International Journal of Information & Computation Technology.
ISSN 0974-2239 Volume 4, Number 7 (2014), pp. 671-676

http://www. irphouse.com

Security Need for Pluggable Applications

Aditya Sengar[1] and Shubhangi Bhadoriya[2]

[1]Server Technology, Oracle India Pvt. Ltd Bangalore, INDIA
[2]Alumni Infosys Technologies Ltd Bangalore, INDIA

ABSTRACT

Pluggable applications have been the basis of enterprise development projects for years. The sole purpose in their life is to get new functionality 'in the field'. But platform security aspect of these applications has got little attention in design and implementation. This is largely due to the fact that plug-in are downloaded from a trusted store and follow good amount of testing before their release. But in a collaborated environment where partners are involved in developing plug-ins for application, aforesaid points may not always be true.

Application upgrade always has security concern and in this case concern becomes especially strong as we download software across the network and executes it locally as part of our application so we see a need for enabling application to identify and control its computational resource like never before.

Rest of this white paper will discuss key points on secure platform design for a Pluggable application by inheriting java security model.

1. INTRODUCTION

For a typical application functionality freezes during design. Post application delivery there is no way for developer to add or customize functionality unless some rework is done for a given module. Pluggable design meets this need by allowing addition or up gradation of modules without breaking existing code base.

"Functionality will emerge when required"

Plug-in is a piece of code that enables application to do something which it couldn't do by itself. In order to support a desired functionality all application administrators have to do is to download and deploy a plug-in for it.

This style of development has manifold advantages,

- On one hand it gives application extensibility to plug-in creator thereby reducing plug-in development time on the other it supports parallel development of plug-ins which can dynamically add new features to application.

- Likewise reduce core application size and increase flexibility in feature customization.
- Plug-in simply provides one functionality hence plug-in developer has single focus.

2. INHERITING JAVA SECURITY MODEL

Java security architecture has ability to grant code with varying degrees of trust and different levels of access to the system. In addition it provides a safe and secure platform for developing and running applications by,

- Inbuilt safety features (like type safety, byte-code verifier, class loader, garbage collector) that leads to more robust code and prevents hostile code from corrupting runtime.
- Fundamental system security features (like protection domain, access control, authorization, security manager) to distinguish unsafe and sensitive operations. This prevents malicious code from interfering with benevolent code.
- Web-based security support (user authentication, code signing, and secure communication protocols such as SSL) for improving trust between two cooperating application.

3. AUTHENTICATING PLUG-IN

"Application must trust plug-in before absorbing it in"

Authentication is a process of identity verification. The participating entity has to provide some proof of identity that system can understand. Three entities involved in plug-in deployment are the plug-in store, user and plug-in itself so to complete authentication we need three levels of check.

3.1 Form based Authentication is used inside application for user authentication needs

- A user comes to Login page of application (or gets redirected here on timeout etc).
- The application web server presents its certificate to the browser.
- The JRE verifies the application's server certificate.
- If verification is successful, user enters Id and password to the server.
- Login module gets a salt, sends hash string created using Id, password and salt.
- The server verifies this hash against the stored hash.
- If the verification is successful, the user is allowed to Login to application.

3.2 Certificate based mutual authentication is used between Online plug-in store and application (In-app or local plug-in store can be considered as trusted)

- Application user requests access to a plug-in jar file from online store.
- Plug-in store server presents its certificate to the application.

- The application verifies the server's certificate.
- If successful, the application sends its certificate to the plug-in store.
- The plug-in store server verifies the application's credentials.
- If successful, the server grants access to the plug-in binary requested by the application.

3.3 We need to verify origin of plug-in binary and make sure that it's not altered during network transmission

- The Plug-in vendor puts digital signature inside the plug-in binary.
- Plug-in store generate hash for this binary and encrypt it with its private key.
- Application downloads plug-in binary and hash value.
- Application decrypts this signed hash with plug-in stores public key
- Application compares this hash with the one generated from plug-in binary.
- If successful, application concludes that it received the correct binary.

Once authentication is successful, plug-in code validations like structure, semantics, metadata, closure etc are performed on success application imports vendor certificate issued by CA.

4. ENSURING RUNTIME SECURITY

"Principal responsibilities of a application is availability"

Availability is the most basic but most vital service metric. It entirely depends on computational resources. In the traditional security scheme, once some software got access to our system, it had full reign. If it was malicious, it can do a great deal of damage if there are no restrictions placed on it by the run-time environment. In similar lines the installed plug-in will compete for limited system resources like memory, disk-space, processor with other plug-ins and also with core application.

In an unprotected environment compromised plug-in (plug-in with vulnerabilities) may deny legitimate use of system resource by keep on requesting system resources without ever releasing them back in other words it can hijack these resources to bring down application or other plug-in.

4.1 Co-existence is achieved only by fair division of resources

"Conflict doesn't occur as long as plug-ins stays out of the way of each other"

- Plug-ins resource references should be partitioned so that garbage collector can free clusters of inter-related objects when they are no longer referenced from the application or plug-ins.
- Similarly, Class memory should be reclaimed by the garbage collector when it finds clusters of inter-related Class instances with no incoming Class references from either plug-in or application.
- In Addition Database objects coming with plug-ins must be created in separate schema (for each plug-in). On one hand it will simplify design and deployment/ undeployment of plug-ins on the other it will secure (isolate) one

plug-in objects from getting altered by other.

- Our application loads each plug-in in a new custom class loader and release reference of this class loader once plug-in is uninstalled. Putting our own classloader gave us an advantage of counting perm space and restricting specific classes from being loaded.

4.2 Putting plug-in security policy in place

We needed a comprehensive set of policies and permissions those can allow application to administer plug-ins with fine-grained access to security-sensitive resources. We come up with set of plug-in security policies that constraints on behavior of plug-in and hence designed to eliminate exploitation of computational resources.

We inherited java policies as they are extremely granular and we could select set of a different policy per plug-in to differentiate between plug-ins. We can modify these policies on the fly without requiring developer to modify the application code. As plug-ins should not halt VM we controlled exitVM property from our custom Security manager. Here is a sample policy set for one plug-in.

```
grant signedBy "samplePlugin"
{
permission java.util.PropertyPermission "${application.root}", "read";
permission java.io.FilePermission "${key.store.path}", "read ";
permission java.io.FilePermission "${log.dir}", "read, write, delete";
permission java.lang.RuntimePermission "accessClassInPackage.sun.io";
permission java.lang.FilePermission "${this.root}", “read,write,delete ";
}
```

Because the Java API always checks with the security manager before it performs a potentially unsafe action, our application will not perform any action forbidden under the security policy established by the security manager. If the security manager forbids an action, application will throw a meaningful exception.

The policy enforcement can be done using plug-in vendor public key certificate that we import on a successful authentication. We used signedBy alias to identify set of policies for a plug-in.

4.3 Sandboxing plug-in environment

“Application must identify and control consumption on all its resource to ensure its services to plug-ins when needed”

The essence of the our sandbox model is that core application code is trusted to have full access to vital system resources while plug-in code is not trusted and can access only the limited resources provided inside the sandbox

Plug-ins job execution happens in a different thread for sandbox environment using Security Manager limited by set of policies. This allows us to define and limit what a plug-in can do in our environment.

```
PluginExecutionEnvironment< PluginCallResult<String>> env
= new PluginExecutionEnvironment< PluginCallResult<String>>()
```

```
    {
            public Object run()
            {
                    /* untrusted code running with current plugin policies */
              return result;
            }
};

PluginContext context = new PluginContext(plugin);
context.setRunConcurrent(true);
    context.setMaxThreadCount(5);
    context.setMaximumRunTime(10, TimeUnit.SECONDS);
    service.invoke(env, context);
```

No Direct resource/ run time access (like Class loader, Thread, Reflection, Socket and File Streams) for plug-in code (restricted by policies), it must use application API to do so. In addition application share guarded object with plug-in if needed. Such an implementation abstracts plug-in from handling low level streams and secures application from DoS attack.

```
FileInputStream fis = new FileInputStream("vendor1.config");
    Guard guard = new PropertyPermission("vendor1", "read");
    GuardedObject go = new GuardedObject(fis, guard);
```

In plug-in code,

```
    FileInputStream fis = (FileInputStream) go.getObject();
```

Our plug-in execution environment is created using a custom class loader.

```
    PluginClassLoader pcl = new PluginClassLoader();
    pcl.setPermMemoryLimit(10*1024)
    pcl.setRestrictPackage("java.*")
```

We can still have fully trusted plug-ins which will run outside sandbox but resource partitioning will still be applicable for them.

4.4 Configuration Isolation

"Compromised plug-in may alter application or other plug-in configuration for malicious purpose"

Keeping default permissions for configuration files make them vulnerable for unauthorized access. Any malicious change in configuration files may affect boot process for application or other plug-in. In our proposed design application creates a hierarchy of directories to store its own configuration information. In addition each plug-in also has a private directory to store its configuration information. Protections are cumulative across these directories. If the policies shows that plug-in have permission to access the resource, system will allow it to access (unless plug-in has been black listed from accessing the files).

Directory hierarchy will be like,

- Application (only) writable configuration
- Application readable (by plug-ins) configuration
- Plug-in (self) writable configuration
- Plug-in (other) readable configuration

In this case only application will have write access to its configuration files. Plug-ins on the other hand can edit their own configuration and if require can read from other application or other plug-ins configuration.

5. CONCLUSIONS

Pluggable applications are security-sensitive as they download remote code and run it locally as part of application. Our proposed design made it possible to build security measures into applications to minimize the likelihood that compromised code can manipulate application for malicious purpose. This is done by establish an additional layer of defense around system resources; In addition we partitioned our system resources to avoid any conflict among downloaded plug-ins.

These security measures ensure that we don't need to figure out what code we can and can't trust. The application itself prevents any malicious or buggy code that we invite into our application from doing any damage hence giving us confidence to welcome code from any partner or 3rd party developer.

6. REFERENCES

[1] Java SE Documentation 'Evolving the SandboxModel' http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc1.html#18314