Advanced security hardening in WebSphere Application Server V7, V8 and V8.5, Part 2: Advanced security considerations

Martin Lansche
Keys Botzum

March 20, 2013

Security consists of more than just some firewalls at the edge of your network protecting you from the outside. It is a difficult and complex set of actions and procedures that strive to strengthen your systems as much as is appropriate. Part 1 of this two-part article discussed many aspects of security in general, with an emphasis on hardening an IBM® WebSphere® Application Server environment. Part 2 covers even more ground, including application-based preventative measures, cell trust, cross-cell trust, administrative and application isolation, identity propagation, desktop development security, and more. This revision of an earlier article has been significantly updated for WebSphere Application Server V7, V8.0 and V8.5. This is part 2 of 2.

View more content in this series

Introduction

Part 1 left off explaining how IBM WebSphere Application Server V7.0 and later versions were designed with the security principle of **secure by default**. While not perfectly achieved, the goal was to release a product whereby, in the most common configurations and simpler environments, the product is configured reasonably securely by default. The previous article ended after describing many significant infrastructure-based preventative security measures that have been incorporated into WebSphere Application Server. This article continues with a description of additional preventative measures that are application-based, and then continues by describing some critical advanced considerations.

While the information in this article is based on IBM WebSphere Application Server V7, V8.0 and V8.5, most of the issues discussed here apply equally to V6.1. Where an issue is unique to a specific version, it will be identified as such. If you are using an earlier version of WebSphere Application Server, refer to the earlier article, as there are **significant version differences, which in turn results in differences between the articles**.

A note about profiles

If you are familiar with WebSphere Application Server prior to V8.5 you know you must create one or more profiles. This could be an Application Server (or Base) profile, a Deployment Manager profile, and so on. For the remainder of this article these will be called "full" profiles. This distinction is made to contrast these profiles to a new addition in V8.5: the Liberty profile. This article also calls out recommendations specific to the new Liberty profile.

Application-based preventative measures

Configuration measures

Up to this point, the focus of these articles has been on the basic steps you can take to ensure that you create a secure IBM WebSphere Application Server infrastructure. This is obviously important, but focusing on the infrastructure alone it is not sufficient. Now that the infrastructure has been hardened, you must now examine the things that applications need to do in order to be secure. Of course, applications must leverage the infrastructure provided by WebSphere Application Server, but there are several other actions that application developers must take (or not take) to make them as secure as possible:

1. Never set Web server document root to WAR
2. Carefully verify that every servlet alias is secure
3. Do not serve servlets by classname
4. Do not place sensitive information in WAR root
5. Define a default error handler
6. Consider disabling file serving and directory browsing
7. Enable session security
8. Beware of custom JMX network access

To help you tie these actions back to specific classes of attack, each item will use the key presented in Part 1 to represent these vulnerabilities

1. Never set Web server document root to WAR

N M E I

WAR files contain application code and lots of sensitive information. Only some of that information is Web-servable content, and so it is inappropriate to set the Web server document root to the WAR root. If you do this, the Web server will serve up all the content of the WAR file without interpretation, resulting in code, raw JSPs, and more being served up to your users. (This action is only relevant when a Web server is co-located with an application server, which typically doesn't occur, if you've followed the guidance in presented in Part 1.)

2. Carefully verify that every servlet alias is secure

N M E I

WebSphere Application Server secures servlets by URL. Each URL that is to be secured must be specified in the web.xml file describing the application. If a servlet has more than one alias (that is, multiple URLs access the same servlet class) or there are many servlets, it is easy to accidentally forget to secure an alias. Be cautious. Since WebSphere Application Server secures URLs and not the underlying classes, if just one servlet URL is insecure, an intruder might be able to bypass your security. To alleviate this, use wildcards to secure servlets wherever possible. If that is not appropriate, carefully doublecheck your web.xml file before deployment.

APAR PK83258 (in WebSphere Application Server Versions 7.0.0.7 and 6.1.0.27) closes part of this vulnerability by checking the authorization on a HEAD request as if it was a GET request.

Related to this, make sure that when you specify authorization constraints for servlets that you either list no methods, or very carefully list (likely in multiple constraints) ALL the methods for a servlet (GET, POST, PUT, HEAD, and so on). Per the Java™ EE specification, if an authorization constraint lists methods explicitly, other unmentioned methods have NO authorization constraint! This is particularly dangerous with methods such as HEAD that default to calling GET to obtain the needed headers -- effectively calling the GET method without checking its authorization constraints. The web.xml code section in Listing 1 is considered unsafe, while that shown in Listing 2 is safe.

Listing 1. web.xml – unsafe

```
<security-constraint>
  <web-resource-collection>
   <web-resource-name>myservlet</web-resource-name>
   <url-pattern>/myservlet</url-pattern>
   <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
   <role-name>arole</role-name>
  </auth-constraint>
</security-constraint>
```

Listing 2. web.xml – safe

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>myservlet</web-resource-name>
  <url-pattern>/myservlet</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <role-name>arole</role-name>
 </auth-constraint>
</security-constraint>
```

The following approach can be taken within every web.xml to ensure that permissions are restrictive:

1. Define an initial security constraint such as that shown in Listing 3. This says all URLs (unless overridden with more specific patterns) are restricted to the NoAccess role. The NoAccess role name can be pre-bound to the None special role.

Listing 3

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>DefaultDeny</web-resource-name>
  <url-pattern>/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <role-name>NoAccess</role-name>
 </auth-constraint>
</security-constraint>
```

2. If you are using a Form-based login, define the permissive constraint to allow the web container to display the Login page (and necessary graphics, and so on) to any user, even those who are not authenticated (Listing 4). AllUsers role name can be pre-bound to the Everyone special role.

Listing 4

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>LoginForm</web-resource-name>
  <url-pattern>/Login.jsp</url-pattern>
  <url-pattern>/images/*.gif</url-pattern>
  <url-pattern>/css/*.css</url-pattern>
  <http-method>GET</http-method>
 </web-resource-collection>
 <auth-constraint>
  <role-name>AllUsers</role-name>
 </auth-constraint>
</security-constraint>
```

3. After the above two “default” constraints are added, add the “real” security constraint element for each servlet added to the application that is appropriate for that URI .

The advantage of this approach is that new end-points added to the web application are unreachable by default. If a new servlet is added, you will have to add a security constraint for the new URL in order for anyone to use it. Such a “default-to-deny” design is a cornerstone of developing secure applications.

If you are using Java annotations to define your security constraints, you can leverage the ServletSecurity.EmptyRoleSemantic.DENY constraint. For example, the /simple URL is allowed for POST and GET methods for the AnyUser role, but the remaining methods are Denied (Listing 5).

Listing 5. Example of using annotations to provide secure defaults for HTTP methods.

```
@WebServlet(description = "Returns msg and user info", urlPatterns = { "/simple*" })
@ServletSecurity(
    value = @HttpConstraint(ServletSecurity.EmptyRoleSemantic.DENY),
    httpMethodConstraints =
       {@HttpMethodConstraint (value = "GET",  rolesAllowed = "AnyUser"),
         @HttpMethodConstraint (value = "POST", rolesAllowed = "AnyUser")})
public class Simple extends HttpServlet {
…
```

According to the Java Servlet Specification 3.0 (Dec, 2009), Section 13.4.1, the value attribute of HttpConstraint(ServletSecurity.EmptyRoleSemantic.DENY) has this affect:

“[The] HttpConstraint that defines the protection to be applied to all HTTP methods that are NOT represented in the array returned by httpMethodConstraints.”

3. Do not serve servlets by classname

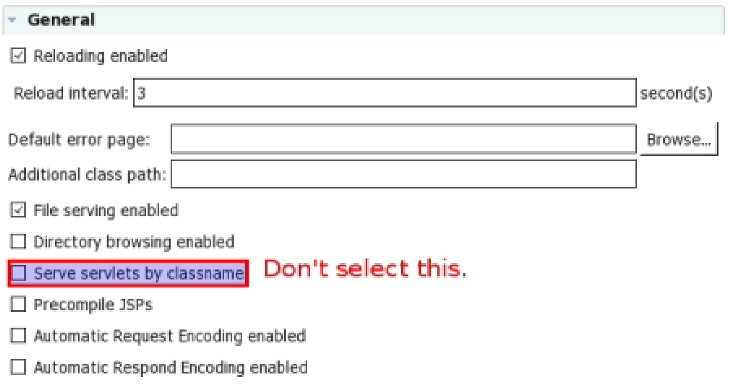
N M E I

Servlets can be served by classname or via a normal URL alias. Typically, applications choose the latter. That is, developers define a precise mapping from each URL to each servlet class in the web.xml file either by hand or by using one of the various WebSphere Application Server development tools.

However, WebSphere Application Server also lets you serve servlets by classname. Instead of defining a mapping for each servlet, a single generic URL (such as /servlet) serves all servlets. The component of the path after the base is assumed to be the classname for the servlet. For example, "/servlet/com.ibm.sample.MyServlet" refers to the servlet class "com.ibm.sample.MyServlet."

Serving servlets by classname is accomplished by setting the **serveServletsByClassnameEnabled** property to true in the ibm-web-ext.xmi file or by selecting serve **servlets by classname** in the WAR editor in IBM Rational® Application Developer. Do not enable this feature — Rational Application Developer enables it by default, so **you must explicitly disable it** if you are using Rational Application Developer (Figure 1).

Figure 1. Do not serve servlets by classname



General

- ☑ Reloading enabled
- Reload interval: 3 second(s)
- Default error page:
- Additional class path:
- ☑ File serving enabled
- ☐ Directory browsing enabled
- ☐ Serve servlets by classname — Don't select this.
- ☐ Precompile JSPs
- ☐ Automatic Request Encoding enabled
- ☐ Automatic Respond Encoding enabled

This feature makes it possible for anyone that knows the classname of any servlet to invoke it directly. Even if your servlet URLs are secured, an attacker might be able to bypass the normal URL-based security. Further, depending on the classloader structure, an attacker might be able to invoke servlets outside of your Web application, such as IBM provided servlets. (Note that not installing the sample applications in production is noted here as a first step to prevent this.)

As an administrator, you might be uncertain as to the setting of various applications with respect to whether they have erroneously enabled serve servlets by classname. You can configure the application server to ignore the setting within the application (thus disabling serving by classname) by setting a custom property com.ibm.ws.webcontainer.disallowAllFileServing on the JVM. The Liberty profile equivalent construct is given in Listing 6.

Listing 6

```
<webcontainer disallowAllFileServing=”true”/>
```

4. Do not place sensitive information in WAR root

N M E I

WAR files contain servable content. The Web container will serve any files found in the root of the WAR file. This is fine as long as you place only servable content in the root. Thus, you should never place content that shouldn't be shown to users in the root of the WAR. For example, don't put property files, class files, or other important information there. If you must place such information in the WAR file, place it within the WEB-INF directory, as permitted by the servlet specification. Information there is never served by the Web container.

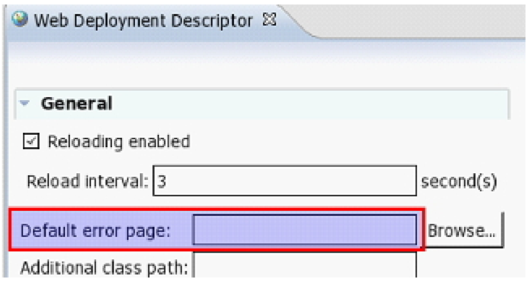
5. Define a default error handler

N M E I

When errors occur in a Web application -- or even before the application dispatch (for example, if WebSphere Application Server can’t find the target servlet) -- an error message is displayed to the user. By default, WebSphere Application Server will display a raw exception stack dump of the error. Not only is this incredibly unfriendly to the user, it also reveals information about the application (names of classes and methods are in the stack information). The exception message text is also displayed, which could contain sensitive information.

It is best to ensure that users never see a raw error message by defining a default error page that displays whenever an unhandled exception occurs. This page can be a user friendly error message rather than a stack trace. The default error page is defined in ibm-web-ext.xmi using the defaultErrorPage attribute, or it can be set in Rational Application Developer using the Web deployment descriptor editor (Extensions tab).

Figure 2. Define default error page


Web Deployment Descriptor

General

☑ Reloading enabled

Reload interval: 3 second(s)

Default error page: Browse...

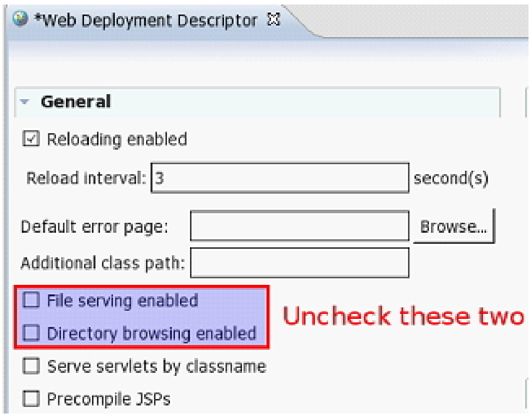
Additional class path:

6. Consider disabling file serving and directory browsing

N M E I

You can further limit the risk of serving inappropriate content by disabling file serving and directory browsing in your Web applications.

Figure 3. Disable file serving and directory browsing



*Web Deployment Descriptor

General

☑ Reloading enabled

Reload interval: 3 second(s)

Default error page: Browse...

Additional class path:

☐ File serving enabled

☐ Directory browsing enabled

Uncheck these two

☐ Serve servlets by classname

☐ Precompile JSPs

Of course, if the WAR contains servable static content, file serving will have to be enabled. There is rarely reason to enable Directory browsing.

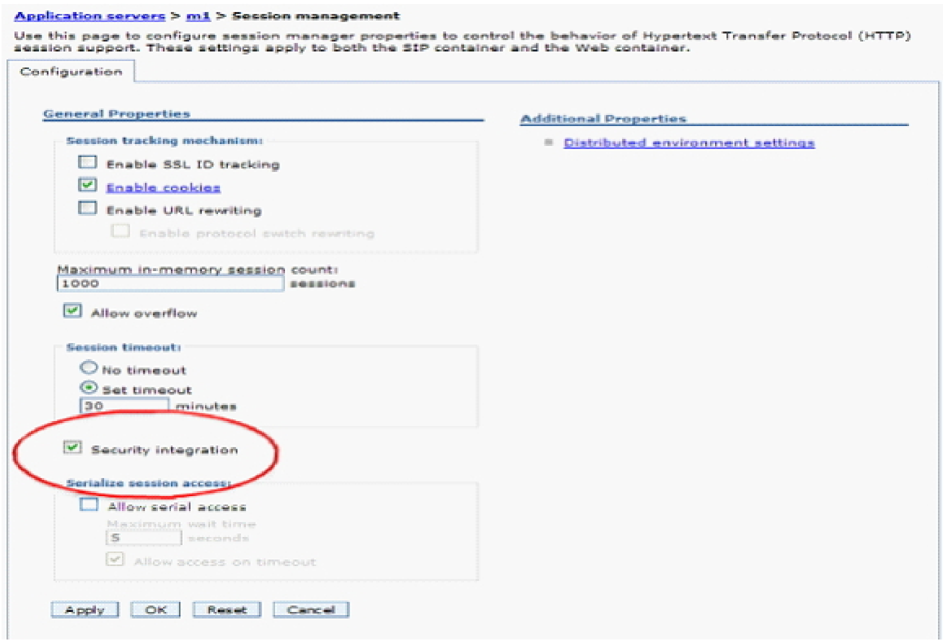
7. Enable session security (Applies only to V7.0 only)

N M E I

Unlike WebSphere Application Server V8.0 (and later), WebSphere Application Server V7.0 does not normally enforce any authorization for HTTP session access. As a result, any request with a valid session identifier can access any session. While session identifiers aren't likely to be guessable, it might be possible to obtain session identifiers through other means. As such, users migrating from earlier versions of WebSphere Application Server who had not enabled session security might see breakage in their applications. Note that this is also the default behavior for the Liberty profile.

To reduce the risk of this form of attack, you should consider enabling session security. This setting is configured by navigating to the **Application server >** ***<server name>*** **> Web container > Session management** panel. Simply select the **Security integration** option, shown in Figure 4.

Figure 4. Session security integration


Application servers > m1 > Session management

Use this page to configure session manager properties to control the behavior of Hypertext Transfer Protocol (HTTP) session support. These settings apply to both the SIP container and the Web container.

Configuration

General Properties

Session tracking mechanism:

- [ ] Enable SSL ID tracking
- [✓] Enable cookies
- [ ] Enable URL rewriting
  - [ ] Enable protocol switch rewriting

Maximum in-memory session count: 1000 sessions

- [✓] Allow overflow

Session timeout:

- ○ No timeout
- ● Set timeout 30 minutes

- [✓] Security integration

Serialize session access:

- [ ] Allow serial access
  - Maximum wait time 5 seconds
  - [✓] Allow access on timeout

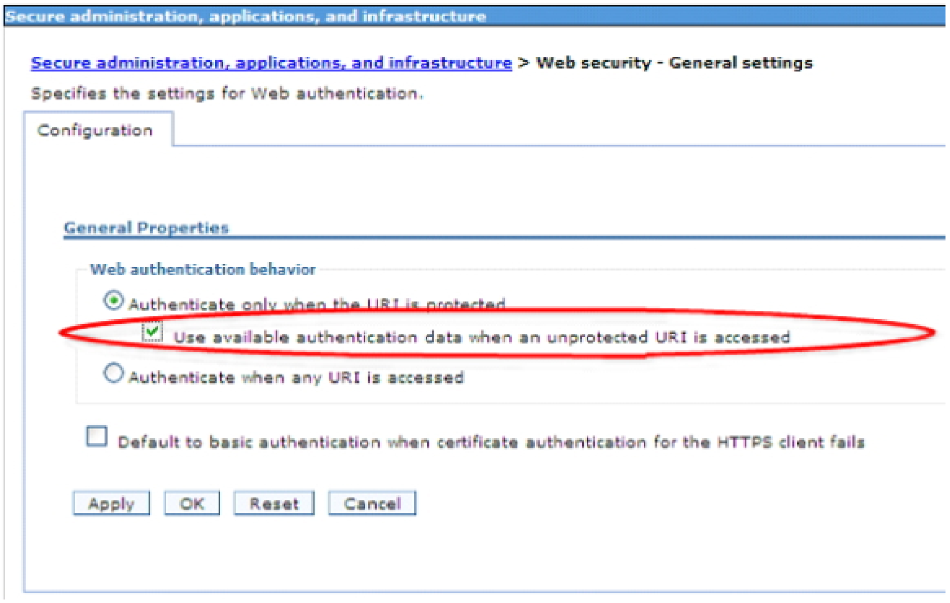
Additional Properties

- Distributed environment settings

WebSphere Application Server will then track which user (as determined by the LTPA credential that the user presents) owns a specific session and will ensure that only requests for that user can access that session.

In rare cases, this setting can break Web applications. If the application contains a mixture of secure (those with authorization constraints) and insecure servlets, the insecure servlets will not be able to access the session object. Once a secure servlet accesses the session, it is marked as “owned” by that user. Insecure servlets that run as anonymous will receive an authorization failure if they try to access those pages. As of WebSphere Application Server V6.1, the authentication behavior can be changed to ensure that servlets without an authorization constraint inherit the current user identity if one has already been established, thus addressing this issue. Figure 5 shows how you can set this feature.

Figure 5. Preserve existing identity


Secure administration, applications, and infrastructure

Secure administration, applications, and infrastructure > Web security - General settings

Specifies the settings for Web authentication.

Configuration

General Properties

Web authentication behavior

(●) Authenticate only when the URI is protected

[✓] Use available authentication data when an unprotected URI is accessed

( ) Authenticate when any URI is accessed

[ ] Default to basic authentication when certificate authentication for the HTTPS client fails

In WebSphere Application Server V8.0 the server defaults have been changed two-fold. It is the combination of enabling session security integration and the “use available authentication data when an unprotected URI is accessed” that avoids the secure/unsecure servlet breakage scenario described earlier.

Preventing an attack

Enabling session security actually prevents one type of attack. Suppose your Web site establishes an HTTP session before the user authenticates and before the use of HTTPS (as with most sites). In this case, the session cookie (JSESSIONID) is exposed in the clear, making it easy for an attacker to capture it. Later, when the user authenticates, you should of course switch to SSL and ensure that the LTPA cookie flows only over SSL, as discussed earlier. The trouble is the attacker already has the JSESSIONID cookie. Depending on how your application is written, he might be able to use that cookie and a different LTPA cookie (perhaps one for his own user identity) to then access the application as the first user. This is a very serious attack and one that is quite easy to implement if an open wireless network is in use. By enabling session security, this attack is blocked because the application server will block access to a session that is not owned by the current user.

8. Beware of custom JMX network access

N	M	E	I

JMX and custom MBeans make it possible to support powerful remote customized administration of your applications. Be aware, however, that JMX MBeans are network accessible. If you choose to deploy them, do so carefully and ensure that they have proper authorization on their operations. WebSphere Application Server will automatically provide default authorization restrictions for

MBeans based on information in the descriptor in the MBean JAR file. Whether this is appropriate or not will depend on your application. See the book IBM WebSphere: Deployment and Advanced Configuration for more information.

Design and implementation measures

Next, let's turn our attention to the actions that application developers and designers must take in order to build a secure application. These steps are crucial and, sadly, often overlooked.

9. Use WebSphere Application Server security to secure applications
10. Do not rely on HTTP session for user identity
11. Secure every layer of the application
12. Validate all user input
13. Write secure applications
14. Store information securely
15. Be sensitive to auditing and tracing
16. Avoid the use of basic authentication for browser clients
17. Avoid widget jacking: Use SSL if you build your browser UI using GETs from third party sites

9. Use WebSphere Application Server security to secure applications

N M E I

Application teams usually recognize that their applications need some amount of security. This is often a business requirement. Unfortunately, many teams decide to develop their own security infrastructure. While it is possible to do this well, it is also very difficult to do so, and as a result most teams do not succeed. Instead, there is the illusion of strong security when in fact the system security is quite weak. Security is simply a difficult problem. There are subtle issues of cryptography, replay attacks, and various other forms of attack that are easily overlooked. The message here is that WebSphere Application Server security should be used. It was recommended earlier that you enable application security; here, it is recommended that you actually use it in your applications.

Perhaps the most common complaint about the Java EE-defined declarative security model is that it is not sufficiently granular. For example, you can only perform authorization at the method level of an EJB or servlet (in this context, a method on a servlet is one of the HTTP methods, such as GET, POST, PUT, and so on), not at the instance level. Thus, for example, all bank accounts have the same security restrictions, but you would prefer that certain users have special permissions on their own accounts.

This problem is addressed by Java EE security APIs **isCallerInRole** and **getCallerPrincipal**. By using these APIs, applications can develop their own powerful and flexible authorization rules but still drive those rules from information that is known to be accurate: the security attributes from the WebSphere Application Server runtime. If this is still not sufficient, you can build (or buy) a more elaborate authorization framework that leverages the existing security information already maintained by the application server (such as groups). If even that is not sufficient, the security infrastructure is highly customizable. Leveraging what exists and works (extending where needed) is the right approach, rather than discarding what you have and attempting to build a secure infrastructure from scratch.

An example of weak security

The WebSphere Application Server security infrastructure can be integrated with other authentication or authorization products. For example, IBM Tivoli® Access Manager can provide authentication and authorization support, and IBM Tivoli Security Policy Manager can provide additional authorization capabilities.

Here is one quick example of a weak security system. Applications that don't use WebSphere Application Server security tend to create their own security tokens and pass them within the application. These tokens typically contain the user's name and some security attributes, such as their group memberships. It is not at all uncommon for these security tokens to have no cryptographically-verifiable information. The presumption is that security decisions can be made based on the information in these tokens. This is false. The tokens simply assert user privileges. The problem here is that any Java program can forge one of these security objects and then possibly sneak into the system through a back door. The best example of this is when the application creates these tokens in the servlet layer and then passes them to an EJB layer. If the EJB layer is not secured (see step 11), intruders can call an EJB directly with forged credentials, rendering the application's security meaningless. Thus, without substantial engineering efforts, the only reliable secure source of user information is the WebSphere Application Server infrastructure.

10. Do not rely on HTTP session for user identity

N M E I

This recommendation is closely related to the previous topic. Far too many applications that implement their own security track the user's authentication session through the use of the HTTP Session. This is dangerous. The session is tracked via a session ID (on the URL or in a cookie). While the ID is randomly generated, it is still subject to replay attacks because it does not have a specific absolute timeout. A timeout only occurs after a period of inactivity -- if the session cookie is intercepted, there is a potentially unlimited time to misuse the cookie. On the other hand, the LTPA token, which is created when WebSphere Application Server security is used by the application, is designed to provide a stronger authentication token. In particular, LTPA tokens have limited lifetimes, use strong encryption, and the security subsystem audits the receipt of potentially forged LTPA tokens.

An even more dangerous yet subtle problem with using the HTTP session for security is that the session cookie (JSESSIONID) is usually created before a user authenticates -- typically when they first access the site. At this point, the cookie is often sent in the clear over HTTP. Once the user authenticates, most applications will switch to HTTPS for all future traffic (protecting cookies and content) -- but the JSESSIONID cookie could already have been stolen because an attacker could have captured the cookie when it was initially sent over HTTP. In addition to the obvious point of not using the HTTP session for security, the risk of stealing an HTTP session can be reduced by enabling session security and restricting the LTPA cookie to HTTPS, as discussed earlier.

In addition, cookies created by your applications should be Secure (restricted to HTTPS), and all cookies should be marked HTTPOnly, with a possible exception (which needs to documented, and signed off in a design review) where an application explicitly requires it to function because of client side JavaScript needing access to the cookie.

11. Secure every layer of the application

N M E I

All too often, Web applications are deployed with some degree of security (home-grown or WebSphere Application Server-based) at the servlet layer, but the other layers that are part of the application are left unsecured. This is done under the false assumption that only servlets need to be secured in the application because they are the front door to the application. But, as any police officer will tell you, you have to lock the back door and the windows to your home as well. There are many ways this can occur, but this is most commonly seen when remotely accessible components (such as EJBs accessible via IIOP or Web services) are used as part of a multi-tier architecture when Java clients aren't part of the application. In this case, developers often assume that the remotely callable components do not need to be secured because they are not "user-accessible" in their application design, but this assumption is dangerously wrong. This is frequently described, tongue firmly in cheek, as “security by obscurity.” An intruder can bypass the servlet interfaces, go directly to the services layer, and wreak havoc if you have no security enforcement at that layer.

Often, the first reaction to this problem is to secure the services via some trivial means, perhaps by marking them accessible to all authenticated users. But, depending on the registry, "all authenticated users" might be every employee in a company. Some take this a step further and restrict access to members of a certain group that means roughly "anyone that can access this application." That's better, but it's usually not sufficient, as everyone that can access the application shouldn't necessarily be able to perform all the operations in the application. The right way to address this is to implement authorization checks in the services. In the case of EJBs, you might also consider implementing EJB components with only local interfaces, making it impossible to connect to them remotely.

12. Validate all user input

N M E I

Cross-site scripting is a fairly insidious attack that takes advantage of the flexibility and power of Web browsers. Most Web browsers can interpret various scripting languages, such as JavaScript™. The browser knows it is looking at something executable based upon a special sequence of escape characters. Therein lies the power -- and the dangerous flaw -- of the Web browser security model.

Intruders take advantage of this hole by tricking a Web site into displaying a script that the intruder wants the site to execute. This is accomplished fairly easily on sites that permit arbitrary user input. For example, if a site includes a form for entering an address, a user can instead input JavaScript. When the site displays the address later, the Web browser will execute the script. That script, since it is running inside the Web browser from the site, has access to secure information, such as the user's cookies.

So far, this doesn't seem terribly dangerous, but intruders can take this one step further and trick a user into going to a Web site and entering the intruding script, perhaps by sending the user an innocent URL in an e-mail or posting it to a public blog. The intruder can now run arbitrary code within the user's browser, typically with access to cookies, and can see all keyboard events and any pages displayed. This is all an intruder needs to do irreparable harm to that user.

This problem is actually a special case of a much larger class of problems related to user input validation. Whenever you permit a user to type in free text, you must ensure that the text doesn't contain special characters that could cause harm. For example, if a user were to type in a string that is used to search some index, it might be important to filter the string for improper wildcard characters that might cause unbounded searches. In the case of cross-site scripting prevention, you need to filter out the escape characters for the scripting languages supported by the browser. The message here is that all external input should be considered suspect and should be carefully validated. A discussion of all the issues that must be addressed is well beyond the scope of this paper.

13. Write secure applications

N M E I

Given space constraints and the intent of this article, it's not possible here to enumerate all of the application design and coding issues that might impact the security of an application. If you are seriously interested in developing secure applications, see these excellent resources on secure application design, development, and testing:

- Book: Writing Secure Code
- Book: Software Security: Building Security In
- OWASP, which has published these guides:
  - Development Guide
  - Code Review Guide
  - Testing Guide
- Web Application Security Consortium
- Penetration testing frameworks

Also, a recently published IBM Redbook discusses using IBM Rational AppScan as a key element of your development cycle to build secure Web applications.

14. Store information securely

N M E I

To create a secure system, you must consider where information is stored or displayed. Sometimes, fairly serious security leaks can be introduced by accident. For example, be cautious about storing highly confidential information in the HTTP Session object, as this object might be serialized to the database, making this object readable from there. If an intruder has access to your database (or even raw machine-level access to the database volumes), he might be able to see information in the session. Needless to say, such an attack would take a high degree of skill.

15. Be sensitive to auditing and tracing

N M E I

Any non-trivial application will generate substantial logging and tracing information for business purposes and debugging purposes. That's all well and good. Just remember that information in files has a tendency to end up in a lot of places (maybe outside your organization). Extremely sensitive information should not be traced and you should try to avoid auditing it if you can. For example, we've seen clients that print user passwords, driver's license numbers, and even social security numbers to a trace file. If that file is read by the wrong person (perhaps a consultant there to help you), you have just revealed sensitive information. Any customer information system sent from the system to **anyone** in electronic form (including backup images) **MUST** be encrypted. This level of control does not apply only to user passwords, but to any and all customer information.

Just to be safe, limit and review what is logged.

16. Avoid the use of basic authentication for browser clients

N M E I

The problem with basic authentication is that the Web browser caches the user ID and password, and resends it automatically on future requests when needed. This means that it is impossible to log off a user that has used basic authentication. For example, if a user's LTPA cookie expires, the application server challenges again for a user ID and password. The browser will helpfully resend the basic authentication header – rendering timeouts useless.

This phenomenon of automatically resending the user credentials when needed is actually at the heart of most Single Sign On solutions. This includes SSO based on SPNEGO, and also when the user authenticates using Client SSL certificates. In other words, Basic Authentication is just another example where the user can easily log back in, without manually providing credentials. However, it is different than these other authentications in that the user’s password is resent on every request.

To make matters worse, if a request later occurs over HTTP (unencrypted), the browser will happily send the password in the clear over the network.

Basic authentication is a sensible approach when the client is not a browser. For example, Web services or REST clients often uses basic authentication safely. Of course, such requests should flow over SSL.

17. Avoid widget jacking: Use SSL if you build your browser UI using GETs from third party sites

N M E I

Social media sites have infiltrated into “traditional” web applications, and have done so in a way that exposes your users to identity theft with their social media credentials. This is called widget jacking. The underlying scenario is the following:

- An end user logs into a social site, such as Facebook or Twitter, and does not logoff. He or she might close the browser or reboot the computer.
- This user goes to a public wifi location and visits his bank (using the traditional web application). That bank has recently added the ability to “like” the bank on Facebook, Twitter and other social media sites. To help the user, the bank’s application user Interface includes the ubiquitous Facebook and Twitter icons.
- The bank’s page is rendered over SSL, but the returned page includes the code in Listing 7.

  ### Listing 7

  ```
  <iframe src="http://www.facebook.com/plugins/like.php?href=YOUR_URL"
  scrolling="no" frameborder="0" style="border:none; width:450px; height:80px">
  </iframe>
  ```
- Notice how the iframe causes a GET to an unencrypted URL.
- All cookies for www.facebook.com are sent on that HTTP GET request, including the cookie containing the user's authenticated identity to the facebook.com site.
- Eve, the eavesdropper listens for unencrypted traffic visible over the wifi (see Firesheep for details) and gets the unencrypted cookie.

With third party applications leveraging social media logins for authentication, this is becoming a larger vulnerability than you would initially think. There are browser plugins designed to protect users, but it requires a knowledgeable consumer to install and enable such plugins.

I am not advocating that your applications not integrate with such social media sites, or even that your users should not be able to register a “like” to such sites. But be responsible: change the code for getting that content to what you see in Listing 8.

Listing 8

```
<iframe src="https://www.facebook.com/plugins/like.php?href=YOUR_URL"
scrolling="no" frameborder="0" style="border:none; width:450px; height:80px"></iframe>
```

Advanced considerations

We move on now to some advanced issues related to hardening, including cross cell trust, application isolation, identity propagation, and limitations in WebSphere Application Server security. In most cases, you won't find any specific recommendations here, rather you should use this important information to help you in your efforts to maintain and manage a secure infrastructure.

Cell trust and isolation

A WebSphere Application Server cell should not span trust boundaries. If you can't trust someone else totally, don't let them manage your cell or a machine in your cell. Irrespective of the use of fine-grained administrative security, the WebSphere Application Server administrative infrastructure assumes a coarse grained shared trust model throughout the cell between each and every WebSphere process. Every application server contains within it the administrative infrastructure, including internal APIs. On the positive, this makes application servers highly independent and robust by eliminating common points of control and failure, but it also negatively impacts isolation. Implications of this approach include:

- Every application server in a WebSphere Application Server cell has full administrative authority to the entire cell. Should any application server be compromised, they are all compromised.
- Physical machine boundaries (separate computers, LPARs, nodes, and so on) have virtually no impact on cell security. The unit of trust in a cell is all application servers over all nodes.
- Process boundaries have little impact on cell security. Placing applications in separate application server JVMs does little to increase their isolation from a security perspective within a cell.
- Separate operating system identities have little impact on cell security. Since application servers communicate with the rest of the cell using various protocols that aren't managed by the operating systems, normal operating system protections have little effect.

This brings us to two key topics: administrative isolation and application isolation, next.

Administrative isolation

With WebSphere Application Server, all administrators have administrative authority (based on their assigned role) by default over the entire cell. With WebSphere Application Server V6.1, a new feature known as authorization groups was added making it possible to grant administrative authorities at a fine-grained level (server, application, node, cluster, and so on). In V6.1, the definition of enrollment into the authorization groups was supported only by wsadmin. In WebSphere Application Server V7, they are supported by the admin console as well. If you have a large cell with many administrators, take some time to look into these capabilities.

Application isolation

Application isolation in this context is fundamentally about preventing the illicit actions of one application from harming another. These types of attacks are very difficult to prevent. The reality is that infrastructure software products such as Java EE application servers have not yet reached the maturity level of multi-user operating systems. They do not provide the kind of robust isolation that an operating system typically provides between multiple users.

Does this affect you?

First and foremost, the weaknesses discussed here are internal only. These weaknesses can only be exploited by applications that are **installed into your cell**. In IBM's experience, the vast majority of users of WebSphere Application Server, even those with a shared infrastructure, are not impacted by these issues. This is because they recognize that their shared infrastructure is within the company and is running company approved code. Thus, they usually do not require complete security isolation of applications.

Users of WebSphere Application Server that are affected by the issues raised here typically have extremely rigorous security policies, such as:

- Formal policies that require applications not to have access to other applications' data even when running in a shared infrastructure.
- The requirement that every application run under a separate user ID on shared servers (a requirement common for systems that predate the use of WebSphere Application Server in the enterprise), detailed rules about file system permissions, dedicated processes for each application, and strict audit procedures to enforce these requirements.
- Strict corporate security guidelines that are rigorously enforced to ensure compliance, most likely including application architecture and code inspections. Without code inspections, developers can insert code that violates corporate policy.
- Policies designed to ensure that remote attacks that might compromise one application not be able to be used to compromise other applications.

If this doesn't sound like your environment, the issues in this section do not apply to you and you can skip to the next section.

What mitigating actions can you take?

What mitigating actions can I take?

- Enforce rigorous code inspections of all application code. Couple this with source code management systems that are secure. This way every code change is tracked to the person that made the change, and code inspections can be scheduled and tracked against every code change. In this situation, it should be impossible for a single programmer to insert malicious code into your system. Instead, multiple people would have to work together to achieve any compromise. We consider this particularly relevant since malicious application code could easily do great harm within one application, regardless of application isolation.

- If you choose to purchase commercial applications to run on WebSphere Application Server, ensure you purchase products only from reliable and trustworthy vendors, carefully test and monitor applications prior to production deployment, and monitor them while in production.
- If absolutely necessary, deploy applications into private WebSphere Application Server cells. You might consider using different cells for different business units or other organizational units, based on risk and trust. To limit hardware costs, you can safely run nodes from multiple cells on a single machine or LPAR. Simply run each cell with a different operating system user ID, different encryption keys, and different administrative passwords. This will provide complete isolation from a security perspective.

If none of these approaches are acceptable, you can apply application isolation hardening techniques to the infrastructure. Understand that these techniques require a great deal of work. More importantly, they do not provide a guarantee of isolation. Applications in the same cell -- even with administrative security, application security, and Java 2 security enabled -- can potentially compromise other applications in the same cell by accessing resources of those applications or altering the cell configuration. There is no way to guarantee that such compromise is impossible.

Another example of why code inspections are required for effective security is illustrated with the code in Listing 9.

Listing 9

```
Map<String, String> map = new HashMap<String, String>();
map.put(Constants.MAPPING_ALIAS, "thinkNode01/app_alias");
CallbackHandler callbackHandler =
 WSMappingCallbackHandlerFactory.getInstance().getCallbackHandler(map,null);

LoginContext loginContext = new LoginContext("DefaultPrincipalMapping", callbackHandler);
loginContext.login();

Subject subject = loginContext.getSubject();
Set<Object> credentials = subject.getPrivateCredentials();

PasswordCredential passwordCredential = (PasswordCredential)
 credentials.iterator().next();

String user = passwordCredential.getUserName();
String password = new String(passwordCredential.getPassword());
```

This code snippet leverages a WebSphere Application Server security SPI to obtain the user ID and password for any J2C alias in the cell. Since the purpose of this SPI is for application security context extension and customization, the SPI has to have access to this sensitive security information. The result is that the only way to preclude malicious code from leveraging this plug point (and others like it) is to perform code inspections to ensure that applications don’t use these plug points, unless business needs require their use.

In spite of these warnings, the following sections will document the actions you can take to significantly improve the security isolation of applications within a cell. You will likely realize quickly that taking these actions will be difficult and expensive. Arguably, a better and less expensive approach is to ensure your application developers are delivering trusted code through rigorous hiring practices, code reviews, and other management controls. As before, these actions are listed in priority order.

1. Do not specify component-managed aliases on Java EE resources
2. Do not define a default user ID and password on a resource
3. Java 2 security
4. Leverage secure chained delegation
5. Protect the TAM WebSEAL TAI password
6. Beware of heightened permissions in custom JMX code
7. Use DynaCache with caution
8. Use all resources with caution

1. Do not specify component-managed aliases on Java EE resources

N M E I

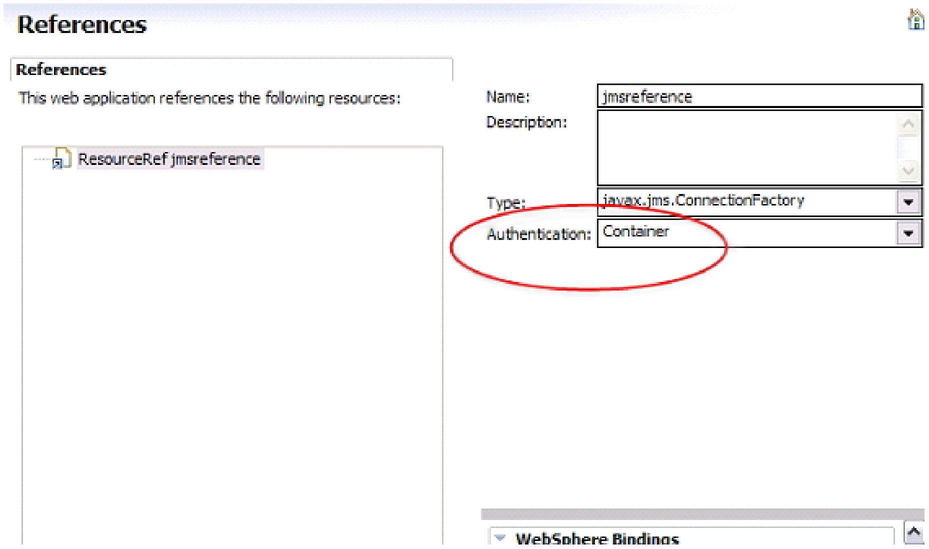
Any Java EE application that runs within the cell can potentially access any Java EE resource. This is because resources have JNDI names that can be looked up by any application and there is no authorization on resource access. Thus, if application A uses an enterprise database simply by defining the database as a data source, it is possible that application B in the same cell can access this database.

When an application tries to access a resource by calling getConnection() on the resource factory (for example, a data source or a JMS connection factory), WebSphere Application Server will implicitly provide authentication information to the underlying resource if it is available. The decision of what authentication information to provide depends upon the authentication mode and the available J2C authentication aliases. The details are quite complex, but in brief, any application can look up any resource in the JNDI namespace. When this is done, the authentication mode of "application" is used implicitly. This in turn means that WebSphere Application Server will use a component-managed authentication alias if one is available. Thus, any resource defined with a component-managed alias is accessible to any application in the cell. Note that defining resources at different scopes has no impact on security. Resource scopes are simply an administrative convenience.

On the other hand, if only a container-managed alias is defined on a resource (or no alias is specified), then a rogue application will not be able to access the resource because when it looks up a resource in the global JNDI namespace, application authentication applies and thus component-managed aliases are used. As a result, no implicit authentication can occur because there is no component-managed alias.

If you choose to use this approach, you will still want the appropriate applications to have access to these resources. To do this, those applications must specify local resource references in their deployment descriptors to access the resource. Those references can then be bound at deployment time to the correct resources. If the application specifies container-managed authentication on the reference, then a container-managed alias defined on the resource itself will be used implicitly. A picture might help to make this clear. Figure 6 shows the IBM Rational Application Developer reference editor where you specify container-managed authentication on a JMS connection reference.

Figure 6. Database resource reference using container-managed authentication


References

References

This web application references the following resources:

ResourceRef jmsreference

Name: jmsreference

Description:

Type: javax.jms.ConnectionFactory

Authentication: Container

WebSphere Bindings

There are two problems with this approach. First, container-managed aliases were erroneously deprecated in WebSphere Application Server V6.1 (but are no longer deprecated in V7). Second, and more significantly, you might have noticed that not all resources permit you to specify container-managed aliases (for example, JMS factories in V6.1). In that case, you have no choice but to provide no default authentication information on the resource. Instead, you must specify (probably during the deployment process) the authentication information for each resource reference in the application (the authentication method does not matter in this case). This is tedious, but it can be automated using wsadmin. At least in the case of MDBs, the authentication information can be specified on the activation specification.

XA recovery aliases are not an issue

Do not confuse component-managed aliases with XA recovery aliases specified on resources. The XA recovery alias is used only during certain recovery scenarios involving transactional failures. It is not normally accessible to applications if Java 2 security is enabled and the default permissions have been used.

2. Do not define a default user ID and password on a resource

N M E I

A corollary of the previous item is that you should not define a default user ID and password on a resource. If you do so, then any application within the cell can look up the resource and then implicitly use the provided user ID and password.

3. Java 2 security

N M E I

As discussed earlier, all application servers contain the WebSphere Application Server administrative infrastructure and thus the APIs for performing most administrative operations. An application programmer that learns the APIs can thus write an application that can call any of these APIs, and then perform essentially any administrative operation. In addition, the file system configuration repository contains a great deal of sensitive information (such as passwords). Any application can use ordinary Java I/O to read these files.

WebSphere Application Server includes support for Java 2 security as provided by the standard JDK. IBM has enhanced the Java 2 support to enforce the Java EE specifications as well as to protect the WebSphere Application Server internal APIs from unauthorized access. Simply by enabling Java 2 security, these rules are automatically enforced. Thus, by enabling Java 2 security, substantial additional protections are added to the runtime to prevent illegal application access. **However, without a formal process to review granted permissions, there is absolutely no value in using Java 2 security – in fact enabling it without policy file validation makes the situation worse** because there is no improvement in security, application development is more difficult, and performance suffers slightly.

Once Java 2 security is enabled, applications are limited to a very small set of "safe" permissions by default. If an application needs more permissions, it must typically define those requested permissions in the was.policy file contained within the EAR. When the application is run, the was.policy file is read and those permissions are added to the standard set. As should be obvious, this is a potential security hole. To make this viable, you’ll need a rigorous and well defined process for determining, reviewing, and then enforcing the Java 2 permissions for each application. If any application attempts to request permissions that are not acceptable (even during an emergency patch) reject the application. There should be a formal process that includes a security review that determines what permissions an application will be allowed. Even with a formal process, enabling Java 2 security should be approached with caution, aside from the additional effort involved, oftentimes a mandate to "enable Java 2 security" results in the use of policy files that provide little meaningful protection to make deployment possible, which not only takes additional effort, but also provides a false sense of security.

The process of review and verification upon installation can be tedious. However, there is an alterative approach. First, for a large set of environments, most applications will need a common set of additional permissions. If this is possible, the infrastructure team can place the default permissions for all applications in the app.policy file on that node. Then, only applications that need unusual permissions will place them in was.policy and require additional verification. You can take this even further by forbidding the use of was.policy and requiring that all permissions be added to app.policy by the administration team. That complicates deployment in some ways (editing a common file), but reduces the risk of an application obtaining inappropriate permissions.

A refinement on this approach is to specify Java 2 security differently by lifecycle environment.

- In development, run with Java 2 security enabled, and define the custom property com.ibm.websphere.java2secman.norethrow=true property. This results in warnings in the

logs of what Java 2 security permissions are required, but does not throw any Java 2 security exceptions.
- In test, run with Java 2 security enabled, but without that custom property defined. Applications will not run unless they include the necessary policy files. Changes to acceptable policy files in test must be rigorously scrutinized as described above.
- In production, do not run with Java 2 security enabled, thus avoiding the performance overhead. Only deploy to production those applications that have passed through the scrutiny in test.

Be aware that this refinement does not lessen the need for review and verification of policy changes.

(Enabling Java 2 security incurs non-trivial run time performance costs. Two very important Security custom properties can reduce this cost (but do not totally remove it). They are com.ibm.websphere.security.auth.j2c.cacheReadOnlyAuthDataSubjects=true, and a properly tuned com.ibm.websphere.security.auth.j2c.readOnlyAuthDataSubjectCacheSize. Read about these properties in the Information Center.)

4. Leverage secure chained delegation

N M E I

One of the great benefits of an application server is that user identity information is automatically sent between system layers and across applications. This gives you transparent single sign on (SSO). Unfortunately, this has one, potentially dangerous side effect: inappropriate impersonation.

The problem here is that when a user authenticates to use application A, it is possible that application A will then make a remote EJB call to application B. Application B then sees the credentials of the original user. Ordinarily that’s a good thing. But, what if application A can’t be trusted? In this case, a user that accesses application A has reason to fear that it might access application B on his behalf using delegation. Imagine the implications of this if application A is “unimportant” and therefore is developed in an ad-hoc manner, while application B is a highly sensitive application that manages confidential information. The problem here is that application now has to trust application A simply because they share a common security realm. That’s bad.

There is a way around this problem. You can use a feature in WebSphere Application Server that is roughly described as “secure chained delegation.” By using WSSecurityHelper.getCallerList() or getServerList(), application B can determine what applications and servers a request passed through. If application B is a highly sensitive application, it might require that the list be empty, implying that it is being used directly by the user. See the WebSphere Application Server Information Center for more information on WSSecurityHelper.

5. Protect the TAM WebSEAL TAI password

N M E I

When SSO is configured between IBM Tivoli Access Manager WebSEAL and WebSphere Application Server, WebSEAL sends its secret password in the HTTP header on every request to ensure that the TAI can work when invoked. While this is generally not a concern, because the connection should be encrypted using SSL, this does expose the WebSEAL password to Web applications running in WebSphere Application Server. (While this section discusses the WebSEAL TAI specifically, this exposure occurs with any TAI that sends a password to establish trust.)

The preferred approach, as described in Part 1 is to use the Enhanced Tivoli Access Manager TAI instead of the WebSEAL TAI, and to not use a secret password for the establishment of trust.

A Web application that is running in WebSphere Application Server and cannot be trusted could potentially take this password, open an HTTP connection to an application server and assert any user’s identity. This would make it possible for a cleverly written intrusive application to act as anyone.

If you are concerned about this type of attack (which can easily be prevented through code inspections), you can prevent untrusted clients from connecting to the Web container. To do this, simply configure a mutually authenticated HTTPS listener on the Web container, as we described Part 1. Then, an application will be unable to open an HTTPS connection to the Web container because they won’t have the right private key (only WebSEAL or the Web server will have it).

6. Beware of heightened permissions in custom JMX code

N M E I

An aspect of MBeans that you need to be aware of is that their use implies elevated Java 2 security permissions. If an application registers MBeans programmatically with the application server runtime, you must grant the calling code elevated adminstrative permissions for it to work. Do this with great caution. This ability could be used to perform illicit administrative operations. A good approach is to create a separate module just for registering the MBean and grant only that module the needed permissions.

A second way to load MBeans is to specify them administratively as Extension MBeans (the recommended approach). This eliminates the problem of having to explicitly grant application code administrative authority -- but raises a new issue: The MBeans are now loaded by a fairly low level WebSphere Application Server classloader, which is more trusted. As a result, your MBeans will have substantially more access to WebSphere Application Server APIs then ordinary user code.

If you choose to develop custom MBeans, you must carefully review the code and the usage to ensure you are not introducing security weaknesses into the system.

7. Use DynaCache with caution

N M E I

DynaCache provides a distributed shared cache for WebSphere Application Server applications. However, there is no access control on DynaCache; any application running in an application server can access any cache to which a given application server has access. More precisely, any application can access any cache that is accessible from that server, and then see (or modify) all of its contents.

There are two kinds of caches to consider. The application server itself maintains some internal hidden caches that might contain sensitive information (servlet caches, Web services caches, security caches). There are also user-created caches, such as Object Cache caches that are visible via JNDI.

Internal caches are not registered in JNDI so they are more difficult to find, but you can gain access to them by using internal APIs. On the positive, these caches are replicated only to servers within a cluster by default, so you can be confident that applications in different clusters can't see caches of information from other clusters.

User defined caches are visible in JNDI and can be looked up from any server within the same cell. An application can modify or read the cache once it looks it up. There is no way to prevent this. Therefore, if you are concerned about application isolation, you cannot leverage user-created caches.

8. Use all resources with caution

N M E I

Many other WebSphere Application Server resources, such as work areas, do not provide for application level authorization. As with DynaCache, you need to use these resources with care. If you are concerned about application isolation, you should carefully evaluate every usage scenario and look for potential weaknesses and act accordingly.

Cross-cell trust

Normally, WebSphere Application Server cells do not trust each other, and thus cross-cell SSO is impossible. However, you can configure cells to support cross cell SSO. There are plenty of good reasons to do this, but when you do, you are extending the trust domain of the cells and need to be aware of the security implications. There are three issues to consider:

Normally, a realm is exactly the same as a registry, but WebSphere Application Server V6.0.2 and later relaxes this by specifying the realm name independently of the registry.

- **Shared LTPA keys**
  In order for two cells to transparently participate in an SSO domain, they must share compatible realms (this means the same realm in WebSphere Application Server V6.1, while in V7 this means trusted realms), share the same LTPA encryption keys, and use compatible SSL keyrings (for server to server traffic). Compatible SSL keyrings simply means that the

calling server must have access to the signing certificate that corresponds to the receiving server’s certificate, as with any SSL communication.

Once you’ve ensured that two cells share the same LTPA encryption key, you have created a situation in which each cell has the ability to create credentials for the other cell – including administrative credentials. Therefore, if one cell is compromised, both cells are compromised. If you are using multiple cells as a way to achieve application isolation for security reasons, you need to enable Java 2 security to limit access to WebSphere Application Server internal APIs.

If your goal in sharing the same LTPA encryption key is merely to create Web SSO without sharing customized subjects, you can achieve the same result by using an authenticating proxy server, such as Tivoli Access Manager WebSEAL, and a corresponding TAI (such as the Extended TAM TAI) without sharing the LTPA encryption keys. Achieving Web SSO via a proxy and a TAI is preferable to sharing LTPA keys.

- **CSIv2 identity assertion**

  If you wish, it is possible to make cross-cell IIOP calls and avoid sharing LTPA encryption keys. To do this, you have to use CSIv2 identity assertion (this also works when contacting non-WebSphere Application Server EJB servers).

  Consider this simple scenario: Assume two cells, A and B, each of which contains servers. Also assume that servers in cell A need to make RMI/IIOP calls to servers in cell B, but not the reverse. For this to work, you configure CSIv2 identity assertion. Servers in cell A will assert identities to servers in cell B. How to configure CSIv2 identity assertion will not be described here, just the implications of doing so.

  In order for servers in cell B to accept the identity assertion, the upstream server in cell A must authenticate itself first. There are two ways of doing this with CSIv2: basic authentication, in which the upstream server sends its user ID and password, and client certificate authentication, in which the upstream server authenticates using its own certificate.

  When the authentication is complete, the receiving server will verify that the upstream server is trusted to perform identity assertion. This is configured on the CSIv2 configuration panels, after which the upstream server sends the downstream server the target user’s identity information.

  Let’s consider the trust implications of this approach. Servers in cell B trust cell A as they accept identity assertion from it. If cell A is compromised, so is cell B -- but, what are the implications for cell A?

  - If cell A sends the server user ID and password to establish trust (which is the default), it is revealing its server user ID and password with full administrative authority to servers in cell B. Thus, cell A now completely trusts B. This is not an improvement over just sharing LTPA keys.
  - If cell A sends a specified user ID and password (one that is configured just for this purpose) instead, it is revealing nothing of importance to cell B. Cell A has not trusted cell B.
  - If cell A authenticates using its own certificate, then it reveals nothing to cell B. Cell A has not trusted cell B.

In summary, in order for cell A to assert identities to cell B, cell B must trust cell A. That much is obvious. If you do not want cell A to trust cell B, use a specified user ID and password or certificate authentication for the server-to-server authentication step, not the default, which sends the server ID and password.

- **Subject propagation callbacks**
  If you are taking advantage of subject propagation across cells, you should be aware that the cells might also be making out-of-band calls to each other to obtain subjects. To make this clearer, consider an example: Assume that there are two servers that share an SSO domain. A user accesses server A using a Web browser and obtains an authentication session (represented by an LTPA cookie in the browser). The user then accesses server B. Server B will attempt to obtain the user's subject from server A. This is called subject propagation. This happens trivially if the servers are in the same DynaCache replication domain. If they are not in the same Dynacache replication domain, then JMX callbacks are used to obtain the subject. Of course, servers in different cells are not in the same DynaCache replication domain. Therefore, in this example, server B will make a secure JMX call to server A in order to obtain the user's subject.
  As with any administrative call, the JMX call requires authentication and authorization. In this case (as of WebSphere Application Server V6.1), that trust is established implicitly through the use of the shared LTPA key. Since server B shares LTPA encryption keys with server A, server A trusts it to ask for subject information.
  On balance, the callbacks don't significantly impact security beyond the risks already introduced by sharing LTPA keys, but they do introduce another network path that some might consider a risk.
  This situation does not apply when downstream propagation occurs using IIOP. In that case, the upstream server simply sends the subject to the downstream server. No JMX callbacks are required.

Identity propagation

While this topic isn't directly related to security hardening, it is a common problem that creeps into system designs that don't take security into account early enough. You must always be very careful to track where identities are established and how they are propagated (if they can be). Far too many designs simply assume the identity is known when practical technologies make this infeasible. Make sure you carefully analyze identity flow in your applications to prevent disaster late in the development cycle. Below are two common cases that involve external resources and, for WebSphere Application Server V6 and later, solutions.

- **Database versus WebSphere Application Server authentication**
  One of the key challenges of enterprise systems is properly implementing strong system security controls. In a nutshell, critical data needs to be protected with appropriate authorizations. What makes this challenge particularly difficult with multi-tier Java EE systems in which Java EE application code contacts a database (using JDBC, SQLJ, JPA, or CMP beans) is that, traditionally, the user's identity information is lost. "User" in this case means the user on whose behalf the application is running; that is, if Bob authenticates to the application using standard Java EE security, then Bob is the user.

In typical Java EE-based systems, the container maintains a pool of authenticated connections. While each user authenticates to the application server (using one of several Java EE authentication mechanisms), the user's identity information is not available to the database. This is because all database access is performed using one of several common shared connections from the connection pool. Historically, this has resulted in applications having to reinvent existing database level authorization and auditing functions inside the application layer. This is wasteful when done properly, and probably insecure when done poorly.

With WebSphere Application Server V6, there is an elegant solution to this problem. With V6.1, identity propagation to IBM DB2® is provided out of the box.

- **MDBs do not run under enqueuer’s identity**

  When a message is enqueued to a messaging system, the original caller’s identity is not normally relevant. That is, the messaging engine authorizes access to the queue based on the connection identity, as discussed earlier. The queue doesn’t usually even record the user’s identity.

  When the message is dequeued, typically by an MDB in Java EE, the original caller’s identity is not available -- even if it were available, it would be ignored. MDBs run under either an anonymous Java EE identity or a static run-as identity. They do not execute under the identity of the message enqueuer.

  There is no direct support in WebSphere Application Server for addressing this problem. However, you can make this work if you are willing to write some custom code. As of WebSphere Application Server V6.0.2, WebSphere Application Server now supports server side identity assertion. That is, server side code can change its Java EE run-as identity using JAAS without providing a password. Instead, it simply asserts user identity information. Here is a simple example of this using Java code:

  Listing 10

```
CallbackHandler wscbh =
 WSCallbackHandlerFactory.getInstance().getCallbackHandler(username, null);
LoginContext lc = new LoginContext("system.RMI_INBOUND", wscbh);
lc.login();
```

  Notice that no password is provided.

  This can be combined with the ideas from this article on advanced authentication to assert custom subjects. You can also cause WebSphere Application Server to create an LTPA cookie, making this work for Web applications, as described in this Information Center article.

  Clearly, this has security implications and so this call is blocked by default when using Java 2 security. However, you can provide for it by granting the application code the needed Java 2 security permissions.

WebSphere Application Server limitations

In general, through careful configuration, you can create a robust, highly secure environment using WebSphere Application Server. However, you should be aware of one typically minor and obscure limitation that might impact you.

Target identity not verified

One very subtle aspect of secure systems is the concept of validating the target of a request. Normally, when you think of authentication, you think of the server authenticating the client, but what about the reverse? How does the client know that the server is in fact the intended server? Most of us don't realize it, but Web browsers implicitly perform this check every day. When HTTPS is used, the Web browser validates that the hostname of the Web server is the same as the Web server's subject in its certificate. This ensures that the server is actually the server intended by the user (assuming, of course, that the user knows what hostname was just used; many aren't that careful).

What might not be obvious is that the check performed by Web browsers isn't some inherent part of SSL, it's a browser-specific check performed outside of SSL. The initiator (the calling server) has to specifically perform this check of the certificate. WebSphere Application Server does not perform this check. Therefore, when an application server (or client) opens an SSL connection to a server, it isn't ensuring that the server is really the intended server. Although highly unlikely, it is possible that a hacker who can compromise your internal DNS or network could then insert a rogue server into a WebSphere Application Server cell and steal information. This would be an incredibly difficult attack -- many other attacks are far easier -- but you should nonetheless be aware of this possibility.

You can remove any unneeded signing keys from the application server trust store files to prevent this. Only certificates issued by trusted signers can be used to execute this attack. Unverifiable server side certificates will be rejected by clients during the SSL handshake. If the trusted list is small (perhaps containing only the default self signed certificates), then this attack becomes very difficult. Because the trust stores do not include any CA signers by default, the only default signer to be concerned with is if you choose to add a CA signing certificate to the shared cell level trust store, you now expose yourself to this (albeit minor) risk.

Protect your desktop development environment

When thinking about security hardening, most people tend to focus on production systems, which are naturally the most important. However, you should also spend some time ensuring that other computers, including your desktop systems, are also reasonably secure.

For those running IBM Rational Application Developer (or the earlier IBM WebSphere Studio), this is a serious concern. The desktop IDE contains a fully functional embedded WebSphere Application Server runtime environment. This application server has open ports and is vulnerable to remote access in precisely the ways described earlier in this article. You need to address this.

Hardening the embedded application server

At a minimum, you should enable administrative security in the embedded application server test environment. Rational Application Developer 7.5 does this by default. This will prevent the most egregious types of attack in which an intruder can use the built-in administrative infrastructure to deploy rogue applications onto your desktop. If you are running Rational Application Developer under an OS user ID with administrative authority, this is quite serious. You might also wish to take some of the other hardening steps indicated in this article, although ensuring that the administrative infrastructure is secure is by far the most important step.

As an alternative to hardening the embedded application server, you might instead choose to install a desktop firewall product that blocks access to your computer. Such an approach can be quite effective if it is configured properly. However, a firewall that trusts your entire internal corporate network is of little value in this context.

Migrating from previous versions

Migrating from version to version of WebSphere Application Server raises the issue of backward compatibility. From a security perspective, the recommended approach to application migration is to build a complete new cell (from scratch) and then install your applications into the new cell (after rigorous testing, of course). By taking this approach, you benefit from all the default configuration changes that you made between releases.

If instead you choose to use WebSphere Application Server cell migration tools, which copy the existing configuration and applications to a new cell, you will **not** benefit from many of the improvements that have been made to security hardening with each major release. As you should expect, this is because we endeavor to make as few changes as possible to the existing configuration to ensure that applications continue to work unchanged. The side effect of this is that changes that have been made to security by default will not take effect. Thus, for example, if you migrate using the tools from WebSphere Application Server V6 to V7, then you should read both the V6 and V7 versions of this hardening article, rather than just the latest version.

Conclusion

This two-part article has covered a great deal of ground. While many aspects of security were discussed here, the core theme of hardening a WebSphere Application Server environment has been the main focus. Hopefully, you now have the basic information you need to secure your Java EE systems. Although not discussed here, you should look for other sources of information to harden other parts of your infrastructure. WebSphere Application Server is only one piece of a much larger puzzle.

A subset of the items described in this paper can now be checked using an automated tool known as the IBM Security Scanner for WebSphere Application Server, which is available for download.

Finally, if you are interested in learning more about WebSphere Application Server security, you can contact IBM Software Services for WebSphere for a customized on site class in WebSphere Application Server security. The class covers security hardening, customizing authentication, integration, single sign on, and a variety of other topics in depth.

Acknowledgements

I would like to thank my colleagues Bill O’Donnell, Paul Glezen, Simon Kapadia, and Tom Alcott for their valuable input and assistance.

I would especially like to thank Keys Botzum who has left me with big shoes to fill, and whose fundamental ideas will live on through future versions of this article.

Related topics

- Advanced security hardening in WebSphere Application Server V7, V8 and V8.5, Part 1: Overview and approach to security hardening
- All articles in this series, for all versions of WebSphere Application Server
- WebSphere Application Server V7 advanced security hardening, Part 1: Overview and approach to security hardening
- WebSphere Application Server V7 advanced security hardening, Part 2: Advanced security considerations
- WebSphere Application Server V6 advanced security hardening, Part 1
- WebSphere Application Server V6 advanced security hardening, Part 2
- Rational AppScan information
- The Top 5 Internal Security Threatsby Cindy Waxer
- Enterprise Application Security
- Redbook: IBM WebSphere 6.0 Security, SG- 246316, IBM Corp, 2005
- Understanding and Deploying LDAP Directory Services by Timothy Howes, et al, ISBN 0672323168
- Buffer Overflows – What Are They and What Can I Do About Them? by Larry Rodgers, CERT, 2001
- Understanding Malicious Content Mitigation for Web Developers
- Java Security Coding Guidelines
- Messaging security
- Securing connections between WebSphere Application Server and WebSphere MQ
- IBM Security Scanner Tool for WebSphere Application Server from IBM support (For V7.0 and earlier)
- IBM WebSphere: Deployment and Advanced Configuration by Roland Barcia, Tom Alcott, Bill Hines, and Keys Botzum, ISBN 0-131468-626
- Advanced Authentication in WebSphere Application Server
- Database identity propagation in WebSphere Application Server V6
- Apache hardening information (applies to IHS as well)
- Building Secure Servers with LINUX
- Firewall Port Assignments in WebSphere Application Server
- The (XML) threat is out there
- DB2 Technical Tip: Setting up SSL
- FIPS Configuration
- SSL, Certificate, and Key Management Enhancements in WebSphere Application Server V6.1
- WebSphere MQ Security heats up
- OWASP Web site discussing the risks of cookie stealing and the HTTP Only flag
- www.keysbotzum.com
- Download WebSphere Application Server V8.5 trial version