

Security Aspects in Software Development

KU 2013/2014

“Applied Cryptography and PKI (J)”

Daniel Hein Johannes Winter

Thomas Kastner

Institute for Applied Information Processing and Communications

Inffeldgasse 16a, 8020 Graz, Austria

sicherheitsaspekte@iaik.tugraz.at

November 6, 2013

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Objectives	2
1.3	Plagiarism	3
2	Background story	3
3	Software requirements and setup	4
3.1	Updating your repository	4
3.2	Implementation details	4
3.3	Running the applications	5
3.4	External libraries	5
3.5	Java cryptography	5
3.6	Creating test certificates	5
3.7	JUnit	6
3.8	Ant	6
4	Submission	6
4.1	Creating the submission branch	6
4.2	Working with the submission branch	7
4.3	Automated compilation	7
4.4	Verifying your submission	7
4.5	Checklist	8
4.6	Deadline	8
5	Tasks	8
5.1	(1.0 points) Symmetric encryption	8
5.2	(1.0 points) Keyed-Hashed Message Authentication Codes (HMACs)	9
5.3	(2.0 points) RSA signature generation and verification	9
5.4	(2.0 points) Session key encipherment with RSA	9
5.5	(8.0 points) Simple certificate validation	9
5.6	(6.0 points) Cross certification	10
5.7	(5.0 points) Protocol Analysis	10

1 Introduction

Assignment title: Applied Cryptography and PKI (J)
Group size: 2 students
Start date: November 6, 2013
Maximum score: 25 points (25% percent of final* mark)

HARD SUBMISSION DEADLINE: December 4, 2013 (23:59:59 CET)

See Section 4 for details on the submission process.

See Section 5 for the subtasks and questions of this assignment.

* Bar the oral exam.

1.1 Motivation

If applied correctly, cryptography provides a number of useful building blocks (primitives) for constructing secure systems. Unfortunately, if used incorrectly, cryptography might not achieve the targeted security goal at all, while lulling the user into a false sense of security. Sadly, there is significant potential for misusing and misconfiguring cryptographic building blocks in a way that breaks security. The same notion holds true for the infrastructure used in support of cryptography. The Public Key Infrastructure (PKI) in use today to authenticate public keys is one example of such an infrastructure. Here, misconfiguration or a faulty implementation can lead to catastrophic effects on security (cf. Georgiev et al. [3]).

In this assignment we concentrate on correct use of primitives like symmetric ciphers, asymmetric ciphers, and hashed message authentication codes in typical application scenarios. Additionally, we address correctly authenticating public keys using PKI components. To further foster the correct use of cryptography we have prepared a protocol analysis tasks. The protocol analysis task uses several different versions of the same protocol to demonstrate the necessity of cryptographic primitives to protect against a man-in-the-middle attacker. In addition, the protocol analysis task highlights why each of the different primitives is in place.

1.2 Objectives

The main goal of this exercise is to build up basic skills in applied cryptography and public key infrastructures. After this assignment you should be able to:

- correctly use symmetric and asymmetric cryptography primitives. Specifically you should be able to use the Advanced Encryption Standard (AES) in the Counter with CBC-MAC (CCM) mode of operation.
- better understand when confidentiality and integrity are required and how to achieve those security goals using the right combination of cryptographic primitives.
- explain and avoid the basics and pitfalls of X.509 certificate validation.
- perform a simplified X.509 certificate validation
- explain typical problems associated with PKI and X.509 certificates

Also, after completing this assignment you should be able to recognize the differences between symmetric ciphers, hashed message authentication codes, and asymmetric ciphers. Moreover, you should be able to correctly apply these cryptographic primitives to implement applications which require digital signatures, message encryption, message authentication and key encipherment.

Another goal of this assignment is to teach you how to correctly use public key infrastructures and how to validate certificate chains. At the end of the task you should be aware of the basic steps required to validate X.509 public key certificates.

1.3 Plagiarism

△ *Plagiarism will not be tolerated and any contributions containing copied code will automatically receive a negative rating. It is explicitly allowed to use material and code snippets shown during the lecture or the exercises.*

2 Background story

A prospective security expert, you, has been asked to secure an existing unidirectional message system - called *J* - by integrating cryptographic primitives to ensure data integrity and confidentiality. The message system uses an untrusted channel to transmit its messages. The message system consists of a client component, which sends compiled programs, so-called *codelets*, to a *dispatcher server*. This *dispatcher server* validates the compiled *codelets*, and forwards them to one of the *codelet execution servers*. This *codelet execution server* then executes the *codelet*.

From a security point of view, executing code sent over an untrusted channel is dangerous to the *codelet execution server*. The receiver must ensure that he, or she trusts the sender. To establish this trust the receiver should at least authenticate the source of the packet, and ensure that the *codelet* has not been modified in transit.

The sender on the other hand wants to ensure the confidentiality of the sent code. The sent code consists of mission critical intellectual property, and loss of this code could have fatal effects for the company. Hence the requirement to add integrity and confidentiality mechanisms to the existing messaging framework *J*. As an upshot of this, the client also needs to be sure that the *codelet* is sent to the correct server.

To actually authenticate the *dispatcher server*, the client maintains a list of trusted server certificates. To ensure the authenticity of a server certificate, the client uses a Keyed-Hash Message Authentication Code (HMAC) on the server certificate's fingerprint. On encountering a new server certificate, the client computes the HMAC and stores the HMAC locally. Next time the client sees the same server certificate it recomputes the HMAC and compares it with its local copy. Access to the HMAC's key is password based cryptography protected. Thus, the client can be sure that the server certificate has not been changed, since first seeing it.

The *dispatcher server* authenticates the client using a PKI. The client signs the *codelet* before sending it to the server. The server then verifies the signature on the *codelet* to ensure its origin is trusted. The server uses X.509 certificates to authenticate the public key used to verify the signature on the *codelet*. As part of the certificate validation process the server also maintains a simple implementation of an Online Certificate Status Protocol (OCSP) server, which checks if a received certificate has not been revoked, by searching the given certificate on a blacklist and returning the state. This implementation only serves the purpose of demonstrating how an OSCP protocol could basically be realized. You are not requested to change anything in this code for the practical exercise.

Additionally to the message system *J* task, we have implemented a simple client-server protocol, designed to check if a given resource is valid or not. The client asks the server if a given resource, identified by a specific identifier, is still valid. The server checks this and responds with a simple yes (1)/no (0) answer. You do not know how the client and the server work, but you do have control over the transmission channel. This allows you to observe and also change the communication.

3 Software requirements and setup

The reference platform for this exercise is the VMware[2] image, which can be found in the download section ¹ of the course website. The reference image is based on Xubuntu(Precise Pangolin 12.04 LTS) Linux with Apache Tomcat 7.0.12, Oracle JAVA SDK 1.7.0_40, MySQL 5.5.32, lemon 3.7.9, gperf 3.0.3, clang 3.0.6 and CUnit 2.1.2(unstable) and is shipped with a pre-installed Eclipse(Kepler).

△ If you prefer to use your own setup, bear in mind that we can and will only support the reference platform virtual appliance. We can give no guarantees about the compatibility of the software used in this assignment with non-reference platform setups.

In the remainder of this document we assume that you are using our VMware reference image. Unless explicitly noted otherwise, all command line examples shown here are to be run inside the virtual environment of the reference image and *not on your host computer*.

3.1 Updating your repository

The source code for this assignment is distributed as a GIT patch against the WS tag in your repositories. To integrate the updates for this task into your repository you simply have to download the patch-file <https://bigfiles.iaik.tugraz.at/get/a2cb6aaf4108da2b20b18ee6cc42d3e8> and merge it into your local working copy. To merge the changes for this assignment run the following commands in your working directory:

```
# Download the patch-file for task J into your home-directory
sase@SASE-reference-image:~$ wget -O ~/assignment-j.patch \
https://bigfiles.iaik.tugraz.at/get/a2cb6aaf4108da2b20b18ee6cc42d3e8

# Switch to your local working copy.
sase@SASE-reference-image:~$ cd ~/<your_sase_repo>

# Merge the patch into your working directory using "git am".
sase@SASE-reference-image:~/<your_sase_repo>$ git am ~/assignment-j.patch
```

3.2 Implementation details

The Applied Cryptography and PKI assignment consists of two separate sub-tasks. First you have to implement a client-server communication part consisting of following packages:

- Client sending binaries to a server:
at.iaik.teaching.sase.ku2013.client.Client
- Server receiving the binaries:
at.iaik.teaching.sase.ku2013.server.Server
- Codlet processing the binaries:
at.iaik.teaching.sase.ku2013.codlet.Codlet
- OCSF server/client for certificate validation:
at.iaik.teaching.sase.ku2013.ocspmini.*
- Cryptographic library:
at.iaik.teaching.sase.ku2013.crypto

The second task provides you with a client and a server communicating with each other, using our own protocol, to obtain the state of a resource:

¹http://www.iaik.tugraz.at/content/teaching/master_courses/sicherheitsaspekte_in_der_softwareentwicklung/practicals/downloads/

- Simple protocol:
`at.iaik.teaching.sase.ku2013.protocolAnalysis.*`
- Cryptographic library:
`at.iaik.teaching.sase.ku2013.crypto`

3.3 Running the applications

- The client-server application can be run from Eclipse, by right-clicking the `client.Client` class and choosing *Run As > Java Application*.
- The protocol analysis task can be run in the same way by right-clicking the `ProtocolAnalysis.Validator` class and choosing *Run As > Java Application*.

☞ *If you prefer to use command line tools, we have also prepared an ant build file. For usage see 3.8.*

3.4 External libraries

For this assignment your submission *SHOULD NOT* use any external Java libraries, except those already installed by the assignment patch. These are located in the `/lib` folder within the Eclipse project. It must be possible to successfully compile and run your programs on the VMware reference image! For details on how you can find out if your submission compiles on our submission system, see 4.3.

3.5 Java cryptography

Java cryptography primitives and X.509 certificate handling is provided by version 5.01 of IAIK's JCE library. A copy of the non-commercial version of this library is included in the source package for this assignment.

The IAIK JCE library implements a standard Java JCE provider, which seamlessly integrates with the `java.security` and `javax.crypto` packages. In addition IAIK's JCE exposes a rich set of classes for handling X.509 public key certificates (`iaik.x509` package in IAIK's JCE).

Complete documentation for the IAIK JCE library can be found at: http://javadoc.iaik.tugraz.at/iaik_jce/current/index.html

3.6 Creating test certificates

Keys and certificates can, in principle, be created with any tool or library which support the X.509 and PKCS#12 standards.

A very convenient way to create test certificates and keys is the graphical XCA² tool, which by itself is a front end for the OpenSSL command line tools. The test certificates found in the source package have been created with this tool and the "XCA database" is included in the examples directory. The graphical XCA tool can be installed on the VMware reference image by simply running:

```
sase@vm$ sudo apt-get install xca
```

²<http://xca.sourceforge.net/>

3.7 JUnit

To help you test your implementation we added a set of JUnit tests, that will perform some basic checks on the cryptographic functions you implemented. The tests can be found in the

- `at.iaik.teaching.sase.ku2013.test`

package and can be run by right-clicking one of the test classes or the whole package and choosing:

- *RUN AS > JUnit Test.*

⚠ Please be aware that passing all of those checks does not automatically mean that your implementation is fully correct, they only provide testcases for the minimal functional requirements of this task.

3.8 Ant

In the root directory of the Eclipse project you will also find a `build.xml` file for running your implementation on the command line. The possible targets are:

- `build` – Builds the whole project.
- `run-client` – Starts the client/server communication application.
- `run-validator` – Starts the revocation check application.
- `junit` – Runs all available JUnit tests.

⚠ Please make sure, that your submission can be built using ant without any errors!

4 Submission

The submission for this tasks consists of all source files, which are required to compile your solutions to the tasks discussed in section 5. Your solutions *MUST* be compilable with the unmodified `build.xml` ANT build-script from the assignment source-patch.

Your submission should at least contain:

- All Java source files required to build your submission. These source files *MUST* reside below the `j/src/` directory of your repository. They *MUST* be compilable with the unmodified `build.xml` file from the assignment source patch.

4.1 Creating the submission branch

The submission itself is done via a separate `submission-j` branch in your GIT repository. To create this branch on the server, you need to execute the following steps *once*:

```
sase@vm:~/<your sase repo>$ git checkout -b submission-j
sase@vm:~/<your sase repo>$ git push -u origin submission-j
```

The “checkout” command (with `-b` option) first creates a new local branch in your local repository. The “push” command then publishes your local branch on the server. The `-u` option to the “push” command tells Git to setup tracking for your new branch.

Your group partner needs to setup a local copy of the submission branch as well. To do so, run the following steps in your group partner’s local Git repository *once* after you have pushed the submission branch to the server:

```
sase@vm:~/<partner's sase repo>$ git pull
sase@vm:~/<partner's sase repo>$ git checkout -t -b submission-j remotes/origin/submission-j
```

The “pull” command in your colleague’s repository fetches changes from the server. The “checkout” command creates a local “submission-j” branch (-b option), which is configured to track (-t option) the “submission-j” branch on the server.

You can use the normal `git push` and `git pull` commands, after you and your colleague have finished the one-time setup of the submission branch.

☞ We strongly recommend to read <http://git-scm.com/book/en/Git-Branching-Remote-Branches> if you have not worked with GIT (remote) branches before.

4.2 Working with the submission branch

Once you and your colleague have configured the submission branch, you can work with it like any other Git branch. You can directly commit to the submission branch via “git commit” or merge changes from (local) working branches via “git merge”.

4.3 Automated compilation

For this task, we will start to use an automated build system, to regularly compile your submissions in our reference environment.

Our build bot tries to checkout your submission branch using the steps discussed in section 4.4. On success, the bot then tries to build your submission using our reference build environment. The commit considered by the build bot will be tagged with a Git tag named `iaik/buildbot/j/XX` where `XX` increases with each scheduled build.

You can fetch the tags generated by the build-bot using either `git pull` or `git fetch --tags`. The build history of your submission branch is accessible via the `git tag -l` command:

```
sase@vm:~/<your sase repo>$ git fetch --tags
sase@vm:~/<your sase repo>$ git tag -n3 -l iaik/buildbot/j/*
iaik/buildbot/j/1 This commit was built successfully by IAIK's Build-Bot
  Build-Schedule: Wednesday, October 30, 2013 2:58:43 PM CET
  Build-Status: SUCCESS
iaik/buildbot/j/2 This commit was built successfully by IAIK's Build-Bot
  Build-Schedule: Monday, November 4, 2013 11:03:06 AM CET
  Build-Status: SUCCESS
...
```

4.4 Verifying your submission

You can easily verify your submission by cloning a fresh copy of your repository into a new directory, and trying to check out the submission-j branch. The Git commands used by our build system to clone your submissions are semantically equivalent to:

```
$ git clone git@teaching.student.iaik.tugraz.at:sase2013gXX.git
$ cd sase2013gXX
$ git fetch origin
$ git checkout -b submission-j origin/submission-j
```

Our build-system automatically tags the commit considered as final submission, with an `iaik/submission/j` tag, once the submission deadline has passed. We will not consider any commits after the `iaik/submission/j` tag for this task.


4.5 Checklist

Here is a checklist with all steps required to successfully complete this task:

- ☐ Download the source-patch for this task from the IAIK website and apply it to your GIT repository as discussed in section 3.1.
- ☐ Create and setup your submission branch as discussed in section 4.1.
- ☐ Solve the tasks discussed in section 5 and commit the code of your solutions to your GIT repository.
- ☐ Checkout your local `submission-j` branch and ensure, that everything you want to submit is merged. Ensure that your code can be compiled with the unmodified `build.xml` from the assignment source patch.
- ☐ Push your `submission-j` branch to our servers. You can push the submission branch as often as you like. Any changes after the submission deadline will be ignored.
- ☐ Verify that your submission worked, using the steps discussed in 4.4 to verify, that you pushed the correct code to the server.


4.6 Deadline

HARD SUBMISSION DEADLINE: December 4, 2013 (23:59:59 (CET))


 *We highly recommend to submit your solution at least one or two days before the ultimate submission deadline. Last minute submissions are always risky with respect to unforeseen technical difficulties.*

5 Tasks

This section discusses the practical tasks to be done as part of the Java cryptography and PKI assignment. There is no theoretical part for this assignment.

 *The classes in the `...sase.ku2013.test` package define the minimal functional requirements of the cryptography and simple certificate handling functions.*

You may change or adapt any parts of the Java source code, as long your program remains compatible with the existing classes in the `...sase.ku2013.test` Java packages.

 *Most parts of the J system are already implemented in the files supplied with the source-code patch. Parts which need to be implemented as part of this assignment are explicitly marked with “TODO” code comments. The informative “BEGIN/END STUDENT CODE” markers should help you to identify the locations which need changes.*

5.1 (1.0 points) Symmetric encryption

Correctly implement the skeleton functions for encrypting and decrypting data packets in the `CryptoToolBox` class.

`CryptoToolBox.aesCcmEncrypt` encrypts an arbitrary length block of data with the Advanced Encryption Standard (AES) cipher in counter mode with CBC-MAC (CCM).

`CryptoToolBox.aesCcmDecrypt` decrypts an encrypted block of data with the AES cipher in CCM mode.

5.2 (1.0 points) Keyed-Hashed Message Authentication Codes (HMACs)

Correctly implement the skeleton functions for computing and verifying HMACs in the `CryptoToolBox` class.

`CryptoToolBox.computeHmac` computes the HMAC-SHA1 message authentication code of a given byte array slice.

`CryptoToolBox.verifyHmac` verifies the integrity of a message by comparing a given (expected) HMAC with the HMAC computed over the message data.

5.3 (2.0 points) RSA signature generation and verification

Correctly implement the skeleton functions found in the `CryptoToolBox` class for signing data-blocks with the RSA signature algorithm.

`CryptoToolBox.packAndSignBlob` digitally signs a byte blob using the RSA signature algorithm with SHA-1 as digest and PKCS#1 padding.

`CryptoToolBox.verifyBlobAndUnpack` verifies the RSA signature on a signed blob and returns the payload data of the blob on success.

5.4 (2.0 points) Session key encipherment with RSA

Correctly implement the skeleton functions found in the `CryptoToolBox` class for encrypting (wrapping) secret session keys (e.g. AES or HMAC keys) with the RSA encryption algorithm.

`CryptoToolBox.wrapKey` encrypts a secret session key using RSA in ECB mode with PKCS#1 padding. The public key of the receiver is used for encryption. The function returns a byte-blob which can be exchanged over an external channel.

`CryptoToolBox.unwrapKey` decrypts a wrapped key blob using an RSA private key and returns the extracted secret session key on success.

5.5 (8.0 points) Simple certificate validation

Extend the `Certificates.checkUsageInChain` method to validate given certificate chains. This method receives an ordered array of `X509Certificates` with the leaf (user) certificate at position zero, and the root certificate at the last array position. These checks are based on, but not fully consistent with, the PKIX RFC 5280 [1] standard. Indeed, the full standard would be much too complicated to be implemented in this exercise. In case of our directives deviating from the standard, our directives take precedence!

Your implementation must check that:

- The root certificate is a trusted certificate.
- Each certificate in the chain (including the root) has a valid signature.
- Each certificate is within its validity period at the time of calling the method.
- A `BasicConstraints` extension is present and marked as critical.
- The `BasicConstraints` extension properly reflects the usage as CA or non-CA certificate.
- Check the path length constraints found in `BasicConstraints` for CA certificates.

- A `KeyUsage` extension is present and matches the intended key usage. See the TODOs in the `IntendedUsage` enumeration class for more details.
- No unhandled critical certificate extensions are present in any of the certificates. The only allowed critical extensions are `BasicConstraints`, `KeyUsage` and `ExtendedKeyUsage`. The `ExtendedKeyUsage` extension may be missing.
- No certificate in the chain has been revoked. Use the `Certificates.isRevoked` method to invoke the `OCSPMiniClient` for performing this check.

5.6 (6.0 points) Cross certification

When constructing a certificate chain, it is necessary to repeatedly match the issuer name of a child certificate with the subject name of its issuer certificate. Commonly, certificates have exactly one issuer (parent) certificate, implying that a single unique path to a root CA certificate can be constructed. In this case all certificates form a tree.

With cross-certification it is possible that more than one certificate exists for the same subject-name/public key combination. This situation occurs, for example, when an intermediate CA *C* or end-user *C* asks two distinct CAs *A* and *B* to sign the certificate of *C*. *C* now gets two distinct certificates for the same subject name “*C*” and public key.

For any certificates signed by CA *C* we now have two distinct certificate chains, one ending at CA *A* and one ending at CA *B* - the certificates now form a directed graph instead of a tree.

To support cross-certification you have to extend the `Certificates.getChains` method. Currently, the method fails if more than one possible certificate chain exists for a given leaf certificate. Your new implementation should instead construct all possible valid chains starting at the given leaf certificate and ending in a root CA certificate. Note that there may be more than one “fork” on the way to the root certificate. Also be aware of infinite loops.

For simplicity reasons, we require that all certificate chains found by the `Certificates.getChains` method are valid. The existing implementation of `Certificates.checkUsage` acknowledges this requirement.

5.7 (5.0 points) Protocol Analysis

The `protocolAnalysis` package consists of four distinct implementations of a client sending a request to a server, asking if a given resource is valid (*1*) or not (*0*). For details on how to select one of the four client-server configurations see below. You do not know the implementation of those classes, but since the developers did not care about securing the communication, you are able to intercept and modify the communication in all four cases. This is the typical assumption for a man-in-the-middle attacker. Thus, your task now, is to attack the communication. The client will request the status of a specific resource from the server. The server will respond with an invalid statement (*0*). You have to change the response in a way that the client believes the resource to be valid. For each of the following 4 cases modify the server's response on the channel to make the client accept even invalid resources as valid.

PlainBlackBox For this task, modify the `protocolAnalysis.PlainChannel` class. This client-server implementation (`PlainBlackBox`) uses no channel security mechanism.

SignatureBlackBox Modify the `protocolAnalysis.SignatureChannel` class. This client-server implementation (`SignatureBlackBox`) tries to implement origin integrity using a digital signature, but fails.

BetterSignatureBlackBox Modify the `protocolAnalysis.BetterSignatureChannel` class. This client-server implementation (`BetterSignatureBlackBox`) tries to do origin integrity right, by using the digital signature more intelligently, but still fails.

EvenBetterSignatureBlackBox Modify the `protocolAnalysis.EvenBetterSignatureChannel` class. This client-server implementation (`EvenBetterSignatureBlackBox`) learns from the previous fails, but is still beatable.

☞ You can switch between the different implementations, by uncommenting the appropriate blackbox line in the `validator.properties` file. If you just want to see some traffic between the server and the client, set the `validator.run.multiple` property to **true**. If you want to perform your attack and do not need that much output, set it to **false**.

References

- [1] *PKIX RFC 5280*. <http://www.ietf.org/rfc/rfc5280.txt>.
- [2] *VMWare*. <http://www.vmware.com/>.
- [3] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.