The Definitive Guide to Security in Jakarta EE

Securing Java-based Enterprise Applications with Jakarta Security, Authorization, Authentication and More

Arjan Tijms
Teo Bais
Werner Keil

Apress®

The Definitive Guide to Security in Jakarta EE

Securing Java-based Enterprise Applications with Jakarta Security, Authorization, Authentication and More

Arjan Tijms
Teo Bais
Werner Keil

Apress®

The Definitive Guide to Security in Jakarta EE: Securing Java-based Enterprise Applications with Jakarta Security, Authorization, Authentication and More

Arjan Tijms
AMSTERDAM, Noord-Holland, The Netherlands

Teo Bais
Utrecht, Utrecht, The Netherlands

Werner Keil
Bad Homburg vdH, Hessen, Germany

ISBN-13 (pbk): 978-1-4842-7944-1
ISBN-13 (electronic): 978-1-4842-7945-8
https://doi.org/10.1007/978-1-4842-7945-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Laura Berendson
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Vincent Law on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (www.github.com). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors

Arjan Tijms was a JSF (JSR 372) and Security API (JSR 375) EG member and is currently project lead for a number of Jakarta projects including Jakarta Security, Authentication, Authorization, and Faces and Expression Language. He is the cocreator of the popular OmniFaces library for JSF that was a 2015 Duke's Choice Award winner and has coauthored two books: *The Definitive Guide to JSF in Java EE 8* and *Pro CDI 2 in Java EE 8*. Arjan holds an MSc degree in computer science from the University of Leiden, the Netherlands. He has been involved with Jakarta EE Security since around 2010, has created a set of tests that most well-known vendors have used (IBM, Oracle, Red Hat) to improve their offerings, was part of the JSR 375 (EE Security) EG, and has been the main architect of the security API and its initial RI implementation Soteria. Arjan has also written and certified the MicroProfile JWT implementation for Payara. He has been mentored by Sun's (later Oracle's) security expert Ron Monzillo. He has written a large series of blog posts about EE Security that have attracted a lot of views. As such, writing a book about Jakarta EE Security is very natural to him.

Teo Bais is a software development manager, Scrum master, and programmer who contributes to the prosperity of the (software) community in several ways. He is the founder and leader of Utrecht Java User Group, which counts over 2600 members and has hosted over 45 events and amazing speakers (James Gosling, Uncle Bob, and over 20 Java Champions, among others), and is running three programs: Devoxx4kids, Speaker Incubator, and uJCP. Teo served JSR-385 (JSR of the Year 2019) as an EG member and was nominated as JCP Participant of the Year in 2019. Teo Bais enjoys sharing his knowledge as a public speaker to help others achieve their goals in career and life.

Werner Keil is a cloud architect, Eclipse RCP, and Microservices expert for a large bank. He helps Global 500 Enterprises across industries and leading IT vendors. He worked for over 30 years as IT manager, PM, coach, SW architect, and consultant for the finance, mobile, media, transport, and public sectors. Werner develops enterprise systems using Java, Java/Jakarta EE, Oracle, IBM, Spring or Microsoft technologies, JavaScript, Node.js, Angular, and dynamic or functional languages. Werner is Committer

at the Apache Foundation and the Eclipse Foundation, Babel Language Champion, UOMo Project Lead, and active member of the Java Community Process in JSRs like 321 (Trusted Java), 344 (JSF 2.2), 354 (Money, also Maintenance Lead), 358/364 (JCP.next), 362 (Portlet 3), 363 (Unit-API 1), 365 (CDI 2), 366 (Java EE 8), 375 (Java EE Security), 380 (Bean Validation 2), and 385 (Unit-API 2, also Spec Lead), and was the longest serving Individual Member of the Executive Committee for nine years in a row till 2017. Werner is currently the community representative in the Jakarta EE Specification Committee. He was among the first five Jakarta EE ambassadors when it was founded as Java EE Guardians and is a member of its Leadership Council.

About the Technical Reviewer

Yogesh Shetty works as a senior software engineer for a European financial institution based in Amsterdam. He is currently involved with designing and developing applications in the payments area. He loves to connect problems in the business domain with technologies in the solution domain – mainly using the Java/JEE suite of technologies.

When not working, he unwinds by cycling through the idyllic Dutch countryside or with a book in hand.

CHAPTER 1

Security History

This chapter describes the history of security in Jakarta EE, starting from its early conception and ending where we are today at the moment of writing. We'll take a look at how the security APIs themselves evolved, how various frameworks were created in response to restrictions and shortcomings in Jakarta EE security APIs, and who some of the people were that were involved in this. Note that we'll be mostly using the term "EE" throughout this chapter, even for those moments in time where it was called "Java 2 Enterprise Edition (J2EE)," or "Java EE." Likewise we'll be using Jakarta Authentication for the moment in time where it was called "JMAC" (Java Message Authentication SPI for Container(s)) or JASPIC (Java Authentication SPI for Containers) and use "Jakarta Authorization" for when it was called JACC (Jakarta Authorization Contract for Containers).

The Beginning

The story of security in Jakarta EE starts with how security in Java SE itself was conceived. Java was originally designed to support embedded usage such as running inside set-top boxes. At the time of its introduction, this shifted to so-called applets, which were small applications embedded in web pages that executed on the computer of the user visiting such web page. In that environment, the applet code is foreign and potentially hostile to the user. The local JVM on the user's computer therefore employs a security model that protects the user and the computer against this downloaded code doing anything harmful. In broad lines, this works by a system of permissions being assigned to downloaded application code, like a *java.io.FilePermission* that gives that code permission to access only certain files instead of permission to read and write all files on the file system. This kind of security is often called "code-based security."

A. Tijms et al., *The Definitive Guide to Security in Jakarta EE*, https://doi.org/10.1007/978-1-4842-7945-8_1

Security in Jakarta EE, and actually Jakarta EE itself, started with the release of the Servlet API in November 1998. At that time, there was no Jakarta EE yet, and Servlet was a stand-alone API. It was based on the work pioneered by early Java servers such as the Kiva Enterprise Server from January 1996, which later became the Netscape Application Server (and even later, eventually, became GlassFish), and more directly by the Java Web Server, which was released in December 1996.

In November 1998, the first version of Servlet backed by a specification, Servlet 2.1, was released. It did not contain much, if any, security provisions. The only thing present that remotely had any association with security was the *HttpServletRequest.getRemoteUser()* method. This originated from the time when there was barely such a thing as a Servlet application, let alone a .war archive. At that time, Servlets were akin to CGI scripts, which were called by a separate HTTP server. This HTTP server would carry out authentication itself if needed (mostly BASIC or DIGEST) and, when succeeded, would set the environment variable REMOTE_USER before calling the CGI script, or in this case, the Servlet. It's this CGI era variable that *HttpServletRequest.getRemoteUser()* corresponds to.

Around April 1999, on the Javasoft website, operated by Sun Microsystems, the first mention of a new security extension library for Java appeared:

> *We welcome comments and suggestions regarding the newly proposed Java™ Authentication and Authorization Service (JAAS) APIs. The JAAS framework augments the Java™ 2 Platform with support for both user-based authentication and user-based access controls.*

During the JavaOne 99 conference held between June 15 and 19, 1999, a software engineer working for Sun Microsystems called Charlie Lai held a presentation about that new security extension library for Java. Four years earlier at Sun, in 1995, Lai had been one of the people defining and developing the Pluggable Authentication Modules (PAM) library for Unix and Unix-like systems. PAM is a general API for authentication services. It allows a system administrator to add new identity stores (such as for the Unix password database, NIS, LDAP, and Radius) by means of installing new PAM modules. Central to PAM is the ability to modify authentication policies by editing configuration files.

The new security extension Lai was presenting about was essentially a Java-based implementation of PAM indeed called Java Authentication and Authorization Service (JAAS). Lai explained that Java 1.0 security was about downloaded code and sandbox restricted access, Java 1.1 adding signing to downloaded code and Java 2 adding a security

policy and domains. JAAS adds the concept of users and allows one to add permissions to a user via "running as" in a policy file, in addition to "code location" (where the code was downloaded from) and "code signers" (who signed the code). JAAS also adds "action code" as source next to "downloaded code." Action code here means a segment of code that is passed to a specific method as an (inner) class. To model this new user in code, JAAS introduced a new type: Subject. Its name is taken from the X.509 public key certificate that uses the Subject term for any kind of entity (not just persons). More specifically, a Subject is a bag of Principals and Credentials. At this early date, people asked why Subject had no explicit support for the Role concept. The answer given by Lai was that Roles should be specific types of Principals; to be in a role means having a Principal matching that role in this bag of Principals. Unfortunately, nobody at the time asked how one can determine that a specific Principal is in fact a role Principal. A seemingly trivial thing that would come to haunt Jakarta EE for well over 20 years to come. The identity stores that PAM just called modules are called login modules in JAAS. They came with a somewhat controversial concept at the time called a Callback, which is a very open-ended interface without methods (a marker interface) which the login module can use to communicate with its environment. When asked about this, the JAAS team responded that it allows for extensibility and that this design allows for a less painful evolution going forward.

Around October that year, the JAAS 1.0 Early Access class libraries were made available for download, allowing developers to get an early glimpse of the technology.

Enter Jakarta EE

Two months later, in December 1999, just before the turn of the millennium, the first version of Jakarta EE, EE 1.2, was released, containing Servlet 2.2 with a minimal but functioning security framework that seems to have been inspired by the way the aforementioned CGI servers worked; security is largely configured outside of the application. This is either fully outside the application by means of configuring the server on which the application runs or semi-outside the application by means of a deployment descriptor (xml file) that is stored inside the application archive, the latter often being split in two parts: a standard one such as web.xml and ejb-jar.xml and a nonstandard one such as sun-web.xml.

This type of security was soon after dubbed "container security," as it was the container (server) that was configured, as opposed to the application containing code to set up and check security. With it came two other terms as well: "declarative security"

and "programmatic security." In declarative security, one defines in a configuration file (typically an XML file) that a resource such as a URL or a method on an EJB bean is constrained (secured). In programmatic security, the code does explicit checks at certain places using the Jakarta APIs.

This initial version of Jakarta EE has a concept of security roles that are used to declaratively constrain URL patterns in web.xml or EJB bean methods in ejb-jar.xml. For the web layer, it introduces four authentication mechanisms, BASIC, DIGEST, FORM, and CLIENT-CERT, which can be set in the web.xml file. Furthermore, there are four main methods for programmatic security: HttpServletRequest.isUserInRole(), HttpServletRequest.getUserPrincipal(), EJBContext.isCallerInRole(), and EJBContext.getCallerPrincipal (and two variants involving the deprecated java.security.Identity type).

A few things are quite striking in this initial release. For starters, there's no explicit concept of the "identity store" (database of user details and roles). This is implied to be an implementation detail. Furthermore, it's not possible to extend the authentication mechanisms and neither is there any type or interface defined for the authentication mechanisms that are there. Striking is also the mirroring of essentially the same methods in the HttpServletRequest and EJBContext this early on. It is indicative of a fundamental problem in Jakarta EE where Jakarta EE is both a platform as a whole, where users would like to see a single *getUserPrincipal()* for the entire platform, and the desire to use specifically Servlet stand-alone (outside Jakarta EE). In other words, if a platform applicable method is in a class that's not part of Servlet, then a stand-alone container like Jetty would not implement it. However, if it's in Servlet-only class that's not applicable to other contexts, it needs to be duplicated.

That same month Lai together with others from Sun Microsystems and IBM published a paper about JAAS titled "Making Login Services Independent of Authentication Technologies." In it, using JAAS for servers and enterprises is mentioned, though there's no explicit reference to Jakarta EE.

Shortly after the release of EE 1.2, the work for EE 1.3 was started in February 2000. As for security enhancements, the plan states that "[...] 1.2 defines a basic architecture for declarative authentication and authorization. There have been requests to provide more flexible control of these in [...] 1.3."

Around March 1, 2000, JAAS 1.0 was released as a stand-alone companion library alongside JDK 1.2.1 and two months later for JDK 1.3.

Enter Jakarta Authorization

On April 3, 2001, Ron Monzillo, a security expert working for Sun Microsystems, became the spec lead of a new JSR filed by Sun: Java Authorization Contract for Containers (later simply called "Jakarta Authorization"). This JSR started before EE 1.3 was released but, due to the anticipated amount of work, targeted the next version of EE, EE 1.4. Its aim is to provide a much more rigid specification of the authorization aspects in security. Specifically, Jakarta Authorization has the aim to map declarative security constraints (such as *auth-constraint* in web.xml) to *java.security.Permission* instances, store these in a repository, and make these available to an authorization module. Furthermore, existing authorization queries in both Servlet and Enterprise Beans, like "has user access to URL" and "is user in role," can be answered by this authorization module by presenting these queries as checks for permissions.

On September 24, 2001, EE 1.3 came out, including Servlet 2.3 and Enterprise Beans 2.0, and the new Connectors 1.0. Connectors 1.0 has an elaborate security model and API. Specifically, it depends on JAAS, which at the time was still an independent framework. This dependency is, however, quite minimal, as the requirement consists merely of using the *javax.security.auth.Subject* type. Furthermore, Connectors 1.0, recommends (but does not mandate) that an application server uses JAAS modules (instances of javax. security.auth.spi.LoginModule, a.k.a. identity stores, a.k.a. authentication modules) for predominantly outbound security. There's unfortunately no concrete specification of how the JAAS modules should be developed, and the Connectors spec even says they are typically specific to an application server. Likewise, the configuration of a JAAS module is left to be specific to an application server as well. Because of Connectors, EE 1.3 has an official dependency on JAAS, but in practice, there's barely anything specified for it. For Servlet and Enterprise Beans, there's even nothing specified at all.

Among EE users, there's some confusion at the time about this, and this confusion would never really go away. Specifically, it results in the belief that the EE Servlet security elements such as *security-constraint* in web.xml and *HttpServletRequest.isUserInRole()* are part of JAAS, which is of course not the case. Not rarely people would report in, for example, support forums that they're using JAAS to secure their application while in reality only using Servlet security. As an example, the Wikipedia page for JAAS had for many years a section on Servlet's FORM authentication, stating that it was a part of JAAS.

On February 6, 2002, JDK 1.4 was released. The former stand-alone JAAS 1.0 library is now integrated into it, meaning later versions of Jakarta EE that depend on JDK 1.4 or higher would no longer need to have the explicit JAAS 1.0 dependency.

Enter Jakarta Authentication

With the Authorization JSR still in the works, on October 22, 2002, Ron Monzillo started on a JSR called the Java Authentication Service Provider Interface for Containers, later simply Jakarta Authentication. This JSR specifically aims to standardize the authentication mechanism concept, specifically the interface that must be implemented by an authentication mechanism and how to add this to a container. Like Jakarta Authorization, Jakarta Authentication from the onset targets both Servlet and Enterprise Beans. Also working on this JSR is Charlie Lai.

The JSR mentions that the authentication mechanism interface could be used to create, for example, a Liberty SSO authentication mechanism. This causes a little row, and eventually IBM votes against the JSR with the following comment related to this:

1) *Potential for adding the Liberty Bindings/Profiles into J2EE.*

   We do understand that Liberty is only one of many potential input specifications into this JSR. We feel that such a decision may have enormous implications and it is too big to be left to an individual expert group to make. The significance is such that the EC should give some direction, as a subset of major J2EE licensees support the WS-Security approach, and potentially a subset support the Liberty Alliance. One solution could be for the JSR to only deal with defining a provider interface for the establishment (process of authentication i.e validating authentication data like userid/password, or tokens) of an authenticated identity. Therefore, it would be authentication mechanism agnostic.

But it's not just Liberty that concerns IBM, the scope of the JSR also doesn't sit quite well with them:

2) *Over-broad scope for this JSR.*

   a) *The 1st major bullet has a sub bullet that is out of scope "the binding of received transport specific security credentials to authentication modules" as this should be "the binding of received transport mechanism to authentication modules". We feel that we should not mandate semantics in this JSR.*

b) *The 2nd major bullet opens up the scope:*

- *"validating mechanism specific credentials received from container transport mechanisms":*

* *This is out of scope and should be left up to the modules.*

- *The following bullets also talk about "run-as" and this JSR is performing identity and credential mapping / translation.*

* *This is out of scope for a "Authentication Service Provider Interface" JSR, this should be done in a separate JSR.*

- *"creating transport specific responses as necessary to implement authentication mechanisms"*

* *This is also out of scope as this JSR should not be defining any new transport specific mechanisms.*

A very early Spring user, Isabelle Muszynski, asked in April 2003 whether JAAS, with its pluggable authentication and authorization, would be a viable API to support in Spring. At that point, the Spring mailing list had just been opened a few months ago, and it would be almost a year until Spring 1.0 was released. Rod Johnson, the founder of the Spring framework, agreed it might be interesting. Though Spring at that point didn't have any security infrastructure yet, Johnson already had experience with security from his work on Spring's predecessor, the MVC framework he wrote for the FT group, making him knowledgeable on the subject.

Jürgen Höller in May of that same year took a look at Jakarta Authorization, but it left him somewhat disappointed. The main value that Jakarta Authorization adds, providing a repository of permissions based on, for instance, constraints in web.xml, isn't that much needed according to Höller. What he needs above all is an API for portable authentication. In a conversation with Muszynski and others, he argued that an "identity store" (which he called an "authenticator") is now always vendor specific. (Most servers ship with say a ready to use identity store backed by an XML file containing users and roles, but creating a custom one, especially one that's owned by the application, requires the use of vendor specific APIs.)

Here, Höller essentially very early on drew the conclusion that the J2EE security model is essentially targeted at integrating applications into an intranet situation, where a few user types such as admin and employee are centrally managed and shared by different, typically externally obtained, applications used by an office. The opposite

situation, a single rich web application with a tightly integrated application user base, where those users subscribe via the Internet and are fully handled by the application, doesn't fit this model very well. Rod Johson agrees security in J2EE is not really standardized well.

Foreshadowing Shiro Part I - IL DRBAC

Somewhere later that month, Les Hazlewood, a J2EE enterprise architect working at Transdyn Controls, Inc., was plowing through Google and tons of security books in a quest to find a suitable security framework for an enterprise application he was working on. This particular application has very strict but at the same time very dynamic security requirements. After a long search, he ended up finding very little to his liking. Disappointed, Hazlewood started to think about creating something himself and came up with a system he called "Instance-Level, Dynamic Role-Based Access Control (IL DRBAC)." Although intended to run on J2EE, it totally avoids using its existing security system. In his system, there are users having one or more roles, which are each mapped to java.security.Permissions. Permissions give access to instances of a resource, for instance, a web page. Incidentally, this is exactly what JACC does, though it's not clear whether Hazlewood is aware of it at that moment. Hazlewood's system specifically emphasizes dynamic behavior, in that all these elements can be created and modified during the runtime of an application and are immediately applied, even if users are already logged in.

In August 2003, a discussion took place among Tomcat users and committers. The topic of discussion was the almost mythical "j_security_check", which is the path the FORM authentication mechanism in Jakarta EE listens to. A Tomcat user, Al Sutton, was convinced Servlet filters should be invoked when mapped to paths containing this, but legendary Tomcat committer and Servlet EG member Craig McClanahan wasn't easily convinced. In that discussion, John T. Bell, who was writing the J2EE Open Source Toolkit book at that time, pointed out that the current state of security in Jakarta EE Security is too limited. Bell provided the following examples of limitations to make his point clear:

- Logins that require more than just user id and password.
- Support for optional login or login directly from a home page form

- Logins supporting a "remember me" feature
- Situations that require logging and retrieval or recording of information upon each login

Tomcat committer Remy Maucherat pointed out that Tomcat can be extended to accommodate this using Tomcat APIs such as Realm, Authenticator, and the Valve. Bell didn't buy this, however, and argued that this is hardly a valid way, as they are Tomcat-specific APIs. What was needed according to him was a standard API to do those things. Near the end of that month, Servlet spec lead Yutaka Yoshida provided a clarification:

> *In regards to this issue, servlet EG had a consensus that Filter must not be applied for j_security_check. We believe the application component should not be involved in the container-managed security. Although we understand why people are using filter to manipulate the authentication mechanism, it doesn't solve all issues related to the security and must be addressed in a larger scope of the portable authentication mechanism, which I expect to have in the next version of the specification.*

There are a few interesting things to note here besides just the actual clarification regarding j_security_check. The first is that in 2003, the Servlet EG was firmly believing in the total separation between security code and application code, which would be a recurring issue for years to come. Secondly, we see here that the Servlet EG had a plan to introduce portable authentication mechanisms in the Servlet spec. Eagle-eyed readers of the Servlet could have noticed this on April 17, 2003, in the Proposed Final Draft 3 for the Servlet 2.4 spec, where the following somewhat cryptic change note appeared in the appendix:

> *HttpSession.logout method was removed. The portable authentication mechanism will be addressed in the next version of this specification and logout will also be discussed in that scope.(12.10)*

On November 24, 2003, EE 1.4 was released, which included Servlet 2.4 and Authorization 1.0. There were no major security features added to Servlet, though the work on Authorization, and specifically the work to translate the existing constraints to the new set of permissions, resulted in various clarifications:

- Clarification that the security model is also applied to filter (12.2)
- Change the status code from 401 to 200 when FORM authentication is failed as there is no appropriate error status code in HTTP/1.1 (12.5.3)

- Clarification: “run-as” identity must apply to all calls from a servlet including init() and destroy() (12.7)
- Clarification of security constraints, especially in the case of overlapping constraints (12.8)
- Change the word “auth constraint” to “authorization constraint” (12.8)
- Clarification of “overlapping constraint” (12.8.1, 12.8.2)
- Login/Logout description added (12.10)

Authorization 1.0 itself greatly enriches the security framework in Jakarta EE. Having a permission store available makes for a much more powerful authorization model, and having pluggable authorization modules, which can essentially make any kind of authorization decision transparently to the applications running on a server, is indeed powerful. However, not all is well. The fixation on the concept of container security, where security is strictly separated from the application, has led to a setup where authorization modules can only be plugged in at the server level, or actually, only at the JVM level. Furthermore, the specification mentions system properties and a classpath to place authorization module jars on, but for modular servers, it’s not clear at all which location a server considers its classpath (if there even is such a location). More problematic is the fact that authorization modules are an all-or-nothing approach; they completely replace the existing authorization structure and are thus not able to simply add some additional rules. As if these issues aren’t problematic enough, there are even more serious issues in practice. Though the specification states that a default authorization module should both be present, as well as actually used in a Jakarta implementation, in practice, implementations ignore this. Some implementations don’t have a default authorization module present at all, just seeing it as an extension point to the server, while others have such authorization module but then by default use their own authorization code and have server specific switches to enable Jakarta Authorization.

Jakarta Authorization also suffers from the opposite problem we saw with *HttpServletRequest.getUserPrincipal()*, which was duplicated all over the place. Jakarta Authorization is a single, separate, spec that Servlet containers can use to enrich their security model. Except, they don’t.

Painfully missing from the EE 1.4 release is the highly anticipated Jakarta Authentication. In fact, nothing has ever been heard of it. According to the plan, a Community Draft would have to be delivered earlier in the year, followed by a Public Draft and a Proposed Final Draft, but nothing happened.

Enter Spring Security

In the same month that EE 1.4 was released, November 2003, Ben Alex, the managing director of his own company, Acegi Technology, looked at the new, at the time not yet officially released, framework sitting on top of Servlets called Spring. Alex inquired about its security features, but its creators, Ron Johnson and Juergen Hoeller, didn't have the time yet to fully look into it. Alex then started writing his own security code using Spring right away. He proposed it a month later but initially didn't get any response.

The following months, Alex gave his proposal more thoughts and struggled with questions such as whether to use JAAS or not, whether to use the authentication data from the container (the principal and its roles) or let the application populate some object for this, and more.

In February 2004 when people asked again about security and Spring, both Alex and Hazlewood seized the opportunity to pitch their work as a base for a Spring security initiative. Alex was very proactive and sent his work in a zip file to Johnson and Höller, proposing it to be simply called "Spring Security," and made a case for it to be part of Spring. At that time, however, Spring committers were frequently debating about the focus of Spring: Should it remain a small DI container, or should it include higher-level frameworks? For the time being, it was decided to develop "Spring Security" outside Spring as Acegi Security, after Alex's company.

On March 3, 2004, the initial public release of Acegi, Acegi 0.1, was presented to the world. At this early point, source code wasn't available from version control yet.

On March 6, 2004, Alex mentioned on the JBoss forum that for Acegi Security, he needed to write security adapters specifically for JBoss, since there was still no standard API for many security-related things. For instance, in this JBoss adapter, Alex obtained an import security object called the "Subject" in a nonstandard way using a JNDI lookup for "java:comp/env/security/subject" and then iterated over all the principles in this Subject until he found the Acegi one. Scott Stark replied that JBoss now marks the caller principal specifically by putting it in a group called "CallerPrincipal," although this is still JBoss specific and not standardized. More than 16 years later, this issue would still not be

solved in what's then called Jakarta EE, and Soteria, a Jakarta Security implementation, would use this exact same server-specific knowledge to obtain the caller principal.

Less than two weeks later, on March 17, 2004, Alex checked the code he had developed offline into version control. Its central class for authentication is AuthenticationManager:

```
/*
 * The Acegi Security System for Spring is published under the terms
 * of the Apache Software License.
 *
 * Visit http://acegisecurity.sourceforge.net for further details.
 */

package net.sf.acegisecurity;

/**
 * Processes an {@link Authentication} request.
 *
 * @author Ben Alex
 * @version $Id$
 */
public interface AuthenticationManager {
    //~ Methods ========================================================

     /**
     * Attempts to authenticate the passed {@link Authentication} object,
     * returning a fully populated <code>Authentication</code> object
     * (including granted authorities) if successful.
     *
     * (exceptions omitted for brevity)
     *
     * @param authentication the authentication request object
     * @return a fully authenticated object including credentials
     * @throws AuthenticationException if authentication fails
     */
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
}
```

The *Authentication* class it uses doubles as a credential, and an authentication outcome:

```
/**
  * Represents an authentication request.
  *
  * <p>
  * An <code>Authentication</code> object is not considered
    authenticated until
  * it is processed by an {@link AuthenticationManager}.
  * </p>
  *
  * <p>
  * Stored in a request {@link net.sf.acegisecurity.context.SecureContext}.
  * </p>
  *
  * @author Ben Alex
  * @version $Id$
  */
 public interface Authentication {
     //~ Methods ========================================================

      public void setAuthenticated(boolean isAuthenticated);

      /**
      * Indicates whether or not authentication was attempted by the {@link
      * net.sf.acegisecurity.SecurityInterceptor}. Note that  classes should
      * not rely on this value as being valid unless it has been set by a
      * trusted <code>SecurityInterceptor</code>.
      *
      * @return true if authenticated by the <code>SecurityInterceptor</code>
      */
     public boolean isAuthenticated();

      /**
      * Set by an <code>AuthenticationManager</code> to indicate the
        authorities
      * that the principal has been granted. Note that classes should not rely
```