KIP-371: Add a configuration to build custom SSL principal name

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *"Adopted*

Discussion thread: *here*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

KafkaPrincipal is formed from the name of the principal received from the underlying channel. In the case of SSL, it is string representation of the X.500 distinguished name using the format defined in RFC 2253. By default, the user name associated with an SSL connection looks like the following. "CN=writeuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown".

Often, users want to extract one of the field (e.g., CN) as the principal name. CN is the commonly used field. Currently in order to customize SSL username a customized principal builder class is required. For this simple change, users need to build and maintain custom principal builder class and also package and deploy the jar to the all brokers. Having built-in SSL principal builder configs/rules that allows to customize SSL principal name will be useful.

Public Interfaces

Option 1 : Check Rejected Alternatives section below.

Option 2:

Similar to SASL kerberos principal rules, we can have list of SSL principal mapping rules.

ssl.principal.mapping.rules: This config allows a list of rules for mapping distinguished name to short principal name. The rules are evaluated in order and the first rule that matches a principal name is used to map it to a short name.
Any later rules in the list are ignored. By default, string representation of the X.500 certificate will be the principal.

The possible values are:

RULE:exp The principal name will be formulated from exp.

The format for exp is **pattern/replacement/[LU].** If the string matches the pattern, then the replacement command will be run over the string.

Each rules starts with "RULE:" and contains an expression as the following formats.

RULE:pattern/replacement/
RULE:pattern/replacement/[LU]

DEFAULT: string representation of the X.500 certificate will be the principal.

Similar to SASL kerberos principal rules, we can also support lowercase/uppercase rule, to force the result to be all lower/upper case. This is done by adding a "/L" , "/U" to the end of the rule.

Example Rules:

```
ssl.principal.mapping.rules=
RULE:^CN=(.*?),OU=ServiceUsers.*$/$1/,
RULE:^CN=(.*?),OU=(.*?),O=(.*?),L=(.*?),ST=(.*?),C=(.*?)$/$1@$2/,
RULE:^cn=(.*?),ou=(.*?),dc=(.*?),dc=(.*?)$/$1@$2/L,
RULE:^.*[Cc][Nn]=([a-zA-Z0-9.]*).*$/$1/L,
DEFAULT
```

This is option supports multiple mapping patterns. Some sample mapping patterns are given below.

Distinguished Name	Mapping pattern	Mapping replacement	mapped name
CN=kafka-server1, OU=KAFKA	^CN=(.*?),OU=(.*?)$	$1	kafka-server1
CN=kafka1, OU=SME, O=mycp, L=Fulton, ST=MD, C=US	^CN=(.*?), OU=(.*?), O=(.*?), L=(.*?), ST=(.*?), C=(.*?)$	$1@$2	kafka1@SME
cn=kafka1,ou=SME,dc=mycp,dc=com	^cn=(.*?),ou=(.*?),dc=(.*?),dc=(.*?)$	$1	kafka1

Notes:

Proposed mapping rules works on string representation of the X.500 distinguished name(RFC2253 format) [1].
Mapping rules can use the attribute types keywords defined in RFC 2253 (CN, L, ST, O, OU, C, STREET, DC, UID).

Any additional/custom attribute types are emitted as OIDs. To emit additional attribute type keys,
we need to have OID -> attribute type keyword String mapping.[2]

For example, String representation of X500Principal("CN=Duke, OU=JavaSoft, O=Sun Microsystems, C=US, EMAILADDRESS=test@test.com")
will be "CN=Duke,OU=JavaSoft,O=Sun Microsystems,C=US,1.2.840.113549.1.9.1=#160d746573744074657374652e636f6d"

If we have the OID - key mapping ("1.2.840.113549.1.9.1", "emailAddress"), then the string will be
"CN=Duke,OU=JavaSoft,O=Sun Microsystems,C=US,emailAddress=test@test.com"

Since we don't have OID - key mapping, we can not use additional attribute type keyword string in our rules.
If the user want to extract additional attribute keys, users need to write custom principal builder class.

[1] https://docs.oracle.com/javase/7/docs/api/javax/security/auth/x500/X500Principal.html#getName(java.lang.String)

[2] https://docs.oracle.com/javase/7/docs/api/javax/security/auth/x500/X500Principal.html#getName(java.lang.String,%20java.util.Map)

Proposed Changes

1. We will update DefaultKafkaPrincipalBuilder class to handle above proposed configuration options.

2. Proposed mapping rules works on string representation of the X.500 distinguished name(RFC2253 format)

3. Proposed configuration will be ignored, if SSL client authentication is disabled. (In this case principal name is ANONYMOUS).

4. Proposed configuration will be ignored, if an extension of KafkaPrincipalBuilder is provided by the principal.builder.class configuration.

Compatibility, Deprecation, and Migration Plan

- There won't be any change of current behavior. By default, string representation of the X.500 certificate will be returned.

Rejected Alternatives

Option 1:

Add new configuration parameter pair of the form:

ssl.principal.mapping.pattern=^.*[Cc][Nn]=([a-zA-Z0-9.]*).*$
ssl.principal.mapping.value=$1

For the pattern portion of the pairing, regular expression syntax is used to parse the original identity into components. The value portion of the pairing uses these parsed components in variable substitution format to build the translated version of the identity.

So a login with "CN=localhost, OU=OrgUnit, DC=Company" matches with pattern above and the mapping value $1 is applied. The principal name will normalized to "localhost".

Examples:

```
ssl.principal.mapping.pattern=^CN=(.*?),OU=ServiceUsers.*$
ssl.principal.mapping.value=$1

ssl.principal.mapping.pattern=^CN=(.*?),OU=(.*?),O=(.*?),L=(.*?),ST=(.*?),C=(.*?)$
ssl.principal.mapping.value=$1@$2

ssl.principal.mapping.pattern=^CN=(.*?),OU=(.*?)$
ssl.principal.mapping.value=$1@$2
```

This option supports single mapping pattern. This handles the common use case.

For consistency and flexibility reasons, we will be going with option 2.