An Oracle White Paper
October 2014

Security in Oracle ADF: Addressing the OWASP Top 10 Security Vulnerabilities

Overview

No application developer strives to write bad code that might compromise a company's security. To help guide developers on how to write secure software and to raise developer security awareness, the Open Web Application Security Project (OWASP) publishes a list of top 10 critical web application security vulnerabilities identified each year.

The OWASP Top 10 vulnerability listing is technology agnostic and does not contain language or framework specific examples, explanations, hints or tips. This paper provides framework specific hints and tips for the Oracle Application Development Framework (ADF) that can be applied to each of the top 10 security vulnerabilities documented in the OWASP Top 10 for 2013.

Disclaimer

This whitepaper discusses the security options and features available in Oracle ADF that help mitigate security risks published in the OWASP Top 10 list of security vulnerabilities for the year 2013. Note that the set of recommendations in this paper is not exhaustive and that no guarantee is given that implementing all the suggestions in this paper provides sufficient protection for all security threats listed in the OWASP Top 10. The reason for this disclaimer is that you cannot delegate responsibility for secure application development to a 3rd party or a single document. This document is to help developers that know security identify tools and features in Oracle ADF that they can use to implement application security. This paper does not replace a formal code review process.

Introduction

"The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted."

- OWASP Top 10, 2013[1]

Oracle ADF contains ADF Security, an integrated application security framework that leverages Java EE authentication and Java Authentication and Authorization Service (JAAS) for authorization. The ADF Security feature provides a declarative and visual development environment for building Oracle platform security services-based security into Oracle ADF applications. Together, Oracle ADF Security and Oracle platform security services (OPSS) enable developers to focus more on *what* needs to be protected rather than on *how* it should be protected.Security is not a one-way street. Many features in Oracle ADF – like bounded task flows – that are not primarily designed as a security feature can also be used by the security aware application developer to protect against known security threats.

To guide developers for what they need to protect against, the Open Web Application Security Project publishes an annual document that lists the 10 most critical security vulnerabilities identified for a year.

Addressing ten security vulnerabilities doesn't provide for total security, but is a good start in raising awareness to the current security threats. This whitepaper explains how to address security vulnerabilities and risks documented by OWASP for 2013, for Oracle ADF

OWASP Top 10 Security Vulnerabilities 2013

This paper addresses security vulnerabilities documented by the OWASP for the year 2013[2]. Lucky enough, the OWASP Top 10 list for the year 2013 doesn't differ much from lists published for previous years, except for changes in ranking. The listed security threats are probably the most severe threats and application developers to be aware of and protect against these threats

To help application developers to protect ADF web applications from security threats documented in the OWASP list of vulnerabilities for the year 2013, this paper discusses Oracle ADF practices and risk mitigation strategies for the following threats:

1. **SQL Injection** – The ability for users to add SQL commands in the application user interface that then are executed against the connected database. The risk associated with free SQL entry in an application is that users accidentally or deliberately may execute a harmful command that reveals, deletes or manipulates sensitive data.

[1] https://www.owasp.org/index.php/Top_10_2013

[2] https://www.owasp.org/index.php/Top_10_2013-Top_10

2. **Broken Authentication and Session Management** – The risk of application user impersonation and session hijacking that is associated with do-it-yourself session handling and authentication in an application if tokens or credentials are not secured.
3. **Cross-Site Scripting (XSS)** – The ability of users to input HTML snippets and JavaScript in an application for display in web views requested by other users. Typical candidates for this kind of attack are discussion forums or social networking sites that allow users to share information with others.
4. **Insecure Direct Object References** – An application may provide printed reports or invoices to authenticated users. If the file reference is not secured or is predictable in the addressing then the associated risk is for users to access other users' documents.
5. **Security Misconfiguration** – Security is a configurable feature in many ways: user IDs are stored in configurable identity stores, permissions are kept in configurable policy stores and options may exist for developers to temporarily disable security for debugging purposes. Applications that don't follow an approach of being secure by default will risk exposing sensitive information and data to unauthorized users in the case of a failed security configuration in production.
6. **Sensitive Data Exposure** – Not all data is public and caution should be used to hide sensitive information from unauthorized users. Failure in security configuration and the selection of insecure defaults may pose a risk data leakage.
7. **Missing Function Level Access Control** – To "ship or not to ship" is the question to be answered by authorized users e.g. in an online shop or manufacturing. Within applications, the shipment of an order is encapsulated in functions that are invoked by an application on behalf of a user. Therefore, sensitive functions must enforce authorization on the authenticated user.
8. **Cross-Site Request Forgery (CSRF)** – The attack attempts to forge an http request in the context of an authenticated user session. For example, a user could be presented with a link that, when clicked, issues a product order on his or her behalf. Often CSRF is implemented by means of Cross-Site-Scripting (XSS) in which the attack injects HTML content to a user page that, when loaded, automatically issues a request.
9. **Using Known Vulnerable Components** – Only in rare cases are larger applications completely built from scratch by a developer. Java EE applications usually leverage some 3rd party components, such as persistence engines, web frameworks, JavaScript and Java libraries. Defects in such components are documented, be it in bug reports or list of bugs fixed in the documentation of a newer version. Applications that utilize old versions of these frameworks or libraries are put at risk by these known vulnerabilities.
10. **Unvalidated Redirects and Forwards** – Java EE applications can be architected as monolithic applications. However, for better maintenance and performance, individual Java EE deployments are often used. These then call each other, passing the user application and authentication context along. Such Java EE modules that assume trust and don't validate the incoming request, risk unauthorized access or access outside of a valid application context.

Security Awareness and Education

The best application security money can buy is education. Developers and project leads need awareness of security issues as well as an understanding of secure coding practices. Training must include an in depth explanation of the potential risks as well as features of the development and deployment platforms that help mitigate exploits.

The most important design principle for application security is to implement security by design and default. Secure coding guidelines should be made available, adhered to and enforced in all development organizations, irrespective of the tools and platforms being used.

A good example for security by default is the expectation that we all have for how elevators behave in case of a power outage. Instead of releasing the breaks, we expect elevators to apply the breaks for the safety of passengers in the cabin. But how would the elevator know that it should apply the brakes if no one defined this as the default behavior? So before thinking about how to prevent external attacks, it makes sense to identify secure defaults for an application to protect it from the inside. This however does not work well without training and awareness.

The Risk Associated with "Build Your Own"

It is not always that developers immediately find the security they need for an application within the security toolset provided by a platform or built into a framework. As a result "build your own security" is not uncommon among development projects. This is especially true if the application is a replacement of an existing system that uses a specific non-standard security infrastructure. An example for this is database table based authentication and authorization in combination with user provisioning and resource granting at runtime.

The risk associated with building your own security is that you are also on your own when it comes to; quality assurance of the security layer, application security propagation and single sign-on, as well as being responsible for bug fixing and maintenance of the security layer.

Not all developers are security experts, but experts are what it takes to build a custom security layer.

Time spent investigating existing security solutions is probably time well spent. Existing solutions can be applied to custom applications easier and more cost effectively than creating an error-prone, self-written mechanism.

Oracle Platform Security Services (OPSS)

The Java EE security model provides high degree of portability and API consistency. However since it protects resources by request URI, it does not consider parameters or server side redirects, it fails to be a complete security solution for modern web applications. Java Authentication and Authorization Service (JAAS) provides better authorization but does not integrate well with Java EE.

If authentication is performed through JAAS, the Java EE containers are not required to manage the authenticated user.

To overcome the limitations in Java EE security, Oracle provides OPSS, a vendor specific security platform service that provides a consistent, portable and standards based solution for authentication, authorization, auditing, role and credential management. OPSS integrates security across Oracle WebLogic Server, Oracle SOA, Oracle WebCenter and Oracle Web Service Manager within Oracle Fusion Middleware (OFM).

With OPSS, application developers get the best of both worlds, the ease of authentication provided by container-managed authentication and the superior authorization support of JAAS. The Oracle ADF Security framework, which is an integral part of the Oracle ADF development platform, uses OPSS to perform user authentication based on Java EE and authorization based on JAAS.

Note: The Oracle Platform Security Services platform provides more than just JAAS integration to web application development. For more information please follow up with the Oracle Fusion Middleware Application Security Guide[3].

ADF Security Overview

The Oracle ADF Security framework provides several out of the box, mostly declarative features that make it easier for a developer to build secure ADF applications.

ADF Security automatically performs a JAAS security check whenever users access protected ADF bounded task flows or protected top-level ADF bound JSF pages. Authentication is not handled by ADF Security but delegated to the Java EE container. Application developers, however, retain full control over when user login and logout actions are processed.

Later in this paper, you will see how ADF Security can be usedto enforce authorization on user interface components and task flow activities. Leveraging OPSS on top of Java EE container security, Oracle ADF Security supports file, RDBMS and LDAP based policy stores for safely storing application user permissions.

It's important to mention that ADF Security is architected such that it supports separation of concern and responsibility by decoupling the development side of implementing security at design time from security administration and maintenance at runtime.

Note: Among other resources referenced later in this paper, a great source of information about ADF Security is chapter 21 of the Oracle Fusion Developer Guide: Building Rich Internet Applications with Oracle ADF Business Components and ADF Faces book published by McGraw Hill in 2010[4].

[3] http://docs.oracle.com/cd/E28280_01/core.1111/e10043/toc.htm

[4] Oracle Fusion Developer Guide, McGraw Hill 2010, ISBN 9780071622547

Security Design Patterns

In software engineering, design patterns are blueprints of best practices that guide developers to writing efficient, robust and scalable software. Design patterns are not implementation specific but provide enough guidance for developers to implement the documented best practices in any language or platform they use for building their applications. The following section touches briefly on a short list of commonly known patterns.

Pattern: Defense in Depth

Security should not be implemented in only a single layer, but should be backed up on other layers. For example, validation of user data input should happen on the view layer for convenience and immediate response, but then also enforced in the business service layer and the data layer (for example the database). The Oracle ADF architecture consists of multiple technology layers that can each use ADF Security to check for user authorization against JAAS policies.

Note: This paper is primarily concerned with addressing the OWASP Top 10 security vulnerabilities rather than being an in-depth discussion of ADF security. For a specific introduction to, and a hands-on tutorial on, ADF Security, read "Security for Everyone", an article published by Oracle Magazine[5]. For a video introduction to ADF Security see the ADF Insider "ADF Security" recording[6].

Pattern: Least Privileged Access

Users that work within application don't need to be assigned administrator privileges to the underlying database system. Instead application users should run an application with the least amount of privileges required to finish their tasks.

Pattern: Single Access Point

How many guards are required to secure Buckingham Palace? Certainly there are many guards assigned to protect the entries to the building and grounds, as well as an organizational infrastructure supporting it. The more entry points an application has the bigger the attack surface becomes that developers need to protect. Ideally, applications offer a single entry point. In Oracle ADF, a single access point is easily implemented using bounded task flows and setting their URL Invoke property to false or by using bounded task flows with page fragments.

What if an application needs to have multiple access points? If there is an architectural requirement to allow multiple access points – for example when parts of an application used by an internal audience are also available to external users – you should think compartmentalization.

---

[5] http://www.oracle.com/technetwork/issue-archive/2012/12-jan/o12adf-1364748.html

[6] http://download.oracle.com/otn_hosted_doc/jdeveloper/11gdemos/AdfSecurity/AdfSecurity.html

To compartmentalize, treat each of the access points as a module and ensure users cannot navigate from one module to the other without getting authorization checked. The OWASP security vulnerability ranked as 10 "Unvalidated Redirects and Forwards" points out the risk associated with compartmentalization if done wrong.

Pattern: Checkpoint

Checkpoint Charlie in Berlin achieved fame during the cold war as a connection between East and West, being used by both sides to exchange agents and prisoners. The role of a checkpoint is to enforce authorized communications between the layers, like the view and the business service in a web application. For this you can wrap method calls in security policy calls that determine whether a user is allowed to process the action he or she requested. In Oracle ADF, checkpoints can be created in Java, using a direct access to the ADF Security policy, or declaratively using Expression Language (EL) on view or controller components.

Pattern: Full and Limited View

Different users have different responsibilities and these responsibilities should mirror what they can see within the application. Information filtering should not only be for the UI but should also be performed on informative and error messages. Error messages that, for example, contain database constraint detail information, are not useful to the normal end user but quite handy to have for customer support assistance. People that try to exploit the vulnerability of a system also enjoy detailed error messages and stack traces as they can provide useful information about the software and implementations.

Pattern: Auditing

Users need to leave a record of their actions for administrators to audit a system for security breaches or to satisfy statutory regulations. How, for example, would you be able to tell a SQL injection attempt has been made if nothing is logged for later diagnosis? The same may be required for data changes. If you only keep track of the most recent change in a table, then you won't be able to understand the way in which that data has evolved. Depending on the application you build, you need to find hook points, like doDML in ADF Business Components (ADF BC) entities, or triggers in the database.

Pattern: Roles

Security administration can be a weak link in the application security chain as it involves human interaction and oversight. Complexity leads to mistakes, so simplification is a great policy. Using role based access control, systems are easier to manage than those which leverage individual access grants. In Oracle ADF applications, security is granted to roles defined as part of the application. These application specific roles are then are mapped to enterprise roles which in turn are assigned to users outside of the application purview. This way application security administration is decoupled from the identity management infrastructure, allowing developers to focus on what grants are needed to run specific parts of the application, without having to think about who actually will run a task. The

concept of enterprise users, especially in larger organizations, is helpful for administrators to not have to understand the role of an employee or to be aware of when employees join or leave.

Role based security keeps it simple and secure. Oracle ADF not only supports role based authorization, it also supports nesting of roles as well as entitlements, which is are groups of multiple permissions that can be issued with a single grant statement

Pattern: Session

When talking about a session, Java EE developers usually have the view layer HTTP session in mind. However, a session also provides a security context for the authenticated user that may span across the layers in an application.

Oracle ADF Security uses Java EE container managed authentication for tracking users and user group memberships. Based on the session context, authorization is enforced for a user automatically.

ADF Security Layers

In ADF, there are several places in which protection can be added. Figure 1 shows an example of protection areas commonly used when building Oracle ADF applications.

View / Controller	ADF Security	Business Service	Database
• Authentication • Page Authorization • Field Authorization • Input Validation	• Page Security • Task Flow Security	• Business method authorization • CRUD authorization • Input Validation	• DML authorization • Read authorization • PLSQL authorization

Figure 1: ADF security layers

View / Controller – The view layer represents the Oracle ADF application user interface. ADF bound views are subject to direct access control if they are top level documents or indirect authorization management through the ADF Controller if the views are fragments exposed in ADF regions. In either case, for users that are not authenticated, the view layer automatically triggers an authentication check on the first occasion that a protected view is accessed. In addition, the view layer can be used to hide components users are not allowed to see or invoke actions on. If you have a choice between disabling, hiding or removing components from a screen, the latter option should be your decision. Components of the view layer can be used to perform input validation to ensure no malicious data entry is passed on to the business service.

ADF Security – ADF Security is the tool that ADF application developers use to enforce authorization on views, controller actions and the ADF BC business service ADF Security is a declarative security framework for defining access control for views, task flows and business objects, but it also supplies an API that simplifies the process of working with JAAS for authorization.

Three key concepts are critical in understanding Oracle ADF Security and Oracle platform security services: *user identities, enterprise roles*, and *application roles*.

User identities define users in an enterprise. Users, such as company employees, usually have a single username/password pair they use to authenticate themselves to applications within an organization. A user identity defines only who the user is—it does not define any access privileges.

To ease system deployment and management, administrators often organize users into **enterprise roles**, which provide a way to manage groups of users who have similar requirements when accessing enterprise resources. For example, employees may all be grouped into an enterprise role called Employees to give them access to all employee self-service applications within an enterprise. From an administrative point of view, it is easier to add users to or remove them from an enterprise role than to maintain individual user grants for an application.

Application roles are specific to an application and are used to grant privileges to users defined in enterprise roles. Application roles make it possible for all users who belong to an enterprise role (such as Employees) to have specific access privileges defined for various applications. For users within an enterprise role to work within an application, application roles must be granted to the enterprise role. Application roles can be granted directly to users, but this practice is rare and is not considered good programming design.

Business Service – There are several options in ADF to access the business data and services needed by the application. The business service option used the most is the ADF Business Components object-relational mapping layer. This part of ADF is tightly integrated with ADF Security for protecting business entities, controlling modification and read privileges. Because ADF Security also provides a programming API, programmers can also develop more specific security rules using Java code within the ADF Business Components framework. For example, a rule might allows users to update a specific attribute of an entity when the entity is new, but prevent the update of that attribute once the record has been saved.[7] Similar ADF Security permissions can be defined to be checked before invoking lifecycle or custom methods within the ADF BC Service layer.

Database – The database (by which we mean persistence store in a general sense rather than specifically a SQL based RDBMS) may provide security features that ADF applications can use. The Oracle database, for example, provides label security, proxy user support and PL/SQL APIs that could be used with ADF to protect data.

Creativity

The requirement to ensure security in web applications and a desire to empower developer creativity are not mutually exclusive. As with any application development technology, to unleash the full power

[7] http://www.oracle.com/technetwork/developer-tools/adf/learnmore/76-insert-update-entity-protection-334421.pdf

of application security in Oracle ADF it helps if developers are able to think out of the box incorporating framework features that are not primarily designed for security in the overall application security strategy. In ADF, such features include

Servlet filters – ADF uses JavaServer Faces for the web user interface and JSF access is through a standard web servlet. Servlet filters allow you to look at incoming requests to block, redirect or allow the request to pass through.

JSF or ADF phase listeners – Similar to servlet filters, though slightly later in the request lifecycle, phase listeners in JSF allow you to listen and respond to incoming requests within the context of the JavaServer Faces request lifecycle. Phase listeners could, for example, enforce authorization on views or control the set of UI components to be rendered.

Component validators - JSF supports validation components to be added declaratively to user input components. Such validators can be used to check for suspicious patterns in the user provided input, for example to prevent scripting and SQL injection attacks.

Bind variables – SQL queries may use bind variables for the construction of where clauses to filter the data available to the user. The use of bind variables has a fortunate side effect of helping to protect against SQL injection attacks.

WEB-INF – All files that are stored under the WEB-INF directory located in HTML root folder in a web application project cannot be accessed directly from a browser URL field. Instead they can only be accessed through a servlet. This provides access control capabilities for those resources to the developer

MDS customization classes – Meta Data Services is the customization and personalization engine used by Oracle ADF. Customization classes are Java based decision points that determine the meta-data to be added to or modified in a view or business object for a particular request. Customization classes have access to the ADF security context so that user granted roles or permissions can be used to determine whether or not specific information is added or hidden from the user.

Task flow oriented architecture – Building ADF applications is all about task oriented development. Apart from ADF Security, bounded task flows provide additional features that can be used to improve security. First of all, bounded task flows enforce a single point of access, thus preventing users from randomly accessing content by manipulating the URL. Secondly, developers can control if their task flows can be used in isolation or if they must always be surfaces within the controlled context of a consuming application.

UI component rendered property – All JSF components expose a rendered property that you can set programmatically or declaratively using information from the ADF security context. This way a component or group of components can be removed from the page if the user of the page does not have a suitable role or permission grant.

Note: In the following this paper explains how security aware developers can address threats as listed in the OWASP Top 10 in Oracle ADF. Keep in mind that where there is a top 10 most likely also is a top 100. Addressing the OWASP Top 10 in Oracle ADF is a big step forward towards securing your ADF web applications but is not a single checklist that guarantees invulnerability.

How-to address the OWASP Top 10 with Oracle ADF

In the following, this paper addresses the OWASP Top 10 security vulnerabilities in the order they appear in the OWASP document published for the year 2013

OWASP #1 - SQL Injection

All data input that define or modify server-side queries to a system must be validated to avoid the risk of SQL injection. Note the use of the word *system* instead of *application* for this requirement. Nowadays, the term *application* no longer sufficiently describes the boundaries of where user interaction starts and where it ends. Different access channels may be used independently to query and write the application data. For example, Web Services may be used from mobile frontends, whereas direct SQL and PL/SQL statements may be used from Oracle Application Express (APEX) or SOA may use the database adapter to access the database as part of a business process. All of these channels need to be protected against SQL injection attacks. ADF can contribute to this protection by ensuring the ADF Business Component model is secured against SQL query manipulation.

Validate user input

The best place to protect against SQL injection is at the point where the query is defined, which in ADF mostly is the ADF Business Components business service layer. In situations where the application developer does not control the business service layer used with Oracle ADF, for example when using services, you can only hope that the web service model mitigates the risk and assist with client side validation added to those input fields that directly feed into web services calls.

With Oracle ADF there are several layers for you to perform user input validation and most of them are explained in more detailed in the section that discusses OWASP #3 – Cross-Site Scripting (XSS).

Commonly, SQL injection attempts are detected by comparing user provided input against a list of known SQL expressions allowed within the specific context. The technique in which user input is validated against a list of allowed values and keywords is known as whitelisting. The counterpart of whitelisting is blacklisting. In blacklisting user input is checked against a list of values and keys that are not allowed in a specific context.

Note that usually whitelisting is preferred because if it fails it doesn't open a security hole and instead most likely only annoy the application user who tried a valid SQL keyword that however is missing in the list.

User input comparison to white- or blacklists are through regular expression validation, which you can add to ADF Faces components and the ADF binding layer.

Hint: It does not make sense to validate all user input fields against SQL injection. Data input that is only persisted in the database but not executed does not cause harm even if the provided input contains SQL keywords. It's your responsibility as a security aware developer to identify areas that require SQL injection protection and to add protection.

Use Bind Variables

ADF Business Components view objects hold the SQL expression used to query the underlying database. As usual in SQL, ADF BC queries are filtered using a WHERE clause.

For example, let's explore the following query defined in a view object:

Select EMPLOYEES.LAST_ID , EMPLOYEES.LAST_NAME from EMPLOYEES;

If developers append the WHERE clause at runtime based on direct and un-validated user input provided in an user interface input component, the query may look as shown below:

Select EMPLOYEES.LAST_ID , EMPLOYEES.LAST_NAME from EMPLOYEES WHERE EMPLOYEES.DEPARTMENT_ID = 60;

There's nothing wrong here, however, the query could also be defined as shown below, which actually finalizes the initial query to the drop whole table.

Select EMPLOYEES.LAST_ID, EMPLOYEES.LAST_NAME from EMPLOYEES WHERE EMPLOYEES.DEPARTMENT_ID = 999 or 1=1; Drop table EMPLOYEES;

Another use of SQL injection, less destructive but arguably just as harmful, is to sniff for information about data that is not displayed in the application but available in the data schema.

For example, to get information about the salary of Steven King (and for the sake of simplicity, let's assume there is only a single employee with the name "King" in the system), a SQL string like shown below could be used iteratively

Select EMPLOYEES.LAST_ID , EMPLOYEES.LAST_NAME from EMPLOYEES WHERE DEPARTMENT_ID=60 and exists (select "x" from SALARIES where name="King" and salary between 1000 and 20000);

If the query above returns a result displayed in the application user interface then an attacker would knows that the salary of Steven King is somewhere between 1000 and 20000. Subsequently, the attacker would narrow the salary range until he or she has obtained the required information.

This example shows that no matter how good authorization might be when applied to entity changes and function execution, it's worthless if SQL can be successfully injected into the application via an input field on the UI.

To avoid SQL injection attacks as shown above, you can validate the user provided data entry for the WHERE clause or even better use a bind variable instead:

Select EMPLOYEES.LAST_ID , EMPLOYEES.LAST_NAME from EMPLOYEES WHERE EMPLOYEES.DEPARTMENT_ID = :DepartmentIdVar;

Be careful with dynamic View Objects

As stated above, using ADF Business Components declarative development options for VOs in combination with bind variables to filter queries is the best option for protecting against SQL injection attacks. However, there are business cases in which developers need to create view object definitions dynamically at runtime. By doing so, the SQL query to be executed by the view object is often derived

from user provided input. Such code requires extra scrutiny and to ensure that any such composed query does not contain fragments of a SQL injection attack.

To reduce the risk of SQL injection when building view objects dynamically, we suggest you …

- Handle the SQL query, WHERE clause and ORDER BY strings separately. If the query statement is provided as a single string, split the string at SQL keywords; WHERE, ORDER BY, UNION, GROUP BY etc. The Select strings provided in a UNION and EXISTS statements should be checked separately.
- Use regular expressions in Java to analyze the composed query string against a list of SQL expressions you don't allow to be part of the query. Example expressions that could be forbidden in user input might include: "where", ":=", "==", ";" (semicolon), "drop", "grant" etc. Alternatively, check against allowed vales (whitelists)
- Use bind variables in the WHERE clause to add the filter value. Using bind variables to fill argument values helps people from injecting SQL as the query argument (in case you re-use the same query with different query filters). The WHERE clause string should be checked with Regular Expressions for black listed expressions like '; (semicolon)', "drop table" etc.

OWASP #2 - Broken Authentication and Session Management

A user session is a working context that holds instance specific application data for a user. If a user session is authenticated, then authorization can be enforced so application functionality executes in the boundaries of a user's permissions and privileges. ADF does not handle authentication or manage user sessions itself but delegates these tasks to Java EE servers such as Web logic Server and thus to experts.

Still there are practices for you to consider for your deployed ADF applications to ensure the legitimate owner owns the user sessions.

Passwords – Ensure passwords are securely saved in an encrypted format in an identity store. Policies should be applied to the quality of passwords that enforces strong password using mixed characters, mixed cases and a decent password length. The lifecycle of passwords should be such that passwords need to be frequently changed by the user and that the same password cannot be used subsequently. To change passwords, a single mechanism should be used. Applications should not be storing passwords either in memory or in the database in an unencrypted form (if at all)

SSL – Authentication is where the user provides username and password in a login form. This login form must submit the user credentials with transport layer security (SSL) applied to ensure passwords are not sent in clear text format. Keep in mind, you own the application but may not necessarily control the whole network end-to-end. Always assume that all traffic between the user's browser and the application server is observable.

Authentication – There are several ways to perform authentication, ranging from weak forms such as Java EE basic authentication, to stronger approaches, using certificates or single sign-on. Ensure strong authentication to be used for your ADF applications for maximum protection.

> **Note** that you cannot disconnect from basic authentication. Basic authentication is also known as browser SSO because it's the browser that authenticates the client for a specific realm and domain. Even after invalidating the applications session, with the next request a new authenticated session is created, which leaves a lot of room for cross-site scripting and request forgery attacks. The only way to disconnect from basic authentication is to close the browser.

Trust relationship – Oracle ADF supports different application architectures based on the application size and business requirements. "Pillar" is an architecture in which a large application is broken up into multiple individual ADF applications that each is deployed as a Java EE application. When doing so, it is important that security does not break at the joins. Each Java EE application must authenticate the user and not rely on trust relationship created through tokens being between the calling and the called application. In such architectures, single sign-on should be used.

Identity management generally extends beyond the scope of a single application and thus should not be part of the application functionality but handled on an administrative level. Applications that maintain their own user and policy tables may be vulnerable if the developer implementing authentication and session handling is not a security expert and if the implementation hasn't been tested by ethical hackers. It's always best to delegate authentication and session management to a professional security and server environment.

Note: ADF does not handle single sign-on itself but delegates the task to the security provider, which is OPSS in Oracle Web logic Server.

Get error messages right – Application login that authenticates users based on the provided username and password pair, plus an additional check, for example an account lock flag, may inform users why authentication fails. Being too specific in the error reporting may provide a hacker with useful information about the target system. For example, if the login logic first checks whether an account is locked based on the provided username without validating the provided password to be correct then any error report on the locked status of the account provides the hacker with a confirmation that they had a valid username (from which they may be able to deduce other valid usernames) and that there is a locking policy on the system. The most secure form of message in this situation is to only state that the system could not authenticate the request. Any more detailed information could go into an application log in case the user failing to authenticate files a support ticket or request.

ADF allows you to configure a custom error handler in the `DataBinding.cpx` file, for you to filter exceptions and messages. The custom error handler should be configured in the `DataBinding.cpx` file of the deployed application and not in the same file within reusable bounded task flow project. The product documentation provides an example of how to create and configure a custom error handler.

OWASP #3 - Cross-Site Scripting (XSS)

As mentioned before, all data input to a system must be validated before it is persisted in the database. Especially data input that is later re-displayed on the application user interface exposes a risk of cross-site scripting.

To enforce valid data input in Oracle ADF, you should implement the following strategy

1. Limit the number of free-text input fields and replace them with select choices
2. Validate all user input on more than one layer
3. Escape output data

Validate all user input

Data input validation is key when it comes to preventing cross-site scripting. In Oracle ADF, input validation can be handled with JavaScript on the browser client, on ADF Faces components, in the ADF binding layer, the business service, for example ADF Business Components, and the database.

The business service layer is the gatekeeper and must validate all data input independently of how the data is accessed. Other application layers can provide additional complementary support, implementing defense in depth and providing timely feedback to end users about invalid data inputs.

JavaScript validation in Application Security

JavaScript executes in browsers, which is an inherently hostile environment that application developers have no control of. SQL injection and cross-site scripting attacks cannot be prevented using JavaScript.

For example, let's try and protect against SQL injection and cross-site scripting attacks by ensuring that a user can only add numeric data input to a text field. For this, the following JavaScript could be used in ADF Faces:

```
function onlyNumbers(evt) {
    var keyCode = evt.getKeyCode();
    isNumeric = (keyCode >= 48 && keyCode <= 57)
                ||(keyCode >= 96 && keyCode <= 105);
    var filterField = evt.getCurrentTarget();
    var oldValue = filterField.getValue();

    if (!isNumeric) {
        filterField.setValue(oldValue);
        evt.cancel();
    }
}
```

Shown below, the ADF Faces client listener behavior tag needs to be added to the input component for the JavaScript function to be invoked whenever a user presses a key on the computer keyboard while the focus is in the input field.

```
<af:inputText label="Filter Character Keys" id="it1">
   <af:clientListener method="onlyNumbers" type="keyDown"/>
</af:inputText>
```

While the above sample does suppress non-numeric data input, should you bet a farm on this level of security? Certainly not! There are at least three vulnerability issues with this sample:

- Browsers may behave differently when executing JavaScript and a script that does what it is supposed to do on one browser may fail on another.
- The listener could be rendered inactive in the page by a knowledgeable hacker and the check bypassed
- The communication between the client and the server can be intercepted using a debugging tool for malicious users to add SQL injection or cross-site scripting scripts.

So what is JavaScript validation in ADF Faces (or any browser deployed application) good for?

JavaScript in ADF Faces is good for implementing client-side validation for immediate user feedback. It's a convenience for end-users that may have mistyped data entered in a form. In any case, client-side JavaScript validation must be re-enforced on the server in the business service or view layer to provide serious protection.

ADF Business Components Entity Validation

Because entity objects are referenced from one or many view objects, adding validation on the entities level ensures consistent validation to be performed throughout an application. When defining validation rules, ADF Business Components provides both, declarative and programmatic options, including a set of built-in rules. In addition to this, lifecycle methods can be overridden in custom base classes to add security as a generic functionality.

The pre-built set of declarative validation rules in Oracle ADF BC is usually sufficiently rich to meet most of developer requirements. To use declarative validation, open the entity in the entity editor and select the **Business Rules** menu entry).

You use the **Business Rules** editor to declaratively define entity- and entity attribute validators, as well as entity lifecycle callback triggers.

Entity Validator – Validation rules that are defined for the entity will execute on row currency change for the selected entity row and when a transaction is committed. The benefit of entity level validation is that all attributes are available for you to cross-validate a single place.

Figure 2 shows the Business Rules editor with the dialog to create or edit entity validation rules. The validation rule illustrated in the image, references a Java method that exposed on the entity implementation class to determine whether the user provided input to attributes of the entity is allowed or not.

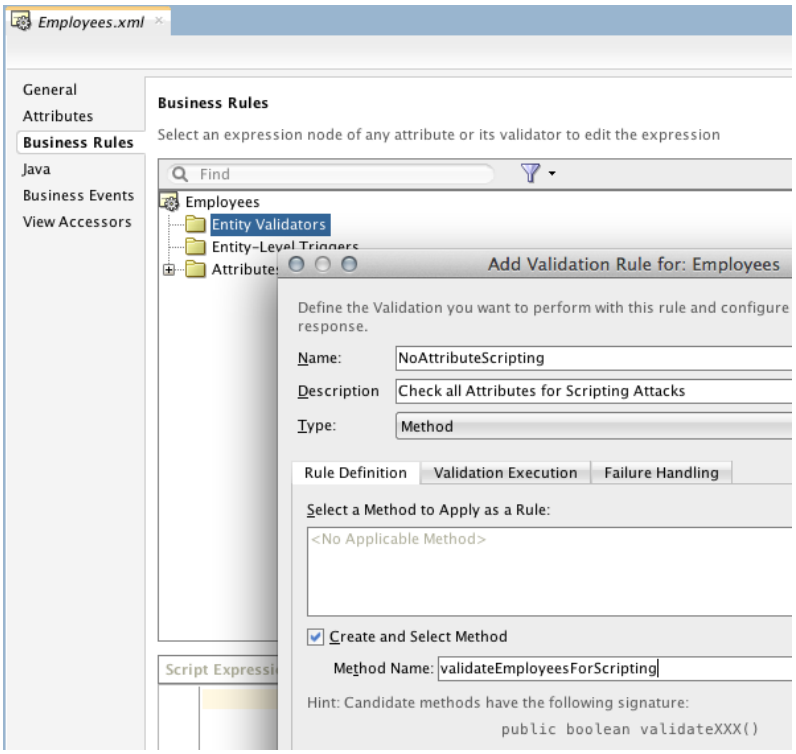


Figure 2: ADF Business Components entity validation

For the purpose of this paper, let's have a look at a simple but effective scripting prevention logic that looks for the opening '<' and closing '>' markup characters:

```
public boolean validateEmployeesForScripting() {
  //check attribute values for scripts
  Object[] attributeValues =  this.getAttributeValues();
  //pattern to check for '<', '>' and the hex encoded equivalent
  Pattern _pattern =
       Pattern.compile("(\\%3E)|(\\%3e)|(\\%3C)|(\\%3c)|(<)|(>)");
  Matcher _matcher = null;
  //check all string attributes if they match the pattern. Return
  //false on first match
  for(Object attributeValue : attributeValues){
    if(attributeValue instanceof String){
      _matcher = _pattern.matcher((String) attributeValue );
      boolean found = _matcher.find();
      if(found){
        //todo: incident reporting here
        return false;
      }
```

```
    }
  }
  return true;
}
```

When invoked, the method accesses all attributes available for an entity to check their data type to be `String`. If a string type is found, the attribute value is then checked against a regular expression string that looks for the '<' and '>' characters and their encoded versions. The method returns Boolean false if validation failed and true if the string passed validation.

The above example code assumes that no attribute value requires the use of ">" or "<" as valid data entries. If this were the case, then a more defined regular expression would be needed to avoid false positives. Commonly required validation methods can be defined in a custom base class to enable declarative reuse across multiple entity definitions.

Note the example above uses blacklisting, which is not as strong as whitelisting. Blacklisting is used in the sample for brevity and because the key point being to show how regular expressions can be used in ADF Business Components.

Entity Attribute Validator – The validation rules that are defined on the entity attribute level execute before the entity is updated with user input. Attribute level validation is the most declarative and centralized option to check for scripting attacks. This validator is called before whole-entity validation.

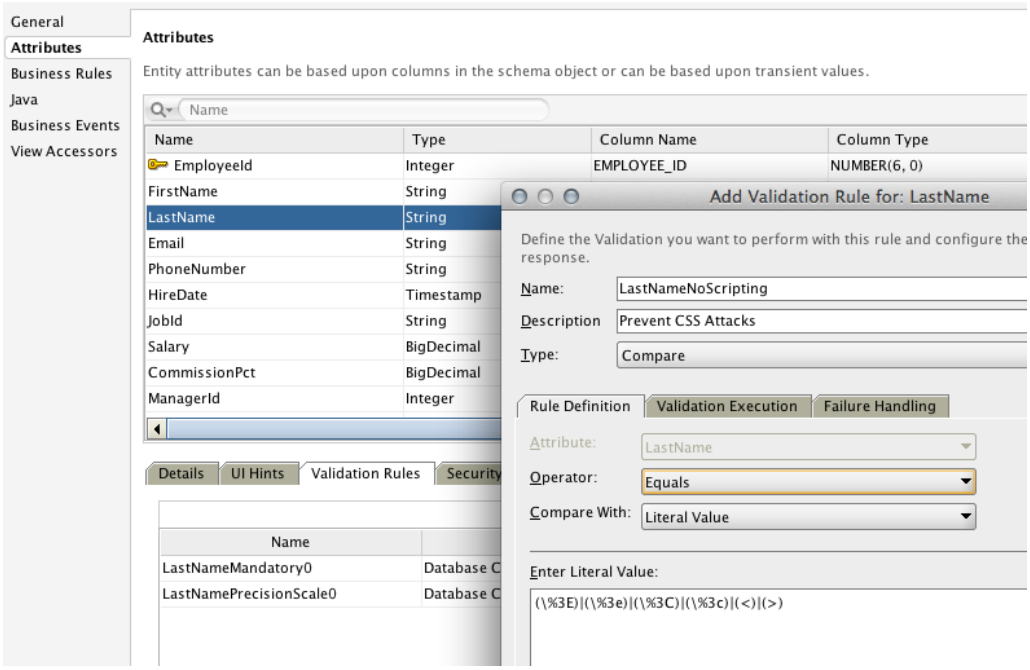


Figure 3: ADF Business Components entity attribute validation

The attribute level validator definition in Figure 3 uses the same regular expression as used in the Java method validator on the entity level: (\%3E)|(\%3e)|(\%3C)|(\%3c)|(<)|(>). The difference here is that the validator is defined for a specific attribute and that it uses one of the provided declarative

validators for regular expression matches. However it is also possible to implement method validation for attributes.

Entity Lifecycle Callback – entities follow a specific order of processing when data is created, changed or deleted. This processing order is referred to as the entity lifecycle. One of these lifecycle methods is `validateEntity` and is called when entity attributes are set or changed or when the entity row is removed. You can override the `validateEntity` method for individual entities in a custom entity class, or for all entities using a custom framework base class. A custom framework base class for example, could be used to add code that checks for cross-site Scripting attempts on all string attributes so the same validation code doesn't need to be added to all entities individually.

Declarative Validation on the ADF Data Control Layer

As discussed above, ADF Business Components provides its own declarative validation framework that is tightly integrated with the ADF request lifecycle. For other business services, like POJO, Web Service and Enterprise JavaBean (EJB), Oracle ADF data controls provide the option for developers to declaratively add validation rules to attributes exposed in a collection. To mitigate the risk of cross-site Scripting, developers can use RegularExpressions in a similar way to how it can be done in ADF Business Components.

All non-ADF Business Components data controls are defined in the `DataControls.dcx` file, a registry of data controls, located in the model project. Open the Data Controls Registry editor with a double click on the `DataControls.dcx` file to see the data controls and the collections they contain.

To add validations to attributes of a collection, select a collection and click the pencil icon shown in Figure 4.

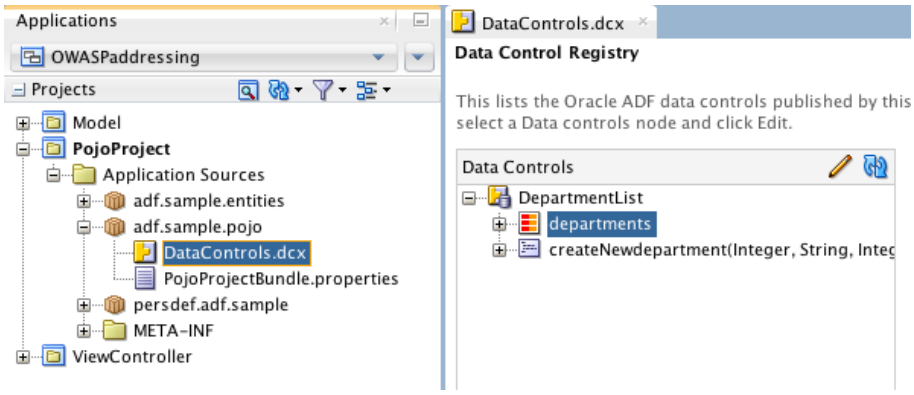


Figure 4: POJO data control definition in DataBindings.cpx editor

In the collection editor, click the **Attributes** menu option and select the attribute to define validation for. Switch to the **Validation Rules** tab and press the green plus icon to create a new validation rule. For XSS prevention, choose the **RegularExpression** validator and add the expression to be executed when the attribute is updated. The same dialog is used define the error message that is shown in case validation fails.

All validation rules defined on the data control are enforced independently of the view that accesses an entity or attribute therein. So unless validation is specific to a page, all validation should be added to the data control.

Hint: ADF data controls don't allow defining validation on the arguments of an exposed method. To validate the fields of an input form created from a method binding, you use declarative validation on the ADF binding layer or in ADF Faces.

Declarative Validation on the ADF Binding Layer

Developers can define declarative validation rules directly on attribute value bindings defined on the ADF binding created for ADF pages. This approach can be useful in some cases (such as the aforementioned parameter form validation); however, it generally makes more sense to apply the validation more centrally on the ADF Business Components entity or the data control.

JSF Validators

ADF Faces provides a regular expression validator to check user input against a pattern allowed for the value of a particular item. The regular expression validator behavior tag is contained in the JDeveloper Component Panel for ADF Faces where it is listed under the "Operations" tag category (Figure 5).

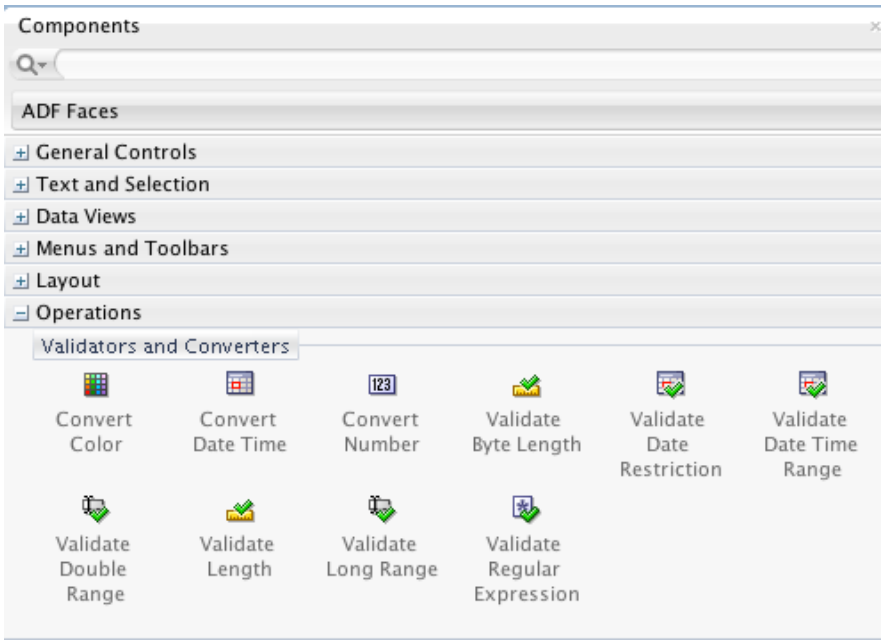


Figure 5: JSF validators in ADF Faces

The validator is dragged from the component palette and dropped onto an input field component. The Java regular expression to validate the user data entry could be added in the Property Inspector.

Any user entry that doesn't meet the regular expression leads to a validation failure that shortcuts the JSF request lifecycle, preventing the update of the model and ensuring that no invalid input finds its way into the business service.

Cross-site scripting prevention may be needed on multiple input components within a screen or application. Because the pattern to prevent such scripting injection in user input doesn't change, you should consider creating a custom JSF validator that then can be reused without the developer having to be relied up on to specify the correct regular expression.

To create a custom JSF validator, you write a Java class that implements the `javax.faces.validator.Validator` interface.

The validation logic itself is coded in the `validate` method of the custom validator and throws a `ValidatorException` in case of validation failure. In the case of XSS prevention you probably will use Java RegularExpression in the `validate` method to validate the user input.

The custom validator then is registered in the JSF faces-config.xml configuration file so it can be used within an application. In addition to server side component validation, ADF Faces also provides client side validation defined in JavaScript. To learn about how-to write custom ADF Faces validators, refer to the "Developing Web User Interfaces with Oracle ADF Faces" book in the Fusion Middleware product documentation.

Note: ADF Faces user interface components that have their *immediate* property set to true, bypass client side validation. If the component uses a value change listener to access submitted values of other user interface components that have their *immediate* property set to *false* (the default), then those values are provided in a raw string format. No validation and object conversion has been processed on the other components at the time that the value change listener fires on the component with *immediate* set to true. In such cases, either ensure that components that you need to access values from within a value change listener also have their *immediate* property set to true, or call the `processValidators` method on the component Java reference.

JSF Converters

On the web, all user input is submitted as a string to the server where it is converted to the target data type such as number, date, or similar. Conversely, data objects retrieved from a business service are converted into a string format before data is displayed in the browser. Converters are invoked in the context of the JSF request lifecycle. Like for JSF validators, you can create custom converters that, for example, escape user input or data output to protect against XSS attacks or XML encoding for content that should be passed on to a SOAP service.

ADF Faces Component Out-of-the-box Security

Output text components in ADF Faces run in a safe mode by default and escape their values. This means that any HTML markup inside of the string is encoded at runtime to display the HTML as source rather than interpreting it. So, for example, a value of hello will be converted to hello rather than sending the markup intact to display **hello** in bold. Developers may disable this character escaping by setting the component "escape" property to false but should think twice before doing so, and you should pay special attention to these cases in code reviews.

The Risk of Custom Ajax Calls

ADF Faces provides partial refresh and partial submit functionality for page content to issue requests to the server without refreshing the whole page. A new behavior tag in JDeveloper 12c, `af:target`, provides even finer control over which components to execute for a request and which components to refresh in response. Using the ADF Faces JavaScript client framework, you can use JavaScript to do asynchronous calls to the server that still operates in the context of the JSF request cycle.

Developers who implement their own Ajax calls to the server, e.g. calling a server side servlet, for querying data need to be aware that they are bypassing the JSF request lifecycle and therefore all the provided framework validation and conversion. Oracle recommends using only ADF Faces for user input communication with the server. If you nevertheless want to use custom Ajax calls in your ADF applications, be aware that you need to protect this access channel yourself. Don't rely on JavaScript validation for this but implement server side protection.

OWASP #4 - Insecure Direct Object References

Direct object references allow users to access documents or data record based on provided value input, such as a parameter in a request URL. An example of direct object references is the retrieval of images or documents from a database is demonstrated in the Oracle ADF Fusion Order Demo (FOD)[8].

```
<af:image source="/imageservlet?thumbnail=#{row.ProductId}"
          id="i7" shortDesc="#{row.ProductId}"/>
```

With the source example above, the FOD sample queries thumbnail images from the database using a servlet to query the image from the database. In the sample, the servlet path is not protected and this accessible to both anonymous and authenticated users. As you can see, it is not difficult to use this object reference directly in a browser URL field to exploit the system for images not displayed on a page. In the FOD example case, however, there is no sensitive data accessed through the servlet so the approach is acceptable. However, what if the images were not product pictures but sensitive documents such business or sales reports? In this case code as shown above would impose a security risk.

Another common example of URL parameter based addressing is to access business reports that are created on the server on a user's behalf and then downloaded as a document, for example in Adobe PDF format, to the client. A bad design for this would be to have documents saved in an unprotected folder on the server with predictable document names.

To protect such URL based references, you can apply the following security measures in Oracle ADF using ADF Security and Oracle database protection.

[8] http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html

Use protected folders and resources – Save documents and files in protected folders only. All access to files, documents and database objects should require authorization. If you use a servlet to download documents to the client, protect the servlet path with a security constraint defined in the web.xml file (or annotations if you developing with Servlet 3.0 in Java EE 6).

Indirect object references – Virtualized access to sensitive documents and files by mapping randomly generated names to a document. Using a database table in a database scheme that is different from the application schema you can access control the mapping between a virtual document name and the real file or object. The schema – for example – could be accessible through stored procedures only and run in the context of the authenticated web user. Another way to virtualize object references is through the use of content management systems that enforce security.

Choose random file or document reference names – Don't follow any sequence or pattern in creating document names or references. Names should be random so they are nor predictable.

Monitor access attempts – Assuming that your locations are protected and need authentication is not enough. A hacker may obtain valid credentials for a user and once inside of the system start to systematically fish for sensitive data. You should have measures in place to detect and react to this in a suitable way

Apply data security – Oracle label security ensures authenticated users to only see what they are allowed to see by dynamically adding a predicate to the query. Using ADF Security and ADF BC, the same can be implemented on a View Object as well. For this you create a ViewCriteria that uses a bind variable to filter the query as shown in figure 6.

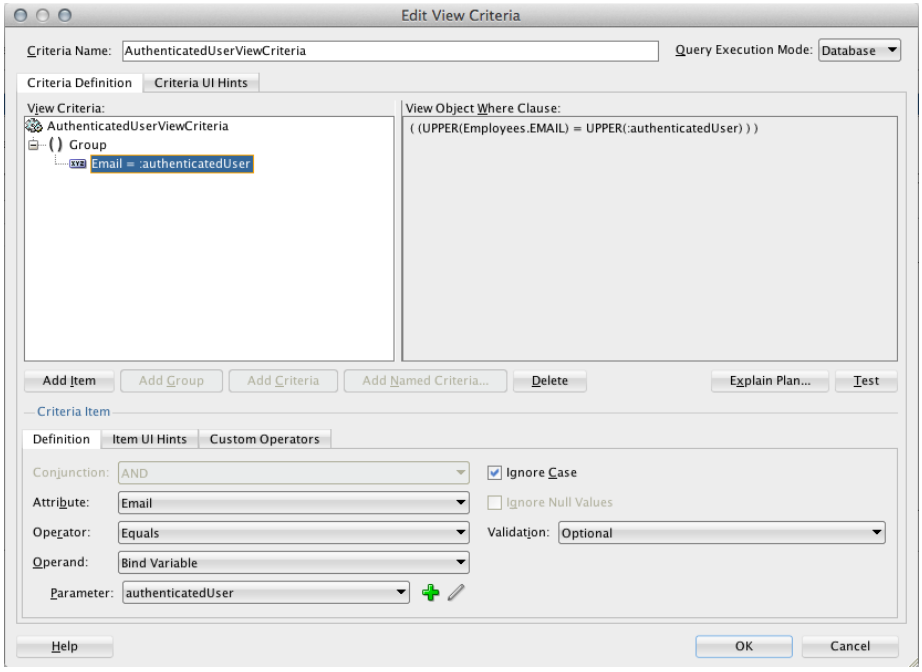


Figure 6: View Object view criteria definition with bind variable

The bind variable uses a Groovy expression in its value field similar to the one shown in figure 7 for queries to execute with a username predicate:

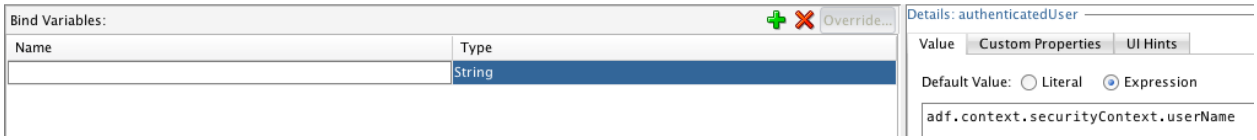


Figure 7: Groovy expression to access authenticated user from bind variable

The ViewCriteria can then be added to a ViewObject instance using the Application Module editor as shown in figure 8. In JDeveloper 12c and later, view criteria can be assigned such that they cannot be removed at runtime.

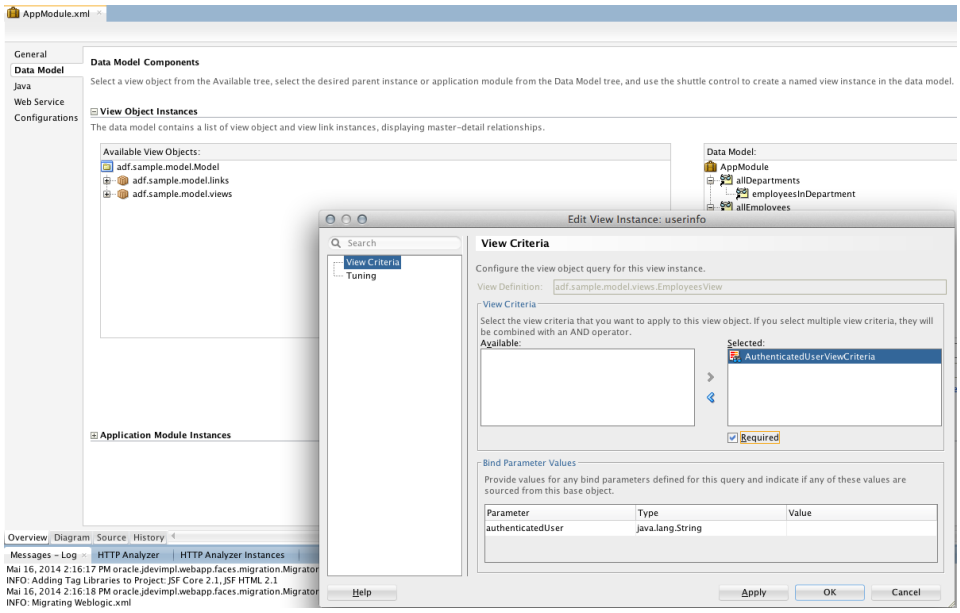


Figure 8: ADF BC View Object instance editor to assign mandatory view criteria

Note: In cases where you directly access documents and referenced objects from a JDBC connection or content management system, you still have access to the authenticated user to pass this information along with the query.

Expire document references – References to sensitive documents and objects should expire within a reasonable time after creation.

OWASP #5 - Security Misconfiguration

Security misconfiguration could happen on all layers, including the database, the Java EE server, the network, but also the application itself. For this paper, all configurations that are not related to ADF are considered out of scope and it is assumed that you have procedures in place that ensure ADF runs

in a secured environment. Therefore, for this section on security misconfiguration, only Oracle ADF is covered.

ADF Security Misconfiguration

In ADF, ADF Security is enabled to enforce authentication and authorization on ADF applications. For this, the ADF configuration file, `adf-config.xml`, is updated with the security setting shown in figure 9 when the ADF Security wizard in Oracle JDeveloper is completed for an application.

```
<sec:JaasSecurityContext initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"
                         jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                         authorizationEnforce="true" authenticationRequire="true"/>
```

Figure 9: ADF configuration that enforces authentication and authorization

Misconfiguration leading to ADF security being disabled for a running ADF applications, may occur in the following situations

Application Not Configured for Production – ADF security may be disabled in the adf-config.xml file during development to test and debug ADF application or to diagnose a particular issue. If in an oversight security is not re-enabled, the application could inadvertently be published to production unprotected. To avoid this, there should be a mandatory checklist that ensures protected applications have the correct security settings (as shown in figure 8) configured.

Incomplete security configuration – Developers may diligently configure security on bounded task flows, which provide reusable components for application to use within Oracle ADF. However, if such a flow is being reused, this security configuration will only apply if the consuming application enables security. Developers that are not aware of this requirement may forget to enable authentication and authorization when assembling applications out of such reusable artifacts. Any secured bounded task flow should have its security requirements documented along with the policy definitions the application assembler must add to the assembling application's jazn-data.xml file before deployment.

Test-role oversight – ADF Security ensures no default role created for testing purposes gets deployed with the application. However development teams may create their own test roles and grant it to the anonymous user identity for security agnostic testing and development. **In such cases, if the test roles are inadvertently deployed with the application, disabling authorization at runtime for production applications.** To avoid problems like this, ensure naming conventions are put in place that indicate a role is to be used only for testing so that administrators can ensure that it is not deployed with the application.

No hardcoded users or back doors - For ease of development, testing and defect diagnosis, developers may be tempted to build in back-doors or special user accounts into the system. Features such as this represent a huge security risk to the application, even if created with the best intentions. You cannot rely on these backdoors remaining confidential and their presence could undermine all of your other security precautions.

Implicit Defaults

All frameworks, including Oracle ADF, have a default behavior. In Oracle ADF, for example, default behaviors are defined for the task flow "URL Invoke" and the "Library Internal" properties, which have a bearing on the security topic.

URL Invoke – The URL Invoke property's default setting is "calculated", which means that a bounded task flow is accessible from a http GET request issued in a browser URL field, if the default activity is a view and the view uses a whole-page document format such as JSPX or JSF. This default value for the property is necessitated for the framework to maintain backwards compatibility with code produced to run on earlier versions. To ensure maximum security, however, the recommendation is to set this property to "disallowed".

Library Internal – bounded task flows can be hidden from display in the JDeveloper Resource palette. This way you can ensure that only top-level task flows, and no sub-flows, are visible to the application developer allowing you full control over the APIs that they consume and preventing misuse of components out of context. The default setting for Library Internal is false meaning that the task flow is visible.

But why are the two settings above mentioned in a section about security misconfiguration? Well, the two are examples for where default settings are not explicitly shown in the framework metadata. If an application developer does not know about the setting or has a different understanding about the default value, mistakes can be made. For example, a developer might assume that the default setting for URL Invoke is "disallowed", which could leave direct access to a task flow page unintentionally possible, bypassing logical parts of the application flow. In such cases the results may be errors (which can reveal information in their own right to a hacker) or even direct access to unauthorized data or functions. So the recommendation is to explicitly set all values for all of the features that you identify as security relevant, even if you think the implicit default is the correct value.

Setting explicit defaults for framework properties also ensures that the configuration is easier to read and understand for the purposes of code security reviews and future maintenance.

ADF Faces Version Output

Oracle ADF Faces provides a web.xml context parameter to display ADF and JDeveloper version information in a page output. Setting the `oracle.adf.view.rich.versionString.HIDDEN` parameter value to **false** will print version information to the HTML of an ADF Faces page. To avoid the leaking critical information for production deployments, ensure the parameter value is set to **true**.

Note: This risk is described in detail in the later *OWASP #9 - Using Known Vulnerable Components* section.

Project Stage

In a related setting to the ADF Faces version number topic, when deploying your application to production you should ensure that the javax.faces.PROJECT_STAGE

parameter in web.xml is switched to the value "Production". As a side effect this will actually also suppress the version number display as well as improving performance.

OWASP #6 - Sensitive Data Exposure

Data that can be classified as sensitive is, of course, dependent on the business modeled by an application and government legislation but usually includes all data that is restricted to a job responsibility, data which is hard to recover if damaged or lost and data that is considered internal. Applications must be designed such that data storage, data queries, data delivery and data manipulation don't expose any surfaces for data to leak, to be tempered with or to be changed without permission.

Sensitive data exposure is not a topic to be handled during application development alone but also requires administrative oversight by system, network and database administrators. This paper only lists tasks that are within the control of the ADF application developer. Topics such as, for example, transport layer security (SSL) — mandatory protection to apply to all communication that involves sensitive data — are out of scope for this paper and can be read up in the Oracle® Fusion Middleware Administrator's Guide[9].

Authentication

Data access authorization has to be enforced in an environment where users are authenticated. However, authentication alone doesn't help if there is no guarantee that users cannot be impersonated. As a matter of course, ensure user identities are stored and managed safely in an identity management system, that strong passwords are used and that the login process is protected with SSL.

Session

Unlike desktop applications, users don't always close their web applications when they are finished in a task. They are just as likely to simply re-use the browser window for something else and even if an application provides a "log-out" option to allow the developer to perform cleanup it may not be used. This is why it is important to ensure that sessions are not maintained for long periods when idle. In all Java EE applications including those written using ADF, the session timeout is set within the `web.xml` configuration file.

ADF sessions are web sessions and the default setting for their expiry is 35 minutes. Ideally this value should further reduced for applications that deal with sensitive data. Here are some examples of what you can do in Oracle ADF to ensure sessions terminate early.

Use the pillar architecture – Pillar is an ADF architecture pattern where the logical application is to split-up into smaller individual Java EE applications. As part of this division of function you can

[9] http://docs.oracle.com/cd/E25178_01/core.1111/e10105/toc.htm

partition access to particularly sensitive data or operations into separate modules and in doing so, you can set the session timeout for different parts of the application according to its security requirements.

To learn more about the pillar architecture, please see "Oracle ADF Architecture Fundamentals"[10], an educational video available from the ADF Development team that explains the different ADF application architecture styles.

Use the ADF Faces session time out warning – A possible concern in your application design is that the web session might expire whilst the user in the middle of a long running task. To respond to this possibility, instead of changing the `web.xml` to set a long, and often unrealistic, session expiry time, you should make use of the session timeout warning feature. Using an ADF Faces specific context parameter[11], users are given the opportunity to extend a session by pressing a button in a warning dialog. If a user session expires, a notification is sent to inform the user about the discontinued session. The user can then refresh the page and be directed to re-authenticate. This allows applications to run with realistically small session expiry times without frustrating users by causing them to lose their work.

Ensure no client-side operations pin the user session – Application logic may require a client to automatically check for server side updates of the data. The ADF `af:poll` component provides a convenient way to achieve this. However, a regular ping to the server from the poll will prevent a user session from expiring even though the user may have left their screen and not be actively working with the application. To allow user sessions to expire in this case, you should ensure the `af:poll` component also times out in response to user inactivity. For this the `af:poll` component exposes a "timeout" property that you can set to interrupt the poll. The "timeout" property allows you to set the timeout for each instance of this component so that different use cases can have different time out settings. A web.xml parameter `oracle.adf.view.rich.poll.TIMEOUT` can be used to set a global timeout value for `af:poll` components that don't have their timeout property set. The default setting of this parameter is 10 minutes until a poll expires. To learn more, please read the ADF Faces tag documentation for the `af:poll` component[12].

Query predicates

One of the best ways to ensure sensitive data is kept that way is to only retrieve it from the database on a "need to know" basis. This is much safer than fetching all the data to the middle tier and then to filter or redact information that the current user is not supposed to have access to. Strategies to achieve such control include filtered query based on the user identify or role membership and the processing of computation in the database, instead of the middleware.

---

[10] https://www.youtube.com/watch?v=toEuQvp73h8

[11] http://docs.oracle.com/cd/E17904_01/web.1111/b31973/ap_config.htm#autoId24

[12] http://docs.oracle.com/cd/E28280_01/apirefs.1111/e12419/tagdoc/af_poll.html

Query predicates in ADF can be enforced through Oracle database label security (aka. Virtual Private Database (VPD)) and view criteria applied to an ADF Business Component view object.

Virtual Private Database – With VPD, the way protection works is that the database appends a user specific where clause to any query issued by the user. In Oracle ADF you override the `prepareSession` method on the Application Module to set the context information used by VPD so queries issued by the authenticated web user are filter based on his or her identity. To set the context information, a prepared statement is used in the Java method to call a PL/SQL procedure stored in the database that sets the predicate.

View Criteria – View criteria in ADF Business Components can be declaratively or programmatically applied to ADF BC view object instances, where they are applied as a where clause. View criteria can be parameterized using bind variables, which in turn can obtain their values through a Groovy script.

The script `adf.context.securityContext.username`, shown in Figure 7, obtains the name of the authenticated user to use in the view criteria filter. For non-username predicates, you can expose a method on the view object implementation class that accesses the username from Java to determine and return the bind variable value. To obtain the username in Java, you can use the following code:

```
ADFContext adfctx = AdfContext.getCurrent();

String username = adfctx.getSecurityContext().getUserName();
```

Note that a new feature in JDeveloper 12.1.3 allows you to configure view criteria so that they cannot be removed from a view object instance at runtime. This type of criteria is ideal to use with such identity based filtering..

UI protection

In Oracle ADF, ADF Faces is used to render the application web user interface. ADF Faces is based on JavaServer Faces, which means that it has a server side object representation of a view as well as a client side representation in HTML and JavaScript.

ADF provides several ways to lock down data access at the user interface layer (remember that security in this layer should always be in addition too, not a replacement for, security checks on the model layer and database). By setting the ADF Faces component `rendered` property to false, UI components are excluded from rendering and thus their data is never delivered to the client and the framework will reject all attempts to update those components through crafted HTTP POST requests. Rendering can be dynamically controlled in a couple of standard ways:

- Using ADF Security expressions that check for the authenticated user permission or application role membership to control access to specific information. You can add EL expressions to the `rendered` property of an ADF Faces component to add it to the

rendered page or not. Note the "add it to the rendered page" in the previous sentence — An important design principle for application developers, not only for those building applications with Oracle ADF, is to implement security by design and default[13]. To ensure sensitive data does not leak if application security is disabled, you want to make sure that the default behavior is to not add a component with sensitive data exposed to the rendered page.

- Runtime Personalization is another option to control the page rendering for sensitive data. This can be achieved using Oracle Metadata Services (MDS), which in Oracle ADF is used to define pre-seeded customization for application pages and views. As mentioned before, when discussing the `rendered` property, you can check ADF Security permissions and user application role membership before adding UI content to a rendered page or view. To control this, you build a custom customization class, which checks against ADF Security, in a way that is similar to the sample shown below. Again, in case of failure, a customization, if used to implement security should always add content to a page but not remove it.

In the sample code shown below, the customization class checks for the user permission to override a default configuration option before adding UI content to the rendered page. For this, a custom ADF Security resource permission is used.

```
public static final String ACTION = "override";
public static final String RESOURCE_NAME = "adf.sample.cc.Site";
public static final String RESOURCE_TYPE = "Customization";

private boolean isAllowedOverride(){
  boolean hasPermissionGranted = false;
  ADFContext adfCtx = ADFContext.getCurrent();
  SecurityContext securityCtx = adfCtx.getSecurityContext();
  ResourcePermission sitePermission = null;
  sitePermission = new ResourcePermission(RESOURCE_TYPE,RESOURCE_NAME
                       ,ACTION);
  hasPermissionGranted = securityCtx.hasPermission(sitePermission);
  return hasPermissionGranted;
}
```

The full sample of how to use ADF Security within Oracle MDS, plus associated documentation, is available as sample "031" on the ADF Code Corner[14] website.

[13] Secure Coding: Principles and Practices, ISBN-13: 978-0596002428

[14] http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html

Note: For more information on Metadata Services, please read the "Building Customizable Oracle ADF Business Applications with Oracle Metadata Services (MDS)" whitepaper available on the Oracle Technology Network (OTN)[15].

OWASP #7 - Missing Function Level Access Control

The defense in depth design pattern specifies that multiple layers of security to be implemented in an application. This also means that application functionality that executes methods and operations should be guarded by authorization checks even if the underlying data object is protected through entity security. For example, methods on the business service may bypass entity validation by issuing a prepared JDBC statement to carry out operations on the database directly and so the invocation of such methods must be protected with authorization checks.

In a similar fashion, applications my call out to remote SOAP or REST services to update data or application state information, such calls are not protected by ADF Security by default so you should look to protect them if used.

As a best practice, never assume that a specific method will only be called within a context that it was initially designed for. All access to functionality that manipulates data must be protected either by access control on the entity or by guarding the invocation of methods using ADF Security or JAAS permission checks.

ADF Security provides the following types of options to protect function calls (and user interface components) from unauthorized use:

- **Custom resource permissions** – Function level security relates to high-level actions performed within an application such as placing an order that may go beyond simple entity updates. To encapsulate the protection of these functions, ADF Security allows developers to create custom permissions in addition to the existing permissions provided by framework. The method calls that need to be protected can check against these permissions before proceeding.
- **ADF Security EL Expressions** – ADF Security provides EL expressions that developers can use on the rendered property of user interface components, or on task flow router activities to hide operations and navigations for unauthorized users. It's important to remove the temptation from users to carry out actions which you will know that they will lack the privileges to execute

With these capabilities Oracle ADF provides ample tools to effectively manage your functional security.

Note: In section we provide a more in-detail discussion on the options ADF security provides to check authorization from Java, in expression language and Groovy. This, for reasons of space, will not

[15] http://www.oracle.com/technetwork/developer-tools/jdev/adfmds-128339.pdf

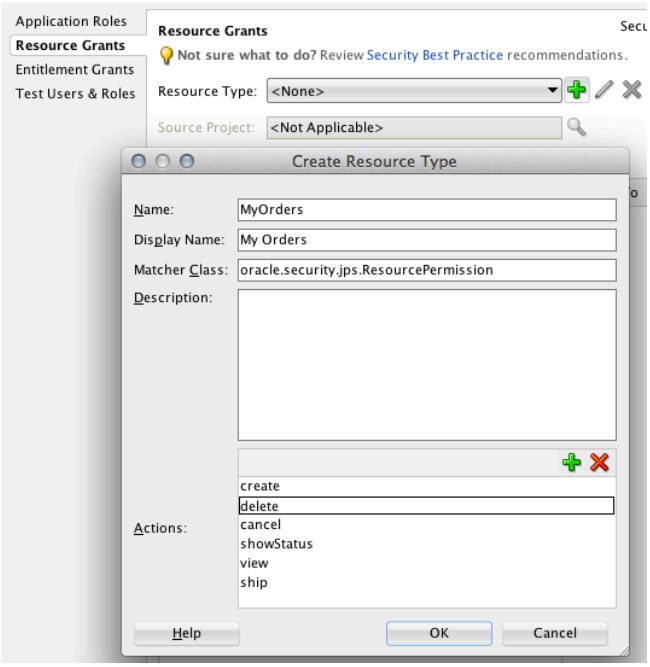
be an exhaustive discussion of the topic, however, and so it is recommended you consult the following resources: the Oracle ADF product documentation, the ADF Insider[16] website and Oracle Magazine[17] [18].

Custom Resource Permissions

ADF custom resource permissions are defined using a name, a custom resource type, one or more actions selected from the resource type definition, and a grant statement. The custom resource type is a blue print for a set of resource permissions and defined by a name, a physical Java permission class and a list of actions to use for protection application resources and functions. The permission class – oracle.security.jps.ResourcePermission – is always the same and provided by OPSS.

For the customer-order example mentioned earlier, a custom *MyOrders* resource type could be created that defines *create*, *delete*, *ship*, *cancel, view* and *showStatus* as protectable actions.

To create custom Resource type, developers use the ADF Security resource editor from the **Application > Secure > Resource Grants** menu in JDeveloper. Pressing the green plus icon next to the *ResourceType* fields, as shown in Figure 9, brings up the JDeveloper *Create Resource Type* dialog to define a custom permission type and the actions to protect.



[16] http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adfinsider-093342.html

[17] http://www.oracle.com/technetwork/issue-archive/2012/12-jan/o12adf-1364748.html

[18] http://www.oracle.com/technetwork/issue-archive/2012/12-may/o32adf-1577987.html

Figure 9: Creating a custom MyOrders resource type

Figure 10 shows how the *MyOrders* type referenced in different resource permissions along with the grant statement and the actions the statement is for.

To create resource permissions, application developers press the green plus icon next to the *Resources* label in the ADF Security editor. Resource permission is defined by a name, one or more application roles to where the permission is granted and a selection of actions exposed on the referenced custom Resource type.

The "shipOrder" permission shown in figure 10 is granted to the "OrderEmployee" application role, a custom application role created in an ADF application. The application role is mapped to an enterprise role (a user group) during post-deployment to abstract application security from users and groups defined in an identity management system.

At runtime, ADF Security can be used from expression language or code to check custom resource permission for the authenticated user, thus enforcing, entity, object and functional security.

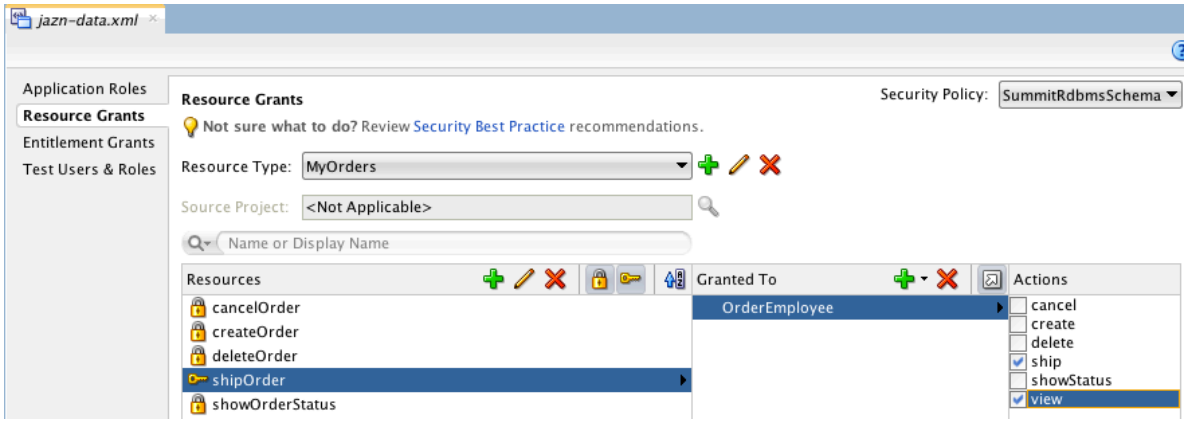


Figure 10: Creating resources as a container to grant permissions to users

ADF SecurityContext

The OPSS `SecurityContext` interface defines a programming APIs for applications developers to access security relevant information such the authenticated user `Principal` and JAAS `Subject`, or to perform authorization checks from Java or EL.

Among others, the key methods exposed by the ADF Security context include

- **getUserPrincipal –** Retrieves the authenticated user principal object
- **getUsername** – Retrieves the username of the authenticated user or "anonymous" in case the user is not authenticated.
- **hasPermission(Permission)** – Returns true if the authenticated user has the specified permission granted
- **isAuthorizationEnabled** – Returns true if ADF Security is configured with authorization enabled

- **isAuthenticationEnabled** – Returns true if ADF Security is configured with authentication required.
- **isAuthenticated** –Returns true if the user has been successfully authenticated
- **isUserInRole(String)** – Returns true if the user is a member of the specified enterprise or application role.

The `SecurityContext` object is accessed from Java on the view layer or the ADF BC business layer using the code lines shown below

```
SecurityContext securityContext =
                ADFContext.getCurrent().getSecurityContext();
```

To access the security context from expression language added to a user interface component or used in a task flow view or router activity, you simply use

```
#{securityContext. …}
```

A third way to access the security context is using Groovy expressions from Oracle ADF Business Components:

```
adf.context.securityContext
```

Note: Figure 6 on page 28 in the "OWASP #3 - Cross-Site Scripting (XSS)" section shows an example of how to use Groovy in application security

ADF Security EL Expressions

To simplify authorization checking on the user interface, ADF Security provides a special set of EL expressions. With these, authorization checks can be made, for example on the rendered property of a navigation command.

As another example, Applications may conceal or reveal functionality depending on the authentication state of the user.

For example, on an online flight-booking portal, a text box may be shown that informs users that login is required before a booking can be performed. After the user is authenticated, the text box is removed using ADF Security EL.

The ADF Faces component that can be used for this use case is the `af:switcher`[19].

The `af:switcher` component changes the visibility of custom facets areas based on the outcome of an expression provided in the switcher *facet* property. The component *defaultFacet* property defines the facet to render in case no facet name matches the EL expression outcome. For the above example of a

[19] http://docs.oracle.com/middleware/1212/adf/TROAF/tagdoc/af_switcher.html

flight-booking portal you would set the *defaultFacet* property value to the name of the facet holding the unauthenticated state text box.

Note: Any component that contains sensitive information should not be hidden but removed from the JSF component tree by executing the ADF Security EL in its *rendered* property.

The list below shows the available ADF Security expressions:

#{*securityContext.authenticated*}

Expression that returns true or false based on whether the user is authenticated or not.

#{*securityContext.userName*}

Expression that retrieves the name provided by the user during login.

#{*securityContext.userInRole['roleList']*}

The expression returns true when the authenticated user is a member in one of the listed security roles.

#{*securityContext.userInAllRoles['roleList']*}

The expression returns true when the authenticated user is a member in all security roles in the list.

#{*securityContext.taskflowViewable['target']*}

Based on the user privilege for the task flow defined as the target, this expression string returns true or false. The target is the name of a task flow definition file and ID including the WEB-INF directory name.

Example:
```
#{securityContext.taskflowViewable
        ['/WEB-INF/EmployeeUpdateFlow.xml#EmployeeUpdateFlow']}
```

#{*securityContext.regionViewable['target']*}

The expression checks the user granted permission to view the ADF PageDef file associated with a page. The PageDef file is referenced by its name and package name as the "target" argument.

Example:
```
#{securityContext.regionViewable
        ['myorg.adf.sample.views.pageDefs.EmployeesPageDef']}
```

#{*securityContext.userGrantedResource['permission']*}

This expression checks a user's permission for an action defined on a custom resource permission. The "permission" argument is a semicolon delimited list containing the name of the resource permission, the resource type and the name of the action to authorize

Example: `#{securityContext.userGrantedResource[`

```
'resourceName=shipOrder;resourceType=MyOrders;
 action=ship']}
```

#{*securityContext.userGrantedPermission['permission']*}

ADF applications may not be the only Java EE web application in an organization. In such cases, there may be existing Java permission classes that developers need to reuse within ADF applications. The *userGrantedPermission* expression allows ADF developers to check authorization against such permission classes. The "permission" argument in the EL call defines the physical permission class, the target and action to perform authorization for.

#{*securityContext.userGrantedPermission* ['permissionClass=<class>; target=<target>; action=<action>']}

For example, to hide a command button if a user is not allowed to delete an ADF BC entity, for example "Orders" , you could add the following EL to the command button's *rendered* property

#{*securityContext.userGrantedPermission* ['
permissionClass=oracle.adf.share.security.authorization.EntityPermission;
target=adf.sample.model.entities.Orders;
action=delete']}

ADF Programmatic Security

A Java class at runtime represents all permissions in ADF Security. This class can be instantiated in application code and dynamically checked. The commonly used classes are

- `oracle.security.jps.ResourcePermission` for custom resource definitions
- `oracle.adf.controller.security.TaskFlowPermission` for bounded task flows
- `oracle.adf.share.[…].RegionPermission` for page permissions in unbounded task flows
- `oracle.adf.share.[…].EntityPermission` for entities
- `oracle.adf.share.[…].EntityAttributePermission` for entity attribute permissions

The code example below shows how to guard an action invoked through the action listener of a JSF command component:

```
public void onCancelOrder(ActionEvent actionEvent) {
 //get the ADF Security context from the ADFContext class
 SecurityContext securityCtx =
                ADFContext.getCurrent().getSecurityContext();
 //create an instance of the ResourcePermission created earlier for
 //orders. Note that this permission is not used on the entity but on
```

```
 //program logic as it defines whether the authenticated user is
 //allowed to cancel an order: ResourcePermission(String rtype,
 //String name, action)
  ResourcePermission resourcePermission =
          new ResourcePermission("MyOrders","cancelOrder","cancel");

 //check if user has permission granted
  boolean userHasPermission =
                      securityCtx.hasPermission(resourcePermission);
 //if user has the permission granted, execute the associated logic
  if (userHasPermission){
      //execute privileged logic here

  }
  //or log the attempt to improve the UI e.g. to not show the cancel
  //option to users that are not authorized
  else{
  // … log failed attempt

 }
}
```

The code snippet above checks authorization against the "action" permission. A menu item in an application menu may also check for the "view" permission to determine whether or not the cancel option should be displayed at all to a user. For the latter use case you would use ADF Security expressions explained earlier.

Hint: ADF Security does not automatically protect custom ADF BC client methods exposed on the application module or view object implementation class. To protect custom methods, you use custom `ResourcePermission` as explained earlier. Similar, to protect ADF BC framework operations like Create, CreateInsert or Delete, you override the framework base classes and their methods to add Java permission checks. Another option to add security to ADF bound UI command components is to not directly invoke the associated ADF method or operation binding from the command component but to use a managed bean in between. To create the bean, in the JDeveloper visual editor, double click on the ADF bound component, which displays a dialog for creating or selecting a managed bean to create the component action method in. When creating the action method, JDeveloper provides the option to generate Java code that invokes the ADF binding layer method or operation bound to the component to preserve the configured behavior. To protect the method or operation binding invocation you then surround the generated code with a check against a custom `ResourcePermission` similar to the example shown above.

OWASP #8 - Cross-Site Request Forgery (CSRF)

Cross-site request forgery describes the risk of a third-party request to a web application on behalf of an authenticated user who unknowingly executes the link on a phishing site or is the victim of cross-

site scripting injecting HTML content to the view to perform the attack. In JavaServer Faces, it's the view state you should pay attention to.

Page tokens

The view state in JavaServer Faces tracks the client state so in subsequent requests the server side view state can be identified and updated with any changed data values. For the view state, there are two options, to save the view state: to use a hidden field on the client and to use a client token.

Hidden field – Using a hidden field in most JavaServer Faces application saves the view state in a base64 encoded value saved in a hidden UI field. This option is considered less safe because the view state can be manipulated on the client unless it has been explicitly encrypted.

Prior to JSF 2.2 encryption of such view state was possible but was not enabled by default. In JSF 2.2 encryption is now enabled by default based on a developer chosen algorithm. The view state encryption is defined in web.xml.

Client token – Using client tokens, the view state is saved on the server, in the user session, and identified by the client only by a token. This is the default and recommended state saving method used in ADF applications. The tokens are encrypted using the MyFaces Trinidad state manager by for each view. It is recommended that you keep the default setting.

XSS

An additional, mitigation strategy for CSRF attacks is to watch out for cross-site scripting risks. As discussed within the XSS section of this paper all input and all output must be encoded to avoid the exploit of a user session or to perform unauthorized actions using JavaScript or added HTML form content.

Framebusting

Phishing is a CSRF attack that spies on user password or sensitive data information by embedding a popular website, for example the site of a credit card company or bank, in a "frame" on malicious page. The user will not be aware that the site is embedded in this way and may be lured into using such sites by following links in emails or documents. The malicious page can then monitor the user activity on the hosted page and capture information. A related phishing attack is "clickjacking". In clickjacking, a malicious page makes users to click a button that belongs to the surrounding malicious page and not the embedded site to execute code.

To neutralize the risk of such phishing techniques, ADF Faces provides the "framebusting[20]" feature that developers configure in the ADF application web.xml file using one of the following context parameters:

[20] http://docs.oracle.com/middleware/1212/adf/ADFUI/ap_config.htm#BABDHGEJ

`oracle.adf.view.rich.security.FRAME_BUSTING` – You use this context parameter if your ADF application runs on ADF 11g.

`org.apache.myfaces.trinidad.security.FRAME_BUSTING` - You use this context parameter if your ADF application runs on ADF 12c or later.

Both of these parameters have the same effect, which is to ensure that the application cannot be wrapped in another page in this way.

OWASP #9 - Using Known Vulnerable Components

Software vendors frequently release software patches in response to security issues in their software. It's imperative that you plan your application maintenance processes to adopt patches and vendor updates as quickly as possible, especially if a patch contains such a security related fixe. Staying on an old software release that has known and documented issues leaves you vulnerable.

A good example for such a patching requirement is the recently discovered "heartbleed"[21] OpenSSL bug. The vulnerability associated with the bug has been documented in early 2014 along with the release of a patch. Companies that used OpenSSL and didn't install the patch put systems at risk because of the existence of this known vulnerability.

The most effective mitigation strategy for this kind of security vulnerability is to be proactive about to keeping up to date with Oracle ADF releases and Oracle Critical Patch Update process.

Risk: JSF project stage and ADF Faces version number

Information you might be interested to see during development and testing is the version of Oracle ADF Faces, ADF and the JDeveloper build number that the application has been developed with. This information can be printed as a hidden field into the ADF Faces page output by setting the following web.xml context parameter:

```
<context-param>
  <description>
    Whether the 'Generated by...' comment at the bottom of ADF Faces HTML pages
      should contain version number information.
  </description>
  <param-name>
   oracle.adf.view.rich.versionString.HIDDEN
  </param-name>
  <param-value>false</param-value>
</context-param>
```

[21] http://en.wikipedia.org/wiki/Heartbleed

Setting the `oracle.adf.view.rich.versionString.HIDDEN` parameter to "false", as shown above, will print version information as part of the rendered page. The recommendation is to always set this parameter to **true** for applications deployed to production.

In JDeveloper 11g R2, 12c and later, the JavaServer Faces `PROJECT_STAGE` parameter can be used to switch the ADF Faces `HIDDEN` parameter based on the deployment type. The default deployment type is production, in case the `PROJECT_STAGE` parameter is not set, and ADF Faces ignores the `HIDDEN` parameter even if it is set to false. However even with this safe default behavior of ADF Faces, its best to always set the `HIDDEN` parameter to true in your production deployments[22].

Note: The above risk also fits to *"OWASP #5 - Security Misconfiguration"* as by security misconfiguration the application provides unwanted information to a hacker.

Defend against zero-day-exploits

Although you may be diligent about keeping up to date with your patches there is always a possibility that a novel as-yet-unknown vulnerability or zero-day[23] attack is used against you. No software is ever going to be totally safe from such exploits. All that we can do is to code as securely as possible within the application and reduce the amount of information that you leak about your application to make it harder for hackers searching for systems with a certain profile.

To reduce the risk of zero-day attacks, don't publicly advertise the version of software you use (see the section above on hiding the version number). Additionally for any issue you find in a production system, use customer support instead of reporting details of versions and errors publicly on the Internet.

Common sense dictates that if information needs to be shared in an open forum, ensure the information you provide doesn't compromise your security. Often major release versions are enough information and no patch set level information is required.

Note: You cannot simply update the JSF or MyFaces Trinidad version yourself in Oracle ADF. Any patch in this area needs to be provided by Oracle for ADF Faces.

OWASP #10 - Unvalidated Redirects and Forwards

ADF web applications use JavaServer Faces post-back navigation for navigating between views in the browser or exposed in ADF regions. If navigation takes place between applications, (i.e. when using a pillar architecture), or when ADF applications need to call into Forms applications, then http GET

[22] https://blogs.oracle.com/groundside/entry/basic_weblogic_deployment_plans_for

[23] http://en.wikipedia.org/wiki/Zero-day_attack

requests are used, This can expose the risk of an additional attack surface that hackers may attempt to exploit, bypassing the conventional paths through your systems.

To mitigate the risk when using Oracle ADF:

- Ensure any request that is issued or received through a direct GET request is validated for the identity of the request origin or target.
- If data needs to be passed to a separate Java EE application, ensure it is passed such that it cannot be tempered with and so the receiving application knows how to verify the validness of the data. A possible solution for this could be a table in the database that is accessed through a stored procedure. The only parameter that needs to be passed between applications in such a case is a secured token (encrypted with a short lifetime).
- Avoid any GET request or redirect for navigation in ADF applications. Use post-back navigation only.
- Limit the number of access points to the ADF application. This is easily achieved by using bounded task flows and by setting the "URL Invoke" property on all task flows to disallowed using the JDeveloper Property Inspector.
- If there is a requirement to call into an ADF application so that the application does not start from its beginning but somewhere in its middle, use bounded task flows with the URL Invoke property set to "allowed" and ADF Security enabled. This ensures access control on the task flow as well as a single point of access to the application sub-functionality.
- All bounded task flows that can be accessed from browsers directly should save input parameter values in managed beans in pageFlowScope and not directly in memory attributes. Managed bean properties that hold input values should have code added in their "set<Name>" method that guards and validates input values provided in the request.
- ADF pages in unbounded task flows can be configured as bookmarkable. The bookmark configuration is on the view activity definition and allows the definition of input parameters to display the bookmarked page with the restored query state. It is important that all parameters have a converter defined that checks the parameter value for validness and correctness. Even if the expected input parameter type is string and no object conversion is required, use converters for validating the request parameters. Note that the bookmark input parameters have no extra validator property, which is why the converter needs to be used for validation as well.

Watch your back

Application security is useless if the application itself runs in an insecure environment. Perimeter security describes the levels of protection that you add on servers, the network and other data access channels outside of the ADF application. As pointed out in this paper, not all of the OWASP Top 10 security vulnerabilities documented for 2013 are topics that are relevant for application developers. Transport layer security and professional identity management clearly fall into the domain of a security

administrator in Fusion Middleware administration. Developers can follow best practices in secure coding as outlined in this paper, but this is not enough if other areas are left unprotected.

How much security do you need?

"The degree of assurance required in your applications is very strongly related to the size and nature of your unique risks, as well as to the cost of the countermeasures you might program in. How secure does your application need to be? Just secure enough."

- Mark G. Graff; Kenneth R. van Wyk[24]

In his book *Effective Oracle Database 10g Security by Design*[25], author David Knox depicts a triangle that shows the interaction and dependency between security, usability and performance. His point is that security does not live on its own and that strong security probably will have an impact on how users perceive an application in respect to performance and ease of use. There is no guarantee of a 100% secure application and often it is not needed. All your application needs is enough security to protect it from the risks you have identified for it and the nature of data you want or are legislated to protect.

This paper showed many options for example to where to put input validation. It also mentioned the multi-lines-of-defense design pattern. But does it mean that all technology and business layers involved in an application require input validation? Probably not and to some point you will consider it enough.

So the question is how much is enough? Enough can only be determined if you have a plan based on a risk analysis and based on a clear understanding about common security threats, such as those documented by the OWASP. However, OWASP is not the only organization writing about security risks in software development and it's worth looking at others as well.

This paper focused on the OWASP Top 10 list of critical security vulnerabilities because of its high profile. This does not mean, however, that this top 10 is all you need to take care of. For example, a vulnerability not covered in the OWASP Top 10 is *Social Engineering*. Social engineering has many facets, and ranges from phishing and shoulder surfing to the tricking information out of users.

Education is, in fact, the best protection you can buy for your custom application developments. It helps to identify risks, work on mitigation strategies and also to tell how much security is enough.

Summary

For Oracle ADF, this paper addresses the relevant sections of the OWASP list of top 10 security vulnerability for the year 2013. The paper lists Oracle Application Development Framework (ADF) features and tools that help security aware developers to protect browser web and mobile applications.

---

[24] Secure Coding: Principles and Practices, ISBN-13: 978-0-596-00242-8

[25] Effective Oracle Database 10g Security by Design, ISBN-13: 978-0-07-223130-4

For application developers and project managers it is important to understand that 100% security doesn't come as a fully formed feature with any platform or framework. Security is a mix of awareness, education and the technology in use. In the case of Oracle ADF you have all of the tools that you need to at least provide the technology for application developers to write secure web applications.

Appendix: Recommended Readings

1. OWASP Home Page

   https://www.owasp.org/index.php/Main_Page

2. "ADF Architecture Square: ADF Code Guidelines v2.00"

   http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adf-code-guidelines-v2-00-2096456.pdf

3. "Handling the OWASP Top Ten Application Security Risks with Oracle Fusion Middleware"

   http://antonfroehlich.blogspot.de/2012/06/handling-owasp-top-ten-application.html

4. Oracle ADF Security Documentation

   http://docs.oracle.com/middleware/1212/adf/adf-secure.htm

Security in Oracle ADF: Addressing the OWASP Top 10 Security Vulnerabilities

October 2014
Author: Frank Nimphius
Contributing Authors: Duncan Mills, Gary Williams, Denis Pilipchuk

ECCN: EAR99

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together