The Anatomy of a Secure Web Application Using Java

ApacheCon NA 2015

John Field, Pivotal Services @EMC
Introductions

- John Field
  - Security Architect at @EMC2
  - Twitter: @architectedsec
  - Website: https://johnpfield.wordpress.com

*Joint work with Shawn McKinney, Principal at Symas.*
Agenda

• Introduction
  • Quick Demonstration
• Step-by-Step “How To” Security Guidance
  • Based upon the “FortressDemo2” sample application
• Survey of Security Architecture Patterns
  • Requirements, Capabilities
• Conclusion
  • Project coordinates
Tutorial Approach

• Examine a typical enterprise Java Web application, one architectural layer at a time.

• Goals:
  • Be able to recognize and identify some well-known security architecture patterns.
  • Understand how each pattern contributes to satisfying the overall security requirements.
  • Learn how to implement these patterns via pragmatic, hands-on configuration guidance.
Be a “Full Stack” Developer

“No one can know everything about everything, but you should be able to visualize what happens up and down the stack as an application does its thing.”

-- C. Bueno of Facebook, c. 2010
Our Business Use Case

- Deployment of an Enterprise Java Web Application.
  - Assumes a standard User Agent / Browser-based HTTPS access path.
- We have requirements for:
  - User Authentication
  - User Authorization
  - Audit Logging
  - Confidentiality and Integrity
Core Security Architecture Patterns

1. Use HTTPS / TLS on a shared network.
2. Use Container-based Enforcement
3. Delegate to a Trusted Third Party (TTP).
4. Use RBAC to express access control policy.
5. Create an audit log.

The patterns remain the same, whether deploying on standalone servers, or to the cloud.
High-Level Deployment

Firefox

HTTPS

X509 Certificate

Tomcat Server

Java Web Application

LDAPv3 Server

LDAPS

RDBM Server

JDBC

PAAS

ApacheCon NA 2015
Demonstration

- Communications secured via SSL/TLS.
- Users authenticate via enterprise LDAP.
- Resource authorization via RBAC.
  - Static (type-based) and dynamic (instance-based)
    - Including Static and Dynamic Separation of Duties
- Audit logging for all application events
  - i.e. any change of state
High-Level Deployment

PEP: Confidentiality, Integrity

Firefox

HTTPS

PEP: AuthN & coarse-grained AuthZ

PEP: Pages, controls, datum, AuthZ

PEP: Audit Logging

PDP: Policy Decision

PAP: Identities Policies

Tomcat Server

Java Web Application

LDAPv3 Server

LDAPS

RDBM Server

JDBC
The Anatomy of a Secure Java Web Application
The Anatomy of a Secure Java Web Application
Here we go!
Need to:

- Create certificates.
- Put them in the right place.
Step 1: Issue Certificates

If date < Mon Apr 7 2014, and version = 1.0.1, then installation is vulnerable to Heart Bleed and must be upgraded.

# openssl version -b

# sudo apt-get upgrade openssl
Step 1: Issue Certificates

# mkdir certs

# cd certs

# openssl genrsa 2048 > pse-ca-key.pem

# openssl req -new -x509 -nodes -days 3600 -key pse-ca-key.pem -out pse-ca-cert.pem

# openssl req -newkey rsa:2048 -days 1825 -nodes -keyout oreo-server-key.pem -out oreo-server-req.pem
Step 1: Issue Certificates

```bash
# openssl rsa -in oreo-server-key.pem -out oreo-server-key.pem
Remove passphrase from private key

# openssl x509 -req -in oreo-server.req.pem -days 1825 -CA pse-ca-cert.pem -CAkey pse-ca-key.pem -set_serial 01 -out oreo-server-cert.pem
Sign server certificate request

# openssl pkcs12 -export -name fortressDemo2ServerCACert -in oreo-server-cert.pem -inkey oreo-server-key.pem -out mykeystore.p12
Generate a temporary PKCS12 keystore.
```
Step 1: Issue Certificates

Use Java keytool to import PKCS12 into JKS key store for Web server

# keytool -importkeystore -destkeystore mykeystore -srckeystore mykeystore.p12 -srcstoretype pkcs12 -alias fortressDemo2ServerCACert

Use Java keytool to import CA cert into JKS truststore for client application

# keytool -import -alias fortressDemo2ServerCACert -file pse-ca-cert.pem -keystore mytruststore
Certificate Summary

- **Server-side:** 4 Files.
  - Used by OpenLDAP and MySQL to offer TLS.
    1. pse-ca-cert.pem
    2. oreo-server-cert.pem
    3. oreo-server-key.pem
  - Used by Tomcat JSSE to offer HTTPS.
    4. mykeystore

- **Client-side:** 1 File.
  - Used by the Web application JSSE to negotiate HTTPS / TLS with OpenLDAP and MySQL servers.
    - mytruststore
Step 2: Tomcat HTTPS
Step 2: Tomcat HTTPS

# sudo apt-get install tomcat7 tomcat7-admin tomcat7-docs
# vi /usr/share/tomcat7/conf/server.xml
Step 2: Tomcat HTTPS

- Add the following to `server.xml`:

```xml
<Connector port="8443" maxThreads="200"
    scheme="https" secure="true"
    SSLEnabled="true"
    keystoreFile="conf/mykeystore"
    keystorePass="changeit"
    clientAuth="false" sslProtocol="TLS"/>
```
Step 2: Tomcat HTTPS

```
# sudo cp certs/mykeystore /usr/share/tomcat7/conf
# sudo cp sentry-1.0-RC39-proxy.jar /usr/share/tomcat7/lib

# sudo service tomcat7 restart
```

While you are at it, add the JEE Security Realm Provider Proxy jar.

Put mykeystore in the indicated place.
Step 3: Enable Java EE Security
Add JEE Security To Web.xml

Add JEE Security To Web.xml

<security-constraint>
  <display-name>My Security Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/secured/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_DEMO_USER</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>MySecurityRealm</realm-name>
  <form-login-config>
    <form-login-page>/login/login.html</form-login-page>
    <form-error-page>/login/error.html</form-error-page>
  </form-login-config>
</login-config>

Declarative coarse-grained authorization.
Enforced high in the stack.

HTML Form-based Authentication
Step 4: Enable Policy Decision Point
Step 4: Enable Policy Decision Point

Assume an LDAP server is already deployed. Focus on the PDP integration via Fortress Sentry.
Fortress Sentry RBAC PDP

- Sentry is a standards-compliant RBAC PDP
  - Conforms to NIST / ANSI / INCITS 359
- Integrates into Tomcat
  - JEE Custom Realm Provider
- Integrates into application
  - As a standard Java component.
    - Add the dependency to Maven pom.xml
    - Add the Bean definition to Spring applicationContext.xml
ANSI RBAC – the TLDR
ANSI RBAC – INCITS 359

RBAC0:
Users, Roles, Perms, Sessions

RBAC1:
Hierarchical Roles

RBAC2:
Static Separation of Duties

RBAC3:
Dynamic Separation of Duties
ANSI RBAC Object Model

Six basic elements:

1. User – human or machine entity
2. Role – a job function within an organization
3. Object – maps to system resources
4. Operation – executable image of program
5. Permission – approval to perform an Operation on one or more Objects
6. Session – contains set of activated roles for User
ANSI RBAC Functional Model

Three standard interface definitions:

1. Administrative
   - Policy CRUD
2. Review
   - Policy Interrogation
3. System
   - Policy Enforcement
ANSI RBAC PDP

Standards-compliant System Interface Definition:

1. createSession
2. **checkAccess**
3. sessionPermissions
4. sessionRoles
5. getUser
6. addActiveRole
7. dropActiveRole
Step 4: Enable Policy Decision Point

Focus on the PDP integration via Fortress Sentry
Configure Tomcat Custom Realm

• Add context.xml file to the META-INF folder:

```xml
<Context reloadable="true">
  <Realm className="org.openldap.sentry.tomcat.Tc7AccessMgrProxy"
    debug="0"
    resourceName="UserDatabase"
    defaultRoles="ROLE DEMO2_SUPER_USER,
    DEMO2_ALL_PAGES, ROLE_PAGE1, ROLE_PAGE2,
    ROLE_PAGE3"
    containerType="TomcatContext"
    realmClasspath="" />
</Context>
```

• Copy sentry jar (cf. slide 23):

```bash
# sudo cp sentry-1.0-RC39-proxy.jar /usr/share/tomcat/lib
```
Configure Application Dependency

• Add Maven dependency to application pom.xml:

```xml
<dependency>
    <groupId>org.openldap</groupId>
    <artifactId>sentry</artifactId>
    <version>1.0-RC39</version>
</dependency>
```

• Add bean definition to applicationContext.xml

```xml
<bean id="accessMgr" class="org.openldap.fortress.AccessMgrFactory" scope="prototype" factory-method="createInstance">
    <constructor-arg value="HOME"/>
</bean>
```
Enable LDAP SSL
Enable OpenLDAP SSL Server

- Add certificate artifacts to OpenLDAP slapd.conf:

  ```bash
  # sudo vi /opt/symas/etc/openldap/slapd.conf
  TLSCACertificateFile /path/pse-ca-cert.pem
  TLSCertificateFile /path/oreo-server-cert.pem
  TLSCertificateKeyFile /path/oreo-server-key.pem
  ```

- Add ldaps to OpenLDAP startup params:

  ```bash
  # slapd … –h “ldaps://oreo:636”
  ```
Enable LDAP SSL Client

• Import CA PKI certificate into Java truststore
  • Cf. slide 16

• Tell client where to find LDAP

```bash
# vi /src/main/resources/fortress.properties
host=oreo
port=636
enable.ldap.ssl=true
trust.store=/path/mytruststore
trust.store.password=changeit
```
Enable Spring Security
Enable Spring Security

- Add URL pattern(s) and corresponding role(s) to Spring Security to applicationContext.xml:

```xml
<bean id="fsi" class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
    <property name="securityMetadataSource">
        <sec:filter-invocation-definition-source>
            <sec:intercept-url pattern="/com.mycompany.page1" access="ROLE_PAGE1"/>
        </sec:filter-invocation-definition-source>
    </property>
</bean>
```

• Add URL pattern(s) and corresponding role(s) to Spring Security to applicationContext.xml:
Enable Spring Security

• Add Maven dependencies for Spring Security to web app’s pom.xml:

```xml
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>${spring.security.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>Spring-security-config</artifactId>
  <version>${spring.security.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>${spring.security.version}</version>
</dependency>
```
Add Security-Aware Web Framework Components

[Diagram showing layers of security components, including Java Servlet Container, Java Secure Socket Extension, Java EE Security, Spring Security, Open SSL LDAPv3 Server, and protocol details.]

Legend:
- security function:
  1. HTTPS server
  2. HTTPS server keystore (private key)
  3. Java EE AuthN & AuthZ
  4. RBAC Policy Decision Point
  5. LDAP SSL client
  6. LDAP SSL client truststore (public key)
  7. LDAP SSL server
  8. LDAP server (private key)
  9. Spring AuthZ
  10. Web App AuthZ

- security concern:
  - Confidentiality
  - Authorization

- protocol:
  - LDAPv3
  - HTTP

ApacheCon NA 2015
Add Security-Aware Web Framework Components

Page1.java, addButtons() private method

```java
add( new SecureIndicatingAjaxButton( "Page1", "Add" )
{
    @Override
    protected void.onSubmit( ... )
    {
        if( checkAccess( customerNumber ) )
        {
            // do something here:
        }
        else
        {
            target.appendJavaScript( ";alert('Unauthorized');" );
        }
    }
});
```

As page is rendered, buttons are activated, per the user’s cached permissions.

On submit, do programmatic authorization (instance-based)
Add Security Aware DAO components
Add Security-Aware DAO Components

Page1DaoMgr.java, updatePage1() public method

```java
public Page1EO updatePage1( Page1EO entity )
{
    ...
    if(checkAccess("Page1","Update",entity.getCust()))
    {
        // Call DAO.update method...
    } else
    {
        throw new RuntimeException("Unauthorized");
    }
    ...
    return entity;
}
```

Just prior to database update, re-verify authorization for this instance.
Enable DB SSL
Enable MySQL SSL Server

# sudo vi /etc/mysql/my.cnf

- In the MySQL my.cnf file, instruct listener to use host name in certificate:
  
  bind-address = oreo

- Add the certificate artifacts generated previously:

  ssl-ca=/path/pse-ca-cert.pem
  ssl-cert=/path/oreo-server-cert.pem
  ssl-key=/path/oreo-server-key.pem
Enable MySQL SSL Client

• Edit the fortress.properties in Web application:

```sh
# sudo vi fortressdemo2/src/main/resources

trust.store.set.prop=true
database.driver=com.mysql.jdbc.Driver
database.url=jdbc:mysql://oreo:3306/demoDB?useSSL=true&requireSSL=true
```
The Anatomy of a Secure Java Web Application
Core Security Architecture Patterns

1. Use HTTPS / TLS on a shared network.

2. Use Container-based Enforcement
   - As appropriate, at each architectural layer.
     - Declarative for static (type-based) resources.
     - Programmatic for dynamic (instance-based) resources.

3. Delegate to a Trusted Third Party (TTP).
   - User Authentication, and Container-based Authorization

4. Use RBAC to express access control policy.

5. Create an audit log record for every application state change.
Part 2

How is it different in Cloud Foundry?
FortressDemo2 in Cloud Foundry

• Cloud Foundry
  • a Platform As A Service offering.

• Enables enterprises to optimize:
  • Infrastructure utilization
  • Application developer productivity

• What changes when deploying to Cloud?
FortressDemo2 in Cloud Foundry
FortressDemo2 in Cloud Foundry

PAAS

Java Servlet Container

app.war

LDAP

RDMBS

ApacheCon NA 2015
FortressDemo2 in Cloud Foundry
FortressDemo2 in Cloud Foundry

VM

Warden Container

Java Servlet Container

app.war

RDMBS

LDAP

HTTP://

HTTPS://

Provided by PAAS

Existing Enterprise Service

Provided by PAAS

ApacheCon NA 2015
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:
- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:
- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Cloud Foundry Request Routing

Remapping of Application URLs

- Original deployment URL:
  - https://host.enterprise.com:8443/fortressdemo2/login.jsp
- Cloud Foundry URL:
  - https://fortressdemo2.pcf.com/login.jsp
- What does this imply?
  - Application context becomes a subdomain.
    - No host or port specified
  - DNS configured to resolve *.pcf.com to IP address of the Cloud Foundry entry point.
Cloud Foundry Request Routing

Client’s DNS resolves *.pcf.com to the CF entry point.

SSL Terminated

HA Proxy forwards request to the CF Router

CF Router maps app name "fortressdemo2" to IP and port.

CF Router

DEA

app.war

HA Proxy

https://fortressdemo2.pcf.com/login.jsp

http://10.110.57.60:12345/login.jsp

Not visible outside cloud
1. SSL Termination

- The SSL connection from user’s browser terminates at the Cloud Foundry HA Proxy.
- Only one certificate needed for all applications hosted in the PAAS.
- No SSL from HA Proxy to our Tomcat instance.
  - All PAAS VMs on the same virtual subnet.
  - No access to application ports from outside cloud.
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:
- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Cloud Foundry Service Bindings

Enabling Distributed Dependency Injection

• Cloud Foundry has the concept of a Service
  • Services enable on-demand provisioning of resources.
  • Services can be Managed or User-Provided.
• Developers declare app’s external service dependencies
  • using `cf bind-service <app> <svc>`.
• At deployment time, war file is scanned
  • Determine the required runtime stack, (i.e. JRE + tomcat)
  • Inject any required connection strings, service credentials
2. MySQL Credential Provisioning

"Auto-Reconfiguration" eliminates hardcoded credentials

- In CF, there are no hardcoded JDBC credentials
  - i.e. in application-Context.xml or fortress.properties.
- Required database credentials are randomly generated and injected at deployment time.
  - e.g.: **User = 5GhaoxJwtCymalOI / Password = 9Bg4tIrEuInZQFVs**
- Feature requires an explicit JDBC DataSource
  - Needed a minor (back-compatible) change to FortressDemo2 application source code
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:
- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Cloud Foundry Buildpacks

What are they? And Why do I care?

• The artifact that contains your runtime stack.
  • Can be auto-detected, or explicitly specified.
  • Example:

    shell> mvn clean package

    shell> cf push fortressdemo2 -p target/fortressdemo2.war
    –b https://github.com/johnpfield/java-buildpack.git

• Deployed application = app.war + buildpack.
  • Intended to support a class of applications, not a single application.
Cloud Foundry Buildpacks

Customizing the Java Buildpack

• The Buildpack is essentially a structure for your runtime libraries.
  • Plus a small amount of “glue code” for interface contract.

• Deployment modes: Easy, Expert, Offline.
  • Tradeoff of having somewhat tighter configuration control, versus using latest-and-greatest.

• Enterprise-specific customizations expected
  • Clone the repo, make changes, & re-bundle archive.
  • Put enterprise-specific jars or security artifacts in the designated place.
3. JEE Realm Configuration
4. JSSE Truststore Management

Configuring the Java Buildpack

- For FortressDemo2 application, we needed 2 configurations of the CF Java Buildpack:
  - Added Sentry jar into the designated place
    - E.g. java-buildpack/resources/tomcat/lib/fortressProxyTomcat7-1.0-RC39.jar
  - Specified our enterprise-specific Truststore in JRE
    - E.g. java-buildpack/resources/open_jdk_jre/lib/security/mycacerts
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:

- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Warden Container Isolation

Providing runtime partitioning and resource control

- Cloud Foundry apps are deployed inside a Warden container.
  - Multiple managed & isolated runtime environments on a single host. (a.k.a: Linux Container, LXC)

- Isolation is via the name-spacing of kernel resources.
  - CPU, memory, disk, & network access.

- Think: “chroot on steroids”
  - Management API integrates: cgroups, iptables, & overlayfs.
## Warden Container Isolation

### Secure Cloud Multi-tenancy at a glance

#### Warden Container “A”
- **Hostname:** 17ruu5224qa
- **VNIC:** w-17ruu5224-qa-1
- **Filesystem:** /home/vcap
- **IP:** 10.254.0.2
- **Memory:** 1Gb

#### Warden Container “B”
- **Hostname:** 17ruu5224qb
- **VNIC:** w-17ruu5224-qb-1
- **Filesystem:** /home/vcap
- **IP:** 10.254.0.6
- **Memory:** 1Gb

---

### DEA VM

- **Hostname:** vm-09bf580a-69a0-431c-9741-bb49c4f318b8
- **VNIC:** eth0
- **Filesystem:** /var/vcap/data/warden/depot/
- **IP:** 10.110.57.60
- **Memory:** 4Gb

---

VNIC: w-17ruu5224qa-0
IP: 10.254.0.1
Filesystem: ./w-17ruu5224qa/tmp/rootfs

VNIC: w-17ruu5334qb-0
IP: 10.254.0.5
Filesystem: ./17ruu5224qb/tmp/rootfs
Summary of Differences

1. SSL Termination
2. MySQL Credential Provisioning
3. JEE Realm Configuration
4. JSSE Truststore Management
5. Warden Container Isolation

What We Need to Understand:
- CF security perimeter, and request routing
- CF service bindings
- CF Build packs
- Linux containers
Network Isolation

- Is like oregano...
- ...you can never have too much.
- You don’t need PCF to do this, but it’s much easier.

- PCF v1.3 features:
  2. Multiple Networks: separate infrastructure VMs from application VMs.
  3. Availability Zones: improved DR availability across geographic locations.
Conclusions from Cloud Deployment

• Core application security patterns are the same, whether doing standalone, or cloud.

  • Application-Container contract is unchanged.

• No changes to the internal security architecture of the application itself.

  • Layered security design is motivated by use cases, and maintainability, not the deployment environment.

• Less platform config. required in a CF deploy.

  • Configure a class of container, rather than an instance.
Coordinates

@architectedsec  mailto: jfield@pivotal.io

https://johnpfield.wordpress.com

GitHub  https://github.com/johnpfield/fortressdemo2

GitHub  https://github.com/shawnmckinney/apache-fortress-demo

http://directory.apache.org/fortress/

https://symas.com/downloads/

https://github.com/cloudfoundry