GTARC
GERMAN TURKISH
ADVANCED RESEARCH CENTRE FOR ICT

# *Intermodal Mobility Assistance for Megacities*

# *IMA*

## *Technical Documentation*

## *Kerberos*

# Alphabetical Index

Kerberos is an authentication protocol that allows JIAC Agents to communicate with each other securely over a non secure network. It is based on a ticket system which can prove the agents identity to one another in a secure manner. Kerberos provides a centralized third party authentication and uses symmetric key cryptography to encrypt and decrypt the agents message payload. As encryption algorithmn Advanced Encryption Standard (AES) is used. It is the de facto standard for symmetric cryptography, which comes with 3 different key length: 128, 192, 256 bits. The national institute of standard and technology (NIST) has announced AES as a standard in 2001 [1]. Since then on AES has never been broken and even the smallest (128 bit) among the recommended keys can be used securely at least until 2030 [2].

The following sections describe all components and configurations needed to activate Kerberos in JIAC.

## *1. Kerberos Server*

The Kerberos Server is the trusted third party which acts as a key distributor between two communicating entities. It consists of an Authentication Service (AS) and a Ticket Granting Service (TGS). The AS is required for the initial authentication of the participants to the system and for providing each registered requester a Ticket Granting Ticket (TGT) and a Session Key (SK). With that TGT and SK an agent can expel his identity and securely communicate with the TGS whose function is to verify the requester's identity and to assign him a session ticket including a new SK in order to establish a secure communication with his desired communication partner. The granting ticket as well as session ticket contain information about the requester, the communicating partner, the requesting timestamp and of course the SK and its expiration date. For the secure transmission of those sensitive information two kinds of encryption keys are needed. First, is the pre-mentioned SK which is used to encrypt the message before transmission. Second, is the Secret Key (K) of each registered agent. It is used to encrypt the initial SK and also to verify the tickets. To use Kerberos in the agent framework JIAC it is necessary that all agents should be registered on the server beforehand. The registration ensures that the server knows about the new agent and a secret key only known by that agent and the server is interchanged. In classical approach this shared secret can be a password, pass phrase or a random Byte array. Also it is not specified how this secret is shared. In the scope of IMA we asume that all agents are registered to the server and the server itself is secure and trustworthy. Therefore the server

must be set up with the secrets from all agents. Each agent creates his own secret key. The same function is used to produce the same secret that can be stored on server side. An agents secret consists of the hash of a password, a salt and a given number of hashing iteration as shown below in Listing 1 .

```
public static String encodeSha256(String password, String salt, int iterations) {
        MessageDigest md;
        try {
                md = MessageDigest.getInstance("SHA-256");
                md.reset();
                md.update(salt.getBytes());
                byte[] digest = md.digest(password.getBytes());
                for (int i = 0; i < iterations; i++) {
                        md.reset();
                        digest = md.digest(digest);
                }
                String test = DatatypeConverter.printHexBinary(digest);
                return test.toLowerCase();

        } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
        }
        return null;
}
```

***Listing 1: SecretKey generation method***

In our case the password is the agents name, the salt is the agents name in lowercase representation and the iteration is set to 1024 rounds. It is up to the admin either to create the secrets manually one by one or automatically. In both cases a generated secret must be inserted into the servers MySQL database table 'services' and mapped to the corresponding agent. The database is explained in section 2. This process of creating and persisting the agents secret key must be done only once at the very first set up of the servers database or in case new agents should be integrated.

The KerberosServer project makes use of the project management tool Maven[1] and the application framework Spring[2] that supports inversion of control. A XML-based configuration file exists in the resources package folder *de.dailab.security.service.conf* where several bean properties can be set, but more on that later.

The main components of this project are the classes KerberosAuthenticationService and the TicketGrantingService. As already mentioned the KerberosAuthenticationService

1*https://maven.apache.org/*
2*http://projects.spring.io/spring-framework/*

authenticates the requesting service, let's say Service 1. It checks whether the service is registered and retrieve the SessionKey ($SK_{S1\text{-}TGS}$) so the service can communicate with the TGS in case it is not expired or it creates a new one. Futhermore it creates two kinds of information tickets. One is known as TGT, the other is the so called SessionTicket (ST) which contains the SessionKey ($SK_{S1\text{-}TGS}$) to be delivered to Service 1 and is additionally encrypted with Service 1's SecretKey ($K_{S1}$). Sure the TGT also contains the SessionKey ($SK_{S1\text{-}TGS}$) and other information about Service 1, but it is encrypted with the TGS's SecretKey ($K_{TGS}$). It ensures that no one else than the TGS can read and verify that ticket.

The task of the TicketGrantingService is as mentioned to verify incoming granting tickets and to deliver session keys to the service partners. Similar to AS it checks for existing sessions between the requesting service and his communicating partner, let's say Service 2. Depending on the exsistence of a session, it retrieves, updates or creates a new SessionKey for both partner. To deliver this SessionKey ($SK_{S1\text{-}S2}$) to both, it is necessary to protect this sensitive information against unauthorized access. Therefore the SK including addiontional attributes must be encrypted in different ways before it can be hand over to the communicating partners. One version of the SessionKey ($SK_{S1\text{-}S2}$) must be encrypted with the SessionKey ($SK_{S1\text{-}TGS}$) that only the TGS and Service 1 knows in order to deliver it to Service 1 securely. Another version of the SessionKey ($SK_{S1\text{-}S2}$) must be encrypted with the SecretKey ($K_{S2}$) of Service 2 which is the only instance to decrypt it. In addition to permit Service 2 to start its own communication the TGS has to distribute a TGT to Service 2 as well. Since the direct way is not possible because the server has no push function and the support for Single Sign On, meaning to avoid reauthentication during an active session, the TGS has to pass the granting ticket for Service 2 over communication line of Service 1. This granting ticket, to distinguish it from TGT let's call it Service Granting Ticket, contains in addition to some service specific attributes a SessionKey ($SK_{S2\text{-}TGS}$) for the communication between Service 2 and TGS. It is dedicated to Service 2 in order to prove its access permission to TGS, wherefore this ticket is encrypted with TGS's SecretKey ($K_{TGS}$). Also Service 2 gets a copy of that SessionKey ($SK_{S2\text{-}TGS}$) to exchange information with TGS.

For the KerberosServer project some configurations according database and session expiration time are mandatory. In the configuration file ( Listing 2) the following red marked bean properties must be changed according the project specifications, where the Kerberos Server is used.

```
<beans:bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        scope="singleton">
            <beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />
            <beans:property name="url" value="jdbc:mysql://host:port/database_schema" />
            <beans:property name="username" value="username" />
            <beans:property name="password" value="password" />
</beans:bean>

<beans:bean name="KerberosAuthenticationServer"
        class="de.dailab.security.service.kerberos.KerberosAuthenticationServer" scope="prototype">
            <beans:property name="userDAO" ref="kerberosDatabase" />
            <beans:property name="tgsSessionExpiration" value="36000000" /> <!-- 600 minutes -->
</beans:bean>

<beans:bean name="TicketGrantingService"
        class="de.dailab.security.service.kerberos.TicketGrantingService" scope="prototype">
            <beans:property name="userDAO" ref="kerberosDatabase" />
            <beans:property name="tgsSessionExpiration" value="36000000" /> <!-- 600 minutes -->
            <beans:property name="sessionExpiration" value="7200000" /> <!-- 120 minutes -->
</beans:bean>
```

***Listing 2: KerberosServer configuration***

The tgsSessionExpiration property is the durability of a granting ticket. Usually it last a little bit longer than a working day. The sessionExpiration property stands for the durability of a SessionTicket which also includes the lifetime of the session key. The maximum lifetime should not outlast the tgsSessionExpiration duration.

All tickets and session keys which are distributed from the kerberos server must be handled, managed and persisted by each single service agent. Therefore a key manager is required which is explained in section 3.

## *2. Database*

For Kerberos it is necessary to have access to the pre-defined shared secret of each registered agent in order to construct outgoing and verify incoming tickets. We are using a MySQL database schema with following tables: services and sessions.

All registered agents including their secret are stored in the table 'services' ( Table 1).

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| id | smallint(6) | NO | PRI | NULL | auto_increment |
| name | char(50) | NO | UNI | NULL | |
| skey | tinyblob | NO | | NULL | |

***Table 1: Services table to store service agent names and their secrets***

As previously mentioned in case a new agent should be integrated into the system the generated secret which the agent also own, can manually be assigned to an agent by following query:

*insert into `services` (name, skey) values ('<agentName>','<secretKey>');*

The table 'sessions' ( Table 1) stores the session keys including its expiration date for each connection made between two communicating entities.

Communication can take place between two services or between a service and the TGS. Table : Sessions table to store session keysBoth must be secured hence session keys are needed. In the outlook section at the end of this document we describe an approach to avoid shared secret exchange and manual key setup. This approach is out of IMA scope and can be seen as a future work.

## *3. Key Manager*

The main function of that part is as its name implies to manage the keys each agent receives from the kerberos server. The session keys are used to encrypt and decrypt the agents message payload before it is send to and after it is received from its communication partner. Therefore it is necessary that each agent has its own key manager that can persist and map the session keys to a sender-receiver bonding. Besides this the key manager also manage all received tickets in order to prove its identity and permission to its communication partner or to the TGS. In case the key manager could not find any keys and tickets or the keys and

tickets are expired, it informs the agent to contact the kerberos server for new keys and tickets.

For the purpose of encrypting and decrypting the message payload it is necessary to go deeper in the internal of JIAC. Although it is possible to modify internal processes of JIAC communication, however the intention is to create a plugable concept where it is only necessary to make changes in the configs to enable kerberos. This will be describe in the next section SecureAgent.

## *4. SecureAgent*

SecureAgent is an extension of the JIAC agent. So each SecureAgent has a Communication Bean which provides access to the message bus and is configurable with at least one message transport which than again is responsible for the transmission of a message to the message bus. The SecureAgent implementation uses JMS message transport which holds a KerberosJMSSender for sending a message to an address and a KerberosJMSReceiver for listening to a message at an address. Also part of the modification is to extend the MessageTransport by a pack and unpack method. Those enables the encryption and decryption of the message's payload before it gets transmitted. For the encryption/decryption to take place, the key manager has to deliver the correct session key for each agent individually. Therefore each SecureAgent holds his own key manager. In the configuration file below (Listing 3) the configs for SecureAgent are defined.

```
<beans>
	<import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentNode.xml" />
	<import resource="classpath:de/dailab/jiactng/agentcore/conf/Agent.xml" />
	<import resource="classpath:de/dailab/jiactng/agentcore/conf/JMSMessaging.xml" />
	<import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentBeans.xml" />

	<bean name="KeyManager" class="de.dailab.jiactng.agentcore.comm.SessionKeyManager"
		singleton="false">
	</bean>
	<bean name="KerberosJMSMessageTransport"
class="de.dailab.jiactng.agentcore.comm.KerberosJMSMessageTransport" singleton="false">

		<property name="connectionFactory" ref="StandardJMSConnectionFactory" />
		<property name="host" value="${kerberos.server}"></property>
	</bean>
	<bean name="KerberosCommunicationBean"
class="de.dailab.jiactng.agentcore.comm.KerberosCommunicationBean" singleton="false">
		<property name="transports">
			<set>
				<ref bean="KerberosJMSMessageTransport" />
			</set>
		</property>
		<property name="keyManager" ref="KeyManager" />

	</bean>
	<bean abstract="true" name="SecureAgent" class="de.dailab.jiactng.agentcore.Agent">
		<property name="memory" ref="Memory" />
		<property name="execution" ref="NonBlockingExecutionCycle" />
		<property name="executionInterval" value="100" />
		<property name="communication" ref="KerberosCommunicationBean" />
	</bean>
</beans>
```

***Listing 3: SecureAgent configuration file***

With the initialization of KerberosCommunicationBean the KeyManager gets initialized at the same time. At the same moment a SecretKey is generated for that agent using the same mechanism describe in the chapter KerberosServer. This SecretKey is going to be stored in the agent's own key manger.

As to be seen in the configuration the KerberosCommunicationBean refers to KerberosJMSMessageTransport which contains a placeholder for the KerberosServer location. This placeholder can be replaced by providing a property file containing the following input inside the agent project that uses SecureAgent (see chapter 5):

*kerberos.server = <host>:<port>*

The Kerberos module for JIAC is a Maven project which can be included in other projects using Maven dependency management:

```
<dependency>
        <groupId>de.dailab.jiactng.security</groupId>
        <artifactId>JiacCrypto</artifactId>
        <version>1.0.0-SNAPSHOT</version>
</dependency>
```

## 5. Configuration

Referring to the well-known Ping-Pong example from JIAC V we create a kerberos-activated version out of that.  To do so include the dependency stated above.

### Agent Preparation

First is to define a node on which the agents Ping and Pong can act. This PingPongNode (Listing 4) has the reference to PingAgent and PongAgent:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <import resource="classpath:examples/pingpong/agents/PingAgent.xml" />
        <import resource="classpath:examples/pingpong/agents/PongAgent.xml" />

        <bean name="PingPongNode" parent="NodeWithJMX">
                <property name="loggingConfig" value="classpath:log4j.properties" />
                <property name="agents">
                        <list>
                                <ref bean="PingAgent" />
                                <ref bean="PongAgent" />
                        </list>
                </property>
        </bean>
</beans>
```

***Listing 4: PingPongNode.xml***

PingAgent (Listing 5) and PongAgent (analogous) are both descending from SecureAgent. So they must import SecureAgent.xml and use it as parent instead of Agent.xml:

*<import resource="classpath:nodes/SecureAgent.xml" />*

*and set*

*parent="SecureAgent"*

For the communication to KerberosServer it is necessary to inform the agent where to find the server. This can be done by adding a PlaceholderConfigurer which is provided by Spring. In that case the placeholder points to the property file 'kerberos.properties' which contains an one-line property :

*kerberos.server = <host>:<port>*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<import resource="classpath:nodes/SecureAgent.xml" />

	<bean name="PingAgent" parent="SecureAgent" singleton="false">
		<property name="agentBeans">
			<list>
				<ref bean="PingBean"/>
			</list>
		</property>
	</bean>
	<bean name="PingBean" class="examples.pingpong.PingBean" scope="singleton" >
		<property name="textMessage" value="Here is Ping" />
		<property name="executionInterval" value="5000"/>
		<property name="logLevel" value="INFO" />
	</bean>
	<bean id="kerberosServer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
		<property name="locations">
			<list>
				<value>classpath:kerberos.properties</value>
			</list>
		</property>
		<property name="ignoreUnresolvablePlaceholders" value="true" />
	</bean>
</beans>
```

***Listing 5: PingAgent.xml***

Before PingAgent and PongAgent can talk to each other securely the database has to be set up and initialized with the secret keys of all agents and the kerberos server should be running.

## *Database Preparation*

Create a database and put two tables as shown in Listing 6 in it.

```
CREATE TABLE IF NOT EXISTS `services` (
`id` SMALLINT NOT NULL AUTO_INCREMENT ,
`name` CHAR(50) NOT NULL ,
`skey` BLOB(16) NOT NULL ,
PRIMARY KEY (`id`) ,
UNIQUE INDEX `name_UNIQUE` (`name` ASC) );



CREATE TABLE IF NOT EXISTS `sessions` (
`id` INT NOT NULL AUTO_INCREMENT ,
`service_id` SMALLINT NULL ,
`service2_id` SMALLINT NULL ,
`tgs_id` SMALLINT NULL ,
`skey` BLOB(16) NOT NULL ,
`expirationDate` BIGINT NOT NULL ,
PRIMARY KEY (`id`) );
```

***Listing 6: MySQL queries to create 'services' and 'sessions' table***

Use the Method in Listing 1 to create a SecretKey for the desired agent and insert it to table 'services' with

mysql> *insert into `services` (name, skey) values ('<agentName>','<secretKey>');*

## *Kerberos Server Preparation*

The KerberosServer contained in KerberosServer project can be started via Eclipse with VM Arguments:

*-Dcom.sun.management.jmxremote*

*-Dcom.sun.management.jmxremote.port=9999*
*-Dcom.sun.management.jmxremote.local.only=false*
*-Dcom.sun.management.jmxremote.authenticate=false*
*-Dcom.sun.management.jmxremote.ssl=false*

or just by running the Java archive (Standalone version) or the start script.
The default TgsSession duration is set to 10 hours and a service session expires after 2 hours. If there is a need to modify those parameters, the corresponding configuration file lies in **`de.dailab.security.service.conf.UserMgmtKerberos.xml`** in the resources of KerberosService project.

## *6. Outlook*

In the current version the server needs to be initialized with the secrets of all agents. Even it is not a complex process which happens once for a fix set of agents, the workload of manual insertion may consider whenever new agents enter the system. To prevent from inserting and mapping the secrets for each single agent we may consider the usage of a Public Key Infrastructre (PKI). Unlike the symmetric key approach on what kerberos is based on, there is no need to worry about how to distribute the secret securely from an agent to the server. With the PKI approach an agent generates two interdependent keys, so-called PrivateKey and PublicKey. A PrivateKey is something that the agent should keep secure and only known by the agent himself. The PublicKey in contrast can be distributed openly.
On the other side a trusted key server can be created who keeps all the PublicKeys. The kerberos server can interact with that key server in order to request for the desired PublicKey. A SessionKey can be encrypted by using the requested PublicKey. The only instance who ever can decrypted the SessionKey is the owner of the corresponing PrivateKey. Furthermore the PrivateKey can also be used to sign data e.g to proof ones identity.

## *Bibliography*

[1] FIPS, Announcing the advanced encryption standard (AES), 2001, Technology Laboratory, National Institute of Standards, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] NIST, Recommendation for Key Management, 2012, NIST Special Publication 800-57, http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf