Оглавление

Безопасность (Security)

Безопасность приложения заключается в предоставлении пользователю доступа только к тем данным и операциям, на которые у него есть права. Безопасность следует по-возможности выносить из бизнес-логики для обеспечения переносимости и масштабируемости приложения, для чего обычно используется ролевая модель делегирования прав.

В программной безопасности приняты кое-какие общие термины:

- субъект (subject) — обобщенное понятие того или чего, что совершает запрос к защищенному объекту, т.е. клиента. Например, стучащий в дверь является субъектом, желающим получить доступ в дом — объект.
- принципал (principal) – одно из свойств или представлений субъекта, понятное системе безопасности. С клиентской стороны это может быть уникальный ID, учетная запись, роль, подписанный ключ и т.д., представляющие уникального человека, приложение, соединение, подписанта ключа и т.п. Со стороны системы безопасности это уникальный ключ, используемый для составления списков контроля доступа (ACL).
- пользователь (user) – это представление конкретного субъекта, связанное с определенным множеством принципалов. В предположении о нераспространении конфиденциальной информации, каждому принципалу соответствует только единственный пользователь.
- учетная запись или аккаунт/учетка (credentials) — это регистрационные данные, по которым определяются права субъекта при обращении к защищенному объекту

Rem: Если, например, один из операторов приложения обращается к некоторому защищенному программному модулю по подписанному ключу, то ключ является учетными данными, подписант этого ключа (тот кто может удостовериться в его валидности) по сути является принципалом, приложение является субъектом, а конкретный оператор,

пытающийся получить доступ к модулю, является пользователем.

- группа пользователей — множество пользователей, наделенное системой безопасности некоторыми одинаковыми групповыми правами.
- роль — множество принципалов (и их объединений в виде пользователей и групп), наделенное системой безопасности некоторыми одинаковыми ролевыми правами для выполнения определенной деятельности.

Rem : в файловых системах ОС вроде Unix или службах каталогов LDAP поддерживается деление пользователей на группы для упрощения реализации ACL (Access Control List, списки контроля доступа). Предполагается, что для приложения группы являются сущностями внешнего мира, поэтому в самом приложении используются внутренние логические эквиваленты групп - роли. Каждой такой прикладной роли можно сопоставить множество групп и отдельных принципалов (user principal). Связь между группами и ролями имеет слабый (через конфигурационный файл) характер, что позволяет получить сопряжение с системой и при этом сохранить переносимость.

Спецификации JEE и EJB определяют базовый набор служб безопасности, которые разработчик приложения может интегрировать декларативно или программно или смешано. Они включают аутентификацию и авторизацию.

- Аутентификация (authentication) – процесс проверки и определения пользователя (обычно через механизм login/password);
- Авторизация (authorization) – процесс наделения пользователя правами внутри системы. Это могут быть разнообразные режимы доступа к данным (чтение-запись-выполнение / r-w-x), допуск пользователей с определенных IP-адресов к определенным (по URL) службам или методам EJB.

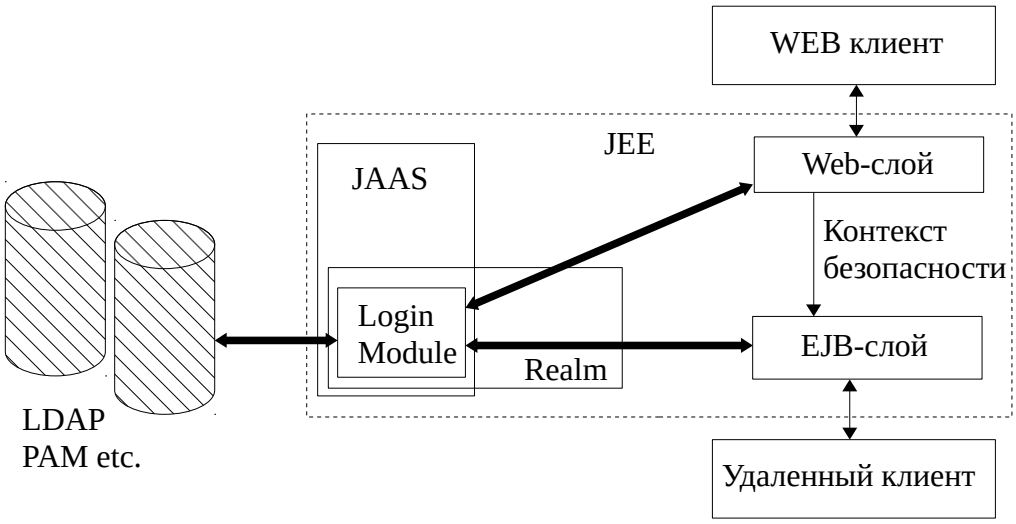
Схема безопасности JSE/JEE

Начиная с версии 1.3, в JEE появилась инфраструктура безопасности в виде фрэймворка JAAS (Java Autentification and Authorization Service) - API аутентификации и авторизации, представленного в пакетах java.security и javax.security JSE (начиная с J2SE 1.4).

В модели безопасности JEE помимо стандартных терминов используется понятие области безопасности (realm). Под областью безопасности (realm) подразумевается некоторое множество групп либо ролей, предполагается, что в каждом realm действует своя политика безопасности (механизм аутентификации и система авторизации).

Также в модели безопасности JEE определены два уровня: системный и прикладной. Для каждой области безопасности (realm) системный уровень определяется набором пользовательских групп и отдельных принципалов. Прикладной уровень определяется набором ролей. При развертывании приложения происходит отображение системных групп/принципалов на прикладные роли в рамках выбранной приложением области безопасности. В JAAS определен интерфейс javax.security.auth.spi.LoginModule,

который и должен заниматься подобным отображением.



В JAAS используется двухфазная аутентификация. Клиентское приложение имеет дело с диспетчером аутентификации приложения, представленным классом javax.security.auth.login.LoginContext. Непосредственной аутентификацией в real-time занимаются встраиваемые (внедряемые) в приложение модули регистрации, представленные интерфейсом javax.security.auth.spi.LoginModule. Клиент передает LoginContext регистрационные данные, LoginContext инициализирует соответствующие модули регистрации и начинает аутентификацию: в первой фазе проводится локальная аутентификация во всех LoginModule, если она всюду прошла успешно, то проводится вторая фаза — фиксация, т.е. превращение клиента в аутентифицированного пользователя с закреплением за ним прикладных ролей.

Защита JEE взаимодействует только с модулем безопасности, построенным на основе фреймворка JAAS, а уже модуль JAAS реализуется вендором EJB-контейнера и отвечает за детали вроде защиты паролей или взаимодействие с внешними службами аутентификации типа LDAP. Для облегчения жизни и избежания повторных аутентификаций, контекст безопасности автоматически распространяется по приложению и становится доступен в любых компонентах, которые его поддерживают.

В JEE6 доступен небольшой API служб защиты для программного создания системы безопасности. Но обычно используется декларативный подход с использованием статичной метаинформации. В EJB-спецификации защита определяется только для сессионных бинов.

Аутентификация в JEE

Аутентификация позволяет JEE-серверу убедиться в личности пользователя. При обращении пользователя к JEE-серверу, ему на время сессии автоматически присваивается ID безопасности (например, через настройку соединения JNDI или вендор-специфической реализации RMI). После получения этого ID безопасности, пользователь может дергать методы EJB. При вызове метода, JEE-сервер неявно

передает EJB-контейнеру вместе с вызовом и пользовательский ID безопасности. Когда EJB-контейнер получает вызов, то он проверяет допустимость ID безопасности и его право на требуемый метод.

Спецификация EJB 3.0 (JEE5) описывает способ передачи учетных данных от клиента к JEE-серверу. Но ее дальнейшая обработка и хранение оставлены на усмотрение вендора. Также не описано, как пользователь должен связывать свой ID безопасности и учетные данные с вызовами EJB, это также решает вендор.

Так или иначе JEE-сервер реализует JAAS-модуль, который и обеспечивает безопасность и может иметь предопределенные реализации, что позволяет обойтись их простым конфигурированием через вендорные файлы-дескрипторы.

Типичный способ аутентификации пользователя происходит через web-слой. В этом web-слое через XML-дескрипторы настраивается алгоритм работы JAAS-модуля JEE-сервера. JAAS-модуль проверяет клиентские данные, в случае успеха создает контекст аутентификации пользователя в виде объекта API JAAS java.security.Principal и приписывает ему соответствующие роли. Далее этот контекст безопасности автоматически распространяется по приложению, попадая в т.ч. в EJB-слой.

Для обеспечения передачи регистрационных данных клиента при удаленном вызове EJB, некоторые вендоры JEE-серверов используют JNDI. В этом случае данные можно запихать в параметры контекста javax.naming.Context вызова при получении заглушек. И при первом же обращении к JEE-серверу учетные данные будут приняты, в случае успеха ассоциированы с пользовательской сессией и распространены JEE-сервером вглубь, например, к EJB-контейнеру, что даст пользователю возможность вызывать далее разрешенные EJB-методы в своем процессе.

```
Ex: добавление аутентификационных данных login/password к параметрам соединения
JNDI для получения доступа к EJB-контейнеру, поддерживающему JNDI-регистрации
...
properties.put(Context.SECURITY_PRINCIPAL, "ExUserName");
properties.put(Context.SECURITY_CREDENTIALS, "password");
Context ctx = new InitialContext(properties);
ExSecureEJBRemote exSecureEJB = (ExSecureEJBRemote)
ctx.lookup("ExSecureEJB/remote");
...
```

Некоторые сервера также определяют отличные от JNDI механизмы аутентификации клиентов, например, для передачи отпечатка пальца. В частности JBOSS использует спецификацию JAAS, дающую развитый API аутентификации. Но использование JNDI все еще остается одним из распространенных способов аутентификации.

Авторизация в JEE

После аутентификации пользователь запрашивает метод бина. В этот момент EJB-

контейнер должен проверить допуск пользователя. В EJB-спецификации для этого предусмотрен механизм делегирования прав через роли. Роли — метки логического доступа, связанные только с бизнес-логикой приложения. С ID реального мира (группами и принципалами) роли связываются только в deploy-time вендор-специфическим образом. Это позволяет за счет гибкости настроек улучшить масштабируемость и портируемость приложения.

Для описания ролевых прав можно использовать аннотации или дескриптор развертывания ejb-jar.xml. Также можно задавать ролевую политику непосредственно в коде самих EJB или вспомогательных классов, например, перехватчиков. Спецификация полностью определяет этот механизм.

Декларативная безопасность EJB

Рассмотрим пример защищенного входа в школу, где: есть 3 типа пользователя — студент, вахтер, директор; существуют 2 входа — основной, служебный; определены 2 состояния школы — открыта, закрыта.

Для реализации примера использован @Singleton EJB и аннотации безопасности с ролевой моделью. С помощью интерфейса определены строковые константы ролей администратора, студента и вахтера:

```
public interface Roles {
  String ADMIN = «Administrator»;
  String STUDENT = «Student»;
  String JANITOR = «Janitor»;
}
```

Роли в области безопасности всего приложения (развертки) объявляются при помощи аннотирования класса EJB аннотацией @javax.annotation.security.DeclareRoles

```
@DeclareRoles({Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
```

Вместо @DeclareRoles можно использовать элемент <security-role-ref> в XML - дескрипторе развертки приложения.

Объявление и/или допуск к классу или методу EJB задается при помощи аннотации @javax.annotation.security.RollesAllowed. При этом аннотация уровня метода имеет больший приоритет, чем уровня класса.

Rem: при одновременном использовании аннотаций @DeclareRoles и @RollesAllowed множества объявленных в них ролей объединяются.

Rem: при наследовании классов, множества ролей, указанных в @DeclareRoles и @RollesAllowed будут объединяться (?)

Хорошей практикой является запрет доступа на уровне класса с разрешениями на уровне методов.

```
@Singleton
@Local(SecureSchoolLocal.class)
@DeclareRoles({Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
@RolesAllowed({})
@Startup
public class SecureSchoolBean implements SecureSchoolLocal {
…
 @RolesAllowed(Roles.ADMIN)
 public void close(){ … }

 @PostConstruct
 @RolesAllowed(Roles.ADMIN)
 public void open(){ … }

 @RolesAllowed({Roles.ADMIN, Roles.JANITOR})
 public void openServiceDoor(){ ... }

 @PermitAll
 public boolean isOpen(){ … }
…
}
```

В данном примере запрещен доступ на уровне класса, а разрешения прописаны на уровне методов. Аннотация @javax.annotation.security.PermitAll разрешает полный доступ. Аннотация @javax.annotation.security.DenyAll полностью запрещает доступ и эквивалентна @RolesAllowed({}), данную аннотацию удобно использовать в период тестирования для блокировки «боевых методов» вроде launchNuclearWar() .

При вызове метода бина, EJB-контейнер сперва будет проверять принадлежность пользователя к сконфигурированным ролям, перед тем как будет запрошен целевой метод экземпляра бина. Если пользовательская роль не подойдет, то будет выброшено исключение javax.ejb.EJBAccessException.

Делегирование прав EJB. RunAs

Для любого EJB можно определить роль, под которой он будет дергать другие EJB. Эта роль может быть произвольной и не зависит от той роли, под которой был вызван его собственный метод. Задается роль аннотации класса: @javax.annotation.security.RunAs

Rem: это аналог команды sudo в UNIX

Rem: по-умолчанию распространяется текущий контекст безопасности, т.е. в методе, вызванном с некоторой ролью xxx, все вызовы других EJB будут идти под ролью xxx.

Rem: клиент, воспользовавшийся аннотированным @RunAs EJB, по сути получает временную

роль и может использовать ее в рамках возможностей этого EJB.

Для MDB это вообще единственный способ вызова защищенного метода другого EJB, т.к. в MDB нет возможности назначать права отдельным методам: все принимаемые MDB сообщения пользовательскими вызовами не являются, соответственно никаких ролей, под которыми был совершен вызов, определить не удастся.

В качестве примера делегирования прав можно рассмотреть ситуацию аварийного закрытия школы. В обычном режиме закрыть школу может только директор. Но в случае пожара закрыть школу через оповещение пожарной службы должен мочь любой, даже аноним.

Для реализации вводится @Singleton EJB с админскими правами, который реализует пожарную службу и дает возможность закрывать школу от имени директора в случае пожара.

```
@Singleton
@RunAs(Roles.ADMIN)
@PermitAll
public class FireDepartmentBean implements FireDepartmentLocalBusiness {
  @EJB
  private SecureSchoolLocalBusiness school;

  @Override
  public void declareEmergency() {
    school.close();
  }
}
```

Сам @Singleton может быть вызван клиентом с любой ролью благодаря @PermitAll.

Программная безопасность EJB

При необходимости управлять правами доступа в run-time одной метаинформации, регулирующей доступ к методам и классам EJB будет недостаточно. Такая необходимость может возникать при зависимых от окружения требованиях безопасности. В качестве примера можно рассмотреть школьную систему безопасности. Открытие школы должно быть доступно студентам и вахтеру только в рабочее время, т.е. доступ должен зависеть от реального времени и задаваться программно (например через службу таймеров).

В JEE6 существует несколько методов, образующих систему безопасности. EJB предоставляет небольшой API для сбора информации о защищенной сессии и оформляющий ее при помощи JAAS API из JSE. Конкретнее, в интерфейсе javax.ejb.EjbContext есть метод для получения информации о клиенте, который вызвал EJB и метод для определения принадлежности текущего пользователя конкретной роли.

Ex: декларативно-программная защита от несанкционированного входа в школу вне рабочего времени

```
@RolesAllowed({Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
public void openFrontDoor() {
  @Resource
  private SessionContext context;
  final String callerName = context.getCallerPrincipal().getName();
  if (!this.isOpen()) {
    if (!context.isCallerInRole(Roles.ADMIN)) {
      throw SchoolClosedException.newInstance("Attempt to open the front door after hours " +
        "is prohibited to all but admins, denied to: " + callerName);
    }
  }
  ...
}
```

В данном примере методу открытия двери вначале декларативно устанавливается полный допуск. Но далее в самом методе он начинает программно регулироваться в зависимости от состояния школы. Текущее состояние школы узнается методом isOpen(). Если она работает, то метод isOpen() выдаст true и входить может любой (директор, студент, вахтер). Если же она закрыта, то isOpen() возвратит false и тогда вход будет позволен только директору, а для других (студент, вахтер) будет выброшено исключение SchoolClosedException. Для идентификации вызова используется метод SessionContext.getCallerPrincipal() внедренного сессионного контекста, возвращающий объект интерфейса java.security.Principal из JAAS с описанием клиента Principal.getName(). Для проверки права клиента на вызов используется метод SessionContext.isCallerInRole(String).

Rem: что будет в реальности возвращать метод Principal.getName() зависит от вендора. Это может быть имя пользователя, имя роли или вообще некая идентификационная метка. Конкретику надо узнавать в вендорной документации EJB-контейнера. Использование этого метода в бизнес-логике может привести к проблемам переносимости и обновляемости систем аутентификации.

Перехватчики & программная авторизация EJB

Перехватчики могут использоваться как посредники в вопросах авторизации. Это избавляет бизнес-логику от жесткой привязки к именам и ролям системы безопасности. Остается слабая связь: бизнес-логика – перехватчик – авторизация EJB.

Ex: перехватчик, проверяющий клиента на соответствие роли «Admin»

```
public class SecurityIntercepter {
  @Resource
  private SessionContext sessionContext;
```

```
  @ArroundInvoke
  public Object checkUserRole(InvocationContext context) throws Exception {
    if(!sessionContext.isCallerInRole(«Admin») {
      throw new SecurityException(«Permission denied!»);
    }
    return context.proceed();
  }
}
```

Поскольку перехватчик работает в едином стеке вызова (потоке) с перехватываемым EJB, контекст безопасности у них единый. При несоответствии роли пользователя, выбрасывается исключение, что приводит к прерыванию стека вызова и недопущению доступа к защищенному методу EJB.

```
Ex: перехватываемый EJB

@Stateless
public class ExEJB {

  @Interceptors(SecurityInterceptor.class)
  public void exSecureMethod (…) { … }
}
```

Rem: такой подход годится для простых схем авторизации, в противном случае число специальных перехватчиков может вырасти до неразумных пределов. И тогда будет лучше использовать декларативную модель с вынесенными в дескриптор развертки правилами авторизации.

Схема безопасности на примере Glassfish 4

Поддержка безопасности описывается как спецификациями EJB и сервлетов, так и вендорной документацией серверов приложения, в данном случае Glassfish 4. Непосредственно сам механизм аутентификации и авторизации обеспечивается на уровне сервера приложений и не требует от приложения никакого дополнительного кода.

Для обеспечения аутентификации сервером приложения GlassFish используется пара доменных java-ресурсов: Realm и LoginModule.

Realm (security realm = область безопасности) функционально представляет из себя логический механизм аутентификации пользователя, работающий поверх модуля JAAS. Каждая Realm тем или иным способом формирует коллекцию групп пользователей в deploy-time. В GlassFish 4 определены 3 стандартные Realm: admin-realm,certificate, file. Также в GlassFish 4 предопределены дополнительные Realm для взаимодействия с различными внешними системами безопасности: LDAP, Solaris, JDBC, PAM. Кроме этого можно создавать собственные Realm, используя API GlassFish, конкретнее расширяя абстрактный класс com.sun.appserv.security.AppservRealm .

В своей работе Realm использует LoginModule — модуль проверки в run-time принадлежности пользователя к набору групп, привязываемый к ней при помощи свойства jaas-context. В свою очередь LoginModule может получить ссылку на использующую его Realm при помощи protected-поля _currentRealm. Вендор Glassfish определяет свою базовую реализацию интерфейса JAAS javax.security.auth.spi.LoginModule. Ее и только ее можно использовать для создания собственных LoginModule.

Политика авторизации задается в дескрипторах развертки приложения. Задать соответствие ролей и групп для WEB-слоя можно при помощи вендорного дескриптора приложения WEB-INF/glassfish-web.xml, для EJB-слоя — в вендорном дескрипторе приложения glassfish-ejb-jar.xml. Остальная политика безопасности для WEB-слоя описывается в стандартном дескрипторе WEB-INF/web.xml, для EJB-слоя в стандартном ejb-jar.xml и/или аннотациях EJB.

Для передачи учетной информации от клиентского браузера к Glassfish при авторизации в WEB-слое существуют стандартные механизмы, они настраиваются в web.xml. Для передачи учетных данных при авторизации в EJB-слое, клиентскому приложению требуется либо взаимодействие с клиентским контейнером Glassfish (Application Client Container, ACC), либо программная реализация механизма с использованием API Glassfish.

Подробности настройки стандартных и предопределенных Realm, использование различных схем защиты (целостность, хеш-слепки-дайджесты учетной информации, HTTPS, клиентские и серверные сертификаты и т.д.) Glassfish есть в книжке .

Создание собственных средств аутентификации

Realm и LoginModule представляют собой java-объекты, базовая реализация классов которых com.sun.appserv.security.AppservRealm и com.sun.appserv.security.AppservPasswordLoginModule, соответственно, входят в библиотеку Glassfish 4. Для создания собственных средств авторизации и аутентификации следует расширить эти классы.

Ex.: пример реализации собственного Realm

```
public class ExRealm extends AppservRealm {
// стандартное для Glassfish имя параметра связи с LoginModule
    private static final String PARAM_JAAS_CONTEXT = "jaas-context";

// Установка свойства  PARAM_JAAS_CONTEXT для связи с LoginModule
// Метод будет дергаться JEE-сервером при создании объекта, properties из domain.xml
    @Override
    public void init(Properties properties)
            throws BadRealmException, NoSuchRealmException {
```

```
        String propJaasContext = properties.getProperty(PARAM_JAAS_CONTEXT);
        if (propJaasContext != null) {
            setProperty(PARAM_JAAS_CONTEXT, propJaasContext);
        }
    }

// Задание имени Realm для конфигурирования в дескрипторе
    @Override
    public String getAuthType() {
        return "example_realm";
    }

// Аутентификация и получение набора групп, связанного с пользователем
    @Override
    public Enumeration getGroupNames(String user)
            throws InvalidOperationException, NoSuchUserException {
        return Collections.enumeration(Arrays.asList("users","guests"));
    }
}
```

Самый важный метод тут — getGroupNames(…), который и задает набор групп текущей области безопасности. В данном примере создается простейший и довольно бесполезный механизм аутентификации, присваивающий каждому пользователю группы “users” и “guests”. Вместо этого, например, можно использовать запрос на группы по имени пользователя к некоторой базе данных.

Rem: для экономии ресурсов при аутентификации кучи пользователей, результаты поиска групп можно кэшировать.

Ex.: пример создания модуля аутентификации

```
public class ExLoginModule extends AppservPasswordLoginModule {

//Аутентификация пользователя и фиксация его набора групп
    @Override
    protected void authenticateUser() throws LoginException {
        ExRealm exRealm = (ExRealm) _currentRealm;
        authenticate(_username, _password);
        Enumeration enumeration = null;
        List<String> authenticatedGroups = new LinkedList();
        enumeration = exRealm.getGroupNames(_username);
        for (int i = 0; enumeration != null && enumeration.hasMoreElements(); i++) {
            authenticatedGroups.add((String) enumeration.nextElement());
        }
// Отображение системных групп и принципалов на прикладные роли
        commitUserAuthentication(authenticatedGroups.toArray(new String[0]));
    }

// Гипотетическая аутентификация через LDAP
    private static void authenticate(String login, String password) throws LoginException {
```

```
    LDAP.authenticate(login, password);
  }
}
```

Предполагается, что данный LoginModule будет взаимодействовать с описанной выше ExRealm и в своей работе также использовать аутентификацию некоторой внешней системы (LDAP).

Для включения созданных Realm и LoginModule в домен, надо собрать их в виде jar-библиотечки. При сборке нужно указать компилятору путь к библиотекам Glassfish javaee.jar, appserv-rt.jar и appserv-ext.jar в classpath. Полученный jar-файл нужно уложить в библиотеку домена ... /.domains/domain1/lib/ExAuth.jar и зарегистрировать в домене:

- в domain.xml найти (создать) элемент <java-config>, прописать путь к jar-файлу аутентификации в атрибуте classpath:

```
... .domains/domain1/config/domain.xml:
… <java-config classpath=”... /.domains/domain1/lib/ExAuth.jar»/> …
```

- в login.conf домена добавить созданный LoginModule, указав FQN класса ExLoginModule:

```
... .domains/domain1/config/login.conf:
…
exLM {
  … .ExLoginModule required;
};
```

- в domain.xml надо прописать создание и инициализацию объекта ExRealm, добавив элемент <auth-realm> после уже существующих <auth-realm>. В его атрибуте classname надо указать FQN ExRealm, в его атрибуте name – имя для дальнейшего использования ExRealm в приложениях домена. Во вложенных элементах <property> указываются свойства, которые затем попадут в параметр java.util.Properties метода init создаваемого объекта:

```
... .domains/domain1/config/domain.xml:
<auth-realm classname="… .ExRealm " name="exRealm"
  <property name="jaas-context" value="exLM"/>
  <property name="auth-type" value="example_realm/>
</auth-realm>
```

Для ввода в действие новой конфигурации надо перезапустить домен.

Аутентификация и авторизация в WEB-слое

Пусть у клиента имеются страницы index.html, for-guset.html, secret.html. Страница index.html является стартовой и содержит ссылки на защищенные secret.html и for-guest.html. Пусть также уже имеется настроенный серверный механизм аутентификации Realm/LoginModule — стандартный или собственный, который умеет идентифицировать пользователей и классифицировать их по группам guests и users. Осталось настроить авторизацию, т.е. наделение пользователей ролями и привязку ролей к защищаемым ресурсам.

Соответствие ролей и групп/принципалов задается в вендорном дескрипторе приложения …/WEB-INF/glassfish-web.xml:

```
...
<glassfish-web-app error-url="">
  <context-root>/path/to/our/App</context-root>
  <security-role-mapping>
    <role-name>user</role-name>
    <group-name>users</group-name>
    <principal-name>specguest</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>guest</role-name>
    <group-name>guests</group-name>
  </security-role-mapping>
...
</glassfish-web-app>
```

В этом XML-фрагменте определена привязка роли user к группе users и принципалу specguest, а роли guest к группе guests (в предположении, что группы и принципалы уже определены в Realm).

В стандартном дескрипторе развертки приложения ../WEB-INF/web.xml надо прописать используемые роли, привязать эти роли к защищаемым ресурсам, указать логику аутентификации (Realm) и способ получения идентификационных данных от клиента.

- Роли (roles):

```
<security-role>
  <description/>
  <role-name>user</role-name>
</security-role>
<security-role>
  <description/>
  <role-name>guest</role-name>
</security-role>
```

  В этом XML-фрагменте заданы роли user и guest

- Ограничения доступа к защищенным ресурсам (constraint):

```
<security-constraint>
  <display-name>Constraint1</display-name>
  <web-resource-collection>
    <web-resource-name>secrets</web-resource-name>
    <description/>
    <url-pattern>/secret*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <display-name>Constraint2</display-name>
  <web-resource-collection>
    <web-resource-name>guests</web-resource-name>
    <description/>
    <url-pattern>/for-guests*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>guest</role-name>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
```

В этом XML-фрагменте определены 2 защищенных набора ресурсов (security-constraint): в первый входят страницы, запрашиваемые по url-шаблону /secret* для пользователей с ролью user; во второй попадают страницы с url-шаблоном /for-guests* для пользователей с ролями user и guest. Остальные страницы не защищены и доступны для произвольных запросов.

Rem: при вложении одного security-constraint в другой, приоритетнее правила вложенного.

- Логика аутентификации (Realm):

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>exRealm</realm-name>
</login-config>
```

В данном XML-фрагменте указан способ простой HTTP-авторизации — BASIC, в качестве логики аутентификации выбрана созданная ранее и прописанная в вендорном domain.xml exRealm.

В GlassFish v4 используются следующие способы пользовательской аутентификации:

- BASIC – простейший пользовательский способ аутентификации со стандартным окном браузера login/password. Данные передаются браузером на сервер в незашифрованном виде, в кодировке base64 (латиница, «+», «/»), даже при использовании HTTPS.
- DIGEST – то же, что и BASIC, но вместо login/password на сервер отправляется их md5-хеш-код (дайджест или выжимка сообщения).
- FORM – пользовательский способ аутентификации с отправкой HTML-формы, содержащей login/password. При использовании HTTPS данные шифруются.
- CLIENT-CERT – пользовательский способ аутентификации , использующий отправку на сервер клиентских сертификатов.

Rem: не все стандартные методы пользовательской аутентификации поддерживаются всеми стандартными Realm. Например Realm file поддерживает только BASIC и FORM.

Также можно написать свою страницу входа с login/password, вместо BASIC.

Rem: JEE-сервер обязан иметь стандартную реализацию обработки формы login/password (в виде фильтра или сервлета) по запросу action=”j_security_check”.

Аутентификация и авторизация в EJB-слое

Пусть на сервере уже имеется настроенный серверный механизм аутентификации — стандартный или собственный Realm/LoginModule, который умеет идентифицировать пользователей и классифицировать их по группам.

Авторизация для EJB описана в спецификации EJB 3.1 (см. выше) и задается метаинформацией: либо аннотациями сессионных бинов, либо в стандартном XML-дескрипторе ejb-jar.xml.

Остается только привязать роли к группам из внешнего мира и настроить механизм передачи учетной записи пользователя с клиента на сервер. Роли привязываются к группам и отдельным принципалам в вендорном XML-дескрипторе glassfish-ejb-jar.xml, в элементе <security-role-mapping> полностью аналогично glassfish-web.xml для web-слоя.

```
...
 <security-role-mapping>
   <role-name>ExRole</role-name>
   <group-name>users</group-name>
   <principal-name>exUser</principal-name>
  </security-role-mapping>
...
```

Передача учетной информации на сервер Glassfish может быть реализована в клиенте с использованием JAAS и вендор-специфического API. Учетные данные передаются в

классах-оболочках интерфейса javax.security.auth.callback.Callback.

Аутентификацию можно проводить явно, программным способом с использованием библиотеки Glassfish:

```
Ex: пример явного программного клиентского запроса на аутентификацию
...
LoginContextDriver.doClientLogin(1, new ExCallbackHandler(“name”,”password”));
...
```

В данном примере, до первого обращения к серверу, клиент локально (в своем процессе) подготавливает учетную информацию путем вызова метода doClientLogin класса com.sun.enterprise.security.auth.login.LoginContextDriver из состава библиотеки Glassfish. Метод doClientLogin эмулирует серверную аутентификацию на стороне клиента, подготавливая при этом учетную информацию в требуемом виде и зашивая ее в вендор-специфический механизм удаленного вызова Glassfish (собственную реализацю RMI) для последующей передачи на сервер при первом же обращении. В качестве параметров задан тип аутентификации «логин/пароль» и объект обработки обратного вызова стандартного интерфейса JAAS javax.security.auth.callback.CallbackHandler, в который и запихивается учетная информация пользователя.

Класс обработки обратного вызова CallbackHandler надо реализовать вручную, для чего достаточно определить единственный метод void handle(Callback[] callbacks). Параметр callbacks представляет массив используемых оболочек аутентификационных данных, классы которых определены в JAAS в пакете javax.security.auth.callback.

```
Ex: реализация  класса обработки обратного вызова типа логин/пароль

private class ExCallbackHandler implements CallbackHandler {
  private String username;
  private char[] password;

  public ExCallbackHandler(String username, char[] password) {
    super();
    this.username = username;
    this.password = new char[password.length];
    System.arraycopy(password,0,this.password,0,password.length);
  }

  @Override
 public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
    try {
      for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof NameCallback) {
          NameCallback nc = (NameCallback) callbacks[i];
          nc.setName(username);
        }
```

```
      else if (callbacks[i] instanceof PasswordCallback) {
        PasswordCallback pc = (PasswordCallback) callbacks[i];
        pc.setPassword(password);
       }
     }
   } catch (Exception ex) {
     ex.printStackTrace();
   }
  }
}
```

Rem: реальная аутентификация на сервере произойдет только при первом вызове сервера.

При использовании на стороне клиента ACC, процесс аутентификации автоматизирован и проходит неявно. ACC выступает в роли прокси для общения с сервером.

- Пользователь вызывает серверный ejb-метод
- ACC переправляет вызов ejb-метода на сервер
- Сервер в ответ запрашивает у ACC учетную информацию
- ACC при помощи стандартного или собственного CallbackHandler собирает пользовательскую учетную информацию в соответствующие оболочки Callback и отправляет на сервер
- Сервер в соответствующем Realm проводит аутентификацию и авторизацию
- При успешной авторизации запускается ejb-метод, результат возвращается ACC
- ACC передает результат пользователю

Ругательства

- субъект (subject) = обобщенное понятие того или чего, что совершает запрос к защищенному объекту, т.е. клиента. Например, стучащий в дверь является субъектом, желающим получить доступ в дом — объект.
- принципал (principal) = одно из свойств или представлений субъекта, понятное системе безопасности. С клиентской стороны это может быть уникальный ID, учетная запись, роль, подписанный ключ и т.д., представляющие уникального человека, приложение, соединение, подписанта ключа и т.п. Со стороны системы безопасности это уникальный ключ, используемый для составления списков контроля доступа (ACL).
- пользователь (user) = это представление конкретного субъекта, связанное с определенным множеством принципалов. В предположении о нераспространении конфиденциальной информации, каждому принципалу соответствует только единственный пользователь.
- учетная запись или аккаунт/учетка (credentials) = это регистрационные данные, по которым определяются права субъекта при обращении к защищенному объекту
- Группа пользователей = множество пользователей, наделенное системой безопасности некоторыми одинаковыми групповыми правами.
- Роль = множество принципалов (и их объединений в виде пользователей и групп),

наделенное системой безопасности некоторыми одинаковыми ролевыми правами для выполнения определенной деятельности.

- Аутентификация = authentication = процесс идентификации пользователя, в котором он доказывает свою подлинность (обычно через механизм login/password).
- Авторизация = authorization = процесс наделения пользователя правами внутри системы. Это могут быть разнообразные режимы доступа к данным (чтение-запись-выполнение / r-w-x), допуск пользователей с определенных IP-адресов к определенным (по URL) службам или методам EJB.
- JAAS = Java Autentification and Authorization Service = API аутентификации и авторизации в JSE, представленный в пакетах java.security и javax.security
- UNIX PAM = подключаемые модули аутентификации UNIX
- Кластер серверов — группа управляемых серверов приложений, между которыми возможна балансировка нагрузки и репликация сессий. Частный случай кластера — отдельный сервер приложений
- Домен — совокупность серверных кластеров, администрируемых как единое целое с помощью единственного администрирующего сервера (DAS, Domain Administration Server). Все сервера домена за исключением администрирующего называются управляемыми (Managed Servers). Каждый запущенный домен создает свои экземпляры серверов. В каждом домене можно развернуть множество приложений. При установке Glassfish и WebLogic создается домен по-умолчанию domain1.
- ACC = Application Client Container – контейнер клиентского приложения, по сути прокси между клиентским приложением и серверным приложением. Реализация ACC входит в состав Glassfish.

Литература

Применение JAAS в Web-приложениях на glassfish v2 = https://habrahabr.ru/post/92608/

Собственный Security Realm в GlassFish = https://habrahabr.ru/post/152483/

Enterprise JavaBeans 3.1 (6th edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Куправк, Майкл Ремижан, стр. nnn

Java EE 6 и сервер приложений GlassFish v3, Дэвид Хэффельфингер

http://www.pramati.com/docstore/1270002/index.htm

https://www.packtpub.com/books/content/designing-secure-java-ee-applications-glassfish