCHIVE="myArchive.jar" ID="a"> … </SCRIPT>**
- **To sign an inline script, you add both an ARCHIVE attribute and an ID attribute to the SCRIPT tag for the script you want to sign**
- **To sign an event handler, you add an(only one) ID attribute for the event handler without specifying the ARCHIVE to the tag containing the event handler; <INPUT TYPE="button" VALUE="OK" onClick="alert('...') onClick="alert('A signed script')" ID="b">**
- **To sign a JavaScript entity, you do not do anything special to the entity. Instead, the HTML page must also contain a signed inline script preceding the JavaScript entity.**

Using Expanded Privileges

- Signed scripts use calls to Netscape's Java security classes to request expanded privileges.(like Java signed objects) For example,
  netscape.security.PrivilegeManager.enablePrivilege("UniversalSendMail")
- When the script calls this function, the signature is verified, and if the signature is valid, expanded privileges can be granted. If necessary, a dialog displays information about the application's author, and gives the user the option to grant or deny expanded privileges.
- Privileges are granted only in the scope of the requesting function and only after the request has been granted in that function. This scope includes any functions called by the requesting function. When the script leaves the requesting function, privileges no longer apply.

Targets

- **A target typically represents one or more system resources, such as reading files stored on a local disk or sending email o your behalf.**
- **The Privilege Manager, with the aid of the Communicator client, keeps track of which principals are allowed to acces which targets at any given tim**
- **The Privilege Manager enforces a distinction between granting a privilege and enabling a privilege. A script that has been granted a privilege has a potential power that is not yet turned on.**



Target	Description
`UniversalBrowserRead`	Allows reading of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserWrite`	Allows modification of privileged data in a browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserAccess`	Allows both reading and modification of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalFileRead`	Allows a script to read any files stored on hard disks or other storage media connected to your computer.
`UniversalPreferencesRead`	Allows the script to read preferences using the `navigator.preference` method.
`UniversalPreferencesWrite`	Allows the script to set preferences using the `navigator.preference` method.
`UniversalSendMail`	Allows the program to send mail in the user's name.

Targets-2

- The followings are some samples of system targets and the JavaScript methods that require privileges to check them:
- UniversalFileRead:Setting a file upload widget
- UniversalSendMail:Submitting a form to a mailto
- UniversalBrowserRead:Using an about: URL other than about:blank
- UniversalBrowserWrite:Setting any property of event object
- UniversalBrowserRead: Getting the value of the data property DragDrop event
- UniversalBrowserRead: Getting the value of any property of history object
- UniversalPreferencesRead/Write: Getting setting the value of a preference of navigatorobject using the preference method

Importing and Exporting Functions

- **You might want to provide interfaces to call into secure containers (windows and layers). To do so, you use the import and export statements.**
- **Exporting a function name makes it available to be imported by scripts outside the container without being subject to a container test.Importing a function into your scope creates a new function of the same name as the imported function. Calling that function calls the corresponding function from the secure container.**
- **One should be very careful to not inadvertently allow access to an attacker**

Weaknesses in the JavaScript Model

- If one have signed scripts in pages he has posted to his site, it is possible to copy the JAR file from his site and post it on another site. As long as the signed scripts themselves are not altered, the scripts will continue to operate under his signature. “Programmer should force scripts to work only from his side.”

- When you export functions from your signed script, you are in effect transferring any trust the user has placed in you to any script that calls your functions.This means you have a responsibility to ensure that you are not exporting interfaces that can be used in ways you do not want.

Signing Scripts

- For any script to be granted expanded privileges, all scripts on the same HTML page or layer must be signed. If you use layers, you can have both signed and unsigned scripts as long as you keep them in separate layers.
- The Netscape Signing Tool (signtool on the command line) is a stand-alone command-line tool that creates digital signatures and uses the Java Archive (JAR) format to associate them with files in a directory. Previous versions of the signtool are known as zigbert and signPages.
- The signtool script extracts scripts from HTML files, signs them, and places their digital signatures in the archive specified by the ARCHIVE attribute in the SCRIPT tag from the HTML files. It also takes care of copying external JavaScript files loaded by the SRC attribute of the SCRIPT tag.

Signing Scripts-2

- The SCRIPT tags in the HTML pages can specify more than one JAR file; if so, signtool creates as many JAR files as it needs.
- The signtool utility program helps to deal with certificate databases; for example signtool -L -d my_test_dir list the certificates stored in the certificate database *.db files in the specified directory. signtool -l -k nickname verifies an object-signing certificate with the specified nickname.
- To sign a file using the Netscape Signing Tool;
- 1. Create an empty directory: % mkdir signdir
- 2. Put script files into it
- 3. Specify the name of your object-signing certificate and sign the directory:
- % signtool -k MySignCert -Z testjar.jar signdir

Signing Scripts-3

- **Output will be**
- **using key "MySignCert"**
- **using certificate directory: /home/loginname/.netscape**
- **Generating signdir/META-INF/manifest.mf file..**
- **--> test.f**
- **adding signdir/test.f to testjar.jar**
- **Generating signtool.sf file..**
- **Enter Password or Pin for "Communicator Certificate DB":XXXXX**
- **adding signdir/META-INF/manifest.mf to testjar.jar**
- **adding signdir/META-INF/signtool.sf to testjar.jar**
- **adding signdir/META-INF/signtool.rsa to testjar.jar**
- **tree "signdir" signed successfully**

Signing Scripts-4

- 4. Test the archive you just created:% signtool -v testjar.jar
- using certificate directory: /home/loginname/.netscape
- archive "testjar.jar" has passed crypto verification.
- status path
- ------------ -------------------
- verified test.f

- To learn more details about the signtool, visit the URL:
- http://developer.netscape.com/docs/manuals/signedobj/signtool/index.htm
- http://developer.netscape.com/docs/manuals/signedobj/zigbert/index.htm