# LDAP findings summary

## The Problem

Load testing revealed, that repeated LDAP authentication requests keep creating new physical connections to the LDAP server, despite Java's built-in ldap connection pooling being enabled

Under load these connections keep piling up, blocking ports until the OS frees them:

tcp6 0 0 [::]:amqp [::]:\* LISTEN

tcp6 0 0 localhost:58046 ubuntu:ldap TIME\_WAIT

tcp6 0 0 ubuntu:ldap localhost:57993 ESTABLISHED

tcp6 0 0 localhost:58011 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58033 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58018 ubuntu:ldap TIME\_WAIT

tcp6 0 0 ubuntu:ldap localhost:57994 ESTABLISHED

tcp6 0 0 localhost:58065 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58061 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58038 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58014 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58040 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58008 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58009 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58043 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58042 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58022 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58058 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:57997 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58004 ubuntu:ldap TIME\_WAIT

tcp6 0 0 ip6-localhost:57568 ip6-localhost:46310 TIME\_WAIT

tcp6 0 0 localhost:58020 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58034 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58053 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58047 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58001 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58054 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58010 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58048 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58026 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58063 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58027 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58028 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58019 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58006 ubuntu:ldap TIME\_WAIT

getnameinfo failed

getnameinfo failed

tcp6 0 0 [UNKNOWN]:ldap [UNKNOWN]:47035 ESTABLISHED

tcp6 0 0 localhost:58032 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58044 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58057 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58000 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58016 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58031 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58005 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58021 ubuntu:ldap TIME\_WAIT

tcp6 0 0 localhost:58056 ubuntu:ldap TIME\_WAIT

The problem appeared first on Windows Server 2008 R2, on Linux it didn't seem to occur. My own tests though showed the same symptoms on both, Windows and Unix.

## Investigation Results

### Pool Timeout

the connection eviction timeout value was set to 30. The value's unit is milliseconds, so it should rather be -Dcom.sun.jndi.ldap.connect.pool.timeout=300000. See <http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/config.html>

### "Bind" bypasses pool for security reasons

By default, to authenticate a user Spring Security performs a "bind" operation (see <http://docs.oracle.com/javase/jndi/tutorial/ldap/security/ldap.html>). A successful "bind" authenticates the connection and all subsequent operations on this connection happen in the security context of that user. To prevent the system's security from being compromised, Spring Security therefore enforces new physical connections for "bind" operations of different users.

From the javadoc of "DefaultSpringSecurityContextSource": "Spring LDAP 1.3 doesn't have JVM-level LDAP connection pooling enabled by default. This class sets the pooled property to true, but customizes the DirContextAuthenticationStrategy used to disable pooling when the DN doesn't match the userDn property. This prevents pooling for calls to AbstractContextSource.getContext(String, String) to authenticate as specific users."

This is in line with Sun/Oracle's own recommendations, to not use pooling for other than anonymous connections, see "When Not To Use Pooling" <http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/pool.html>

In fact also other LDAP client SDK vendors recommend against pooling for connections that are authenticated for different users.

### JVM built-in pool is not sufficient for HA environments

As stated in <https://wiki.jasig.org/display/CASUM/LDAP>, the built-in ldap conncetion pooling is not sufficient for highly available environments because "[..] com.sun.jndi.ldap.connect.pool=true uses a strategy that will incur unacceptable latency in the case of LDAP node failure"

### Web Service Authentication Requirements

Before suggesting any potential solution, it is worth to review a Web Service's actual authentication requirements. A Web Service's usage profile is significantly different from typical web applications. In case of a large-scale deployment, a Web Service will have to deal with a large number of concurrent clients that each will perform only few operations, as compared to a large number of request from a few clients in Client-Server solutions

## Suggestions

### Solving timeout

solving the short timeout value is trivial, just set a larger value, typically around 5 minutes

### Prevent connections from piling up

Clearly, having tcp connections piling up is not desirable. The following looks at a few potential solutions.

#### Cache authenticated clients

Given a Web Service has to deal with many different clients performing only few operations, the value of any caching efforts is questionable. In any case each client has to be authenticated first, so given 100k clients performing 3 operations each, we still have to perform a minimum of 100k authentications. Of course at a certain scale caching might provide some gain. Even on very fast boxes it makes a significant difference whether they need to perform 1mio or 3mio authentication operations.

Caching could be done e.g.

1. by validating the user's password once against LDAP and keeping a salted hash of the given password in an in-memory cache. Subsequent reqests then can be verified against this cached hash
2. connections over SSL/HTTPS implicitely carry session information - hence such client connections can simply keep all required data in memory once loaded.

The places to look for implementing your own caching are "SecurityContextRepository", which specifies the security context persistence strategy for security contexts.

#### Disable Spring's "protection" mechanism

In theory under certain conditions it might be possible to disable Spring's "protection" mechanism. Given the correct configuration, the bind() operation is always the first operation after obtaining a connection from the pool. Thus any subsequent operation - like fetching user details - should happen in the correct security context.

To achieve this, the following changes need to be done to the typical Spring Security LDAP configuration:

1. instead of

<sec:ldap-server id="ldapServer"

url="${ldap.url}"

manager-dn="${ldap.managerDn}"

manager-password="${ldap.managerPassword}" />

explicitely set the "authenticationStrategy" on the ContextSource:

<bean id="ldapServer"

class="org.springframework.security.ldap.DefaultSpringSecurityContextSource">

<constructor-arg value="${ldap.url}"/>

<property name="userDn" value="${ldap.managerDn}"/>

<property name="password" value="${ldap.managerPassword}"/>

<property name="authenticationStrategy">

<bean class="org.springframework.ldap.core.support.SimpleDirContextAuthenticationStrategy" />

</property>

</bean>

this effectively overrides Spring's default mechanism to suppress pooling for connections using a different account than the ContextSource's original managerDn.

1. The BindAuthenticator must be configured to access a user object directly using the DN. Otherwise it performs a generic ldap search for the user object first in order to obtain this DN for authentication - a search that might potentially happen in a different security context. So instead of

<sec:authentication-manager alias="authenticationManager">

<sec:ldap-authentication-provider server-ref="ldapServer"

user-search-filter="${ldap.userSearchFilter}"

user-search-base="${ldap.userSearchBase}"

group-search-filter="${ldap.groupSearchFilter}"

group-search-base="${ldap.groupSearchBase}"

group-role-attribute="${ldap.groupRoleAttribute}">

</sec:ldap-authentication-provider>

</sec:authentication-manager>

rather use

<sec:authentication-manager alias="authenticationManager">

<sec:ldap-authentication-provider server-ref="ldapServer"

user-dn-pattern="${ldap.userDnPatterns}"

group-search-filter="${ldap.groupSearchFilter}"

group-search-base="${ldap.groupSearchBase}"

group-role-attribute="${ldap.groupRoleAttribute}"

role-prefix="none">

</sec:ldap-authentication-provider>

</sec:authentication-manager>

This solution may work and may be sufficent. Nevertheless it should be carefully evaluated and tested!

#### Performing Password Comparison

This is a technique similar to what is usually done when authenticating against SQL databases. First using an ldap search command the user object is retrieved from LDAP. The provided password is then compared with the password stored in the user record.

Usually (hopefully) passwords of course are not stored in plaintext. Rather what is stored is a hash value of the password, generated by a well-known algorithm. LDAP knows a few standard hashing algorithms, common are md5, {sha} and {ssha}, defined in RFC2307 (http://www.openldap.org/faq/data/cache/347.html).

Unfortunately - out of the box - Spring Security only supports unsalted password algorithms. This is not recommended for use in production. For {sha} and {ssha} it is not difficult to implement, the a web service project should overwrite the class " org.springframework.security.ldap.authentication.PasswordComparisonAuthenticator" to deal with salted passwords (see Javadoc of PasswordComparisonAuthenticator).

While this entirely solves the problem of pooled connections switching security contexts with each user trying to login, there is one downside: the password algorithm used by the LDAP server needs to be known and configured in a Web Service to be able to perform this compare operation.

### Using alternative LDAP client libraries

Instead of relying on Oracle's default ldap client & pooling implementation, it is certainly worth to take a look at alternative LDAP client libraries. During my research I came across in particular

* Apache DS  
  <http://directory.apache.org/api/news.html>
* OpenDJ SDK / ForgeRock
* UnboundID LDAP SDK for Java  
  <https://www.unboundid.com/products/ldap-sdk/>

## General Recommendations

As e.g. suggested by the CAS project, it is recommended to use different ContextSources for authentication and retrieving user details

<https://wiki.jasig.org/display/CASUM/LDAP>

Also note their paragraph under "Connection Pooling":

*"The use of PoolingContextSource is strongly recommended in cases where it is supported. This component uses commons-pool object pooling and has performance characteristics suitable for HA environments. This is in stark contrast to the JNDI pooling feature enabled by com.sun.jndi.ldap.connect.pool=true that uses a strategy that will incur unacceptable latency in the case of LDAP node failure."*