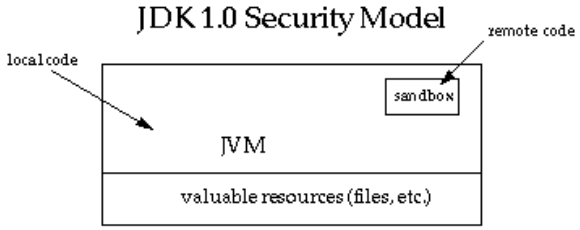
LIYANA ARACHCHIGE RANIL

SHORT NOTES ON JAVA SECURITY – PLATFORM SECURITY AND OTHER

1. JAVA security includes two things
    a. Provide the JAVA platform as a secure , readily built platform on which to run JAVA enabled applications in a secure manner
    b. Provide security tools and services implemented in JAVA
2. The ORIGINAL SAND BOX model introduced in JDK 1.0 SECURITY MODEL is very much restricted. All the **LOCAL code (such as file system)** was **TRUSTED** while downloaded **REMOTE code (such as applets) is NOT TRUSTED** and can access only limited resources provided inside the SANDBOX



3. The sandbox (JDK 1.0) model was deployed through the JDK and was generally adopted by all applications built with JDK 1.0, including JAVA enabled browsers
4. Overall security is enforced through number of mechanisms.
    a. **LANGUAGE LEVEL (JVM)**
        i. The language is designed to be **TYPE – SAFE**
        ii. Language has **AUTOMATIC MEMORY MANAGEMENT**
        iii. Language has **AUTOMATIC GARBAGE COLLECTION**

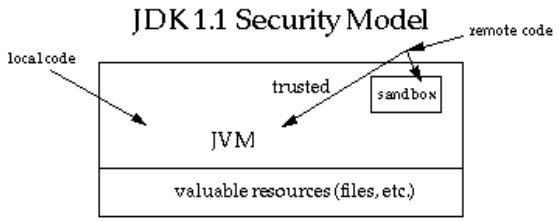
iv. Language has **RANGE CHECKING** on Strings and ARRAYS

b. **COMPILER AND BYTECODE LEVEL**
   i. Compiler and Byte code verifier **ensures that only legitimate** JAVA byte codes are executed
   ii. This would guarantee the language safely at RUN TIME

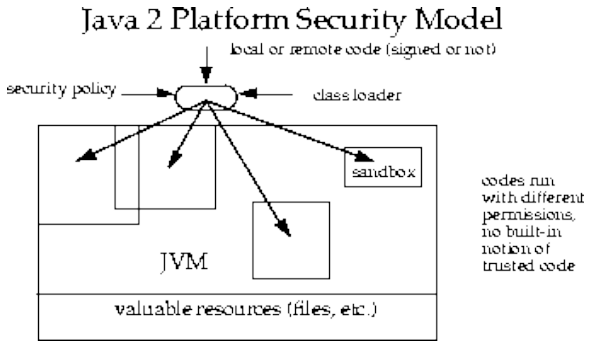
c. **CLASS LOADERS LEVEL**
   i. Class loaders define a local name space, which can be used to ensure that an un trusted applet cannot interfere with the running of the other programs

d. **ACCESS TO CRITICAL SYSTEM RESOURCES ARE PROHIBITED**
   i. Access to crucial system resources is mediated by JVM and is checked in advanced by a SECURITY MANAGER

5. JDK 1.1 introduced the concept of **SIGNED APPLETS**
6. In that a correctly **digitally signed applet is treated as If it is trusted local code** if the key is recognized as trusted by the end system that received the applet
7. SIGNED applets together with their signature are delivered in the JAR format
8. **In JDK 1.1 UNSIGNED applets still run in a SANDBOX**



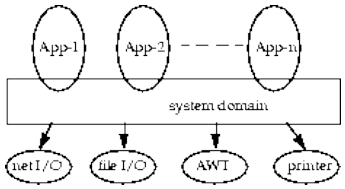
9. New **JAVA 2 PLATFORM SECURITY ARCHITECTURE (1.2)**



10. New **JAVA 2 PLATFORM SECURITY provides**
    a. **Fine grained control**
    b. **Easily configurable security policy**
    c. **Easily accessible access control structure**
    d. **Extension of security checked to all Java programs , including applications as well as applets**

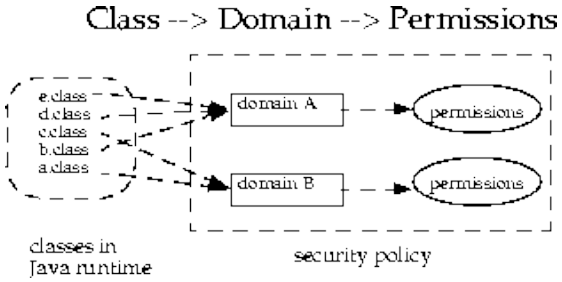
11. This **fine grained** security capability was with JDK from its **beginning**, but to realize it users had to **subclass** java **SecurityManager** and **ClassLoader** classes. **HotJava** browser was one of those which uses old JDK and provide fine grained security features
12. Up to **JDK 1.1** in order to create a **NEW ACCESS PERMISSION**, you had to add a new **CHECK** method to the **SECURITYMANAGER** class. The new architecture allows **TYPED PERMISSIONS** and **AUTOMATIC HANDLING** of all permissions. **No new methods to be created in SecurityManager class**
13. There is **no longer** a build in concept that **all LOCAL code** trusted. Instead local code is subjected to the same security control as applets

14. It is possible to declare that the policy on local code (or remote code) be the most liberal and enabling such code to effectively run as totally trusted
15. The same principle applies to SIGNED APPLETS and any JAVA APPLICATION
16. A **PROTECTION DOMAIN** is set of objects that are currently directly accessible by a **PRINCIPAL** where principle is an **ENTITY** in the computer system to which **PERMISSIONS** are granted
17. A **PROTECTION DOMAIN** is a convenient principle **in grouping** and **isolation** between **units of protection**
18. **PROTECTION DOMAIN** falls in to **two categories**
    a. **SYSTEM DOMAIN**
    b. **APPLICATION DOMAIN**
    c.



19. **PROTECTION DOMAIN(S)** are determined by the **POLICY currently in effect**

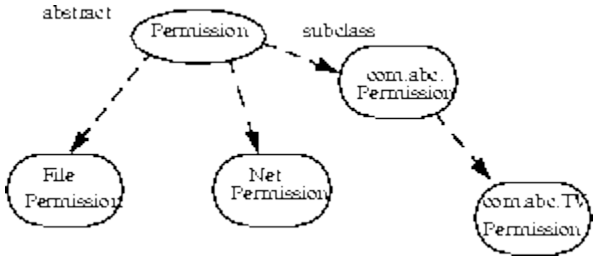
a.



20. A **THREAD of EXECUTION** may occur completely within a single PROTECTION DOMAIN or may involve an APPLICATION DOMAIN and also a SYSTEM DOMAIN

21. APPLICATION DOMAIN **DOES NOT** gain additional permission by calling the SYSTEM DOMAIN
22. Today all the code shipped as **part of JAVA 2 SDK** is considered **SYSTEM CODE**
23. when **SYSTEM DOMAIN** invokes a method from an **APPLICATION DOMAIN** the effective access rights are the same as current rights enabled in the APPLICATION DOMAIN
24. A **less POWERFULL domain** can not gain additional permissions as a result of **CALLING** or **BEING CALLED** by a more powerful domain
25. A simple rule for calculating permissions ,
    a. The permission of an execution thread is considered to be the INTERSECTION of the permissions of all PROTECTION DOMAINS traversed by the execution thread
    b. When a piece of code calls the **doPriviledge** method , the permission set of the execution thread is considered to include a permission , if it is allowed by the said code's protection domain and by all protection domains that are called or entered directly or indirectly subsequently
    c. A caller can be marked as being **PRIVILEDGED** when it calls the **doPrivileged**
26. By Default JAVA programs run without a SECURITY MANGER , to use SECURITY MANAGER you need to Install it before running the program
27. To create a new type of permission ,
    a. Extend the java.security.Permission class and create a new Permission class
    b. Create another Permission class "TVPermission" which extends the fist created permission

c. **com.abc.Permission extends java.security.Permission ; com.abc.TVPermission extends com.abc.Permission**

d. 

e. Add an entry in policy file to give permission to users who want to allow this new type of permission

f. **grant codebase "http://java.sun.com/" { permission com.abc.TVPermission "channel-5", "watch"; }**

g. In Application code , when checking to see if a permission should be granted , call **AccessControlle's checkPermission** method

h. TVPermission tvPerm = new TVPermission("cahnnel-5","watch")
AccessController.checkPermission(tvPerm);

28. **AccessController** is used for three cases ,
    a. To decide whether an access to a critical system resource is to be allowed or denied based on the current security policy in effect
    b. To mark code as being PRIVILEDGE thus effecting subsequent access determinations
    c. To obtain a snapshot of current calling security context to be used in different a context

29. **FilePermission perm = new FilePermission("path/file","read"); AccessController.checkPermission(perm);**

30. JAVA has a class loader hierarchy

31. The ROOT of the class loader is java.lang.ClassLoader (Abstract), this was introduced in JDK 1.0.
32. SecureClassLoader was introduced in JDK 1.2 (JAVA 2)
33. URLClassLoader is a subclass of the SecureClassLoader
34. Each class is loaded by it's classloader , there is a PRIMODIAL class loader that BOOTSTRAPS the class loading process. Primordial class loader is written in a NATIVE language such as C
35. The base JAVA Classes (java.*) which are essential for the correct functioning of the Java Virtual Machine and runtime have a class loader that is NULL
36. When a class loader is asked to LOAD a class, this class loader either loads a class itself or it can ask another class loader to do so. Meaning the FIRST class loader DELEGATES the class loading to SECOND class loader
37. The default implementation of the JAVA 2 SDK Class Loader method for loading classes
    a. Check if the class has already been loaded
    b. If the current class loader has a specified delegation parent, delegate to the parent to try to load this class. If there is no parent then delegate to PREMODIAL class loader
    c. Call a customized method to find the class elsewhere (Developers need to do this)
38. Same class must not be loaded by the same class loader more than once and this is critical for type safety
39. When loading the **FIRST class of an APPLICATION** , a new instance of **URLClassLoader** is used
40. When loading the **FISRT class of an APPLET** , a new instance of **AppletClassLoader** is used

41. When **java.lang.Class.ForName** is directly called , **PRIMODIAL** class loader is used
42. If the request to load a class is triggered by a reference to it from an existing class, the class loader for the existing class is asked to load the class
43. CLASS LOADER always looks at the STANDARD classes first
44. Class loaders keep namespaces of different APPLETS separate
45. Whenever and APPLET is run, a SECURITYMANAGER is installed
46. SECURITY MANAGER represents the concept of a central point of Access Control , while ACCESS CONTROLLER implements a particular access control algorithm
47. To protect ONE method in ALL instances , use SECURITY MANAGER while to PROTECT a REFERENCE to an INDIVIDUAL instance , use GUARDEDOBJECT

48.

```
package com.scea.sec.guadedobj;

import java.io.FilePermission;
import java.security.AccessControlException;
import java.security.GuardedObject;

public class GuardedObjectTest {
    public static void main(String[] args){
        GuardedObject go = new GuardedObject(new TestObject(),
                new FilePermission(
                        "C:\\work\\workspaces\\eclipse-scea-ws\\java-guarded-object\\TestFile.txt",
                        "write"));
        try{
            Object o = go.getObject();
            System.out.println("Can access");
        }catch(AccessControlException ace){
            System.out.println("Can not access..");
        }
    }
}
```

49. You can write your OWN GUADE or use permission class since it is already implementing the GUARGE interface

50. A SIGNEDOBJECT instance acts as a WRAPPER around an instance of another class. A SIGNEDOBJECT instance contains the SERIALIZED representation of the WRAPPED OBJECT. Along with the signature information necessary to validate the WRAPPED OBJECT'S IDENTITY

51.

```
Signature signingEngine = Signature.getInstance(algorithm,
                                                provider);
SignedObject so = new SignedObject(myobject, signingKey,
                                   signingEngine);
```

52. JAR signing is digitally signing jar archive files

53. Classes within the same archive can be signed with different keys , and a class can be unsigned , signed with a different keys ,signed with one key or signed with multiple keys
54. The other resources in a JAR file such as graphic images , audio clips can also be signed or unsigned
55. APPLETS are not allowed to do below things if the applet is not trusted
    a. File related operations such as check for the existence of a file , read file , write file , rename , create a dir , list files , check file type , check the timestamp , check the file size etc
    b. Can not open network connection to other host other than the one from which is it loaded
    c. Can not read certain system properties users folder etc
    d. Can not load libraries or define native methods
    e. Can not start any program on the host that's executing it
    f. Windows that applets bring up are different than the system windows
56. APPLETS can create many number of threads consuming system resources at a larger scale and bringing down the system to a resource scare situation
57. Byte Code verifier checks
    a. JAVA byte codes contain only valid **INSTRUCTIONS** and **REGISTERS** to use [it is an error to load from an uninitialized register etc. The argument of an instruction are always of the type expected by the instruction]
    b. Byte code verification runs by default , but it can be turned off by setting –Xverify:none

c. On JAVA 2 systems , PRIMODIAL class loader is permitted to OMIT byte code verification of the classes loaded from the CLASSPATH

d. By DEFAULT the code loaded from the CLASSPATH is not verified. Hence you can change one class property value to private from public which is accessed in another class and allow illegal access to the said class.

```
package com.scea.bytecodechk;

public class Surfer {

    public String test;

    public Surfer(){
        test = "I am attacked";
    }
}
```

e.

```
package com.scea.bytecodechk;

public class Attacker {

    public static void main(String[] args) {
        Surfer surfer = new Surfer();
        System.out.println(surfer.test);
    }

}
```

f.

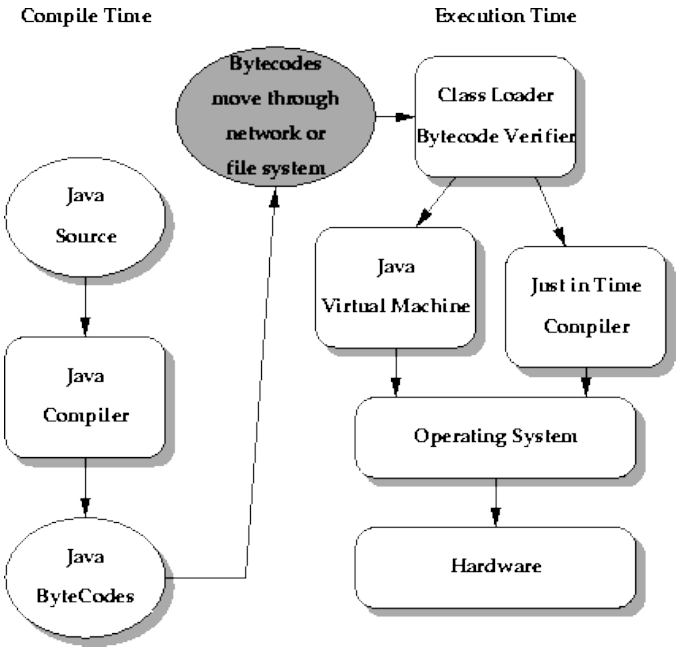
```
package com.scea.bytecodechk;

public class Surfer {

    private String test;

    public Surfer(){
        test = "I am attacked";
    }
}
```

g.

h.

i. Byte code verifier is AUTOMATICALLY applied to RECOMPILED code, for UNMODIFIED code it does not apply.
j. Code does not **OVERFLOW/UNDERFLOW OPERAND STACK** [an instruction never pops an argument off an EMPTY STACK , not pushes a result on a FULL STACK]
k. Does not convert data types illegally or forge pointers
l. Accesses objects as correct type
m. Method calls use correct number and types of arguments
n. Reference to other classes use legal names
o. Does **not VIOLATE access RESTRICTIONS**
p. Class files contain CORRECT FORMAT
q. No illegal data conversion occur (Casts). Although many checks are done my VERIFIER , some are DEFERRED until RUNTIME

58. BYTE CODE VERIFICATION help protect the underline machine of illegal access , crashes etc

59. 

JAVA WEB STRT

60. **JAVA WEB START** applications run in a **RESTRICTED ENVIRONMENT** known as **SAND BOX**
61. In this **SANDBOX (WEB START)** , it protects users against **MALICIOUS** code that could effect **LOCAL** files
62. Also , protects **ENTERPRISES** against code that could attempt to **ACCESS** or **DESTROY** data on **NETWORKS**
63. **UNSIGNED JAR** files launched by **JAVA WEB START** remain in this **SAND BOX**, those **CAN NOT** access **LOCAL** file system or **NETWORK**
64. **UNSIGNED JAR** files launched by **JAVA WEB SRART can** use **FileOpenService** , **FileSaveService** to **REQUEST** user permission to read files from the users system. But this happens at the **discretion** of the user.

65. If a **JAVA WEEB START** application request for <**all-permissions**/> under <security> the **JAR file MUST BE SIGNED**
66. When user FIRST run an APPLICATION as a JAVA WEB START **signed** JAR file , JAVA WEB START opens a DIALOGUE box displaying the APPLICATIONS ORIGIN bases on the SIGNER'S CERTIFICATE
67. With **JAVA WEB START** , a single application can be placed on a **WEB SERVER** for deployment to a wide variety of platforms , LINUX , WINDOWS , SOLARIS etc
68. An APPLET can also be used to RUN using JAVA WEB START
69. JNLP defines how JAVA WEB START applications are launched
70. JNLP stands for JAVA NETWORK LOANCHING PROTOCOLE
71. JNLP is a web centric provisioning PROTOCOL and APPLICATION ENVIRONMENT for web deployed java 2 technology based applications
72. The main concepts of JNLP specification are
    a. Web CENTRIC application model with NO INSTALLATION PHASE , this provides transparent and incremental updates , incremental downloading of the application is also provided
    b. A provisioning PROTOCOL that DESCRIBES how to package and application on a WEB SERVER. The KEY component in this PROVISIONING is the JNLP file which describes HOW to download and launch the application
    c. Specify a STANDARD EXECUTION environment for the application. The execution environment includes both a SAFE environment where ACCESS to the LOCAL disk and

the NETWORK is restricted for UNSTRUSTED applications , and UNRESTRICTED environment for TRUSTED applications

d. The **RESTRICTED** environment is SIMILAR to the **APPLET SANDBOX** , but **EXTENDED** with **ADDITIONAL FUNCTIONALITY** via **JNLP API**

73. A **JNLP client** is an **APPLICATION** or **SERVICE** that can launch application on a client system from **RESOURCES** hosted across the **network**
74. A JNLP file does not contain any binary data , instead it contains URL(S) that point to all BINARY DATA
75. Most commonly a **JNLP** file describes an **APPLICATION**. A JNLP file of this type is referred as **APPLICATION DESCRIPTOR**. It specifies the JAR files that the application consist of , the java 2 platform it requires , optional packages that it depends on, runtime prams , other system props etc
76. A **JNLP** file can also refer to **OTHER JNLP** files. Such fillies are called **EXTENSION DESCRIPTERS**
77. **EXTENSION DESCRIPTORS** usually describes a component that must be used in order to run the application
78. The resources described in the **EXTENSION DESCRIPTOR** becomes part of the **CLASSPATH** for the application

79.

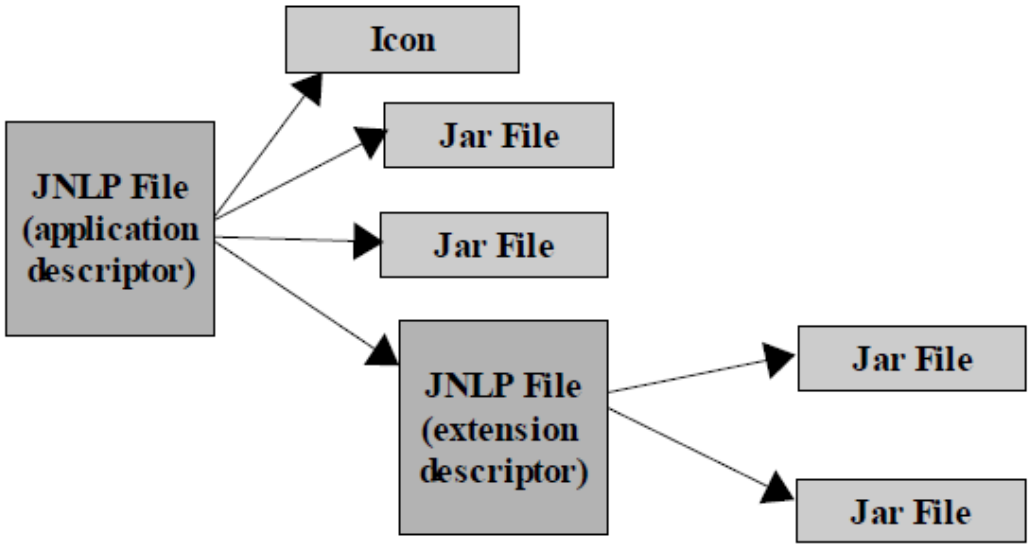


Figure 1: JNLP File and External Resources

80. A JNLP client can download THREE different kind of RESOURCE , JAR Files , IMAGES and Extension Descriptors
81. Applications launched with JNLP do not run in a BROWSER WINDOW. But are instead separate applications that are run on separate JVMs
82. A JNLP file can contain SYSTEM PROPERTY settings as well. Properties defined like this are available via System.getProperty(key).

83.

```
<property name="key" value="overwritten"/>
<property name="key" value="used"/>
```

84. A JNLP file may contain two kind of CODE RESOURCES, **jar** and **nativlib**

85. nativelib , specifies a JAR file that contains NATIVE LIBRARIES

86.

```
<resources>
  <jar href="lib/app.jar" version="3.2" main="true"/>
</resources>
<resources os="Windows"/>
  <nativelib href="lib/windows/corelibs.jar"/>
</resources>
<resources os="SunOS" arch="SPARC">
  <nativelib href="lib/solaris/corelibs.jar"/>
</resources>
```

87. **jar** and **nativelib**(s) can either be downloaded EARGERLY or LAZYLY

88.

```
<jar href="sound.jar" download="lazy"/>
<nativelib href="native-sound.jar" download="eager"/>
```

89. An application launched with JNLP client MUST be run in an environment according to the specification

a. A preconfigured set of proxies for HTTP communication
b. A RESTRICTED environment for UNTRUSTED APPLICATIONS and TWO EXECUTION environments for TRUSTED APPLICATIONS. The trusted environment are "**all-permissions**" and "**j2ee-application-client**" environment
c. A BASIC set of services that are available through the **JAVAX.JNLP** package
    i. **BasicService –** provides set of method for querying and interacting wit the environment. Allows opening of a default browser window etc. Even for applications that are running in a restricted environment
    ii. **ClipBoardService –** provides access to shared – system wide clipboard. Even for applications that are running in a restricted environment
    iii. **DownloadService –** Allows application to control how its own resources are cached
    iv. **FileOpenService –** Provides method for importing files from the LOCAL FILE SYSTEM. Even for applications which are running in a restricted environment
    v. **FileSaveService –** Provides method for exporting files TO DISK. Even for the applications that are running in the restricted environment. Provides the SAME level of DISK ACCESS to potentially UNTRUSTED web deployed applications that a WEB BROWSER provides
    vi. **PrintService –** Provides access to printing, even for application running in the restricted environment.

vii. **PersistanceService –** provides method for storing DATA locally like in Cookies. Even for the applications running in restricted environment.

d. Ability to DOWNLOAD resources lazily as the application executes
e. Validating signing of the JAR files

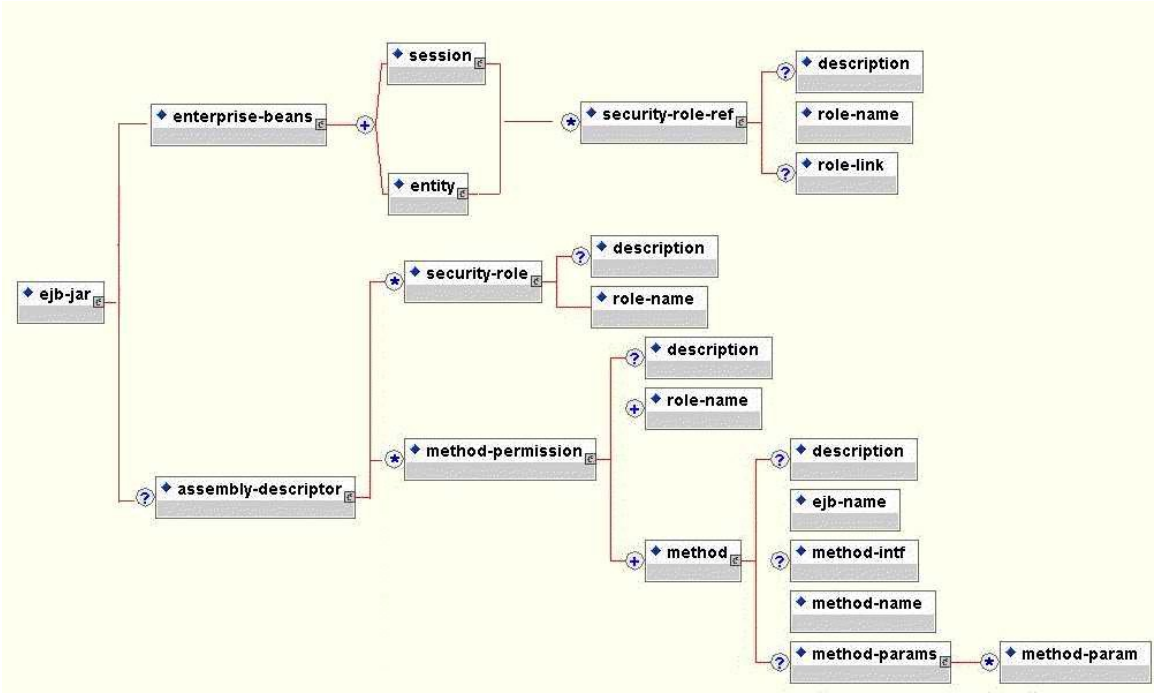
90. An application launched by a JNLP client is considered signed if and only if ,
    a. All the JAR files are SIGNED (both jar and nativlib) and can be verified. A SINGLE CERITFICATE must be used to sign each JAR file
91. The JNLP client must check a JAR file signing information BEFORE it is USED (i.e before a CLASS file or another resource is RETRIEVED from it)
92. A JNLP file can OPTIONALITY BE SIGNED as well
93. if a JNLP is running in an UNTRUSTED environment ,
    a. All JAR files specified in the RESOURCES elements of the JNLP file MUST BE downloaded from the SAME HOST
    b. No **NATIVLIB** element can be used
    c. The application MUST BE run with a SECURITYMANAGER installed
    d. The JNLP file can REQUEST EXTENSIONS and JRE from any HOST, An application CAN NOT make a SOCKET connection back to any of the HOSTS where JREs or EXTENSIONS are DOWNLOADED
94. **JNLP API** is available to all the APPLICATION , whether those are TRUSTED or NOT
95. Services such as BASIC SERVICE , DOWNLOAD SERVICE ,FILE OPEN SERVICE (Allows un-trusted application to import files from the LOCAL DISK), FILE SAVE SERVICE (Allows un-trusted

application to export files to the LOCAL DISK) ,CLIPBOARD SERVICE (Allows un-trusted application to access CLIPBOAD),PRINT SERVICE (Allow un-trusted application to access print service),PERSISTANCE SERVICE and EXTENSION INSTALLER SERVICE

96. JNLP API provides following – Summary
    a. Loading and Saving Files
    b. Accessing the CLIPBOARD
    c. Printing
    d. Downloading a File
    e. Displaying a Document in the default browser
    f. Storing and retrieving persistence configuration information

97. EJB 2.1 security has following capabilities
    a. Declaring **METHOD PERMISSIONS** [Programmatic and Declarative]
    b. Mapping Roles to J2EE users and groups
98. EJB 2.1 has both **PROGRAMATIC** and **DECLARATIVE SECURITY** capability
99. **DECLARATIVE security** can be at the level **of allowing or denying METHOD permissions**. If there is a **conditional security check inside a method itself then PROGRAMMATIC security MUST BE USED**
100. DECLARATIVE SECURITY is done in the ejb-jar.xml , using “assembly-descriptor”,”security-role”,”role-name”,”method-permission”,”method”,”ejb-name”,”method-name” etc.

101. **ejb –jar.xml** security elements



102. **EJB-2.1 MUST have a DEPLOYMENT DESCRIPTOR**

103. EJB 2.1 , PROGRAMMATIC SECURITY was achieved **using isCallerInRole(rolename)** , **getCallerPrincipal()** methods

104. EJB 3.0 has made DEPLOYMENT DESCRIPTORS completely OPTIONAL

105. EJB 3.0 users ANNOTATIONS for SECURITY declarations

106. Annotations used are **@RunAs [can only be applied at the Class level],@RollesAllowed,@PermitAll,@DenyAll,@DeclairRoles**

107. Annotations are considered to be DECLARATIVE
108. Whenever DEPLOYER needs to OVERWRITE annotated values for SECURITY , he may use DEPLOYMENT DESCRIPTOR for that
109. PROGRAMMATIC security in EJB 3.0 is achieved using **getCallerPrincipal()** , **isCallerInRole(rolename)** methods from the **SESSION CONTEXT**
110. **SESSIONCONTEXT** is injected to a SESSION BEAN via **DEPENDENCY INJECTION** (@Resource) in EJB 3.0
111. **AUTHENTICATION** for EJB application client is done using **IOR** (**Interoparable Object Reference**) authentication
112. **IOR** protocol was originally created for **CORBA (COMMON OBJECT REQUEST BROKER) ,** but **ALL JAVA EE compliant CONTAINERS** support it
113. **IOR** (S) are configured in **VENDOR SPECIFIC XML** files
114. **Only AUTHENTICATION** method available for **JAVA EJB clients complying to IOR** authentication is **USERNAME_PASSWORD**
115. In WEB TIER , both PROGRAMMATIC and DECLARATIVE security is SUPPORTED
116. For **PROGRAMMATIC SECURITY in WEB TIRE**, it users **getUserPrincipal()** , **isUserInRole(rolename)** from **HTTPSERVLETREQUEST** interface
117. Above METHOD could be USED in either SERVLET or JSPs
118. The **@RunAs** can also be used with **SERVLET(S)** and make SERVLET run as the GIVEN ROLE
119. There is **no EQUIVALENT** for **WEB-TIER declarative authorization** compared to **BUSINESS TIRE**. **WEB TIER has to use WEB.XML for deployment descriptor** instead

120. in WEB.XML , use “security-constraint”,”web-resource-collection”,”auth-constraint” for DECLARATIVE SECURITY
121. Web Tier applications have **number** of **AUTHENTICATION** mechanisms compared to EJB Client AUTHENTICATION mechanism which is IOR where only USERNAME_PASSWORD mechanism is available
122. WEB TIER authentication mechanisms are **BASIC**,**FORM**,**CLIENT CERT**,**DIGEST**
123. With **DECLARATIVE** authorization , **CONTAINER**(S) do the authorization while **PROGRAMMATIC** authorization , **EJB**(S) do the authorization
124. WEB SERVICE SECURITY is defined in WS-Security standards which is controlled by OASIS
125. WS-Security address **AUTHENTICATION and AUTHORIZATION (Using CREDENTIALS), MESSAGE LEVEL DATA INTERGRITY (USING XML SIGNATURES) , MESSAGE LEVEL AND TRANSPORT CONFIDENTIALITY (USING ENCRYPTION)**
126. In J2EE5 support for WEB SERVICE SECURITY is not COMPLETE
127. J2EE5 compliant servers with an implementation of the XML/HTTP binding MUST support HTTP basic AUTHENTICATION using two properties to configure AUTHENTICATION information (**javax.xml.ws.security.auth.username**) and (**javax.xml.ws.security.auth.password**)
128. In addition to that, TRANSPORT level ENCRYPTION is also supported.
129. MESSAGE LEVEL encryption is not supported or required by standard implementation in J2EE5

130. Usually WS-Security talks about AUTHENTICATION , SINGNATURES and ENCRYPTION

131. WS-SECURITY provides and INFINITE number of ways to validate (Authentication) a user. The specification addresses

a. Username/Password
b. PKI through X.509 Certificates
c. Kerberos

132. Username/Password

```
<wsse:UsernameToken>
          <wsse:Username>scott</wsse:Username>
          <wsse:PasswordType="wsse:PasswordText">password</wsse:Password>
</wsse:UsernameToken>
```

133. PKI through X.509 Certificates

```
<wsse:BinarySecurityToken
   ValueType="wsse:X509v3"
   EncodingType="wsse:Base64Binary"
Id="SecurityToken-f49bd662-59a0-401a-ab23-
1aa12764184f">MIIHdjCCB...</wsse:BinarySecurityToken>
```

134. SIGNING – When a message is signed it is nearly impossible to tamper with the message. Message signing does not PREVENT external parties reading the message

ENCRYPT the message not to expose the content ,

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
   <soap:Header
      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
      xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
      <wsu:Timestamp>
         <wsu:Created
            wsu:Id="Id-3beeb885-16a4-4b65-b14c-0cfe6ad26800"
            >2002-08-22T00:26:15Z</wsu:Created>
         <wsu:Expires
            wsu:Id="Id-10c46143-cb53-4a8e-9e83-ef374e40aa54"
            >2002-08-22T00:31:15Z</wsu:Expires>
      </wsu:Timestamp>
      <wsse:Security soap:mustUnderstand="1" >
         <xenc:ReferenceList>
            <xenc:DataReference
```

```
      URI="#EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51" />
         </xenc:ReferenceList>
         <xenc:ReferenceList>
            <xenc:DataReference
      URI="#EncryptedContent-666b184a-a388-46cc-a9e3-06583b9d43b6" />
         </xenc:ReferenceList>
      </wsse:Security>
   </soap:Header>
   <soap:Body>
      <xenc:EncryptedData
         Id="EncryptedContent-f6f50b24-3458-41d3-aac4-390f476f2e51"
         Type="http://www.w3.org/2001/04/xmlenc#Content">
         <xenc:EncryptionMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
            <KeyName>Symmetric Key</KeyName>
         </KeyInfo>
         <xenc:CipherData>
            <xenc:CipherValue
            >InmSSXQcBV5UiT...  Y7RVZQqnPpZYMg==</xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
   </soap:Body>
</soap:Envelope>
```

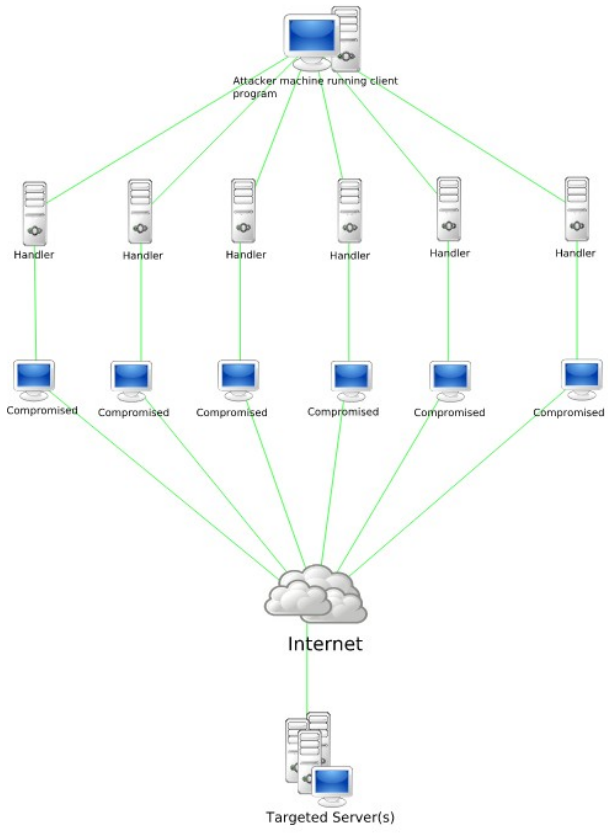
135. LOGIN MODULE FLAGS

   a. REQUIRED – LoginModule is required to succeed, if fails authentication still continues to proceed down to the LoginModule list
   b. REQUISITE – Required to succeed , if it succeeds authentication continues down to the Module list , if fails control immediately return to app
   c. SUFFICIENT – Not required to succeed , if it does , control returns to the application, if fails authentication goes down the module list
   d. OPTIONAL – is not required to succeed, if succeeds or fails still goes down the list

136. COMMONLY ENCOUNTERED SECURITY THREATS are

   a. Main in the Middle attack
   b. Session Hijacking / Replaying Data

c. Password Cracking (Bruce Fort)
d. Phishing
e. Social Hacking (Member of opposite sex taking the password)
f. Network Sniffing (One of the oldest method , un encrypted data is simply read by using a network sniffing tool)
g. XSS on java script based web applications (type 0 , 1, 2) [advent of rich internet applications has this possibility that a J2EE architect should be aware of]
h. DOS attack (Denial of Service) is an attempt to make a service un-available for its users. One common method for this is saturating the target site with requests
i. DDOS (Distributed denial of Service attacks). A high scale of DOS attack.
j.



137. **SAML** stands for **SECURITY ASERTION MARKUP LANGUAGE**; this is being used in SINGLE SIGN ON login mechanism.
138. **SAML** is a standard based on **XML** to exchange **AUTHENTICATION** and **AUTHORIZATION** data between **SECURITY DOMAINS**
139. **SAML** is a product of **OASIS** committee
140. The single **MOST IMPORTATN** problem that **SAML** is trying to solve is **WEB BROWSER SINGLE SIGN ON**
141. **OpenSAML** is a **JAVA** implementation of **SAML** and can be used in SSO authentication
142.