

4. Java Security

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 1

This topic is partly based on chapter 9 (Security) of this textbook: *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature, Prentice Hall International*

The book chapter is also available online at
<http://www.informit.com/articles/article.aspx?p=1187967>

Content

- Java Cryptography Architecture ([JCA](#))
 - How to carry out basically any cryptographic operation in Java?
- Java Secure Sockets Extension ([JSSE](#))
 - How to implement SSL / TLS protected communication channels in Java?

Goals

- You know which [cryptographic algorithms](#) are offered by Java
- You can [use these algorithms](#) to implement cryptography-based security functions in your programs in an efficient and secure way
- You know the functionality offered by the [Java Secure Sockets Extension \(JSSE\)](#)
- You can [secure networked applications](#) using JSSE
- You can use the [keytool](#) to create key pairs and certificates

Java Cryptography Architecture (JCA)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 4

- Java provides functionality to perform a variety of [cryptographic operations](#), including (with examples):
 - Message digests (MD5, SHA-1, SHA-256) message auth. codes (HMAC)
 - Secret and public key cryptography (AES, 3DES, RC4, RSA, ECC)
 - Diffie-Hellman key exchange
 - Pseudo random number generation
- The functions are provided by two Java library components
 - [Java Cryptography Architecture \(JCA\)](#): since Java 1.1
 - [Java Cryptography Extension \(JCE\)](#): optional since 1.2, official since 1.4
- This separation has mainly to do with the [restrictive US crypto export regulations before 2000](#)
 - "Critical" functions such as strong encryption are part of JCE and were not officially part of Java before version 1.4
 - Today, this distinction is no longer relevant and JCE is considered as an integral part of JCA

- JCA uses a “provider”-based architecture
 - This means that the actual implementations of cryptographic algorithms can be provided from different providers in a plug-in manner
 - The providers are called **Cryptographic Service Providers** (CSP)
- When creating a crypto object, e.g. one of type `MessageDigest`, one can **optionally specify the provider**

```
md = MessageDigest.getInstance("MD5");
md = MessageDigest.getInstance("MD5", "ProviderC");
```
- This was especially relevant before Java 1.4, when functions such as symmetric encryption had to be provided by a third-party
- One of the most prominent crypto providers is the **Legion of the Bouncy Castle** (<http://www.bouncycastle.org>)
 - But today, one usually uses the built-in implementations included in Java

- The CSPs – also the standard ones included in Java – must be listed in \${java.home}/jre/lib/security/java.security:

```
# List of providers and their preference orders (see above):
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
...
...
```

- Example: SunJCE is the standard provider of symmetric encryption algorithms (e.g. AES)
- When creating a crypto object without specifying a provider, the highest-priority provider that implements the function is used
- Additionally used providers should be included as follows:
 - Add an entry to the list (with the desired priority)
 - Add the crypto library provided by the CSP in the standard extensions directory \${java.home}/jre/lib/ext

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 7

SunJCE

SunJCE is the standard Java CSP and was added “officially” to Java with version 1.4. It contains cryptographic functions such as encryption, that could not simply be included in Java due to US crypto export regulation until 2000. The corresponding library (sunjce_provider.jar) is included in the standard extensions directory (\${java.home}/lib/ext) – the same place where you would add other CSPs.

- The API provided by JCA is extensive and contains many classes and methods
- We can only provide here a first insight and look at the following:
 - Computing a message digest with SHA-1
 - Generating secret keys and public key pair
 - Symmetric encryption with AES
 - Public key encryption with RSA
- Remember Kingdom 3 “Security Features”?
 - Even when using the predefined security functions of a programming language, it's still easy to make mistakes, e.g. using short key lengths, reusing initialization vectors, or poorly initializing random number generators
 - Therefore, make sure to study the API documentation of the JCA classes and use the appropriate classes and methods to solve your problem
 - The official JCA reference guide provides valuable information as well

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 8

JCA Reference Guide

<http://download.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html>

Computing a Message Digest

Computing a message digest in Java requires the following steps:

- Create the necessary **MessageDigest object**

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

- **MessageDigest** is a factory class to create an object that can compute a specific message digest
- That object is itself a subclass of **MessageDigest**
- Use the **update method** to feed individual bytes or byte arrays into the **MessageDigest** object

```
InputStream in = ...;
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte) ch);
```

```
byte[] bytes = ...;
alg.update(bytes);
```

- Use the **digest method** to compute the hash

```
byte[] hash = alg.digest();
```

- This method computes the **hash of the byte array b using the hash algorithm alg**
 - The output is returned as a “pretty” hex-string: F2 37 9A...

```
public String computeDigest(byte[] b, String alg) {
    try {
        MessageDigest md = MessageDigest.getInstance(alg);
        md.update(b);
        byte[] hash = md.digest();
        String d = "";
        for (int i = 0; i < hash.length; i++) {
            int v = hash[i] & 0xFF;
            if (v < 16) d += "0";
            d += Integer.toHexString(v, 16).toUpperCase() + " ";
        }
        return d;
    } catch (NoSuchAlgorithmException e) {
        return "" + e;
    }
}
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 10

MessageDigest Example

The example is based on the MessageDigestTest example in *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*

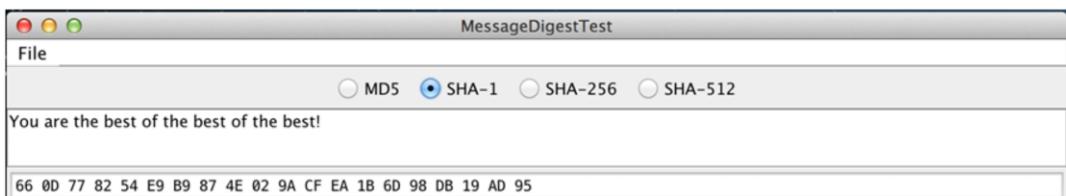
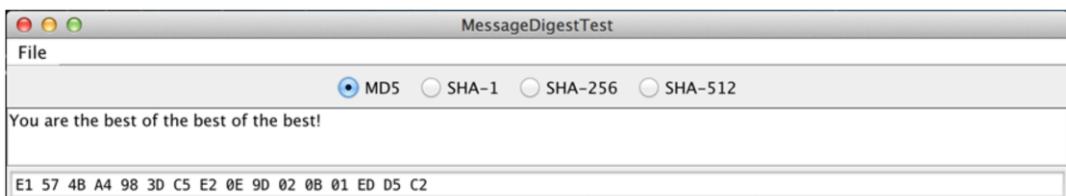
You can get the full source code at <http://www.horstmann.com/corejava.html>

```
int v = hash[i] & 0xFF;
```

byte-values in Java are signed, so they are interpreted as numerical values between -128 and 127. However, when printing the byte-values, we want them to show the actual values between 0 and 255 (00 – FF in hex) that is stored in the byte, i.e. we want the byte-value to be interpreted as an unsigned number. Therefore, we first convert the signed byte value into an int-value between 0 and 255. This can easily done using the line above:

- 0xFF corresponds to the int-Value 00000000 00000000 00000000 11111111
- hash[i] & 0xFF (& is bitwise AND) will first expand hash[i] to an int (which has the same value between -128 and 127 as the byte, but is now 4 bytes long) and then set the first 24 bits to zero.
- If the number was already between 0 and 127, this changes nothing.
- But if the number was between -128 and -1, this converts this value to the correct unsigned number we are expecting, e.g. a value -72 is converted to (256 – 72) = 184. For details, you have to study how the two's-complement system exactly works.

Running the Message Digest Example



Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 11

- Just like with `MessageDigest`, there's a `Cipher factory class` that creates an object (of a `Cipher` subclass) to perform the cryptographic operation

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

- `algorithmName` specifies the `algorithm, cipher mode and padding scheme`, for example:
 - AES/CBC/PKCS5Padding: AES in CBC mode with PKCS5Padding
 - DES: DES using the default cipher mode (ECB) and the default padding scheme (PKCS5Padding)
 - RC4: RC4 stream cipher (which has neither mode nor padding)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 12

PKCS5Padding

A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security Inc. ([ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf](http://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf)). In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if L is the last (incomplete) block, then it is padded as follows (we assume a block length of 8 bytes here):

L 01	if <code>length(L) = 7</code>
L 02 02	if <code>length(L) = 6</code>
L 03 03 03	if <code>length(L) = 5</code>
...	
L 07 07 07 07 07 07 07	if <code>length(L) = 1</code>

Finally, if the length of the input is actually divisible by 8, then one block

08 08 08 08 08 08 08 08

is appended to the input and encrypted. For decryption, the very last byte of the plaintext is a count of the padding characters to discard.

Source: Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature

Algorithms, Modes and Padding Schemes

For more details about supported cipher, modes and padding schemes, refer to
<http://download.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html>

- Once the Cipher object has been created, it must be **initialized** by setting the key and the encryption/decryption mode:

```
int mode = ...;
Key key = ...;
cipher.init(mode, key);
```

- There are 4 modes:
 - Cipher.**ENCRYPT_MODE**: for encryption
 - Cipher.**DECRYPT_MODE**: for decryption
 - Cipher.**WRAP_MODE**: to encrypt one key with another (see later)
 - Cipher.**UNWRAP_MODE**: to decrypt one key with another (see later)

- The encrypt or decrypt data, the `update` method is used
- With `block ciphers`, this is often done block by block:

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
byte[] outBytes = new byte[blockSize];
... // Fill into inBytes the block to encrypt/decrypt
int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
... // outBytes now contains the encrypted/decrypted block
```

- Parameters of `update`:
 - `inBytes`: the `block to encrypt or decrypt`
 - `0, blockSize`: the `range` of `inBytes` to be used (`blockSize` bytes starting at offset `0`, so the entire byte array)
 - `outBytes`: the `buffer` for the decrypted/encrypted block
- The `return value` of `update` specifies the number of bytes written to `outBytes` (the `block size` in this example)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 14

Stream Ciphers

The example above is for a block cipher. Stream ciphers such as RC4 are used slightly different in the sense that any byte-array (any length) can be used as input to the `update` method.

update

In general, the `update` method can process an arbitrary number of bytes. However, with a block cipher, the output is always a number of blocks, i.e. a multiple of the block length. Any remaining data simply “stays” in the Cipher object and is processed together with additional data during a subsequent call of `update`. Also, there are different variants of the `update` method, check out the Java API Specifications for details.

- The **final block** must be processed using the `doFinal` method
 - This causes the final (incomplete) block to be padded correctly before encryption
 - When decrypting it causes the padding to be removed correctly
- There are basically two variants of `doFinal`:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);  
outBytes = cipher.doFinal();
```

- The **first version** is to encrypt the final (incomplete) plaintext block
- The **second version** is used in the following cases:
 - During encryption if the plaintext is a **multiple of the block length**
 - An entire block consisting only of padding data is produced and encrypted
 - Guarantees that padding data can be recognized in any case when decrypting
 - To **decrypt the final ciphertext block** correctly (including padding removal)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 15

Parameters of `doFinal`

- `inBytes`: the block to encrypt
- `0, inLength`: the range of `inBytes` to be used (`inLength` bytes starting at offset 0)

`doFinal`

Just like with `update`, `doFinal` can create multiple blocks, depending on how many bytes were already in the object and how many are added as parameter in the call of `doFinal`. In addition and again just like with `update`, there are different variants of the `doFinal` method, check out the Java API Specifications for details.

Key Generation (1)

- To encrypt or decrypt, a key is required
 - It is paramount for security that keys are random
- In Java, the following steps are used to generate a key:
 - Get a KeyGenerator object for the desired algorithm
 - Initialize the generator with the key length and pseudo random number generator
 - Call the generateKey method
- Example to generate an AES key of length 128 bits:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom();
keygen.init(128, random);
SecretKey key = keygen.generateKey();
```

- SecureRandom is considered to be a cryptographically strong random number generator

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 16

Key and SecretKey

Key is an interface and SecretKey is a subinterface of Key. There are also other subinterfaces for other types of keys, e.g. PublicKey and PrivateKey. The Cipher object requires a Key object when initializing it, so any of the subinterfaces of Key (more specific Keys) can be used for initialization of the Cipher object.

SecureRandom

When constructed without parameters, SecureRandom seeds itself. The random material used for seeding is received from the underlying system and can usually be considered to be of high entropy. However, you can also seed SecureRandom yourselves by passing it seeding material in the constructor or by using the setSeed method.

Key Generation (2)

- One can also produce keys **from existing “raw key material”**
 - E.g. if someone else has given you the key and you are using it to encrypt data for her
 - Producing a `SecretKey` object can then be done with `SecretKeyFactory`

```
// Create a SecretKeyFactory to generate AES keys
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("AES");

// The raw key material (e.g. 128 bits), available as byte array
byte[] keyData = ...;

// Generate a SecretKeySpec from the raw material
SecretKeySpec keySpec = new SecretKeySpec(keyData, "AES");

// Pass the SecretKeySpec to the SecretKeyFactory to generate the
// SecretKey object
SecretKey key = keyFactory.generateSecret(keySpec);
```

- As an example, we implement a command-line program AESTest

- The program can be used as follows:

- Generate a secret key and store it in the file secret.key:

```
java AESTest -genkey secret.key
```

- Encrypt plaintextFile into encryptedFile using the key in secret.key:

```
java AESTest -encrypt plaintextFile encryptedFile secret.key
```

- Decrypt encryptedFile into decryptedFile using the key in secret.key:

```
java AESTest -decrypt encryptedFile decryptedFile secret.key
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 18

AESTest Example

The example is based on the AESTest example in *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*

You can get the full source code at <http://www.horstmann.com/corejava.html>

AESTest.java (1)

```
public class AESTest {  
  
    public static void main(String[] args) {  
  
        try {  
  
            if (args[0].equals("-genkey")) {  
  
                KeyGenerator keygen = KeyGenerator.getInstance("AES");  
                SecureRandom random = new SecureRandom();  
                keygen.init(128, random);  
                SecretKey key = keygen.generateKey();  
  
                ObjectOutputStream out = new ObjectOutputStream(  
                    new FileOutputStream(args[1]));  
                out.writeObject(key);  
                out.close();  
            } else {  
                ...  
            }  
        }  
    }  
}
```

Generate a 128-bit AES key

Write the entire
SecretKey object to the
specified file (using
serialization)

AESTest.java (2)

```
...
} else {
    int mode;
    if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
    else mode = Cipher.DECRYPT_MODE;

    ObjectInputStream keyIn = new ObjectInputStream(
        new FileInputStream(args[3]));
    SecretKey key = (SecretKey) keyIn.readObject();
    keyIn.close();

    InputStream in = new FileInputStream(args[1]);
    OutputStream out = new FileOutputStream(args[2]);
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    cipher.init(mode, key);
    crypt(in, out, cipher);
    in.close();
    out.close();
}
} catch (Exception e) {
    e.printStackTrace();
}
} // end of main
...
```

Specify the cipher mode

Read the SecretKey object from the specified file

Create and initialize the Cipher object

Call the crypt method to perform the encryption or decryption

20

ECB vs. CBC

You probably have heard before that CBC (Cipher Block Chaining) mode is more secure than ECB (Electronic Codebook) mode. You are totally right. Nevertheless, we use ECB here for simplicity so we don't have to deal with initialization vectors.

```

...
public static void crypt(InputStream in, OutputStream out,
    Cipher cipher) throws IOException, GeneralSecurityException {

    int blockSize = cipher.getBlockSize();
    byte[] inBytes = new byte[blockSize];
    byte[] outBytes = new byte[blockSize]; } Create arrays
    to read and
    write a block

    int inLength = 0;
    boolean more = true;
    while (more) {
        inLength = in.read(inBytes);
        if (inLength == blockSize) { Read a block to process
            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
            out.write(outBytes, 0, outLength);
        } else { If it's a full block, process it with the
            update method and write output block
            more = false; } Not a full block (or no data at all),
            so do not enter while loop again
    }
    if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
    else outBytes = cipher.doFinal(); out.write(outBytes); } Process the final block
    and write output
}
}

```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 21

update and doFinal

When doing encryption, the `update` method produces 16 bytes of ciphertext in each round of the while loop above. When reading in the final (incomplete) plaintext block, the `update` method is not used, but the `doFinal` method is used at the end of the `crypt` method. This adds the necessary padding and produces the final ciphertext block. If the plaintext is a multiple of the block length, then `in.read` returned -1 and the `doFinal` method is called without parameters to produce the final ciphertext block (which encrypts only padding bytes, but this corresponds to the standard of block ciphers).

Decryption works basically the same, but there are some differences. First of all, the input ciphertext is always a multiple of the block length (16 bytes here), which means the return value of `in.read` can only be either 16 or -1 (if the end of stream has been reached). The problem is that when reading the final 16 bytes, we don't know yet that these are indeed the final 16 bytes (which we can only know during the next `read` operation when receiving -1 as return value). But when the final 16 bytes are read, the `doFinal` method should be used (to correctly remove the padding) and not the `update` method. It therefore appears that one should use `update` to decrypt only if one has made sure that at least one additional ciphertext block follows.

The designers of Java were clever enough to solve this problem neatly and without having the programmer to check manually if a further ciphertext block is available before using the `update` method. The trick is that during decryption, the `update` method only decrypts a block if the `Cipher` object has received at least two blocks, i.e. there's always at least one ciphertext block kept in the internal buffer. This means that in the while loop above, the first call to `update` does not produce any plaintext (`outLength` is 0). During the next round, when the `Cipher` object gets the next 16 bytes, it decrypts the first 16 bytes and keeps the second plaintext block in its internal buffer. When the end of the input buffer is reached, `in.read` returns -1 and the while loop is left. But the final 16 ciphertext bytes (of which the plaintext contains padding bytes) are still in the buffer of the `Cipher` object – which is good as the call to `doFinal` (without any parameters) causes the final block to be decrypted correctly – including the removal of padding bytes.

One nice side effect of this design is that exactly the same code can be used for encryption and decryption, which is what we did in the `crypt` method above.

Running AESTest

- We generate first a secret key and then encrypt a file and decrypt it again:

```
$ java AESTest -genkey secret.key
$ java AESTest -encrypt plainText cipherText secret.key
$ java AESTest -decrypt cipherText decryptedText secret.key
```

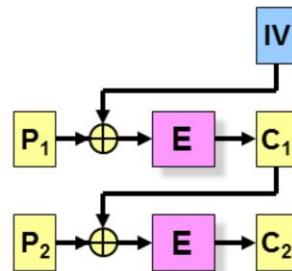
- Listing of the files:

```
$ ls -al
-rw-r--r-- 1 marc staff 3597 Aug 19 19:52 AESTest.class
-rw-r--r-- 1 marc staff 64 Aug 19 20:35 cipherText
-rw-r--r-- 1 marc staff 51 Aug 19 20:36 decryptedText
-rw-r--r-- 1 marc staff 51 Aug 19 17:50 plainText
-rw-r--r-- 1 marc staff 141 Aug 19 20:35 secret.key
```

- plainText and decryptedText have a size of **51 bytes**
- cipherText has a size of **64 bytes** → multiple of block size (padding)

Using Cipher Block Chaining Mode

- When using a cipher in CBC mode, the Cipher object needs a third initialization parameter that contains **the initialization vector (IV)**



```
// Specify a Cipher in CBC mode
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

// Create an initialization vector (IV) with 128 random bits
byte[] iv = new byte[16];
SecureRandom random = new SecureRandom();
random.nextBytes(iv);

// Create an AlgorithmParameters object and initialize it with the IV
AlgorithmParameters algParam =
    AlgorithmParameters.getInstance("AES");
algParam.init(new IvParameterSpec(iv));

// Use the init-Method with 3 Parameters to pass the
// AlgorithmParameters object
cipher.init(mode, key, algParam);
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 23

- Java can also perform [public key operations](#) such as RSA signing and encrypting and Diffie-Hellman key exchange
- As an example, we consider here [RSA encryption](#)
- Public key operations are much slower than secret key operations, so one usually combines public and secret key cryptography → [Hybrid Encryption](#)
 - The bulk data is encrypted with a randomly selected secret key
 - The secret key is encrypted with the recipient's public key, so the public key is only used for key exchange
 - The encrypted data and the encrypted secret key are sent to the recipient
 - The recipient decrypts the secret key with his private key, which allows him to decrypt the data
- In Java public key encryption works [very similar to secret key encryption](#), but different classes are used for key generation

- The following code serves to generate an RSA key pair

```
// Create a KeyPairGenerator for RSA keys
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");

// Create a SecureRandom object as the source of randomness
SecureRandom random = new SecureRandom();

// Init the generator with the key size (e.g. 2048) and the
// pseudo random number generator
pairgen.initialize(2048, random);

// Generate the key pair and extract public and private keys
KeyPair keyPair = pairgen.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
```

- The public key is used to encrypt a secret key with a Cipher in `Cipher.WRAP_MODE`:

```
// The secret key to be encrypted with the public key
SecretKey key = ...;

// The public key used for encryption
PublicKey publicKey = ...;

// The cipher for RSA encryption
Cipher cipher = Cipher.getInstance("RSA");

// Init the cipher with WRAP_MODE and the public key
cipher.init(Cipher.WRAP_MODE, publicKey);

// Encrypt the secret key with the public key
byte[] wrappedKey = cipher.wrap(key);
```

- One could also use `ENCRYPT_MODE` instead of `WRAP_MODE`
 - Using the `WRAP/UNWRAP_MODE` is **more convenient** as the cipher can encrypt `Key` objects and produce `Key` objects when decrypting

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 26

RSA Encryption

RSA encryption uses the PKCS#1 format, see <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf>.

- As an example, we implement a command-line program RSATest

- The program can be used as follows:

- Generate a public key pair and store the keys in the files public.key and private.key:

```
java RSATest -genkey public.key private.key
```

- Encrypt plaintextFile into encryptedFile using the key in public.key:

```
java RSATest -encrypt plaintextFile encryptedFile public.key
```

- Decrypt encryptedFile into decryptedFile using the key in private.key:

```
java RSATest -decrypt encryptedFile decryptedFile private.key
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 27

RSATest Example

The example is based on the RSATest example in *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*

You can get the full source code at <http://www.horstmann.com/corejava.html>

RSATest.java (1)

```
public class RSATest {  
  
    private static final int KEYSIZE = 2048;  
  
    public static void main(String[] args) {  
        try {  
            if (args[0].equals("-genkey")) {  
  
                KeyPairGenerator pairgen =  
                    KeyPairGenerator.getInstance("RSA");  
                SecureRandom random = new SecureRandom();  
                pairgen.initialize(KEYSIZE, random);  
                KeyPair keyPair = pairgen.generateKeyPair();  
  
                ObjectOutputStream out = new ObjectOutputStream(  
                    new FileOutputStream(args[1]));  
                out.writeObject(keyPair.getPublic());  
                out.close();  
                out = new ObjectOutputStream(new FileOutputStream(args[2]));  
                out.writeObject(keyPair.getPrivate());  
                out.close();  
            } else if ...  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Generate a 2048-bit RSA key pair

Write both keys to the specified files (using serialization)

RSATest.java (2)

```
... } else if (args[0].equals("-encrypt")) {  
    KeyGenerator keygen = KeyGenerator.getInstance("AES");  
    SecureRandom random = new SecureRandom();  
    keygen.init(128, random);  
    SecretKey key = keygen.generateKey(); } }  
  
ObjectInputStream keyIn = new ObjectInputStream(  
    new FileInputStream(args[3]));  
PublicKey publicKey = (PublicKey) keyIn.readObject();  
keyIn.close(); } }  
  
Cipher cipher = Cipher.getInstance("RSA");  
cipher.init(Cipher.WRAP_MODE, publicKey);  
byte[] wrappedKey = cipher.wrap(key); } }  
DataOutputStream out = new DataOutputStream(  
    new FileOutputStream(args[2]));  
out.write(wrappedKey); } }  
  
InputStream in = new FileInputStream(args[1]);  
cipher = Cipher.getInstance("AES");  
cipher.init(Cipher.ENCRYPT_MODE, key);  
crypt(in, out, cipher); } }  
in.close();  
out.close(); } }  
} else { ... } }
```

Generate a 128-bit AES key

Read the PublicKey object from the specified file

Encrypt (wrap) the secret key with the public key and write it to the specified file

Encrypt the specified file with AES using the crypt method (same method as in AESTest.java)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 29

RSATest.java (3)

```
... } else {
```

Decrypt a file

```
    DataInputStream in = new DataInputStream(
        new FileInputStream(args[1]));
    byte[] wrappedKey = new byte[KEYSIZE/8];
    in.read(wrappedKey, 0, KEYSIZE/8);
```

Read encrypted secret key from the specified file

```
    ObjectInputStream keyIn = new ObjectInputStream(
        new FileInputStream(args[3]));
    PrivateKey privateKey = (PrivateKey) keyIn.readObject();
    keyIn.close();
```

Read the Private-Key object from the specified file

```
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.UNWRAP_MODE, privateKey);
    Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
```

Decrypt (unwrap) the secret key with the private key

```
    OutputStream out = new FileOutputStream(args[2]);
    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    crypt(in, out, cipher);
    in.close();
    out.close();
}
```

Decrypt the specified file with AES using the crypt method (same method as in AESTest.java)

```
} catch (Exception e) {
    e.printStackTrace();
} // end of main
...
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 30

Cipher.unwrap

The `unwrap` method requires 3 parameters:

- The first is the wrapped key (a byte-array)
- The second is a string that specifies the algorithm associated with the wrapped key, e.g. AES or DES
- The third parameter is the type of the wrapped key, which must be one of `SECRET_KEY`, `PRIVATE_KEY`, or `PUBLIC_KEY`.

Running RSATest

- We generate first a public key pair and then encrypt a file and decrypt it again:

```
$ java RSATest -genkey public.key private.key
$ java RSATest -encrypt plainText cipherText public.key
$ java RSATest -decrypt cipherText decryptedText private.key
```

- Listing of the files:

```
$ ls -al
-rw-r--r-- 1 marc staff 4496 Aug 20 08:43 RSATest.class
-rw-r--r-- 1 marc staff 320 Aug 20 09:20 cipherText
-rw-r--r-- 1 marc staff 51 Aug 20 09:20 decryptedText
-rw-r--r-- 1 marc staff 51 Aug 20 08:22 plainText
-rw-r--r-- 1 marc staff 1478 Aug 20 09:19 private.key
-rw-r--r-- 1 marc staff 551 Aug 20 09:19 public.key
```

- plainText and decryptedText have a size of **51 bytes**
- cipherText has a size of **320 bytes** → this consists of the wrapped key (256 bytes) and the AES-encrypted ciphertext (64 bytes due to padding to multiple of block size)

Java Secure Sockets Extension (JSSE)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 32

- Java Secure Socket Extension (JSSE) provides support for the [Secure Sockets Layer / Transport Layer Security \(SSL/TLS\)](#) protocols
 - Implements SSL 3.0 – TLS 1.2 (only TLS 1.0 before Java 7)
 - JSSE was an optional package for Java 1.3 and is officially part of Java since 1.4
- Provides [all important features](#) of SSL/TLS
 - Support of server- and client-side certificates (via keystore)
 - Specifying of cipher suites
 - Session resumption
- Actual [cryptographic algorithms](#) are provided by JCA
 - JSSE therefore supports the algorithms that are offered by JCA

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 33

JSSE

Official guide, see

<http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html> for the official Java documentation.

Some tutorials to get familiar with JSSE:

- <http://www.javaworld.com/javaworld/jw-05-2001/jw-0511-howto.html>
- http://onjava.com/pub/a/onjava/2001/05/03/java_security.html

Normal, non-secure TCP-based socket communication in Java:

- Server-side:
 - The server creates a `ServerSocket` object and sets its port
 - The `accept` method is called to wait for connection requests by clients
 - When a connection has been established, the method `returns` and delivers a `Socket` object
 - The `Socket`'s `getInputStream` and `getOutputStream` methods return the stream objects for communication
- Client-side
 - The client creates a `Socket` object and specifies the server to connect to (host name/IP address and port)
 - This directly establishes the connection to the server
 - The `Socket`'s `getInputStream` and `getOutputStream` methods return the stream objects for communication

Non-Secure Echo Application – Example

- We start with a simple example of **non-secure communication**
 - The server listens on Port 9999 for incoming TCP connections
 - The client connects to the server
 - Anything that is entered on stdin on the client is sent to the server
 - The server displays the received data on stdout
- It's "**very basic**":
 - No multithreading
 - The server can just handle one client and exits when the client terminates the connection

Non-Secure Echo Server

```
public class EchoServer {  
    public static void main(String[] args) {  
        try {  
  
            // Create a ServerSocket that is bound to port 9999  
            ServerSocket ss = new ServerSocket(9999);  
  
            // Start listening for incoming connections and accept connection  
            // requests (blocking)  
            Socket socket = ss.accept();  
  
            // Echo anything that is received to stdout  
            Scanner reader = new Scanner(socket.getInputStream());  
            String string = null;  
            while ((string = reader.nextLine()) != null) {  
                System.out.println(string);  
                System.out.flush();  
            }  
        } catch (NoSuchElementException exception) {  
            // OK, client disconnected  
        } catch (Exception exception) {  
            exception.printStackTrace();  
        }  
    }  
}
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 36

Non-Secure Echo Client

```
public class EchoClient {  
    public static void main(String [] args) {  
        try {  
            // Create a Socket that connects to localhost:9999  
            Socket socket = new Socket("localhost", 9999);  
  
            // Read lines from stdin and send them to the server  
            Scanner reader = new Scanner(System.in);  
            OutputStream outputstream = socket.getOutputStream();  
            OutputStreamWriter outputstreamwriter =  
                new OutputStreamWriter(outputstream);  
            BufferedWriter bufferedwriter =  
                new BufferedWriter(outputstreamwriter);  
            String string = null;  
            while ((string = reader.nextLine()) != null) {  
                bufferedwriter.write(string + '\n');  
                bufferedwriter.flush();  
            }  
        } catch (Exception exception) {  
            exception.printStackTrace();  
        }  
    }  
}
```

A Secure Echo Application – Server Side

- Extending this example to a secure application that uses SSL/TLS requires only little adaptation on the server side:
 - Create an `SSLServerSocketFactory` and use it to create an `SSLServerSocket`
 - `accept` returns an `SSLSocket` object

```
public class EchoServer {  
    public static void main(String[] args) {  
        try {  
  
            // Create the SSLServerSocketFactory with default settings  
            SSLServerSocketFactory sslSSF =  
                (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();  
  
            // Create an SSLServerSocket that is bound to port 9999  
            SSLServerSocket ss =  
                (SSLServerSocket)sslSSF.createServerSocket(9999);  
  
            // Start listening for incoming connections and accept connection  
            // requests (blocking)  
            SSLSocket socket = (SSLSocket)ss.accept();  
  
            // Echo anything that is received to stdout  
            // No changes from here on...  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

`SSLServerSocketFactory.getDefault()` creates an `SSLServerSocketFactory` object, but the return type of the method is `ServerSocketFactory`. To assign it to a variable of type `SSLServerSocketFactory`, it must be explicitly casted. Similarly, this must also be done with `SSLServerSocket` and `SSLSocket`.

A Secure Echo Application – Client Side

- On the client side, some small changes must be done as well:
 - Create an `SSLSocketFactory` and use it to create an `SSLSocket`

```
public class EchoClient {  
    public static void main(String[] args) {  
        try {  
  
            // Create the SSLSocketFactory with default settings  
            SSLSocketFactory sslSF =  
                (SSLSocketFactory) SSLSocketFactory.getDefault();  
  
            // Creates an SSLSocket that connects to localhost:9999  
            SSLSocket socket = (SSLSocket)sslSF.createSocket("localhost", 9999);  
  
            // Read lines from stdin and send them to the server  
            // No changes from here on...  
        }  
    }  
}
```

- Since **SSL/TLS uses certificates**, we have to create them before the example can be run
 - Per default, we have only **server-side authentication** with SSL/TLS
 - So we need a key-pair and a corresponding certificate for the server
 - And **the client has to trust the certificate** as otherwise, an exception is thrown during the SSL/TLS handshake
- To store this information, Java uses **keystores and truststores**
 - **keystores** contain both private keys and corresponding certificates
 - **truststores** contain certificates only
 - Both are manipulated using the **keytool** command-line tool
 - The party that authenticates itself (here the server) needs a keystore
 - The party that authenticates the other endpoint (here the client) needs a truststore
 - The standard truststore shipped with Java (`$JAVA_HOME/jre/lib/security/cacerts`) contains **root certificates issued by "official" certification authorities (CAs)** → certificates issued by these CAs are always trusted

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 40

keytool

Java contains the keytool, a command-line tool to generate and manage keys and certificates. This tool can be used to administrate keystores and truststores.

A Secure Echo Application – Key Pair and Certificate Creation

Zürcher Hochschule
für Angewandte Wissenschaften



- **Generate** the key pair and self-signed certificate

```
keytool -genkeypair -keystore ks_server -alias localhost
```

- The certificate follows the **X.509 standard** and the naming uses the **X.500 naming scheme**, so corresponding values have to be entered:

```
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
[Unknown]: localhost
What is the name of your organizational unit?
[Unknown]: InIT
What is the name of your organization?
[Unknown]: ZHAW
What is the name of your City or Locality?
[Unknown]: Winterthur
What is the name of your State or Province?
[Unknown]: ZH
What is the two-letter country code for this unit?
[Unknown]: CH
Is CN=localhost, OU=InIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH correct?
[no]: yes

Enter key password for <localhost>
      (RETURN if same as keystore password):
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 41

alias

The alias is used to identify the key pair / certificate, as a keystore may contain several of them.

keytool

Refer to <http://download.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html> for a detailed explanation of keytool and all command-line tool options. Per default, DSA keys with a length of 1024 bits are generated and SHA1withDSA is used for signatures.

Common Name

If you are familiar with certificate handling when browsing the Web using SSL/TLS, you know that browsers display a warning when the Common Name in the server certificate does not match the hostname you entered in the address bar (e.g. <http://www.zhaw.ch/foo/bar>). When using JSSE, this check is not done automatically, so no matter what you enter as a reply to the “What is your first and last name” question (which results in the Common Name entry in the certificate), running the example will not result in any complaint about a mismatch in the name. All JSSE checks is that the certificate can be trusted in the sense that it is included in the client’s truststore or it is signed by a certificate in the truststore. Any further checks of the certificate have to be done by the programmer by accessing the certificate received from the peer and analyzing its content.

Just to give you a start: This can be done by using the `getSession` method of the `SSLocket` object, which provides a `SSLSession` object. Then, the `getPeerCertificates` method of the `SSLSession` object can be used to access the certificate. For details, refer to the JSSE reference guide at <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html>.

Using Certificates from other Sources?

While keytool can be used to create certificates, it's not really powerful. If you want to set specific certificate options, using, e.g., openssl or any other certificate management software is usually the better option. keytool is then still used to import the certificates and keys into keystores and truststores. You can, of course, also import “real” certificates and keys issued by certification authorities.

However, if you just need some keys and certificates to do some tests, keytool is well suited also for certificate and key creation.

A Secure Echo Application – Export and Show Certificate

Zürcher Hochschule
für Angewandte Wissenschaften



- Export the certificate

```
keytool -exportcert -keystore ks_server -alias localhost  
-file server.cer
```

- Display the certificate (just to show the functionality):

```
keytool -printcert -file server.cer
```

```
Owner: CN=localhost, OU=InIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH  
Issuer: CN=localhost, OU=InIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH  
Serial number: 4c721e04  
Valid from: Thu Mar 07 08:40:24 CET 2013 until: Wed Jun 05 09:40:24 CEST 2013  
Certificate fingerprints:  
MD5: FF:61:0E:D5:A3:F6:49:C3:A9:30:C8:39:29:74:FC:F1  
SHA1: 93:9E:58:06:E1:72:D7:C7:5B:23:3B:59:1D:88:D7:B8:D4:BB:1E:A7  
Signature algorithm name: SHA1withDSA  
Version: 3
```

A Secure Echo Application – Truststore Creation

Zürcher Hochschule
für Angewandte Wissenschaften



- Create a **truststore** and import the certificate

```
keytool -importcert -keystore ts_client -alias localhost  
-file server.cer
```

```
Enter keystore password: password  
Re-enter new password: password  
Owner: CN=localhost, OU=INIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH  
Issuer: CN=localhost, OU=INIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH  
Serial number: 4c721e04  
Valid from: Mon Aug 23 09:06:44 CEST 2010 until: Sun Nov 21 08:06:44 CET 2010  
Certificate fingerprints:  
MD5: FF:61:0E:D5:A3:F6:49:C3:A9:30:C8:39:29:74:FC:F1  
SHA1: 93:9E:58:06:E1:72:D7:C7:5B:23:3B:59:1D:88:D7:B8:D4:BB:1E:A7  
Signature algorithm name: SHA1withDSA  
Version: 3  
Trust this certificate? [no]: yes  
Certificate was added to keystore
```

- Sidenote: This example simply uses a self-signed certificate for the server, but **certificate hierarchies** are also possible (see notes below)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 43

Certificate Hierarchies

In the example used above, we simply use a self-signed certificate. However, keystores can also be used with certificate hierarchies, i.e. a root certificate that is used to sign further certificates, which in turn are used for authentication. The following example is from *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*:

- Generate a root certificate

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

- Export the certificate

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file  
acmeroot.cer
```

- Add the root certificate to the keystores of the employees, e.g. Cindy:

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file  
acmeroot.cer
```

Note that when importing the certificate, she is asked whether she trusts it.

- Have Alice generate a key pair and export the certificate:

```
keytool -genkeypair -keystore alice.certs -alias alice
```

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

- Sign Alice's certificate using the root certificate. This cannot be done with keytool, but there's a Java program available on the book's resources pages to do so:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot -infile  
alice.cer -outfile alice_signedby_acmeroot.cer
```

- Have Cindy import the signed certificate into her keystore. This time – as she already trusts the root certificate, she is not asked to trust Alice's certificate

```
keytool -importcert -keystore cindy.certs -alias alice -file alice.cer
```

A Secure Echo Application – Running the Example

- To run the server, specify the keystore and password:

```
$ java -Djavax.net.ssl.keyStore=ks_server  
-Djavax.net.ssl.keyStorePassword=password EchoServer
```

- To run the client, specify the truststore and password:

```
$ java -Djavax.net.ssl.trustStore=ts_client  
-Djavax.net.ssl.trustStorePassword=password EchoClient
```

- Entering a line in the client....

```
rema:bin marc$ java -Djavax.net.ssl.trustStore=ts_client -Djavax.net.ssl.trustSt  
orePassword=password EchoClient  
You are the best of the best of the best!
```

- ...echoes it at the server side

```
rema:bin marc$ java -Djavax.net.ssl.keyStore=ks_server -Djavax.net.ssl.keyStoreP  
assword=password EchoServer  
You are the best of the best of the best!
```

A Secure Echo Application – Analysis (1)

- Wireshark trace:

56 9.628531	127.0.0.1	127.0.0.1	TCP	60583 > distinct [SYN] Seq=0 Win=65535 Len=0 MSS=16 TSval=65535 Len=0 Win=16 TSval=16344
57 9.628619	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344
58 9.628630	127.0.0.1	127.0.0.1	TCP	60583 > distinct [ACK] Seq=1 Ack=1 Win=146976 Len=0 TSval=33957340
59 9.628640	127.0.0.1	127.0.0.1	TCP	[TCP Window Update] distinct > 60583 [ACK] Seq=1 Ack=1 Win=146976
62 10.231677	127.0.0.1	127.0.0.1	TLSv1	Client Hello
63 10.231715	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1 Ack=169 Win=146816 Len=0 TSval=339573
64 10.241403	127.0.0.1	127.0.0.1	TLSv1	Server Hello, [Certificate] Server Key Exchange, Server Hello Done
65 10.241460	127.0.0.1	127.0.0.1	TCP	60583 > distinct [ACK] Seq=169 Ack=1250 Win=145728 Len=0 TSval=339
66 10.253103	127.0.0.1	127.0.0.1	TLSv1	Client Key Exchange
67 10.253157	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1250 Ack=276 Win=146704 Len=0 TSval=339
68 10.262249	127.0.0.1	127.0.0.1	TLSv1	Change Cipher Spec
69 10.262279	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1250 Ack=282 Win=146704 Len=0 TSval=339
70 10.263768	127.0.0.1	127.0.0.1	TLSv1	Encrypted Handshake Message
71 10.263814	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1250 Ack=335 Win=146640 Len=0 TSval=339
72 10.265748	127.0.0.1	127.0.0.1	TLSv1	Change Cipher Spec
73 10.265802	127.0.0.1	127.0.0.1	TCP	60583 > distinct [ACK] Seq=335 Ack=1256 Win=145728 Len=0 TSval=339
74 10.266057	127.0.0.1	127.0.0.1	TLSv1	Encrypted Handshake Message
75 10.266101	127.0.0.1	127.0.0.1	TCP	60583 > distinct [ACK] Seq=335 Ack=1309 Win=145680 Len=0 TSval=339
76 10.266540	127.0.0.1	127.0.0.1	TLSv1	Application Data
77 10.266598	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1309 Ack=372 Win=146608 Len=0 TSval=339
88 12.411119	127.0.0.1	127.0.0.1	TCP	60583 > distinct [FIN, ACK] Seq=372 Ack=1309 Win=145680 Len=0 TSval=339
89 12.411182	127.0.0.1	127.0.0.1	TCP	distinct > 60583 [ACK] Seq=1309 Ack=373 Win=146608 Len=0 TSval=339
90 12.411201	127.0.0.1	127.0.0.1	TCP	[TCP Dup ACK 88#1] 60583 > distinct [ACK] Seq=373 Ack=1309 Win=145
91 12.411799	127.0.0.1	127.0.0.1	TLSv1	Encrypted Alert
92 12.411852	127.0.0.1	127.0.0.1	TCP	60583 > distinct [RST] Seq=373 Win=0 Len=0

- So SSL/TLS is used indeed

- Per default **TLS 1.0** is used (although Java 7 supports TLS 1.2)
 - **Server authentication** takes place (server sends certificate to client)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 45

From <http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html>: Although SunJSSE in the Java SE 7 release supports TLS 1.1 and TLS 1.2, neither version is enabled by default for client connections. Some servers do not implement forward compatibility correctly and refuse to talk to TLS 1.1 or TLS 1.2 clients. For interoperability, SunJSSE does not enable TLS 1.1 or TLS 1.2 by default for client connections.

A Secure Echo Application – Analysis (2)

- The client only offers reasonably strong cipher suites (MD5 though...)

Cipher Suites (28 suites)

```
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0x00c)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x00a)
Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
```

- Before Java 7, this was NOT the case, Java 6:

```
▽ Cipher Specs (22 specs)
Cipher Spec: TLS_RSA_WITH_RC4_128_MD5 (0x000004)
Cipher Spec: SSL2_RC4_128_WITH_MD5 (0x010080)
Cipher Spec: TLS_RSA_WITH_RC4_128_SHA (0x000005)
Cipher Spec: TLS_RSA_WITH_AES_128_CBC_SHA (0x00002f)
Cipher Spec: TLS_RSA_WITH_AES_256_CBC_SHA (0x000035)
Cipher Spec: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x000033)
Cipher Spec: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x000039)
Cipher Spec: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x000032)
Cipher Spec: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x000038)
Cipher Spec: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x00000a)
Cipher Spec: SSL2_DES_192_EDE3_CBC_WITH_MD5 (0x0700c0)
Cipher Spec: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x000016)
Cipher Spec: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x000013)
Cipher Spec: TLS_RSA_WITH_DES_CBC_SHA (0x000009)
Cipher Spec: SSL2_DES_64_CBC_WITH_MD5 (0x060040)
Cipher Spec: TLS_DHE_RSA_WITH_DES_CBC_SHA (0x000015)
Cipher Spec: TLS_DHE_DSS_WITH_DES_CBC_SHA (0x000012)
Cipher Spec: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x000003)
Cipher Spec: SSL2_RC4_128_EXPORT40_WITH_MD5 (0x020080)
Cipher Spec: TLS_RSA_EXPORT_WITH_DES40_CBC_SHA (0x000008)
Cipher Spec: TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA (0x000014)
Cipher Spec: TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA (0x000011)
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 46

TLS_EMPTY_RENEGOTIATION_INFO_SCSV

Sometimes, this special cipher suite is also offered by the client: However, this is not an actual cipher suite but has been added to recent versions to cope with a vulnerability discovered with respect to TLS renegotiation (not discussed here). It is used by client-side implementations that are not vulnerable to the attack to signal the server that renegotiating can be used securely. For details, refer to <http://www.ietf.org/rfc/rfc5746.txt>. For details about the vulnerability, refer to <http://blog.gsec.lu/2009/11/tls-sslv3-renegotiation-vulnerability.html>.

A Secure Echo Application – Analysis (3)

- JSSE debugging can also be done using a command-line option:

```
$ java -Djavax.net.debug=ssl -Djavax.net.ssl.trustStore=ts_client
-Djavax.net.ssl.trustStorePassword=password EchoClient
```

```
rema:bin marc$ java -Djavax.net.debug=ssl -Djavax.net.ssl.trustStore=ts_client -Djavax.net.ssl.trustStorePassword=password
EchoClient
keyStore is :
keyStore type is : jks
keyStore provider is :
init keystore
init keymanager of type SunX509
trustStore is: ts_client
trustStore type is : jks
trustStore provider is :
init truststore
adding as trusted cert:
Subject: CN=localhost, OU=INIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Issuer: CN=localhost, OU=INIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Algorithm: DSA; Serial number: 0x39c133a7
Valid from Mon Oct 07 10:48:47 CEST 2013 until Sun Jan 05 09:48:47 CET 2014

trigger seeding of SecureRandom
done seeding SecureRandom
You are the best of the best of the best!
%% No cached client session
*** ClientHello, TLSv1
RandomCookie:  GMT: 1381136949 bytes = { 127, 250, 66, 246, 230, 179, 15, 101, 205, 79, 119, 5, 24, 19, 175, 148, 214, 65,
12, 189, 163, 85, 80, 178, 54, 199 }
Session ID: {}
Cipher Suites: [TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA, TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA, TLS_ECDH_RSA_WITH_AES_256_CBC_SHA, TLS_DHE_RSA_WITH_AES_256_CBC_SHA,
TLS_DHE_DSS_WITH_AES_256_CBC_SHA, TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA, TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
TLS_RSA_WITH_AES_128_CBC_SHA, TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA, TLS_ECDH_RSA_WITH_AES_128_CBC_SHA,
TLS_DHE_RSA_WITH_AES_128_CBC_SHA, TLS_DHE_DSS_WITH_AES_128_CBC_SHA, TLS_ECDHE_ECDSA_WITH_RC4_128_SHA,
TLS_ECDHE_RSA_WITH_RC4_128_SHA, SSL_RSA_WITH_RC4_128_SHA, TLS_ECDH_ECDSA_WITH_RC4_128_SHA, TLS_ECDH_RSA_WITH_RC4_128_SHA,
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA, TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA, SSL_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA, TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA, SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA,
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA, SSL_RSA_WITH_RC4_128_MDS, TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
Compression Methods: { 0 }
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 47

A more Elaborate Example

- The previous example demonstrated the easiest way to work with SSL/TLS in Java
 - The basic configuration is "quite secure"
- We now modify the secure Echo application to show some additional possibilities you have when working with JSSE
- In particular, we implement the following extension:
 - We do not specify keystores and truststores via the command-line, but access them directly from within the program
 - To do so, we cannot use `SSL(Server)SocketFactory` with default settings, but must create them via an `SSLContext` object
 - This also allows specifying to use TLS 1.2
 - We use additional client authentication
 - We specify the cipher suites to determine the algorithms to be used
 - The server displays some information about the client's certificate

A more Elaborate Example – keystores and truststores

- We need some **keystores and truststores**:

- Both the client and the server need a key pair and a keystore, as both authenticate themselves
- Both need a truststore to verify the peer's certificate

```
Create a key pair for server and client using RSA keys with a 2048-bit modulus (default is DSA with 1024 bits):
```

```
keytool -genkeypair -keystore ks_server -keyalg rsa -keysize 2048  
-alias server  
keytool -genkeypair -keystore ks_client -keyalg rsa -keysize 2048  
-alias client
```

```
Export the certificates:
```

```
keytool -export -keystore ks_server -alias server -file cert_server.cer  
keytool -export -keystore ks_client -alias client -file cert_client.cer
```

```
Import the peer's cerificate in a truststore:
```

```
keytool -import -keystore ts_server -alias client -file cert_client.cer  
keytool -import -keystore ts_client -alias server -file cert_server.cer
```

A more Elaborate Example – Server Side (1)

Load keystore and truststore

```
public class EchoServer {  
    public static void main(String[] args) {  
  
        // The password used for keystores, truststores and private keys  
        final char[] PASSWORD = "password".toCharArray();  
  
        try {  
            // Create a KeyStore object and load it with the keystore data  
            KeyStore keystore = KeyStore.getInstance("JKS");  
            keystore.load(new FileInputStream("ks_server"), PASSWORD);  
  
            // Create a KeyManagerFactory object and initialize it to work with  
            // the keystore; giving it access to private key and certificate  
            KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");  
            kmf.init(keystore, PASSWORD);  
  
            // Create a KeyStore object and load it with the truststore data  
            KeyStore truststore = KeyStore.getInstance("JKS");  
            truststore.load(new FileInputStream("ts_server"), PASSWORD);  
  
            // Create a TrustManagerFactory object and initialize it to work  
            // with the truststore; giving it access to the certificate  
            TrustManagerFactory tmf =  
                TrustManagerFactory.getInstance("SunX509");  
            tmf.init(truststore);  
            ...  
        }  
    }  
}
```

x 50

Keystore Password

Obviously, it's not a good idea to hardcode a password into a program, but it's nevertheless done here for simplicity. And "password" is not a really strong password, I guess...

A more Elaborate Example – Server Side (2)

Create a server socket

```
...  
  
// Create an SSLContext that uses TLS 1.2  
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");  
  
// Create KeyManager and TrustManager objects from the previously  
// created factories and use them to initialize the SSLContext object,  
// which then uses the private key and certificates in our keystore  
// and truststore (the third argument can be used to specify a source  
// of randomness for key material generation, default is SecureRandom)  
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);  
  
// Create an SSLServerSocketFactory from the SSLContext object  
SSLServerSocketFactory sslSSF =  
    (SSLServerSocketFactory)sslContext.getServerSocketFactory();  
  
// Create an SSLServerSocket that is bound to port 9999  
SSLServerSocket ss = (SSLServerSocket)sslSSF.createServerSocket(9999);  
  
// Require client authentication  
ss.setNeedClientAuth(true);  
  
...
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 51

SSLContext.init()

Note that the getKeyManagers and getTrustManagers methods return arrays of KeyManagers/TrustManagers. The reason is that if the used keystore/truststore contains different types of keys and certificates, then one KeyManager/TrustManager per type is returned. In our case, the keystore/truststore only contains X.509 certificates and corresponding private keys and correspondingly, the returned array also only contains one KeyManager/TrustManager (of type X509KeyManager and X509TrustManager).

A more Elaborate Example – Server Side (3) Set cipher suites and print certificate information

```
...  
  
// The server just accepts one cipher suite  
String[] enabledCipherSuites = {"TLS_DHE_RSA_WITH_AES_128_CBC_SHA"};  
ss.setEnabledCipherSuites(enabledCipherSuites);  
  
// Start listening for incoming connections and accept connection  
// requests (blocking)  
SSLocket socket = (SSLocket)ss.accept();  
  
// Get the client's certificate(s)  
X509Certificate[] certificates =  
    (X509Certificate[])socket.getSession().getPeerCertificates();  
System.out.println("Subject: " +  
    certificates[0].getSubjectX500Principal());  
System.out.println("Validity: " + certificates[0].getNotBefore() +  
    " - " + certificates[0].getNotAfter());  
  
// Echo anything that is received to stdout  
// No changes from here on...
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 52

getPeerCertificates()

The method returns an array of X509Certificates. The reason why there are multiple certificates is that the entire certificate chain (if any) is returned. As our example only uses a self-signed certificate and no hierarchy, just one certificate is returned.

A more Elaborate Example – Client Side (1)

Load keystore and truststore

```
public class EchoClient {  
    public static void main(String[] args) {  
  
        // The password used for keystores, truststores and private keys  
        final char[] PASSWORD = "password".toCharArray();  
  
        try {  
            // Create a KeyStore object and load it with the keystore data  
            KeyStore keystore = KeyStore.getInstance("JKS");  
            keystore.load(new FileInputStream("ks_client"), PASSWORD);  
  
            // Create a KeyManagerFactory object and initialize it to work with  
            // the keystore; giving it access to private key and certificate  
            KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");  
            kmf.init(keystore, PASSWORD);  
  
            // Create a KeyStore object and load it with the truststore data  
            KeyStore truststore = KeyStore.getInstance("JKS");  
            truststore.load(new FileInputStream("ts_client"), PASSWORD);  
  
            // Create a TrustManagerFactory object and initialize it to work  
            // with the truststore; giving it access to the certificate  
            TrustManagerFactory tmf =  
                TrustManagerFactory.getInstance("SunX509");  
            tmf.init(truststore);  
            ...  
        }  
    }  
}
```

x 53

Load Keystore and Truststore

The code above is exactly the same as on the server side, except that different files are used.

A more Elaborate Example – Client Side (2)

Create a socket and set cipher suites

```
...
// Create an SSLContext that uses TLS 1.2
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");

// Create KeyManager and TrustManager objects from the previously
// created factories and initialize the SSLContext object, which then
// uses the private key and certificates in our keystore and truststore
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create an SSLSocketFactory from the SSLContext
SSLSocketFactory sslSF =
    (SSLSocketFactory)sslContext.getSocketFactory();

// Creates an SSLSocket that connects to localhost:9999
SSLSocket socket = (SSLSocket)sslSF.createSocket("localhost", 9999);

// The client only wants to use AES cipher suites
String[] enabledCipherSuites = {"TLS_RSA_WITH_AES_128_CBC_SHA",
                                "TLS_RSA_WITH_AES_256_CBC_SHA",
                                "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
                                "TLS_DHE_RSA_WITH_AES_256_CBC_SHA"};
socket.setEnabledCipherSuites(enabledCipherSuites);

// Read lines from stdin and send them to the server
// No changes from here on...
```

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 54

A more Elaborate Example – Running the Example

- No options are necessary to [run server and client](#):

```
$ java EchoServer
```

```
$ java EchoClient
```

- Entering a line in the client....

```
$ java EchoClient
You are the best of the best of the best!
```

- ...results in the following at the server side

```
$ java EchoServer
Subject: CN=Marc Rennhard, OU=InIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Validity: Thu Mar 01 15:08:32 CET 2012 - Wed May 30 16:08:32 CEST 2012
You are the best of the best of the best!
```

- As expected, the output contains [information from the client's certificate](#)

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 55

Performed Certificate Checks

Besides checking whether the trust chain can be successfully built from the certificate received from the peer to a trusted certificate in the truststore (this of course includes checking the correctness of the signatures), virtually no further security checks are performed. It was noted before that with JSSE, the common name is not checked automatically, this has to be done programmatically if desired. The same is true for other checks. For instance, it is not checked whether the current date is within the validity period of all certificates used in the certificate chain. This can be done using the checkValidity() method of a certificate (X509Certificate class). In addition, revocation checks using CRLs or OCSP must also be done programmatically if desired. For more details, refer to the official JSSE Reference Guide.

A more Elaborate Example – Analysis (1)

19	5.931604	127.0.0.1	127.0.0.1	TCP	65308 > distinct [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=16 TSval=58189768
20	5.931675	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=16
21	5.931684	127.0.0.1	127.0.0.1	TCP	65308 > distinct [ACK] Seq=1 Ack=1 Win=146976 Len=0 TSval=581895611 TSec
22	5.931692	127.0.0.1	127.0.0.1	TCP	[TCP Window Update] distinct > 65308 [ACK] Seq=1 Ack=1 Win=146976 Len=0
35	8.139791	127.0.0.1	127.0.0.1	TLSv1.2	client Hello
36	8.139823	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1 Ack=94 Win=146880 Len=0 TSval=581897580 TSe
37	8.172575	127.0.0.1	127.0.0.1	TLSv1.2	Server Hello, Certificate, Server Key Exchange, Certificate Request, Ser
38	8.172599	127.0.0.1	127.0.0.1	TCP	65308 > distinct [ACK] Seq=94 Ack=1628 Win=145360 Len=0 TSval=581897610
39	8.182755	127.0.0.1	127.0.0.1	TLSv1.2	Certificate, Client Key Exchange
40	8.182811	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1628 Ack=1053 Win=145936 Len=0 TSval=58189762
41	8.215569	127.0.0.1	127.0.0.1	TLSv1.2	Certificate Verify
42	8.215625	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1628 Ack=1322 Win=145664 Len=0 TSval=58189764
43	8.215638	127.0.0.1	127.0.0.1	TLSv1.2	Change Cipher Spec
44	8.215648	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1628 Ack=1328 Win=145648 Len=0 TSval=58189764
45	8.217240	127.0.0.1	127.0.0.1	TLSv1.2	Encrypted Handshake Message
46	8.217276	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1628 Ack=1397 Win=145584 Len=0 TSval=58189764
47	8.220765	127.0.0.1	127.0.0.1	TLSv1.2	Change Cipher Spec
48	8.220815	127.0.0.1	127.0.0.1	TCP	65308 > distinct [ACK] Seq=1397 Ack=1634 Win=145344 Len=0 TSval=58189765
49	8.221089	127.0.0.1	127.0.0.1	TLSv1.2	Encrypted Handshake Message
50	8.221132	127.0.0.1	127.0.0.1	TCP	65308 > distinct [ACK] Seq=1397 Ack=1703 Win=145280 Len=0 TSval=58189765
51	8.221480	127.0.0.1	127.0.0.1	TLSv1.2	Application Data
52	8.221524	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1703 Ack=1450 Win=145536 Len=0 TSval=58189765
59	11.230681	127.0.0.1	127.0.0.1	TCP	65308 > distinct [FIN, ACK] Seq=1450 Ack=1703 Win=145280 Len=0 TSval=581
60	11.230727	127.0.0.1	127.0.0.1	TCP	distinct > 65308 [ACK] Seq=1703 Ack=1451 Win=145536 Len=0 TSval=58190032
61	11.230740	127.0.0.1	127.0.0.1	TCP	[TCP Dup ACK 59#1] 65308 > distinct [ACK] Seq=1451 Ack=1703 Win=145280 L
62	11.231316	127.0.0.1	127.0.0.1	TLSv1.2	Encrypted Alert
63	11.231344	127.0.0.1	127.0.0.1	TCP	65308 > distinct [RST] Seq=1451 Win=0 Len=0

- As configured, TLS 1.2 and **client authentication is used**:
 - The server sends a Certificate Request message
 - The client sends its certificate and a Certificate Verify message

Marc Rennhard, 29.05.2014, SSI_JavaSecurity.pptx 56

A Secure Echo Application – Analysis (2)

- Cipher suites offered by the client:

- Cipher Suites (4 suites)
 - Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
 - Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
 - Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
 - Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)

- Cipher suite selected by the server:

- Handshake Protocol: Server Hello
 - Handshake Type: Server Hello (2)
 - Length: 77
 - Version: TLS 1.2 (0x0303)
- ▷ Random
 - Session ID Length: 32
 - Session ID: 524fb980cb99eda6bf6411aafbc5b1d97e9fb1ffaf8f31ed...
 - Cipher Suite: **TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)**
 - Compression Method: null (0)

- **JCA** enables fundamental cryptographic operations, including
 - Message digests, Message authentication codes (HMAC), Symmetric encryption, Public key cryptography, Diffie-Hellman key exchange, Pseudo random number generation
- **JSSE** allows to secure networked applications using SSL/TLS
 - Allows specifying various options such as client-authentication and used cipher suites
- **Use these libraries** when you need the corresponding functionality
 - Don't invent your own cryptography!
 - But: it's still necessary to use the libraries with care (key lengths, cipher modes, cipher suites etc.)