Solutions for a Secure Java Servlet Development

MARIA VLAD, CATALINA NUTA, MADALIN STEFAN VLAD,
VALENTIN SGARCIU
Faculty of Automatic Control and Computers,
"Politehnica" University of Bucharest,
313, Spaiul Independentei, Sector 6, Bucharest,
ROMANIA
E-mail: maria@ac.pub.ro, nuta_catalina@yahoo.com, madalinv@ac.pub.ro, vsgarciu@aii.pub.ro

Abstract: A web application contains resources that can be accessed by many users. These resources often traverse unprotected, open networks such as the Internet. In such an environment, a substantial number of web applications will have security requirements. Although the quality assurances and implementation details may vary, servlet containers have mechanisms and infrastructure for meeting these requirements that share some of the following characteristics: authentication, access control for resources, data integrity, confidentiality or data privacy.

Key words: servlets, security, java, jdk

1. Introduction

Early in the World Wide Web's history, the Common Gateway Interface (CGI) was defined to allow Web servers to process user input and serve dynamic content.

The Servlet API was developed to leverage the advantages of the Java platform to solve the issues of CGI and proprietary APIs. It's a simple API supported by virtually all Web servers and even load-balancing, fault-tolerant Application Servers. It solves the performance problem by executing all requests as threads in one process, or in a load-balanced system, in one process per server in the cluster. Servlets can easily share resources as you will see in this article.

1.1. Defining the concept of Servlet

A servlet is a Java^{TM} technology-based Web component, managed by a container that generates dynamic content. Like other Java technology-based components, servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamically into and run by a Java technology-enabled Web server.

Servlets are server side components. These components can be run on any platform or any server due to the core java technology which is used to implement them. Servlets augment the functionality of a web application. They are dynamically loaded by the server's Java runtime environment when needed. On receiving an incoming request from the client, the web server/container initiates the required servlet. The servlet processes the client request and sends the response back to the server/container, which is routed to the client.

Containers, sometimes called servlet engines, are Web server extensions that provide servlet functionality. Servlets interact with Web clients via a request/response paradigm implemented by the servlet container.

1.2. Defining the concept of Servlet Container

The servlet container is a part of a Web server or application server that provides the network services over which requests and responses are sent, decodes MIME-based requests, and formats MIME-based responses. A servlet container also contains and manages servlets through their lifecycle.

A servlet container can be built into a host Web server, or installed as an add-on component to a Web Server via that server's native extension API. Servlet containers can also be built into or possibly installed into Web-enabled application servers.

All servlet containers must support HTTP as a protocol for requests and responses, but additional request/response-based protocols such as HTTPS (HTTP over SSL) may be supported. The required versions of the HTTP specification that a container must implement are HTTP/1.0 and HTTP/1.1. Because the container may have a caching mechanism described in RFC2616 (HTTP/1.1), it may modify requests from the clients before delivering them to the servlet, may modify responses produced by servlets before sending them to the clients, or may respond to requests without delivering them to the servlet under the compliance with RFC2616.

A servlet container may place security restrictions on the environment in which a servlet executes. In a Java 2 Platform, Standard Edition (J2SETM, v.6.0) or Java 2 Platform, Enterprise Edition (J2EETM, v.6.0) environment, these restrictions should be placed using the permission architecture defined by the Java 2 platform. For example, high-end application servers may limit the creation of a Thread object to insure that other components of the container are not negatively impacted.

1.3. Servlet Lifecycle

A servlet is managed through a well defined life cycle that defines how it is loaded and instantiated, is initialized, handles requests from clients, and is taken out of service. This life cycle is expressed in the API by the init, service, and destroy methods of the javax.servlet.Servlet interface that all servlets must implement directly or indirectly through the GenericServlet or HttpServlet abstract classes.

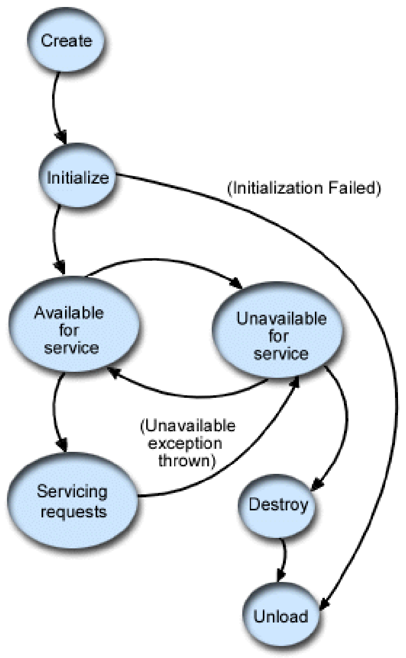

Fig. 1. Servlet lifecycle

1.3.1. Loading and Instantiation

The servlet container is responsible for loading and instantiating servlets. The loading and instantiation can occur when the container is started, or delayed until the container determines the servlet is needed to service a request.

When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class using normal Java class loading facilities. The loading may be from a local file system, a remote file system, or other network services. After loading the Servlet class, the container instantiates it for use.

1.3.2. Initialization

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC™ API based connections), and perform other one-time activities. The container initializes the servlet instance by calling the init method of the Servlet interface with a unique (per servlet declaration) object implementing the ServletConfig interface.

This configuration object allows the servlet to access name-value initialization parameters from the Web application's configuration information. The configuration object also gives the servlet access to an object (implementing the ServletContext interface) that describes the servlet's runtime environment. See Chapter SRV.3, "Servlet Context" for more information about the ServletContext interface.

1.3.3. Request Handling

After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type ServletRequest. The servlet fills out response to requests by calling methods of a provided object of type ServletResponse. These objects are passed as parameters to the service method of the Servlet interface.

In the case of an HTTP request, the objects provided by the container are of types HttpServletRequest and HttpServletResponse. Note that a servlet instance placed into service by a servlet container may handle no requests during its lifetime.

1.3.4. End of Service

The servlet container is not required to keep a servlet loaded for any particular period of time. A servlet instance may be kept active in a servlet container for a period of milliseconds, for the lifetime of the servlet container (which could be a number of days, months, or years), or any amount of time in between.

When the servlet container determines that a servlet should be removed from service, it calls the destroy method of the Servlet interface to allow the servlet to release any resources it is using and save any persistent state. For example, the container may do this when it wants to conserve memory resources, or when it is being shut down.

Before the servlet container calls the destroy method, it must allow any threads that are currently running in the service method of the servlet to complete execution, or exceed a server-defined time limit.

Once the destroy method is called on a servlet instance, the container may not route other requests to that instance of the servlet. If the container needs to enable the servlet again, it must do so with a new instance of the servlet's class.

After the destroy method completes, the servlet container must release the servlet instance so that it is eligible for garbage collection.

1.4. Usages of Servlets

Heterogeneous parallel computing. CGI programming can be done in almost any language, including C and C++. None of these languages are truly portable on different platforms (*e.g.*, Windows and Unix). Only the Java language is 'write once, run everywhere'. As the servlet is a pure Java solution, the actual server in our system can be any combination of hardware and operating system. The only requirement of the server computer is that it supports the execution of servlets. Indeed, we have tested our system in a heterogeneous environment consisting different Pentium computers and Sun Sparc workstations. In terms of operating systems, these included Windows 95, 98, NT, 2000, XP, Unix and Linux. We did not need to change a single line of our programs.

Persistency. A servlet stays in the memory of the computer as an object after it completes its operation. It can thus respond and execute faster than a CGI.

Security means different things to different people, but most will agree that securing an Information Technology system means controlling access to the underlying data so that only authorized users can read and modify the data.

There are several components of security that are needed to achieve this simple aim:

- ***Authentication*** is a means whereby users can identify themselves and be validated by the system. The most common form of IT authentication involves providing a username and password, but other techniques—such as digital certificates, smart cards and pin numbers, and biometrics (fingerprints, etc.)—are alternatives that can be used in some systems.
- ***Authorization*** is the process by which an authenticated user is granted access to various items of data. Authorization allows some users to read data; whereas others can read, modify, or update the data.
- ***Confidentiality*** means that only authorized users can view the data, and typically requires encryption of the data as it is transferred around the network.
- ***Integrity*** means that the data the user views is the same as the data stored in the system. In other words, the data has not been corrupted or changed when transferred from the server to the client. Data integrity is usually achieved by using data encryption. Integrity also means that when a user changes an item of data, that change is permanently made and cannot subsequently be lost. An audit log is used to support this aspect of integrity.
- ***Non repudiation*** means that if a user changes a piece of data, the system can prove who made the change and when, even if the user subsequently denies making the change. Audit trails and logging are used to support non repudiation.

2. SECURITY IN SERVLETS

2.1. Declarative Security

Declarative security refers to the means of expressing an application's security structure including roles, access control and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in web applications.

The Deployer maps the application's logical security requirements to a representation of the security policy that is specific to the runtime environment. At runtime, the servlet container uses the security policy representation to enforce authentication and authorization.

The security model applies to the static content part of the web application and to servlets and filters within the application that is requested by the client. The security model does not apply when a servlet uses the RequestDispatcher to invoke a static resource or servlet using a forward or an include.

A security role is a logical grouping of users defined by the Application Developer or Assembler. When the application is deployed, roles are mapped by a Deployer to principals or groups in the runtime environment.

A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of the principal. This may happen in either of the following ways:

1. A deployer has mapped a security role to a user group in the operational environment. The user group to which the calling principal belongs is retrieved from its security attributes. The principal is in the security role only if the principal's user group matches the user group to which the security role has been mapped by the deployer.
2. A deployer has mapped a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. The principal is in the security role only if the principal name is the same as a principal name to which the security role was mapped.

2.2. Programmatic Security

Programmatic security is used by security aware applications when declarative security alone is not sufficient to express the security model of the application. The word programmatic here means implemented from scratch, which is a good choice for authentication if users must have portability or if they want total control. Because it's more work than relying on their servlet container, programmatic authentication can be a bad choice if they are not interested in those benefits.

Another drawback to programmatic authentication is that HttpServletRequest.getUserPrincipal, HttpServletRequest.getRemoteUser and HttpServletRequest.isUserInRole are rendered useless for applications with programmatic authentication. Programmatic authentication requires you to implement, and use, your own API because setting principals and roles is strictly for servlet containers.

To prevent unauthorized access, *each* servlet or JSP page must either authenticate the user or verify that the user has been authenticated previously. To safeguard network data, each servlet or JSP page has to check the network protocol used to access it. If users try to use a regular HTTP connection to access one of these URLs, the servlet or JSP page must manually redirect them to the HTTPS (SSL) equivalent.

2.3. Authentication

A servlet-based web application can choose from the following types of authentication, from least secure to most:

- Basic authentication
- Form-based authentication
- Digest authentication
- SSL and client certificate authentication

Servlet authentication looks simple:

1. A user tries to access a protected resource, such as a JSP page.
2. If the user has been authenticated, the servlet container makes the resource available; otherwise, the user is asked for a username and password.
3. If the name and password cannot be authenticated, an error is displayed and the user is given the opportunity to enter a new username and password.

The steps outlined above are simple, but vague. It's not apparent who asks for a username and password, which does the authentication, how it's performed, or even how the user is asked for a username and password. Those steps are unspecified because the servlet specification leaves them up to applications and servlet containers. The servlet specification leaves enough security details unspecified that servlet containers must fill in the gaps with non-portable functionality. For example, the servlet specification does not specify a default authentication mechanism, so servlet containers implement their own; for example, Tomcat uses an XML file to specify usernames and passwords, whereas Resin requires implementing an authenticator.

Because of non-portable security aspects of servlet containers and depending upon user's choice for authentication, users may need to write some non-portable code, such as a Resin authenticator or a Tomcat realm. On the other hand, users can use declarative authentication to minimize any code they have to write.

Basic authentication is very weak. It provides no confidentiality, no integrity, and only the most basic authentication. The problem is that passwords are transmitted over the network, thinly disguised by a well-known and easily reversed Base64 encoding. Anyone monitoring the TCP/IP data stream has full and immediate access to all the information being exchanged, including the username and password. Plus, passwords are often stored on the server in clear text, making them vulnerable to anyone cracking into the server's file system. While it's certainly better than nothing, sites that rely exclusively on basic authentication cannot be considered really secure.

Form-based authentication allows you to control the look and feel of the login page. Form-based authentication works like basic authentication, except that you specify a login page that is displayed instead of a dialog and an error page that's displayed if login fails.

Like basic authentication, form-based authentication is not secure because passwords are transmitted as clear text. Unlike basic and digest authentication, form-based authentication is defined in the servlet specification, not the HTTP specification.

2.4. Custom Authorization

There are two aspects to authentication: *challenging* principals for usernames and passwords and *authenticating* usernames and passwords. The servlet specification requires servlet containers to allow customization of the former with form-based authentication. The servlet specification does not require servlet containers to allow customization of the latter, but most servlet containers let you do so. Because the servlet specification does not provide a standard mechanism for customizing authentication of usernames and passwords, that kind of customization is inherently non-portable.

Normally, client authentication is handled by the web server. The server administrator tells the server which resources are to be restricted to which users, and information about those users (such as their passwords) is somehow made available to the server.

This is often good enough, but sometimes the desired security policy cannot be implemented by the server. Maybe the user list needs to be stored in a format that is not readable by the server. Or maybe you want any username to be allowed, as long as it is given with the appropriate "skeleton key" password. To handle these situations, we can use servlets. A servlet can be implemented so that it learns about users from a specially formatted file or a relational database; it can also be written to enforce any security policy you like. Such a servlet can even add, remove, or manipulate user entries--something that isn't supported directly in the Servlet API, except through proprietary server extensions.

A servlet uses status codes and HTTP headers to manage its own security policy. The servlet receives encoded authorization credentials in the Authorization header. If it chooses to deny those credentials, it does so by sending the SC_UNAUTHORIZED status code and a WWW-Authenticate header that describes the desired credentials. A web server normally handles these details without involving its servlets, but for a servlet to do its own authorization, it must handle these details itself, while the server is told not to restrict access to the servlet.

The Authorization header, if sent by the client, contains the client's username and password. With the basic authorization scheme, the Authorization header contains the string of "username:password" encoded in Base64. For example, the username of "webmaster" with the password "try2gueSS" is sent in an Authorization header with the value:

```
        BASIC
d2VibWFzdGVyOnRyeTJndWVTUw
```

If a servlet needs to, it can send an WWW-Authenticate header to tell the client the authorization scheme and the realm against which users will be verified. A realm is simply a collection of user accounts and protected resources. For example, to tell the client to use basic authorization for the realm "Admin", the WWW-Authenticate header is:

```
        BASIC realm="Admin"
```

Custom authorization can be used for more than restricting access to a single servlet. Were we to add this logic to our ViewFile servlet, we could implement a custom access policy for an entire set of files. Were we to create a special subclass of HttpServlet and add this logic to that, we could easily restrict access to every servlet derived from that subclass. Our point is this: with custom authorization, the security policy limitations of the server do not limit the possible security policy implementations of its servlets.

3. Running Servlets Securely

3.1. The Servlet Sandbox

Servlets built using JDK 5.0 generally operate with a security model called the "servlet sandbox." Under this model, servlets are either trusted – and given open access to the server machine – or they're untrusted and have their access limited by a restrictive security manager. The model is very similar to the "applet sandbox", where untrusted applet code has limited access to the client machine.

What's a security manager? It's a class sub classed from java.lang.SecurityManager that is loaded by the Java environment to monitor all security-related operations: opening network connections, reading and writing files, exiting the program, and so on. Whenever an application, applet, or servlet performs an action that could cause a potential security breach, the environment queries the security manager to check its permissions. For a normal Java application, there is no security manager. When a web browser loads an untrusted applet over the network, however, it loads a very restrictive security manager before allowing the applet to execute.

Servlets can use the same technology, if the web server implements it. Local servlets can be trusted to run without a security manager, or with a fairly lenient one. For the Java Web Server 7.0, this is what happens when servlets are placed in the default servlet directory or another local source. Servlets loaded from a remote source, on the other hand, are by nature suspect and untrusted, so the Java Web Server forces them to run in a very restrictive environment where they can't access the local file system, establish network connections, and so on. All this logic is contained within the server and is invisible to the servlet, except that the servlet may see a SecurityException thrown when it tries to access a restricted resource. The servlet sandbox is a simple model, but it is already more potent than any other server extension technology to date.

Using digital signatures, it is possible for remotely loaded servlets to be trusted just like local servlets. Third-party servlets are often packaged using the Java Archive (JAR) file format. A JAR file collects a group of class files and other resources into a single archive for easy maintenance and fast download. Another nice feature of JAR files that is useful to servlets is that they can be digitally signed. This means that anyone with the public key for "Crazy Al's Servlet Shack" can verify that her copy of Al's Guestbook Servlet actually came from Al. On some servers, including the Java Web Server, these authenticated servlets can then be trusted and given extended access to the system. Users can create their owned signed servlets using a certificate generated by the JDK's key management tools. Alternately, they can obtain signed certificates from VeriSign or another certificate authority.

4. Conclusion

Security is an important aspect of applications that transport sensitive data over the Internet. Because of this requirement, the servlet specification requires servlet containers to provide implementations of basic and digest authentication, as defined in the HTTP/1.1 specification. Additionally, servlet containers must provide form-based security that allows developers to control the look and feel of login screens. Finally, servlet containers may provide SSL and client certificate authentication, although containers that are not J2EE compliant are not required to do so.

Unlike other aspects of web applications implemented with JSP and the Java programming language, security typically requires some non-portable code. If portability is a high priority, you can implement security can be implemented from scratch by using JSP and servlets, as illustrated in "Programmatic Authentication"

References:

[1] Allamaraju, S. et al. (2000). *Professional Java Server Programming J2EE Edition.* Wrox Press, Inc., 978-1861004659, U.K.

[2] Basham, B. et. al (2008). Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam, O'Reilly Media, 978-0596516680

[3] Callaway, D. (1999). *Inside Servlets: Server-Side Programming for the Java Platform*, Addison-Wesley Pub (Sd), 978-0201379631, U.S.A.

[4] Crawford, W & Hunter, J. (1998). *Java Servlet Programming*, O'Reilly Media, Inc., 1-56592-391-XE, U.S.A.

[5] Geary, D. (2001). *Java Servlet Authentication,* Prentice Hall, 978-0-13-030704-0, U.S.A

[6] Hall, M. & Brown, L. (2003). *Core Servlets and JavaServer Pages:Core Technologies, vol. 1,* Prentice Hall PTR, 978-0130092298, U.S.A.

[7] Hassan, D.; El-Kassas, S. & Ziedan, I. (2009). Developing a Security Typed Java Servlet, *Journal of Information Assurance and Security*, nr. 4.

[8] Murach, J. & Steelman, A. (2008*). Murach's Java Servlets and JSP, 2nd Edition*, Mike Murach & Associates, 1890774448, U.S.A

[9] Volkheimer, J. (2003). *Introduction to Servlets, JSP, and Servlet Engines*, *Available from* http://devcentral.iftech.com/articles/Java/intro_Servlet_JSP_Engine/default.php *Accessed:* 2009-11-02

[10] Wai-Sing Loo, A. (2007*). Peer-to-Peer Computing. Building Supercomputers with Web Technologies*, Springer London, 978-1-84628-381-9, U.K.