HBS Security Framework API

Background

Security is an emergent property of a system. A secure system needs to be built from the ground up using secure design principles. Because systems depend so heavily on the open source and 3rd party frameworks they're built on, developers must focus on leveraging the right frameworks' security controls correctly, and in the right locations.

A few guiding principles guide the selection of framework security controls, their placement within a design and their correct use. The two most compelling are:

- Prefer automatically applied controls to those a developer must remember ;
- Align controls as closely with convention and framework design as possible.

Expanding on these principles,

Build your controls in such a manner that they are mandatory, invisible and automatically applied. When a framework applies controls automatically, the developer cannot forget to apply the controls, or use them incorrectly. For example, if a developer needs to remember to validate an incoming request represents a logged in user, then the authentication control is **not** invisible, mandatory or automatically applied. However, if using framework configuration (or filters/inceptors) we are able to verify an authenticated resource request even before it hits the resource then developers do not have to worry about authentication. The same argument can be made about input validation, output encoding, and others.

In order to align controls most closely with convention and framework design, follow the following heuristic for choosing control alternatives. Prefer in decreasing order :

1. *Platform-provide* – Controls provided by frameworks become "platform provided" from the application developer's perspective. For example, developers get protection from code injection "for free" because of the type-safety and heap protection provided by the .NET and JEE runtime platforms.
2. *Convention* – Developers, following a natural and JQuery-like JSP tag syntax convention, can get output encoding "for free" by including a contextually-aware output encoding library in their project build;
3. *Configuration* – If security can obtain simply through use of configuration, such as for mandatory SSL use, HTTP-only cookies, and others, developers need not remember to implement use of these feature;
4. *Service* – Certain security controls, such as encryption or tokenization, fit best when abstracted out as separate components. These functions, often requiring a high degree of Subject Matter Expertise to implement correctly, should be built once and centrally, and made straightforward and easy to integrate with.
5. *OO/AD – Annotation, Inheritance, Meta-programming, etc.* – When necessary to place security controls within an application doing so through annotation, such as with Spring Security's authorization constraints, inheritance, or another OO/AD-based abstraction is preferable. These techniques seek to cleanly separate the security from functional aspects of the implementation , making auditing and change easy.
6. *Custom code/Feature-function development* – Instruct developers to call a specific security feature/function library from their code directly (e.g. OWASP's ESAPI), often through development standards or security policy. This option demands developers remember to call and API, do so for the right uses, and do so correctly. The amount of opportunity for failure makes this the obvious option of last resort.

Note : This content is an extract from Cigital's blog.

High Level security requirements for any web-based application (OWASP Top 10)

Category	**Requirements description**
Authentication	Access control for all pages.
	Password rules check validation.
	Audit log on success or failure.
	Password storage (Hashing & Salting)
	Credentials used to access external resources should be stored encrypted.
Session Management	Invalidate the session on user logout or session timeout.
	Securely Manage the session id's/tokens generation and storage.
	Clear the user data on logout.
	Prevent duplicate concurrent user sessions.
Access Control	Pages can be accessed only by the authorized and authenticated users.
	Protect against direct object references.
	Access only secure files.
	Log access control decisions.
	Generate very strong anti CSRF tokens.
Input Validation	Protect against SQL Injection, XSS attacks, LDAP injection, OS Command line injection, buffer overflows (XML).
	Input validation and output encoding.

	HTTP parameter pollution attack.
Cryptography	Implement all the cryptographic functions used to protect secrets like passwords etc.
	Fail securely , in case of failure.
	Generate random number for keys/secrets using approved API's
Secure Error handling and Logging	Implement secure logging control to prevent output error messages or stack traces containing sensitive data.
	Record log events with NIST defined attributes.
	Implement error handling logic for all the security controls.
Data Protection	Disable client side caching of sensitive data.
	Send sensitive data in HTTP message body (POST).
	Purge/Invalidate the sensitive data immediately after use.
Communications Security	Use SSL over HTTP using CA authorized server certificate.
	Log connection failures.
HTTP Security	Accept only a defined set of HTTP request methods such as POST and GET.
	Prevent Click jacking.
Business Logic Security	Protect against brute force or denial of service attacks.
	Process business logic in highly secured environment.
	Protect from transaction spoofing (triggered by session tampering or session replay attacks).

Files and Resources	Protect from path traversal attacks by canonicalizing file names.
	Input validation on the file names passed as a user input.
	File storage outside web-root/as a blob in database.
General Audit Log Requirements	Log the audit events on all the sensitive activities like user managements, access control, security events etc.

HBS Security Utils (library)

SVN path : https://rb-tmp-dev.de.bosch.com/svn/HBS/Security/trunk/hbs-security-util

It defines a web application security library to address commonly defined security requirements across different applications.

This library has some basic design. The design is highly influenced by OWASP ESAPI library (open-source framework)

- There is a set of security **control interfaces**. They define for example types of parameters that are passed to types of security controls.
- There is a **reference implementation** for each security control. The logic is not specific to any application as such. An example: DefaultEncryptor.
- There are optionally your **own implementations** for each security control. This may contain logic specific to an application. This needs to be implemented as per business requirements.

Features currently added to this library.

1. Authentication (Cryptographic operations) and Protection against sensitive data.
   a. Encryption.
   b. Decryption.
   c. Password Hashing with Salt.
2. Encoding Mechanism, Canonicalization
   a. encodeForJavascript.
   b. encodeForHTML.
   c. encodeForXML
   d. encodeForSQL
   e. encodeForURL.
3. Input Validation
   a. Basic input validation for SSN, URL, Email, Date, File, Directory path etc.
   b. Validate HTTP parameters and headers using pre-defined set of patterns.
   c. XSS (Cross site scripting).
4. Denial of Service (TBD)
   a. XML injection.
5. Generation of Secured random tokens, integers, strings, long, UUID etc.

6. Configure the security related attributes, implementations classes, patterns as per the application demands.
7. Error handling for different security controls.

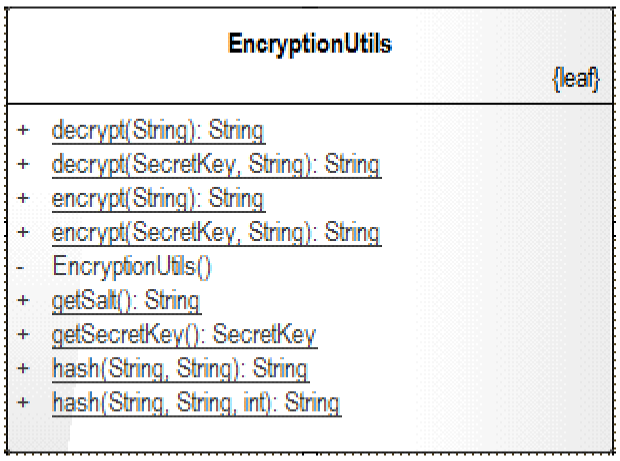
Library Packages and there usages

com.bosch.security.crypto - EncryptionUtils

Features

- Provides utility methods to encrypt or decrypt a message using a secret key.
- Provides method to hash the password strings using salt and returns a hashed value back to user.
- Set of overloaded functions which can be utilized as per the application needs.
- Provide with different methods to create salt or a secret key.

Class Diagram


EncryptionUtils
{leaf}
+ decrypt(String): String
+ decrypt(SecretKey, String): String
+ encrypt(String): String
+ encrypt(SecretKey, String): String
- EncryptionUtils()
+ getSalt(): String
+ getSecretKey(): SecretKey
+ hash(String, String): String
+ hash(String, String, int): String

With this utility, we can easily ecrypt/decrpt a message using the default encryption key or user -defined. We can also create a hashed value for any password with salt and use it to store in database.

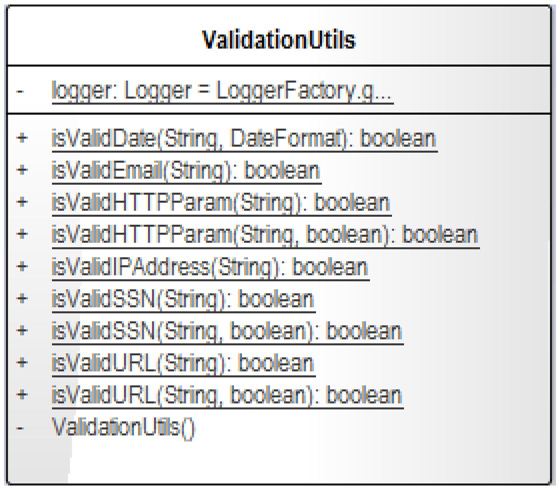
```
//Encryption/Decryption
try
{
 // Default encryption key is referred. Default encryption key is added as static
variable in memory during jar load.
 String encryptedData = EncryptionUtils.encrypt("XYZabh123#$%");
 String decryptedData = EncryptionUtils.decrypt(encryptedData);

    // Encrytpion key can be generated randomly and can be passed as an input argument
to encrypt/decrypt function.
 SecretKey secretKey = EncryptionUtils.getSecretKey();
 String encryptedData = EncryptionUtils.encrypt(secretKey, "XYZabh123#$%");
 String decryptedData = EncryptionUtils.decrypt(secretKey, encryptedData);
}
catch (EncryptionException e) {
 System.out.println("Encryption Failed..");
}

//Hashing
String password1 =  "TestPwd@17198*#";
String salt = EncryptionUtils.getSalt();
try {
 String hash1 = EncryptionUtils.hash(password1, salt);
 String hash2 = EncryptionUtils.hash(password1, salt);
 if(hash1.equals(hash2)) {
    System.out.println("Password matched");
 }
} catch (EncryptionException e) {
 System.out.println("Hashing failed..");
}
```

com.bosch.security.validator - ValidationUtils

Features

- Provides utility methods to validate SSN, Email, UserName, URL, Date, Integer, SerialNumber etc as per the different pattern configured.
- Provides methods to test for HTTP header information and request parameters.
- Overloaded function with an option to canonicalize before input validation.

Class Diagram


ValidationUtils
- logger: Logger = LoggerFactory.g...
+ isValidDate(String, DateFormat): boolean
+ isValidEmail(String): boolean
+ isValidHTTPParam(String): boolean
+ isValidHTTPParam(String, boolean): boolean
+ isValidIPAddress(String): boolean
+ isValidSSN(String): boolean
+ isValidSSN(String, boolean): boolean
+ isValidURL(String): boolean
+ isValidURL(String, boolean): boolean
- ValidationUtils()

Sample code showing few validation steps. All the utility methods return a Boolean flag defining if the validation is successful or not.

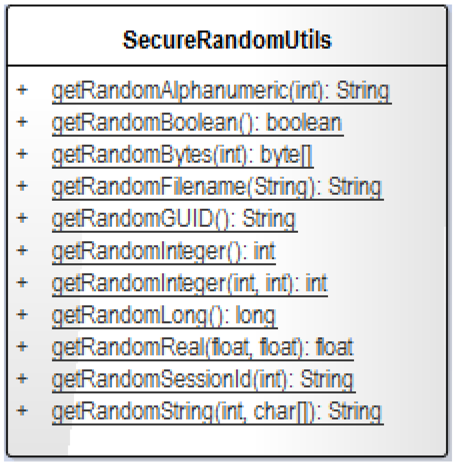
```
//Validate for HTTP request parameter WITHOUT canonicalization (second arg set to
false)
boolean isValidHTTParameter;
isValidHTTParameter =
ValidationUtils.isValidHTTPParam("((\\%3C)|<)((\\%2F)|\\/)object|embed|iframe|base",fa
lse);

//Validate for HTTP request parameter WITH canonicalization (second arg set to true)
boolean isValidHTTParameter;
isValidHTTParameter =
ValidationUtils.isValidHTTPParam("((\\%3C)|<)((\\%2F)|\\/)object|embed|iframe|base",tr
ue);

//Validate SSN
boolean isValidSSN;
isValidSSN = ValidationUtils.isValidSSN("078-05-1120");

//Validate Date
boolean isValidDate;
DateFormat dt = new SimpleDateFormat("MM/dd/yyyy");
isValidDate = ValidationUtils.isValidDate("08/26/2014", dt);
```

com.bosch.security.random - SecureRandomUtils

<u>Features</u>

- Provides utility methods to generate secure random string, integer, long, bytes, UUID, alphanumeric, sessionId etc.

<u>Class Diagram</u>


SecureRandomUtils
+ getRandomAlphanumeric(int): String
+ getRandomBoolean(): boolean
+ getRandomBytes(int): byte[]
+ getRandomFilename(String): String
+ getRandomGUID(): String
+ getRandomInteger(): int
+ getRandomInteger(int, int): int
+ getRandomLong(): long
+ getRandomReal(float, float): float
+ getRandomSessionId(int): String
+ getRandomString(int, char[]): String

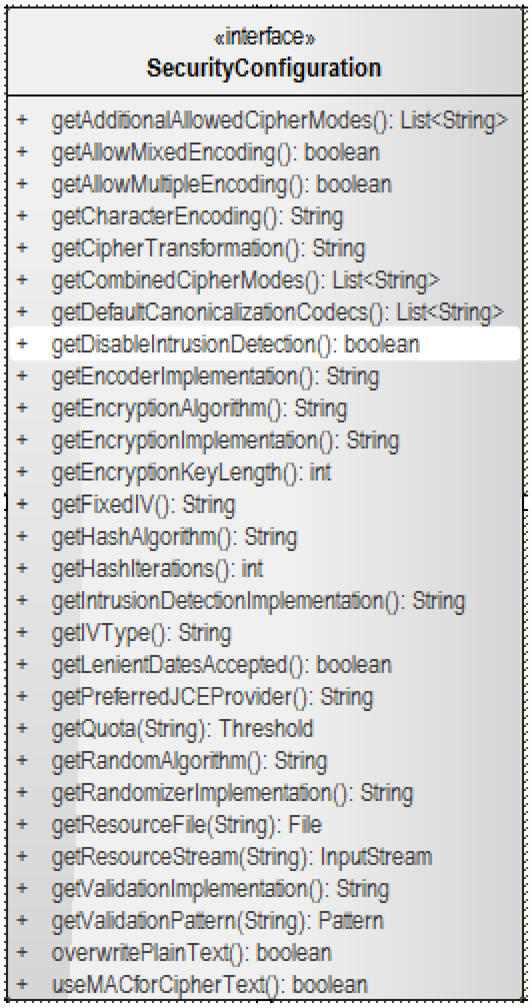
```
//Generate a random session id of desired length..
String sessionId = SecureRandomUtils.getRandomSessionId(50);
Long randomlong = SecureRandomUtils.getRandomLong();
```

com.bosch.security.config - SecurityConfiguration

Features

- Allows to configure various security control attributes.
- Configuration can either be placed in classpath (i.e resources folder) or can be placed in some external path and loaded dynamically by defining the actual path in the JVM argument on the application startup . [**com.bosch.security.resources=D:\config]**
- Configure custom implementation defined for interfaces (related to security controls) as per application demands.
- Add n number of validation patterns for different data types.
- Configure crypto algorithms and related attributes.

Class Diagram


«interface»
SecurityConfiguration

```
+ getAdditionalAllowedCipherModes(): List<String>
+ getAllowMixedEncoding(): boolean
+ getAllowMultipleEncoding(): boolean
+ getCharacterEncoding(): String
+ getCipherTransformation(): String
+ getCombinedCipherModes(): List<String>
+ getDefaultCanonicalizationCodecs(): List<String>
+ getDisableIntrusionDetection(): boolean
+ getEncoderImplementation(): String
+ getEncryptionAlgorithm(): String
+ getEncryptionImplementation(): String
+ getEncryptionKeyLength(): int
+ getFixedIV(): String
+ getHashAlgorithm(): String
+ getHashIterations(): int
+ getIntrusionDetectionImplementation(): String
+ getIVType(): String
+ getLenientDatesAccepted(): boolean
+ getPreferredJCEProvider(): String
+ getQuota(String): Threshold
+ getRandomAlgorithm(): String
+ getRandomizerImplementation(): String
+ getResourceFile(String): File
+ getResourceStream(String): InputStream
+ getValidationImplementation(): String
+ getValidationPattern(String): Pattern
+ overwritePlainText(): boolean
+ useMACforCipherText(): boolean
```

Configuration File (Basic):

```
# If true, then print all the properties set here when they are loaded.
# If false, they are not printed.
Security.printProperties=false

#Data Encoder (Codecs)
Security.Encoder=com.bosch.security.codecs.impl.DefaultEncoder

#Data Encryptor (Encryption/Decryption)
Security.Encryptor=com.bosch.security.crypto.impl.DataEncryptor

#Web intrusion detector.
Security.IntrusionDetector=com.bosch.security.logger.impl.DefaultIntrusionDetector

#Random number/string generator..
Security.Randomizer=com.bosch.security.random.impl.DefaultRandomizer

#Input validator..
Security.Validator=com.bosch.security.validator.impl.DefaultValidator
```

```
#===========================================================================
# Encoder
# Multiple encoding is when a single encoding format is applied multiple times.
Allowing
# multiple encoding is strongly discouraged.
Encoder.AllowMultipleEncoding=false

# Mixed encoding is when multiple different encoding formats are applied, or when
# multiple formats are nested. Allowing multiple encoding is strongly discouraged.
Encoder.AllowMixedEncoding=false

# The default list of codecs to apply when canonicalizing untrusted data.
Encoder.DefaultCodecList=HTMLEntityCodec,PercentCodec,JavaScriptCodec

# Ensure no CRLF injection into logs for forging records
Encoder.LogEncodingRequired=true

#===========================================================================
# Encryption
# Provides the default JCE provider that ESAPI will "prefer" for its symmetric
# encryption and hashing. If left unset, ESAPI will just use your Java VM's current
# preferred JCE provider, which is generally set in the file
# "$JAVA_HOME/jre/lib/security/java.security".
Encryptor.PreferredJCEProvider=BC

# AES is the most widely used and strongest encryption algorithm. This
# should agree with your Encryptor.CipherTransformation property.
Encryptor.EncryptionAlgorithm=AES
Encryptor.CipherTransformation=AES/GCM/NoPadding

# PasswordBasedEncryption(PBE) attributes for the sensitive data stored in the
property files.
Encryptor.PBEEncryptionAlgorithm=PBEWITHSHA256AND128BITAES-CBC-BC
Encryptor.PBEEncryptionJCEProvider=BC

# Comma-separated list of cipher modes that provide *BOTH*
# confidentiality *AND* message authenticity.
Encryptor.cipher_modes.combined_modes=GCM

# Additional cipher modes allowed for  encryption.
Encryptor.cipher_modes.additional_allowed=CBC

# 128-bit is almost always sufficient and appears to be more resistant to
# related key attacks than is 256-bit AES. Use '_' to use default key size
# for cipher algorithms (where it makes sense because the algorithm supports
# a variable key size). Key length must agree to what's provided as the
# cipher transformation
Encryptor.EncryptionKeyLength=256

# If we are using CBC mode by default, it requires an initialization vector (IV).
# (All cipher modes except ECB require an IV.)
# Valid values:  random|fixed|
Encryptor.ChooseIVMethod=random
```

```
# If you choose to use a fixed IV, then you must place a fixed IV here that
# is known to all others who are sharing your secret key. The format should
# be a hex string that is the same length as the cipher block size for the
# cipher algorithm that you are using. The following is an example for AES
# from an AES test vector for AES-128/CBC as described in:
# NIST Special Publication 800-38A (2001 Edition)
# "Recommendation for Block Cipher Modes of Operation".
# (Note that the block size for AES is 16 bytes == 128 bits.)
#
Encryptor.fixedIV=0x000102030405060708090a0b0c0d0e0f

# Whether or not CipherText should use a message authentication code (MAC) with it.
# This prevents an adversary from altering the IV as well as allowing a more
# fool-proof way of determining the decryption failed because of an incorrect
# key being supplied. This refers to the "separate" MAC calculated and stored
# in CipherText, not part of any MAC that is calculated as a result of a
# "combined mode" cipher mode.
# If you are using ESAPI with a FIPS 140-2 cryptographic module, you *must* also
# set this property to false.
Encryptor.CipherText.useMAC=false

# Whether or not the PlainText object may be overwritten and then marked
# eligible for garbage collection. If not set, this is still treated as 'true'.
Encryptor.PlainText.overwrite=true

#HashAlgorithms and other attributes.
Encryptor.HashAlgorithm=SHA-256
Encryptor.HashIterations=1024
Encryptor.RandomAlgorithm=SHA1PRNG
Encryptor.CharacterEncoding=UTF-8
Encryptor.SaltLength=23

# Max allowed cipher text size limit [in bytes] ~ 1KB
Encryptor.MaxAllowedCipherTextSize=2048

#============================================================================
# Validation
#
# The Validator works on regular expressions with defined names. You can define names
# either here, or you may define application specific patterns in a separate file
defined below.
# This allows enterprises to specify both organizational standards as well as
application specific
# validation rules.
#
```

```
Validator.ConfigurationFile=validation.properties
XSS.ConfigurationFile=xss_encoded.properties
XSS.Pattern.encoded=true
```

Validation.properties

```
# The validator does many security checks on input, such as canonicalization
# and whitelist validation. Note that all of these validation rules are applied
*after*
# canonicalization. Double-encoded characters (even with different encodings involved,
# are never allowed.

# Common Validators
Validator.AccountName=^[a-zA-Z0-9]{3,20}$
Validator.SystemCommand=^[a-zA-Z\\-\\/]{1,64}$
Validator.RoleName=^[a-z]{1,20}$
Validator.Redirect=^\\/test.*$
Validator.SafeString=[A-Za-z0-9]{0,1024}$
Validator.Email=^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\\.[a-zA-Z]{2,4}$
Validator.IPAddress=^(?\:(?\:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?\:25[0-5]|2
[0-4][0-9]|[01]?[0-9][0-9]?)$
Validator.URL=^(ht|f)tp(s?)\\\:\\/\\/[0-9a-zA-Z]([-.\\w]*[0-9a-zA-Z])*(\:(0-9)*)*(\\/?
)([a-zA-Z0-9\\-\\.\\?\\,\\\:\\'\\/\\\\\\+\=&amp;%\\$\#_]*)?$
Validator.CreditCard=^(\\d{4}[- ]?){3}\\d{4}$
Validator.SSN=^(?\!000)([0-6]\\d{2}|7([0-6]\\d|7[012]))([
-]?)(?\!00)\\d\\d\\3(?\!0000)\\d{4}$

# Global HTTP Validation Rules
Validator.HTTPScheme=^(http|https)$
Validator.HTTPServerName=^[a-zA-Z0-9_.\\-]*$
Validator.HTTPCookieName=^[a-zA-Z0-9\\-_]{1,32}$
Validator.HTTPCookieValue=^[a-zA-Z0-9\\-\\/+\=_ ]*$
Validator.HTTPHeaderName=^[a-zA-Z0-9\\-_]{1,32}$
Validator.HTTPHeaderValue=^[a-zA-Z0-9()\\-\=\\*\\.\\?;,+\\/\:&_ ]*$
Validator.HTTPServletPath=^[a-zA-Z0-9.\\-\\/_]*$
Validator.HTTPPath=^[a-zA-Z0-9.\\-_]*$
Validator.HTTPURL=^.*$
Validator.HTTPJSESSIONID=^[A-Z0-9]{10,30}$
Validator.HTTPParameterName=^[a-zA-Z0-9_\\-]{1,32}$
Validator.HTTPParameterValue=^[\\p{L}\\p{N}\\p{S}\\p{M}.\\-/+\=_
\!$%*?@&"';\#\:/\\\\\{}\\[\\](),.~'`^<>|(.|\n+\t+\s)®\u2122©]*$
Validator.HTTPContextPath=^/[a-zA-Z0-9.\\-_]*$
Validator.HTTPQueryString=^([a-zA-Z0-9_\\-]{1,32}\=[\\p{L}\\p{N}.\\-/+\=_
\!$*?@&%]*&?)*$
Validator.HTTPURI=^/([a-zA-Z0-9.\\-_]*/?)*$

# Validation of file related input
Validator.FileName=^[a-zA-Z0-9\!@\#$%^&{}\\[\\]()_+\\-\=,.~:'` ]{1,255}$
Validator.DirectoryName=^[a-zA-Z0-9\:/\\\\\!@\#$%^&{}\\[\\]()_+\\-\=,.~'` ]{1,255}$

# Validation of dates.
Validator.AcceptLenientDates=false
```

Dependency Libraries

External Libraries
commons-codec-1.9.jar
slf4j-api-1.7.7.jar
bcprov-jdk16.1.46

Build and Deployment

- Structured as a maven project.
- Package the jar using maven. Execute the following command from the project path
    - mvn clean compile install.
- Copy and export the generated jar **hbs-security-util.jar** from the /target folder. to your classpath.
- Can be exported or added to a class-path as a external library [jar].
- Configuration file is currently included within the JAR. In case, application needs to overwrite this file, then copy the same file in some external path on the server and provide the path asthe JVM argument to your application. Please note, the preference is first provided to the value defined in the JVM argument.
    - **-Dcom.bosch.security.resources=#path**