# Performance of authentication service in SOA

*Kim, Sang Hyun, University of Southern California, shkim9576@gmail.com*

***Abstract*— Service-oriented architecture (SOA) is the software design style that has been rapidly adopted in the industry to build distributed software systems because of its benefits such as maintainability, loose coupling, scalability, etc. While its advantages make SOA popular, authenticating a service or user has been big design concern because authentication service is commonly accessed by most other services in the system if not all and if authentication service becomes bottleneck or unavailable entire system may slow or unavailable as a consequence. Performance of authentication service is more important for the system where a service receives a lot of requests from clients such as Internet of Things. This paper evaluates performance of authentication service in a component level as well as service level with some of commonly used authentication models in SOA.**

***Index Terms*—service oriented architecture, authentication service, performance**

## I. Introduction

### A. What is SOA

Service-oriented architecture (SOA) is software design paradigm that has been adopted broadly in the industry to build distributed software systems. There are no industry standards relating to SOA. However the crux of SOA is a service that has own characteristics.

Below TABLE 1 took the properties of service in SOA from the paper [1].

TABLE 1
SOA Service Properties

| *Property* | *Principle* |
|---|---|
| Self-contained | The service is highly modular and can be independently deployed |
| Distributed component | The service is available over the network and accessible through a name or locator other than the absolute network address |
| Published interface | Users of the service only need to see the interface and can be oblivious to implementation details |
| Stresses interoperability | Service users and providers can use different implementation languages and platforms |
| Discoverable | A special directory service allows the service to be registered, so users can look it up |
| Dynamically bound | A service user does not need to have the service implementation available at build time; the service is located and bound at runtime |

The paper [1] explains SOA that it consists of “service user” – elements that invoke services provided by others - and “service provider” - elements offering services and a service provider can also be a service user of another service.

Typically services refer to web services with REST APIs and hence this paper includes experiments that implements web services with REST APIs for authentication.

Since terms used in SOA may be ambiguous and used differently often, this paper uses terms and definitions in TABLE 2 and some of those (user, server, and client) definitions are from the paper [2].

TABLE 2
Terms

| *Term* | *Definition* |
|---|---|
| User | A human being who uses a program or service |
| Server | A machine processes a service |
| Service | A program runs on a server to handle requests from client |
| Client | A program runs on one machine that sends requests to a remove service |
| Performance | Amount of work accomplished by a system and performance metrics include response time, throughput, and availability. |
| Response time | The total amount of time a service takes to respond to a client request |
| Secret key | A key used for encryption and decryption in symmetric cryptography |
| Public key | A key used to encrypt data in asymmetric cryptography |
| Private key | A key used to decrypt data in asymmetric cryptography |

### B. Authentication service and performance in SOA

According to the paper [3], with the advent of clouding computer, severity and/or existence of SOA vulnerabilities are changed and they bring new security challenges and hence the paper recommends using some form of authentication among services, or between services and the service browser.

The risks in SOA lead to require and implement more secure solutions and often the cost of improving security was loosing performance of the system or investing more hardware and/or human resources to keep comparable performance.

The scope of this paper is evaluating performance of authentication service among those security solutions and this could lead further discussion regarding security requirement, Quality of Service (QoS) requirement by Service Level Agreement (SLA), and its implementation costs that are essential factors to make better business decision.

Like other services in SOA, authentication service is deployed independently and it performs authentication of other services or human users. There are several different ways of authenticating a client and this paper chose some of those mechanisms that are commonly used in the industry today and evaluated their performance.

### C. Why performance is important

According to [8], the number of daily active Facebook users worldwide has been increasing and there are 1.18 billion daily active users for September 2016. Gartner predicts more than 20 billion Internet of Things units by year 2020 [9]. Rapidly increasing the number of requests from clients made high throughput and performance are one of most critical non-functional requirements when a service provider builds a software system including authentication service.

## II. Authentication Model

Authentication means that the process of confirming a claimed identity of either a user or service. There are several factors that can be used to confirming identity and this paper covers below methods.

### *A. Encryption based authentication*

Symmetric and asymmetric cryptography itself can be a method of authentication or major building block of an authentication system. For example, when Kerberos performs client authentication authenticator encrypts a message using the client session key. In case of challenge-response authentication, it also uses the encryption key to build a challenge and only the client who knows same secret key can send a valid response back.

The paper [4] explains encryption based authentication that symmetric method uses a pair of keys, public key for encryption and private key for decryption. The disadvantage of asymmetric authentication is that it has long process time. Symmetric method is based on shared secret key and authentication is performed by proofing the possession of the secret key. The issue of symmetric authentication is the key distribution and key management may not easy. Every update of the key has to be communicated with all participants and this problem and solutions are addressed in [5].

This paper includes experiment that encrypts and decrypts data using AES algorithm with CBC mode of operation for symmetric authentication and RSA for asymmetric authentication to compare their performance.

### *B. Signature based authentication*

Digital signatures can be used to authenticate a user or a service by verifying a signature, typically a hash value, with a signature owner's public key where the signature is generated with a corresponding private key.

For example, X.509 certificate includes information of issuer, public key, subject name, certificate signature, and certificate signature algorithm. A signature included in X.509 certificate is a hash value and hence any change in the data results in a different value. A verifier decrypts this hash value with signer's public key and if it matches it proves origin of source. X.509 certificate also includes certificate authority who binds a public key with an identity therefore it can be used this public key and certificate belong to a specific user or service. In the industry, Internet of Things cloud services often authenticate devices using X.509 client certificate that is presented to the MQTT message broker during TLS handshake.

Pretty Good Privacy (PGP) also supports authentication by a digital signature that determines whether a message was sent by a claimed client.

This paper includes experiment that computes a hash using SAH256 and signing it with RSA algorithm to measure their performance.

### *C. Password based authentication*

The paper [6] explains password based authentication that it is one of the simplest and convenient mechanism. However it is vulnerable to various attacks such as replay attack, guessing attack, modification attack, and stolen verifier attack. The sequence of password authentication is that a client submits a pair of ID and password to the authentication service and authentication service authenticate by comparing it with stored ID/Password pair.

Yet it is weak authentication, lots of servers today use ID and password authentication because it is relatively easy to implement and a user types ID/password through a web browser or a console to have an access to the resources. In case of Internet of Things devices, a device sends a MQTT connect message to MQTT broker and the packet optionally contains 'username' and 'password' fields to support this authentication method.

The experiment in this paper computes a hash value from both a password and a salt before persisting it into local database. Authentication is performed by comparing a stored hash value and computed one by a given password.

### *D. OAuth*

The paper [7] summarizes Open Authorization (OAuth): a user might also want to give a service access to data owned by the user and hosted on some other service without giving the service requiring the access their credentials to the service hosting their data. One possible solution for this problem is the OAuth protocol, which provides a way for the user to grant access to their data hosted on a service.

OAuth emerged from social web and its motivation is allowing a user's information to a third party services without exposing a user's password. OAuth is very popular in the industry today because a service provider may not need to implement user authentication especially managing user's password. Instead, user authentication is performed by OAuth provider. When a user is authenticated OAuth provider redirects back to the client with user access token in a format of JSON Web Token (JWT).

Disadvantage of OAuth is that it requires redirects – once to OAuth provider and one more for callback- and often redirects increase overall user authentication time.

This paper includes experiment that measures End to End OAuth authentication time with several different OAuth providers.

## III. Performance Evaluation Result

### A. Experimental Setup

Below table lists experiment setup used to measure performance.

TABLE 3
EXPERIMENT SETUP

| *Category* | *Subcategory* | *Specification or library* |
|---|---|---|
| Hardware | Processor | 2.8 GHZ Intel Core i7 |
| | Memory | 16 GB 1600 MHz DDR3 |
| | Storage | 1 TB flash storage |
| Network | Upload | 44.92 Mbps |
| | Download | 47.57 Mbps |
| Software | OS | OS X 10.11.2 |
| | Crypto/Sign | Java 1.8.0_66 |
| | Username/Password | Node.js 6.9.1 |
| | | MongoDB 3.2 |
| | | Passport 0.2.1 |
| | | Passport-local 1.0.0 |
| | OAuth | Node.js 6.9.1 |
| | | MongoDB 3.2 |
| | | Passport 0.2.1 |
| | | Passport-facebook 1.0.3 |
| | | Passport-twitter 1.0.2 |
| | | Passport-google-oauth 0.1.5 |
| | | Passport-azure-ad-oauth2 0.1.0 |
| Tool | Apache JMeter | 3.1 |

To measure performance of section B, C, D, E, and F below, tests were executed 1,000 times each and took average values. Java and Node.js applications were used with libraries as listed in TABLE 3 above.

### B. Symmetric vs. Asymmetric encryption

AES CBC mode of operation is used to represent symmetric encryption and RSA is used for asymmetric encryption. As shown in below Fig. 1., RSA encryption time linearly increased by data size while AES encryption time was increased logarithmically.

For instance, AES encryption took 91 microseconds with 1 megabytes data and 198 microseconds with 10 megabytes data. RSA encryption took 734 microseconds with 1 megabytes data and 5,523 microseconds with 10 megabytes data.

This result shows RSA encryption took about 8 times more than AES with 1 megabytes data and performance becomes worse as data size increasing.

See Appendix A and B for Java application source code that used to measure performance.

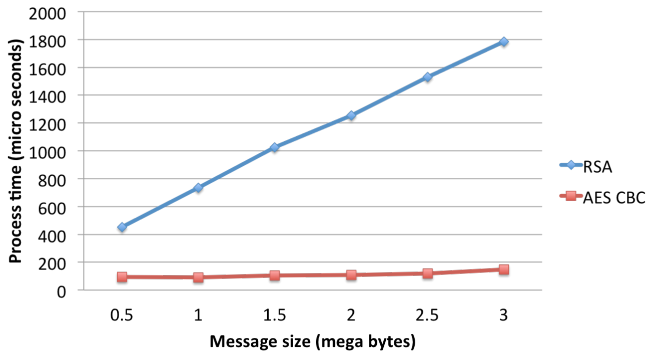

Fig. 1. Performance comparison between symmetric and asymmetric encryption.

TABLE 4
ENCRYPTION PROCESS TIME

| *Data Size (megabytes)* | *RSA (microseconds)* | *AES CBC (microseconds)* |
|---|---|---|
| 0.5 | 453 | 92 |
| 1.0 | 734 | 91 |
| 1.5 | 1,025 | 105 |
| 2.0 | 1,255 | 108 |
| 2.5 | 1,532 | 120 |
| 3.0 | 1,783 | 148 |
| 10.0 | 5,523 | 198 |

### C. Symmetric vs. Asymmetric decryption

Analysis result is almost same as encryption as explained in above section *B. Symmetric vs. Asymmetric encryption* except RSA decryption took much longer than encryption. For instance, RSA decryption took 9,330 microseconds with 1 megabytes data while encryption took 734 microseconds.

See Appendix A and B for Java application source code that used to measure performance.

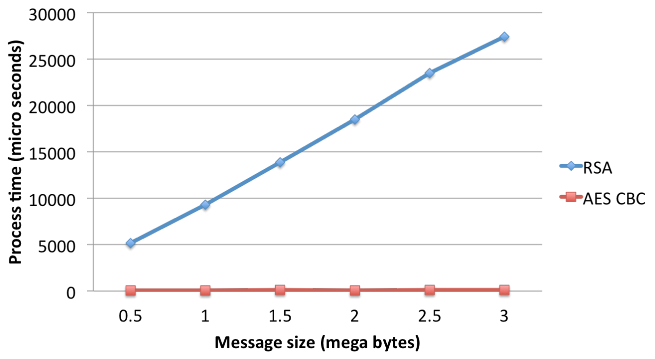

Fig. 2. Performance comparison between symmetric and asymmetric decryption.

TABLE 5
DECRYPTION PROCESS TIME

| *Data Size (megabytes)* | *RSA (microseconds)* | *AES CBC (microseconds)* |
|---|---|---|
| 0.5 | 5,153 | 67 |
| 1.0 | 9,330 | 79 |
| 1.5 | 13,854 | 99 |
| 2.0 | 18,511 | 77 |
| 2.5 | 23,468 | 105 |
| 3.0 | 27,442 | 103 |

## *D. Signing data*

SHA256 is used to compute a hash value and then RSA algorithm is used to generate a signature and process time is in Fig. 3. As shown below, performance took constant time and it was not affected by data size. For instance, signing took 1,065 microseconds with 1 megabytes data and 1,075 microseconds with 10 megabytes data.

See Appendix C for Java application source code that used to measure performance.

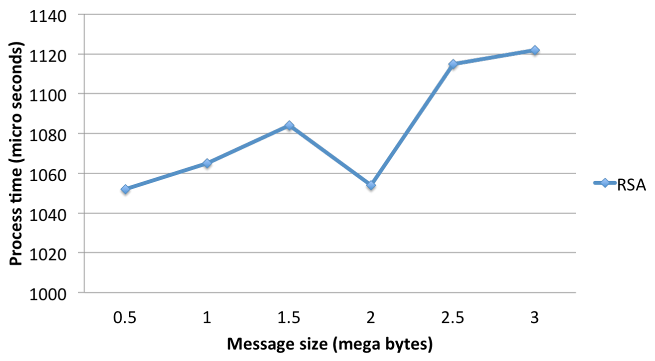

Fig. 3. Performance of RSA SHA256 data signing.

TABLE 6
SIGNING PROCESS TIME

| *Data Size (megabytes)* | *RSA SHA 256 (microseconds)* |
|---|---|
| 0.5 | 1,052 |
| 1.0 | 1,065 |
| 1.5 | 1,084 |
| 2.0 | 1,054 |
| 2.5 | 1,115 |
| 3.0 | 1,122 |
| 10.0 | 1,075 |

## *E. Verifying signature*

Analysis result is almost same as section *D. Signing data* except verification process time is much shorter than signing data. For instance signing 1 megabytes data took 1,065 microseconds while verification took 54 microseconds.

See Appendix C for Java application source code that used to measure performance.

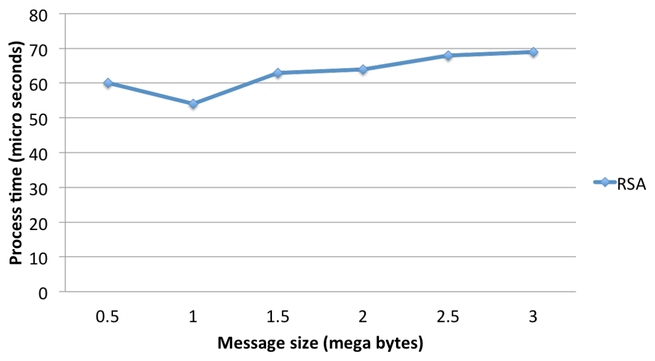

Fig. 4. Performance of RSA SHA256 signature verification.

TABLE 7
VERIFICATION PROCESS TIME

| *Data Size (megabytes)* | *RSA SHA 256 (microseconds)* |
|---|---|
| 0.5 | 60 |
| 1.0 | 54 |
| 1.5 | 63 |
| 2.0 | 64 |
| 2.5 | 68 |
| 3.0 | 69 |

## *F. Password based authentication*

This experiment measures response time and transaction per second (TPS) using JMeter that calls user login API. Node.js web service is used to provide user log in service and MongoDB is used to store user identity and password. Password is hashed with a salt for every password and a hash value is stored instead of a plain password. See Appendix D, E, and F.

Tests were executed 100 times each and took average response time, maximum response time, and minimum response time and its result is shown in below figure and table.

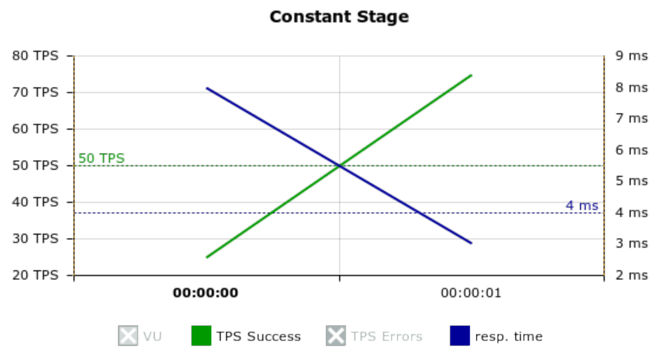

Fig. 5. ID/Password response time

TABLE 8
ID/PASSWORD RESPONSE TIME

| *Type* | *Response time (milliseconds)* |
|---|---|
| Average | 5 |
| Maximum | 82 |
| Minimum | 3 |

## *G. OAuth*

This experiment measures response time and transactions per second (TPS) using JMeter that calls user login API including redirect from each OAuth authentication providers. Node.js web service is used to provide user log in service. Application Source code is based on [10] and available at Appendix D, E, G, H, I, and J.

Tests were executed 100 times each and took average response time, maximum response time, and minimum response time.

Below figures and tables show results from each OAuth authentication providers that is Facebook, Twitter, Google, Azure active directory respectively.

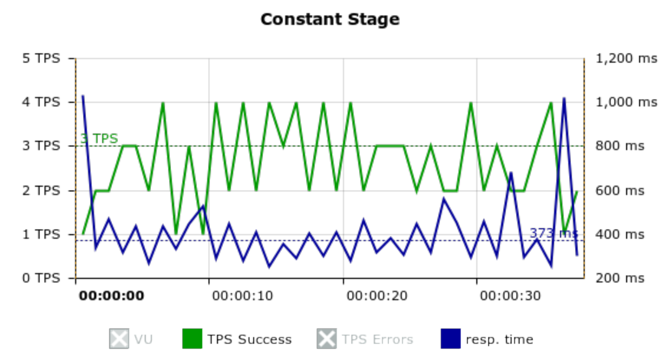

Fig. 6. Facebook OAuth response time

TABLE 9
FACEBOOK RESPONSE TIME

| *Type* | *Response time (milliseconds)* |
|---|---|
| Average | 372 |
| Maximum | 1,065 |
| Minimum | 211 |

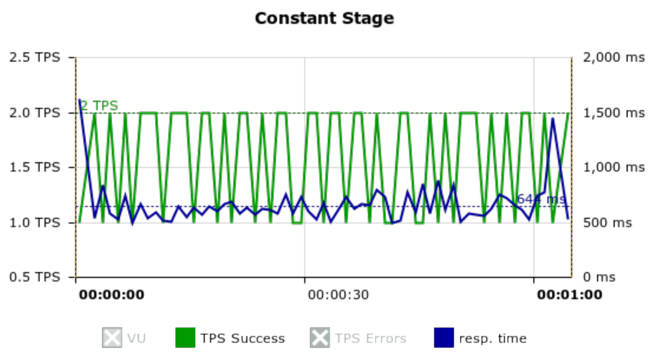

Fig. 7. Twitter OAuth response time

TABLE 10
TWITTER RESPONSE TIME

| *Type* | *Response time (milliseconds)* |
|---|---|
| Average | 644 |
| Maximum | 1,629 |
| Minimum | 456 |

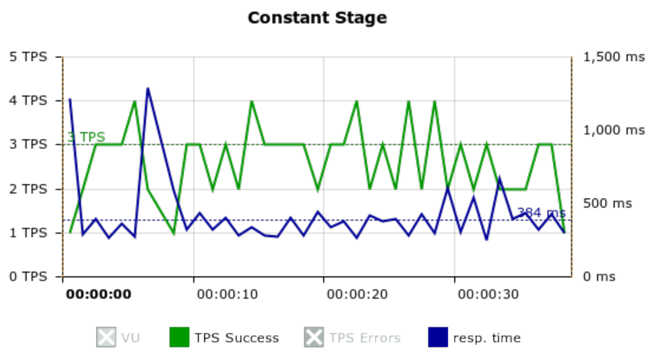

Fig. 8. Google OAuth response time

TABLE 11
GOOGLE RESPONSE TIME

| *Type* | *Response time (milliseconds)* |
|---|---|
| Average | 384 |
| Maximum | 2,328 |
| Minimum | 229 |

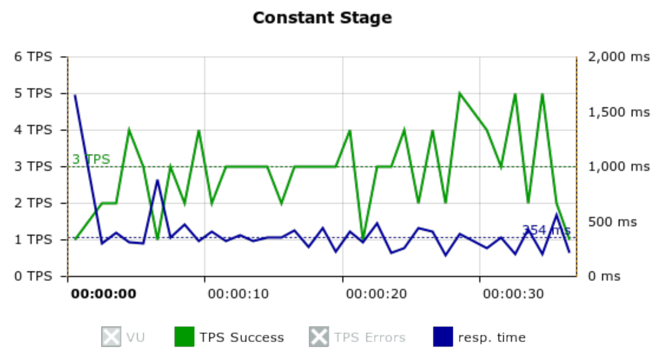

Fig. 9. Azure active directory OAuth response time

TABLE 12
AZURE ACTIVE DIRECTORY RESPONSE TIME

| *Type* | *Response time (milliseconds)* |
|---|---|
| Average | 354 |
| Maximum | 1,661 |
| Minimum | 169 |

Above data shows OAuth user authentication takes around 350 – 400 milliseconds in general and Twitter is slower, 644 milliseconds average, than other providers.

### H. *Summary*

The highlights of above experiment results are as follows.

1) Asymmetric cryptography is significantly slower than symmetric cryptography
2) Asymmetric decryption is significantly slower than Asymmetric encryption
3) Process time of asymmetric cryptography increases linearly by data size: O(n)
4) Process time of symmetric cryptography increases logarithmically by data size: O(logn)
5) Signing data is significantly slower than signature verification
6) Process time of Signing and Verification is constant by data size: O(1)
7) Asymmetric decryption is expensive in terms of performance
8) OAuth is expensive in terms of performance

## IV. CONCLUSION

SOA is widely used software system design that is composed of several services. Due to vulnerabilities in SOA, authentication service has been a required component and its performance is one of most important requirements because of increasing network traffics.

There are several different ways that authentication service in SOA can perform to authenticate a client: either a user or other services. Each method provides different levels of effectiveness against certain attacks such as replay attack and guessing attack and cost of implementing authentication services are also vary. Typically strong authentication cost more than weak authentication and one of major factors of cost evaluation is service performance.

Service providers could make better business decision and requirements for the system if they know precisely what should be protected and what is the performance cost of each implementation.

Therefore, this paper contains performance evaluation with several different authentication mechanisms and as result password based authentication is confirmed fast and easy to implement while it is weak authentication. Service provider may prefer OAuth to avoid cost of implementing authentication service while it is expensive in terms of performance. In case of encryption based authentication asymmetric decryption is expensive that typically performed by authentication service. Signature based authentication is relatively reasonable mechanism in terms of its balance between security and performance.

Recommended further research is evaluating performance for not only other authentication models that are not covered in this paper but also authorization service and ID management service. As the number of users is increasing fast, finding a user from database and applying authorization policies in a very short time is another challenge in SOA today. Another recommendation is building a performance test framework. Since performance measurement result is affected by various variables including hardware specification, network speed, and amount of loads to a service, having accurate measurement tool is most important requirement for correct performance evaluation.

As technology evolving fast, continuously updating evaluation result for performance of computer security systems may require while some basic building blocks discussed in this paper remain valid relatively long time.

## APPENDIX

### A. *Java application - AES CBC cryptography*

```
import com.google.common.base.Stopwatch;

import org.apache.commons.codec.Charsets;
import org.apache.commons.codec.binary.Base64;

import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.util.concurrent.TimeUnit;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class AESTest {

   private static final String ALPHA_NUMERIC_STRING =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
   private static final int NUM_LOOP = 1000;
   private static final int MSG_LENGTH = 1024;
   private static final int KEY_SIZE_BYTES = 16;
   private static final int IV_SIZE_BYTES = 16;
   private static final String ALGORITHM = "AES";
   private static final String TRANSFORMATION =
"AES/CBC/PKCS5PADDING";

   public static void main(String[] args) throws NoSuchPaddingException,
InvalidAlgorithmParameterException, NoSuchAlgorithmException,
         IllegalBlockSizeException, BadPaddingException,
InvalidKeyException {
      String key = generateRandomAlphaNumeric(KEY_SIZE_BYTES);
      String initVector = generateRandomAlphaNumeric(IV_SIZE_BYTES);
      byte[] data =
generateRandomAlphaNumeric(MSG_LENGTH).getBytes();

      if (isValid(key, initVector, data)) {
         encryptMeasure(key, initVector, data);
         decryptMeasure(key, initVector, data);
      }

   }

   private static void encryptMeasure(String key, String initVector, byte[]
data) throws NoSuchPaddingException, InvalidKeyException,
NoSuchAlgorithmException, IllegalBlockSizeException,
BadPaddingException, InvalidAlgorithmParameterException {
      Stopwatch stopwatch = Stopwatch.createStarted();
      for (int i = 0; i < NUM_LOOP; i++) {
         encrypt(key, initVector, data);
      }
      stopwatch.stop();

      long totalTime = stopwatch.elapsed(TimeUnit.MICROSECONDS);
      System.out.println("avg encrypt took (micro seconds): " + (totalTime /
NUM_LOOP));
   }

   private static void decryptMeasure(String key, String initVector, byte[]
data) throws BadPaddingException, InvalidKeyException,
NoSuchAlgorithmException,
         IllegalBlockSizeException, NoSuchPaddingException,
InvalidAlgorithmParameterException {
      byte[] encrypted = encrypt(key, initVector, data);

      Stopwatch stopwatch = Stopwatch.createStarted();
      for (int i = 0; i < NUM_LOOP; i++) {
         decrypt(key, initVector, encrypted);
      }
      stopwatch.stop();

      long totalTime = stopwatch.elapsed(TimeUnit.MICROSECONDS);
      System.out.println("avg decrypt took (micro seconds): " + (totalTime /
NUM_LOOP));
   }

   private static boolean isValid(String key, String initVector, byte[] data)
throws NoSuchPaddingException, InvalidKeyException,
NoSuchAlgorithmException,
         IllegalBlockSizeException, BadPaddingException,
InvalidAlgorithmParameterException {
      byte[] encrypted = encrypt(key, initVector, data);
      byte[] decrypted = decrypt(key, initVector, encrypted);

      return new String(data).equals(new String(decrypted));
   }

   public static byte[] encrypt(String key, String initVector, byte[] data)
throws InvalidAlgorithmParameterException, InvalidKeyException,
         NoSuchPaddingException, NoSuchAlgorithmException,
BadPaddingException, IllegalBlockSizeException {
      IvParameterSpec iv = new
IvParameterSpec(initVector.getBytes(Charsets.UTF_8));
      SecretKeySpec skeySpec = new
SecretKeySpec(key.getBytes(Charsets.UTF_8), ALGORITHM);

      Cipher cipher = Cipher.getInstance(TRANSFORMATION);
      cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);
```

```
        byte[] encrypted = cipher.doFinal(data);
        return Base64.encodeBase64(encrypted);
    }

    public static byte[] decrypt(String key, String initVector, byte[] encrypted)
throws BadPaddingException, IllegalBlockSizeException,
            InvalidAlgorithmParameterException, InvalidKeyException,
NoSuchPaddingException, NoSuchAlgorithmException {

        IvParameterSpec iv = new
IvParameterSpec(initVector.getBytes(Charsets.UTF_8));
        SecretKeySpec skeySpec = new
SecretKeySpec(key.getBytes(Charsets.UTF_8), ALGORITHM);

        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);

        return cipher.doFinal(Base64.decodeBase64(encrypted));
    }

    public static String generateRandomAlphaNumeric(int count) {
        StringBuilder builder = new StringBuilder();
        while (count-- != 0) {
            int character = (int) (Math.random() *
ALPHA_NUMERIC_STRING.length());
            builder.append(ALPHA_NUMERIC_STRING.charAt(character));
        }
        return builder.toString();
    }

}
```

## B. *Java application - RSA cryptography*

```
// Code is based on the source from http://coding.westreicher.org/?p=23
import com.google.common.base.Stopwatch;

import org.apache.commons.codec.binary.Hex;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.concurrent.TimeUnit;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;

public class RSATest2 {

    private static final String ALPHA_NUMERIC_STRING =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    private static final int MSG_LENGTH = 1024 * 10;
    private static final int NUM_LOOP = 1000;

    private static KeyPair keypair;
    private static Cipher cipher;

    public static void main(String[] args) throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(1024);
        keypair = kpg.generateKeyPair();
        cipher = Cipher.getInstance("RSA");

        String orgText = generateRandomAlphaNumeric(MSG_LENGTH);
        String cipherText = null;

        Stopwatch stopwatch = Stopwatch.createStarted();
        for (int i = 0; i < NUM_LOOP; i++) {
            cipherText = encrypt(orgText);
        }
        stopwatch.stop();
        System.out.println("avg encrypt took (micro seconds): " +
(stopwatch.elapsed(TimeUnit.MICROSECONDS)/NUM_LOOP));

        stopwatch.reset();
        stopwatch.start();
        for (int i = 0; i < NUM_LOOP; i++) {
            decrypt(cipherText);
        }
        stopwatch.stop();
        System.out.println("avg decrypt took (micro seconds): " +
(stopwatch.elapsed(TimeUnit.MICROSECONDS)/NUM_LOOP));
    }

    public static String generateRandomAlphaNumeric(int count) {
        StringBuilder builder = new StringBuilder();
        while (count-- != 0) {
            int character = (int) (Math.random() *
ALPHA_NUMERIC_STRING.length());
            builder.append(ALPHA_NUMERIC_STRING.charAt(character));
        }
        return builder.toString();
    }

    public static String encrypt(String plaintext) throws Exception {
        cipher.init(Cipher.ENCRYPT_MODE, keypair.getPublic());
        byte[] bytes = plaintext.getBytes("UTF-8");

        byte[] encrypted = blockCipher(bytes, Cipher.ENCRYPT_MODE);

        char[] encryptedTranspherable = Hex.encodeHex(encrypted);
        return new String(encryptedTranspherable);
    }

    public static String decrypt(String encrypted) throws Exception {
        cipher.init(Cipher.DECRYPT_MODE, keypair.getPrivate());
        byte[] bts = Hex.decodeHex(encrypted.toCharArray());

        byte[] decrypted = blockCipher(bts, Cipher.DECRYPT_MODE);

        return new String(decrypted, "UTF-8");
    }

    private static byte[] blockCipher(byte[] bytes, int mode) throws
IllegalBlockSizeException, BadPaddingException, BadPaddingException,
            IllegalBlockSizeException {
        // string initialize 2 buffers.
        // scrambled will hold intermediate results
        byte[] scrambled = new byte[0];

        // toReturn will hold the total result
        byte[] toReturn = new byte[0];
        // if we encrypt we use 100 byte long blocks. Decryption requires 128 byte
long blocks (because of RSA)
        int length = (mode == Cipher.ENCRYPT_MODE) ? 100 : 128;

        // another buffer. this one will hold the bytes that have to be modified in
this step
        byte[] buffer = new byte[length];

        for (int i = 0; i < bytes.length; i++) {

            // if we filled our buffer array we have our block ready for de- or
encryption
            if ((i > 0) && (i % length == 0)) {
                //execute the operation
                scrambled = cipher.doFinal(buffer);
                // add the result to our total result.
                toReturn = append(toReturn, scrambled);
                // here we calculate the length of the next buffer required
                int newlength = length;

                // if newlength would be longer than remaining bytes in the bytes
array we shorten it.
                if (i + length > bytes.length) {
                    newlength = bytes.length - i;
                }
                // clean the buffer array
                buffer = new byte[newlength];
            }
            // copy byte into our buffer.
            buffer[i % length] = bytes[i];
        }

        // this step is needed if we had a trailing buffer. should only happen when
encrypting.
        // example: we encrypt 110 bytes. 100 bytes per run means we "forgot" the
last 10 bytes. they are in the buffer array
        scrambled = cipher.doFinal(buffer);

        // final step before we can return the modified data.
        toReturn = append(toReturn, scrambled);

        return toReturn;
    }

    private static byte[] append(byte[] prefix, byte[] suffix) {
        byte[] toReturn = new byte[prefix.length + suffix.length];
        for (int i = 0; i < prefix.length; i++) {
            toReturn[i] = prefix[i];
        }
        for (int i = 0; i < suffix.length; i++) {
            toReturn[i + prefix.length] = suffix[i];
        }
        return toReturn;
    }
}
```

## C. Java application - signing and verification

```
package usc.security;

import com.google.common.base.Stopwatch;

import java.security.InvalidKeyException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;
import java.util.concurrent.TimeUnit;

public class SignatureKeyPair {

    private static final String ALPHA_NUMERIC_STRING =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    private static final int MSG_SIZE = 1024;
    private static final int NUM_LOOP = 1000;
    private static final int KEY_SIZE = 1024;
    private static final String ALGORITHM = "RSA";
    private static final String SIGN_ALGORITHM = "SHA256withRSA";

    public static void main(String[] args) throws SignatureException,
InvalidKeyException, NoSuchAlgorithmException {
        KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance(ALGORITHM);
        keyPairGenerator.initialize(KEY_SIZE);

        KeyPair keyPair = keyPairGenerator.generateKeyPair();
        Signature signer = Signature.getInstance(SIGN_ALGORITHM);
        PrivateKey privateKey = keyPair.getPrivate();
        PublicKey publicKey = keyPair.getPublic();
        byte[] data = generateRandomAlphaNumeric(MSG_SIZE).getBytes();

        if (isValid(signer, publicKey, privateKey, data)) {
            byte[] signedData = signMeasure(signer, privateKey, data);
            verifyMeasure(signer, publicKey, data, signedData);
        }

    }
```

```
    private static boolean isValid(Signature signer, PublicKey publicKey,
PrivateKey privateKey, byte[] data) throws NoSuchAlgorithmException,
            InvalidKeyException,
            SignatureException {
        byte[] signedData = sign(signer, privateKey, data);
        return verify(signer, publicKey, data, signedData);
    }

    private static byte[] signMeasure(Signature signer, PrivateKey privateKey,
byte[] data) throws NoSuchAlgorithmException, InvalidKeyException,
            SignatureException {
        byte[] signedData = new byte[0];
        Stopwatch stopwatch = Stopwatch.createStarted();
        for (int i = 0; i < NUM_LOOP; i++) {
            signedData = sign(signer, privateKey, data);
        }
        stopwatch.stop();
        System.out.println("avg sign took (micro seconds): " +
stopwatch.elapsed(TimeUnit.MICROSECONDS) / NUM_LOOP);
        return signedData;
    }

    private static void verifyMeasure(Signature signer, PublicKey publicKey,
byte[] data, byte[] signedData) throws SignatureException,
InvalidKeyException {
        Stopwatch stopwatch = Stopwatch.createStarted();
        for (int i = 0; i < NUM_LOOP; i++) {
            verify(signer, publicKey, data, signedData);
        }
        stopwatch.stop();
        System.out.println("avg verify took (micro seconds): " +
stopwatch.elapsed(TimeUnit.MICROSECONDS) / NUM_LOOP);
    }

    private static byte[] sign(Signature signer, PrivateKey privateKey, byte[]
data) throws NoSuchAlgorithmException, InvalidKeyException,
            SignatureException {
        signer.initSign(privateKey);
        signer.update(data);
        return signer.sign();
    }

    private static boolean verify(Signature signer, PublicKey publicKey,
byte[] data, byte[] signedData) throws InvalidKeyException,
SignatureException {
        signer.initVerify(publicKey);
        signer.update(data);
        return signer.verify(signedData);
    }

    public static String generateRandomAlphaNumeric(int count) {
        StringBuilder builder = new StringBuilder();
        while (count-- != 0) {
            int character = (int) (Math.random() *
ALPHA_NUMERIC_STRING.length());
            builder.append(ALPHA_NUMERIC_STRING.charAt(character));
        }
        return builder.toString();
    }
}
```

## D. Node.js application - passport initialization

```
    // Node.js app code is based on
https://www.safaribooksonline.com/library/view/mean-web-development/9781
783983285/
var passport = require('passport'),
    mongoose = require('mongoose');

module.exports = function() {
    var User = mongoose.model('User');

    passport.serializeUser(function(user, done) {
        done(null, user.id);
```

```
   });

   passport.deserializeUser(function(id, done) {
      User.findOne({
         _id: id
      }, '-password -salt', function(err, user) {
         done(err, user);
      });
   });

   require('./strategies/local.js')();
   require('./strategies/facebook.js')();
   require('./strategies/twitter.js')();
   require('./strategies/google.js')();
   require('./strategies/windows.js')();
};
```

## E. *Node.js application - passport routes*

```
var users = require('../../app/controllers/users.server.controller'),
   passport = require('passport');

module.exports = function(app) {

   // Local sign up/in/out resource
   app.route('/signup')
      .get(users.renderSignup)
      .post(users.signup);

   app.route('/signin')
      .get(users.renderSignin)
      .post(passport.authenticate('local', {
         successRedirect: '/',
         failureRedirect: '/#!/signin',
         failureFlash: true
      }));

   app.get('/signout', users.signout);


   // facebook Oauth resource
   app.get('/oauth/facebook', passport.authenticate('facebook', {
      failureRedirect: '/signin'
   }));
   app.get('/oauth/facebook/callback', passport.authenticate('facebook', {
      failureRedirect: '/signin',
      successRedirect: '/'
   }));


   // twitter Oauth resource
   app.get('/oauth/twitter', passport.authenticate('twitter', {
      failureRedirect: '/signin'
   }));

   app.get('/oauth/twitter/callback', passport.authenticate('twitter', {
      failureRedirect: '/signin',
      successRedirect: '/'
   }));

   // google Oauth resource
   app.get('/oauth/google', passport.authenticate('google', {
      failureRedirect: '/signin',
      scope: [
         'https://www.googleapis.com/auth/userinfo.profile',
         'https://www.googleapis.com/auth/userinfo.email'
      ],
   }));

   app.get('/oauth/google/callback', passport.authenticate('google', {
      failureRedirect: '/signin',
      successRedirect: '/'
   }));
```

```
   // windows active directory
   app.get('/oauth/windows', passport.authenticate('azure_ad_oauth2', {
      failureRedirect: '/#!/signin'
   }));

   app.get('/oauth/windows/callback',
passport.authenticate('azure_ad_oauth2', {
      failureRedirect: '/#!/signin',
      successRedirect: '/#!/'
   }));

};
```

## F. *Node.js application - passport local ID/Password*

```
// local passport strategy - using username/passport authentication

var passport = require('passport'),
   LocalStrategy = require('passport-local').Strategy,
   User = require('mongoose').model('User');

module.exports = function() {
   passport.use(new LocalStrategy(function(username, password, done) {
      User.findOne({
         username: username
      }, function(err, user) {
         if (err) {
            return done(err);
         }

         if (!user) {
            return done(null, false, {
               message: 'Unknown user'
            });
         }
         if (!user.authenticate(password)) {
            return done(null, false, {
               message: 'Invalid password'
            });
         }

         return done(null, user);
      });
   }));
};
```

## G. *Node.js application - passport Facebook OAuth*

```
var passport = require('passport'),
   url = require('url'),
   FacebookStrategy = require('passport-facebook').Strategy,
   config = require('../config');
var User = require('mongoose').model('User');

module.exports = function() {
   passport.use(new FacebookStrategy({
         clientID: config.facebook.clientID,
         clientSecret: config.facebook.clientSecret,
         callbackURL: config.facebook.callbackURL,
         profileFields: config.facebook.profileFields,
         scope: config.facebook.scope,
         passReqToCallback: true
      },
      function(req, accessToken, refreshToken, profile, done) {
         var providerData = profile._json;
         providerData.accessToken = accessToken;
         providerData.refreshToken = refreshToken;

         User.findOne({
            username: profile.id
         }, function(err, user) {
            if (err) {
                return done(err);
            }

            if (!user) {
                var newUser = new User();
                newUser.firstName = profile.name.givenName;
                newUser.lastName = profile.name.familyName;
                newUser.email = profile.emails[0].value;
                newUser.username = profile.id;
                newUser.provider = 'facebook';

                newUser.save(function (err) {
                    if (err) {
                        return done(err);
                    }

                    return done(null, newUser);
                });
            } else {
                return done(null, user);

            }
        });

    }));
};
```

## H. Node.js application - passport Google OAuth

```
var passport = require('passport'),
    url = require('url'),
    GoogleStrategy = require('passport-google-oauth').OAuth2Strategy,
    config = require('../config');
var User = require('mongoose').model('User');

module.exports = function() {
    passport.use(new GoogleStrategy({
            clientID: config.google.clientID,
            clientSecret: config.google.clientSecret,
            callbackURL: config.google.callbackURL,
            passReqToCallback: true
        },
        function(req, accessToken, refreshToken, profile, done) {
            var providerData = profile._json;
            providerData.accessToken = accessToken;
            providerData.refreshToken = refreshToken;

            User.findOne({
                username: profile.id
            }, function(err, user) {
                if (err) {
                    return done(err);
                }

                if (!user) {
                    var newUser = new User();
                    newUser.firstName = profile.name.givenName;
                    newUser.lastName = profile.name.familyName;
                    newUser.email = profile.emails[0].value;
                    newUser.username = profile.id;
                    newUser.provider = 'google';

                    newUser.save(function (err) {
                        if (err) {
                            return done(err);
                        }

                        return done(null, newUser);
                    });
                } else {
                    return done(null, user);

                }
            });
        }));
};
```

## I. Node.js application - passport Twitter OAuth

```
var passport = require('passport'),
    url = require('url'),
    TwitterStrategy = require('passport-twitter').Strategy,
    config = require('../config');
var User = require('mongoose').model('User');

module.exports = function() {
    passport.use(new TwitterStrategy({
            consumerKey: config.twitter.clientID,
            consumerSecret: config.twitter.clientSecret,
            callbackURL: config.twitter.callbackURL,
            passReqToCallback: true
        },
        function(req, token, tokenSecret, profile, done) {
            var providerData = profile._json;
            providerData.token = token;
            providerData.tokenSecret = tokenSecret;

            User.findOne({
                username: profile.id
            }, function(err, user) {
                if (err) {
                    return done(err);
                }

                if (!user) {
                    var newUser = new User();
                    newUser.firstName = profile.displayName;
                    newUser.lastName = '';
                    newUser.email = 'default@gmail.com';
                    newUser.username = profile.id;
                    newUser.provider = 'twitter';

                    newUser.save(function (err) {
                        if (err) {
                            return done(err);
                        }

                        return done(null, newUser);
                    });
                } else {
                    return done(null, user);

                }
            });
        }));
};
```

## J. Node.js application - passport Azure Active Directory OAuth

```
var passport = require('passport');
var AzureAdOAuth2Strategy = require('passport-azure-ad-oauth2');
var User = require('mongoose').model('User');
var config = require('../config');
var jwt = require('jsonwebtoken');

module.exports = function () {
    passport.use(new AzureAdOAuth2Strategy({
            clientID: config.windows.clientID,
            clientSecret: config.windows.clientSecret,
            callbackURL: config.windows.callbackURL,
            resource: '00000002-0000-0000-c000-000000000000',
            tenant: 'shkim9576gmail.onmicrosoft.com'
        },
        function (accessToken, refresh_token, params, profile, done) {
            var token = jwt.decode(params.id_token);
        User.findOne({
            username: token.oid
        }, function(err, user) {
            if (err) {
                return done(err);
            }

            if (!user) {
                var newUser = new User();
                newUser.firstName = token.given_name;
                newUser.lastName = token.family_name;
                newUser.email = token.email;
                newUser.username = token.oid;
                newUser.provider = "windows";

                newUser.save(function (err) {
                    if (err) {
                        return done(err);
                    }

                    return done(null, newUser);
                });
            } else {
                return done(null, user);

            }
        });

    }));
};
```

## REFERENCES

[1] *Bianco, Philip, Rick Kotermanski, and Paulo F. Merson. "Evaluating a service-oriented architecture." (2007).*

[2] *Steiner, Jennifer G., B. Clifford Neuman, and Jeffrey I. Schiller. "Kerberos: An Authentication Service for Open Network Systems." USENIX Winter. 1988.*

[3] *Abuhussein, Abdullah, Harkeerat Bedi, and Sajjan Shiva. "Exploring Security and Privacy Risks of SoA Solutions Deployed on the Cloud." Proceedings of the International Conference on Grid Computing and Applications (GCA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014.*

[4] Feldhofer, Martin, Sandra Dominikus, and Johannes Wolkerstorfer. "Strong authentication for RFID systems using the AES algorithm." *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer Berlin Heidelberg, 2004.

[5] Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[6] Liao, I-En, Cheng-Chi Lee, and Min-Shiang Hwang. "A password authentication scheme over insecure networks." *Journal of Computer and System Sciences* 72.4 (2006): 727-740.

[7] Kaila, Pauli. "Oauth and openid 2.0." *From End-to-End to Trust-to-Trust* 18 (2008): 18-22.

[8] https://www.statista.com/statistics/346167/facebook-global-dau/

[9] http://www.gartner.com/newsroom/id/3165317

[10] Haviv, Amos Q. *MEAN Web Development*. Packt Publishing Ltd, 2014.