Digital Signatures for PDF documents

A White Paper by Bruno Lowagie (iText Software)

Introduction

The main rationale for PDF used to be viewing and printing documents in a reliable way. The technology was conceived with the goal "to provide a collection of utilities, applications, and system software so that a corporation can effectively capture documents from any application, send electronic versions of these documents anywhere, and view and print these documents on any machines." (Warnock, 1991)

Why we need PDF

This mission was set forth in the Camelot paper, and it was accomplished with the first publication of the Portable Document Format Reference (Adobe, 1993) and the availability of the first PDF software products created by Adobe. PDF became renowned as the format that could be trusted to ensure a consistent output, be it on screen or in print.

In the years that followed, an abundance of new tools from Adobe as well as from third party software vendors emerged, and the PDF specification was —and still is— very much alive. Plenty of functionality has been added to the PDF format over the years. Because of this, PDF has become the preferred document format of choice in many professional sectors and industries.

In this paper we'll focus on one specific aspect of PDF files that makes the choice for PDF over any other document format a no-brainer: digital signatures.

Why we need digital signatures

Imagine a document that has legal value. Such a document may contain important information about rights and obligations, in which case you need to ensure its authenticity. You don't want people to deny the commitments they've written down. Furthermore, this document probably has to be mailed to, viewed and stored by different parties. On different places in the workflow, at different moments in time, the document can be altered, be it voluntary, for instance to add an extra signature, involuntary, for example due to a transmission error, or deliberately, if somebody wants to create a forgery from the original document.

For centuries, we've tried to solve this problem by putting a so-called 'wet ink signature' on paper. Nowadays, we can use digital signatures to ensure:

- *the integrity of the document*— we want assurance that the document hasn't been changed somewhere in the workflow,
- *the authenticity of the document*— we want assurance that the author of the document is who we think it is (and not somebody else),
- *non-repudiation*— we want assurance that the author can't deny his or her authorship.

In this paper, we'll focus on documents in the portable document format (PDF).

Preface

I wrote my first book in 2005-2006. A sequal was published in 2010. Both books were published by Manning Publications. I really liked working with a Publisher. As an author, you become part of a whole team of editors, reviewers and proof readers ensuring the quality of your work. It was great working with the fantastic team at Manning Publications.

But there's also a downside. When you write a book for a Publisher, you're no longer free to do what you want with your own work. In many cases, I felt sorry that I couldn't just send somebody the ebook to explain how to solve a specific issue. That's why I decided to write this white paper without a Publisher, even after it became clear that what started as merely a white paper was slowly changing into a real book. Because of that decision, I can choose to distribute this book for free.

To assure the quality of the book, I counted on the goodwill of a number of selected reviewers. One early-access reader helped me out by giving me access to a Hardware Security Module; another one sent me prototypes of smart card readers that weren't on the market yet. To all the people who helped me: I'm really grateful for your support!

Acknowledgments

I want to thank Paulo Soares (Glintt) for redesigning the API for signing, making it ready for PDF 2.0 (ISO-32000-2), Michael Klink (Authentidate) for being an excellent technical editor, Leonard Rosenthol (Adobe) for clarifying the PDF specification when needed, Danny Decock (KULeuven), Bart Hanssens (FedICT) and Frank Cornelis (FedICT) for their insights on the eID, and William Alexander Segraves (a long-time friend) for giving me advice about writing American English.

Special thanks go to GlobalSign and VASCO. GlobalSign's Steve Roylance and Paul van Brouwershaven provided information regarding the role of a Certificate Authority, as well as hardware to test with, such as a USB token with a GlobalSign certificate. The people from VASCO sent me three different models of their smart card readers to experiment with. Thank you very much!

I wouldn't have been able to write this paper without our valued customers. I want to thank Louis Collet from the Belgian Federal Ministry of Finances, Luc Verschraegen and Petra Claeys from Universiteit Gent, and many other project leaders and developers from other companies who encouraged me to write this paper so that they could implement an iText-driven digital signature solution.

Big thanks also to the iText development team. Let's start with the people who did an internship at iText Software: Jeroen Nouws wrote a first Proof of Concept for the eID examples; Bart Verbrugghe tested the eID code and wrote Proof of Concepts extracting information about signatures; Jens Ponnet wrote a Proof of Concept demonstrating how to sign a document on Android. Jens is an employee at iText Software now. He joins our team consisting of Alexander Chingarev, Michaël Demey, Raf Hens, Denis Koleda, and Eugene Markovskyi.

Finally a big thanks to the other staff members at iText Software, Michael Bradbury, Kevin Brown, Chris Heuer, and Garry Vleminckx in sales, Valérie Hillewaere who is responsible for marketing, our COO Frank Gielen, and last but not least, my wife the CFO, Ingeborg Willaert.

Table of Contents

1. Understanding the concept of digital signatures

Let's start by taking a look at a PDF file, and identify possible issues regarding document integrity.

1.1 A simple PDF example

Figure 1.1 shows a simple PDF document, containing nothing but the words "Hello World".

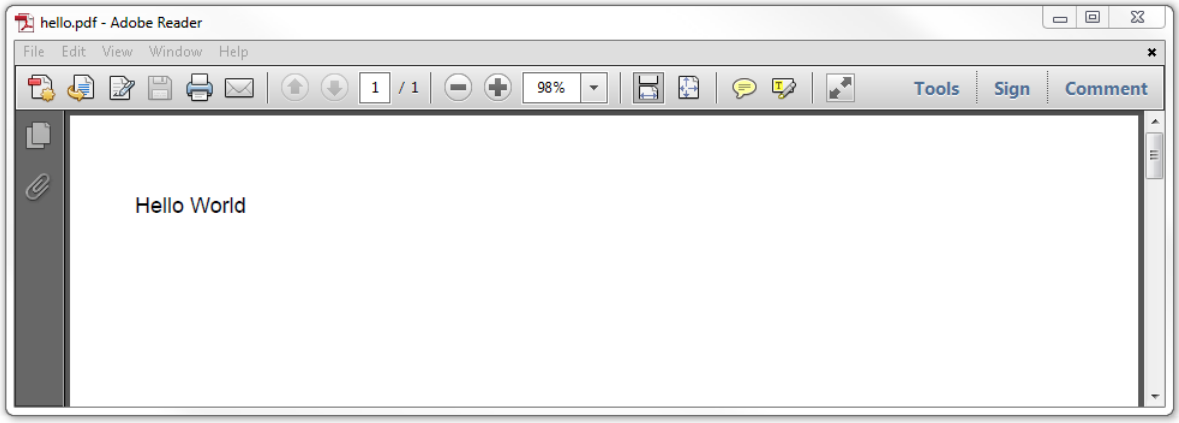

Figure 1.1: A simple Hello World file

If we look inside the PDF file shown in figure 1.1, we see the following PDF syntax:

Code sample 1.1: A PDF file inside-out

```
%PDF-1.4
%âãÏÓ
2 0 obj
<</Length 73 >>stream
BT
36 806 Td
0 -18 Td
/F1 12 Tf
(Hello World)Tj
0 0 Td
ET
Q
endstream
endobj
4 0 obj
<</Parent 3 0 R/Contents 2 0 R/Type/Page/Resources<</ProcSet [/PDF /Text /ImageB
/ImageC /ImageI]/Font<</F1 1 0 R>>>>/MediaBox[0 0 595 842]>>
endobj
1 0 obj
<</BaseFont/Helvetica/Type/Font/Encoding/WinAnsiEncoding/Subtype/Type1>>
endobj
3 0 obj
<</ITXT(5.3.0)/Type/Pages/Count 1/Kids[4 0 R]>>
endobj
5 0 obj
<</Type/Catalog/Pages 3 0 R>>
endobj
6 0 obj
```

```
<</Producer(iText® 5.3.0 ©2000-2012 1T3XT
BVBA)/ModDate(D:20120613102725+02'00')/CreationDate(D:20120613102725+02'00')>>
endobj
xref
0 7
0000000000 65535 f
0000000311 00000 n
0000000015 00000 n
0000000399 00000 n
0000000154 00000 n
0000000462 00000 n
0000000507 00000 n
trailer
<</Root 5 0 R/ID
[<0f6bb651c0480213fbbd13449b40fe8f><e77fb3c3c64c30ea2a908cd181c5f500>]/Info 6 0
R/Size 7>>
startxref
643
%%EOF
```

Every PDF file starts with `%PDF-` followed by a version number, and it ends with `%%EOF`. In-between, you can find different PDF objects that somehow define the document. Explaining the meaning of all the objects shown in code sample 1.1 is outside the scope of this paper.

1.1.1 How to forge the content of a PDF document

Suppose we know how to apply some small changes to the file. For instance: let's change the content, the dimensions, and the metadata of the document. See the parts marked in red in Code sample 1.2.

Code sample 1.2: A manually altered PDF file

```
%PDF-1.4
%âãÏÓ
2 0 obj
<</Length 73 >>stream
BT
36 806 Td
0 -18 Td
/F1 12 Tf
(Hello Bruno)Tj
0 0 Td
ET
Q
endstream
endobj
4 0 obj
<</Parent 3 0 R/Contents 2 0 R/Type/Page/Resources<</ProcSet [/PDF /Text /ImageB
/ImageC /ImageI]/Font<</F1 1 0 R>>>>/MediaBox[0 0 120 806]>>
endobj
1 0 obj
<</BaseFont/Helvetica/Type/Font/Encoding/WinAnsiEncoding/Subtype/Type1>>
endobj
3 0 obj
<</ITXT(5.3.0)/Type/Pages/Count 1/Kids[4 0 R]>>
endobj
5 0 obj
<</Type/Catalog/Pages 3 0 R>>
```

```
endobj
6 0 obj
<</Producer(iText® 1.0.0 ©2000-2012 1T3XT
BVBA)/ModDate(D:20120613102725+02'00')/CreationDate(D:20120613102725+02'00')>>
endobj
xref
0 7
0000000000 65535 f
0000000311 00000 n
0000000015 00000 n
0000000399 00000 n
0000000154 00000 n
0000000462 00000 n
0000000507 00000 n
trailer
<</Root 5 0 R/ID
[<0f6bb651c0480213fbbd13449b40fe8f><e77fb3c3c64c30ea2a908cd181c5f500>]/Info 6 0
R/Size 7>>
startxref
643
%%EOF
```

I manually replaced the word 'World' with 'Bruno', I changed the dimensions of the page from 595 x 842 to 120 x 806, and I changed the version number of iText in the producer line. Figure 1.2 shows the result.

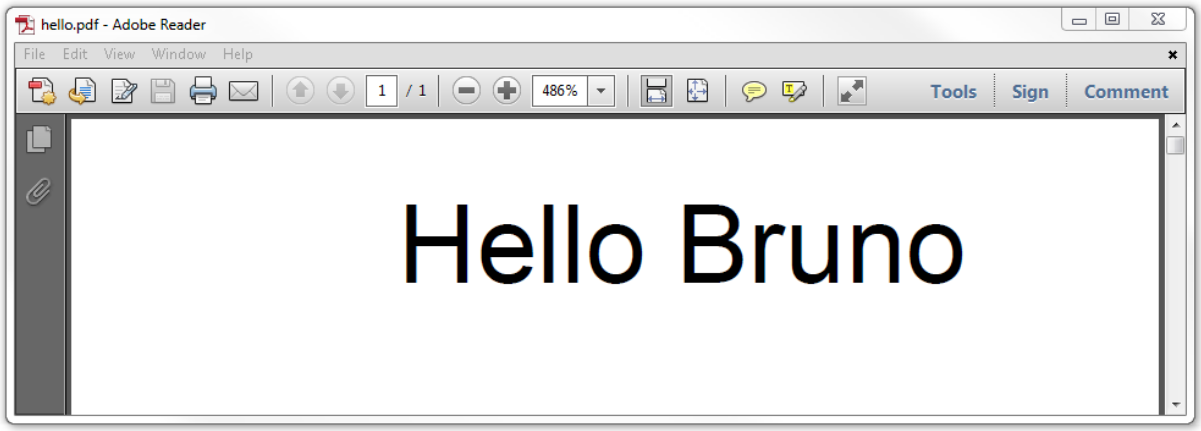

Figure 1.2: an altered Hello World file

Don't ever do this yourself! Changing a PDF manually will corrupt your file in 99.9% of the cases. I'm only doing this to prove that, although PDF isn't a word processing format, although PDF isn't meant for editing a document and although it's not recommended to do so, you can change the content of a document. This is exactly what we try to avoid by introducing a digital signature.

1.1.2 A digitally signed PDF document

Figure 1.3 shows a Hello World document that has been digitally signed. The blue banner tells us the document is *"Signed and all signatures are valid."* The Signature panel informs us that the file was *"Signed by Bruno Specimen"*, and it gives you more signature details.

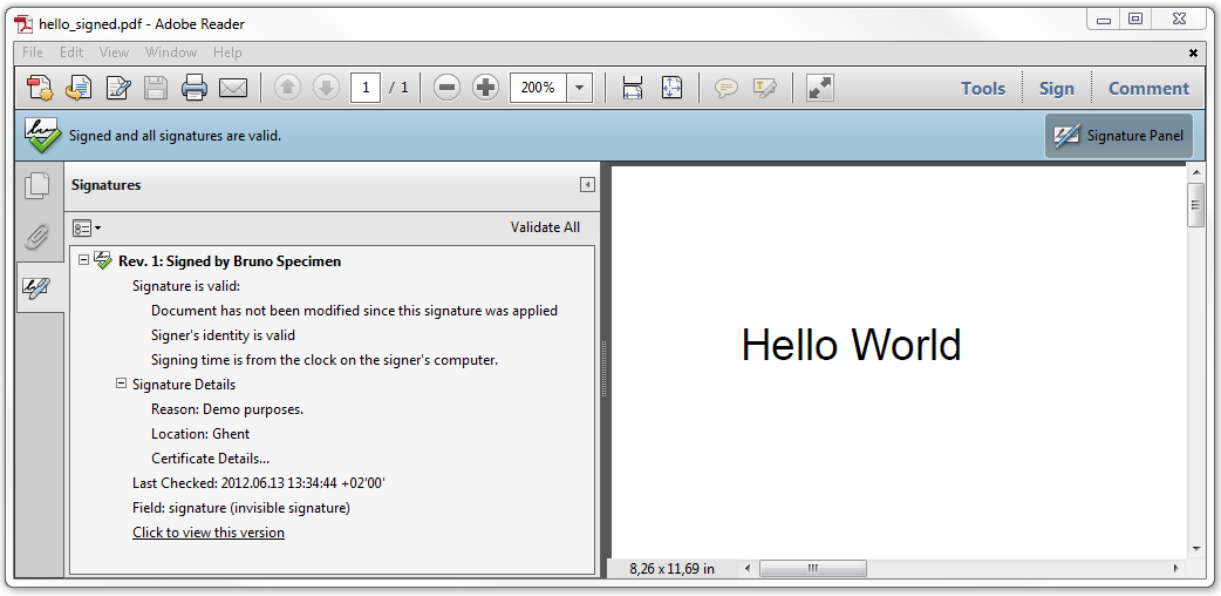

Figure 1.3: a signed Hello World file

The green check mark means that the Document hasn't been modified since the signature was applied and that the Signer's identity is valid.

1.1.3 Inspecting the syntax of the digital signature

Now let's take a look inside this PDF file.

Code sample 1.3: snippets of a signed PDF file

```
%PDF-1.4
%âãÏÓ
3 0 obj
<</F 132/Type/Annot/Subtype/Widget/Rect[0 0 0 0]/FT/Sig
/DR<<>>/T(signature)/V 1 0 R/P 4 0 R/AP<</N 2 0 R>>>>
endobj
1 0 obj
<</Contents <0481801e6d931d561563fb254e27c846e08325570847ed63d6f9e35 ... b2c8788a5>
/Type/Sig/SubFilter/adbe.pkcs7.detached/Location(Ghent)/M(D:20120928104114+02'00')
/ByteRange [0 160 16546 1745 ]/Filter/Adobe.PPKLite/Reason(Test)/ContactInfo()>>
endobj
...
9 0 obj
<</Length 63>>stream
q
BT
36 806 Td
0 -18 Td
/F1 12 Tf
(Hello World!)Tj
0 0 Td
ET
Q
endstream
endobj
...
11 0 obj
<</Type/Catalog/AcroForm<</Fields[3 0 R]/DR<</Font<</Helv 5 0 R
/ZaDb 6 0 R>>>>/DA(/Helv 0 Tf 0 g )/SigFlags 3>>/Pages 10 0 R>>
```

```
endobj
xref
0 12
0000000000 65535 f
...
0000017736 00000 n
trailer
<</Root 11 0 R/ID
[<08ed1afb8ac41e841738c8b24d592465><bd91a30f9c94b8facf5673e7d7c998dc>]/Info 7 0
R/Size 12>>
startxref
17879
%%EOF
```

Note that I've slightly altered the file, removing bytes that aren't relevant when explaining the concept of a digital signature.

First let's inspect the root object of the PDF (aka the Catalog object). It's marked in green in code sample 1.3 (the object with number 11). The Catalog is always represented as a PDF dictionary. Dictionaries can easily be recognized in a PDF file. They start with << and end with >>. In-between you'll find a series of key-value pairs. The key is always a name object. Note that names always start with a /. For example: if the PDF contains a form, you'll find an /AcroForm key in the catalog dictionary. Its value will be (a reference to) a dictionary. In turn, this dictionary will contain a /SigFlags value if the form contains a digital signature.

There's one field in the form. It's referred to from the /Fields array: see object 3 (marked in red). The field named "signature" (/T(signature)) is a field of type signature (/FT/Sig). We didn't see any visual representation of the signature in figure 1.3. That's because Bruno Specimen decided to use an invisible signature. The rectangle (/Rect) defining the widget annotation (/Type/Annot /SubType/Widget) has a zero width and zero height ([0 0 0 0]).

The actual signature can be found in the signature dictionary (marked in blue). This dictionary is referred to from the value (/V) of the signature field. The signature is the value of the /Contents entry. This signature covers all the bytes of the PDF file, except for the signature bytes itself.

> **NOTE:** When using the word *signature*, people may mean different things. In the pure technical sense, a signature in a PDF file consists of the bytes stored in the /Contents entry. However, we also use the word when we refer to the representation of the signature on the page, or even when talking about the complete signature infrastructure in the file (the annotation and the signature dictionary). I'll try to be as precise as possible in this paper, but in some cases, the exact meaning should be clear from the context.

See the /ByteRange entry: the signature covers bytes 0 to 160 and bytes 16546 to 18291. The signature itself takes bytes 161 to 16545.

1.1.4 Making the signature invalid

Now when I change one of the bytes inside the byte range covered by the signature, Adobe Reader will show a red cross instead of a green check mark. Figure 1.4 shows what happens if I manually replace 'World' by 'Bruno'.

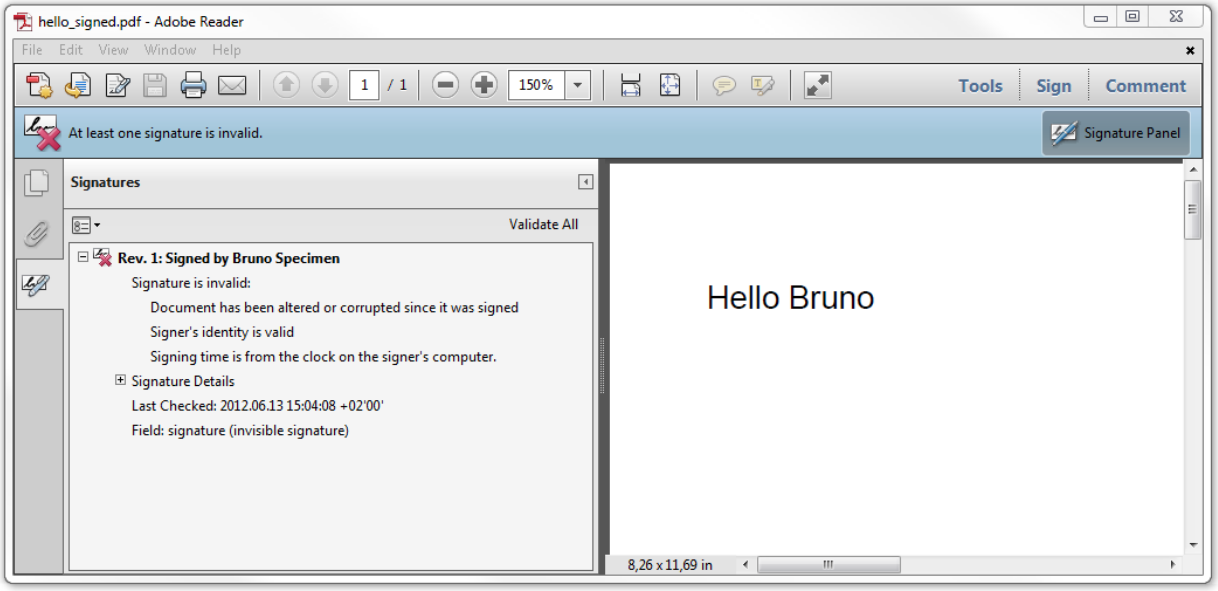

Figure 1.4: Invalidated signature

In this case, the Signer's identity is valid, but the *"Document has been altered or corrupted since it was signed."* How does Adobe Reader know that the document has been altered? To understand this, we need to understand the concept of hashing and we need to know how encryption works.

1.2 Creating a message digest

I have a confession to make: I can't remember all the passwords I'm using to log in into different sites such as Twitter, Facebook, LinkedIn, and so on. I am one of the frequent users of the *"Lost your password?"* functionality. Usually, I get a link that allows me to reset my password, but once in a while I get a mail containing my original password in clear text.

The fact that a service can provide me with my password, means that my actual password can be found somewhere in a database, on a server. That's a dangerous situation: it means that whoever hacks the system can obtain the passwords of all the users.

1.2.1 How to check a password

A simple way to check a password is to store a digest of the password instead of the actual password. Let's create a simple class that demonstrates how it's done:

Code sample 1.4: an example showing how to use the MessageDigest class

```
public class DigestDefault {
    protected byte[] digest;
    protected MessageDigest md;

    protected DigestDefault(String password, String algorithm, String provider)
        throws GeneralSecurityException {
        if (provider == null)
            md = MessageDigest.getInstance(algorithm);
        else
            md = MessageDigest.getInstance(algorithm, provider);
        digest = md.digest(password.getBytes());
    }

    public static DigestDefault getInstance(String password, String algorithm)
```

```
        throws GeneralSecurityException {
        return new DigestDefault(password, algorithm, null);
    }

    public int getDigestSize() {
        return digest.length;
    }

    public String getDigestAsHexString() {
        return new BigInteger(1, digest).toString(16);
    }

    public boolean checkPassword(String password) {
        return Arrays.equals(digest, md.digest(password.getBytes()));
    }

    public static void showTest(String algorithm) {
        try {
            DigestDefault app = getInstance("password", algorithm);
            System.out.println("Digest using " + algorithm + ": "
                    + app.getDigestSize());
            System.out.println("Digest: " + app.getDigestAsHexString());
            System.out.println("Is the password 'password'? "
                    + app.checkPassword("password"));
            System.out.println("Is the password 'secret'? "
                    + app.checkPassword("secret"));
        } catch (NoSuchAlgorithmException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Take a look at the `showTest()` method in code sample 1.4. We create a `DigestDefault` object named `app`, passing a password "`password`" and an algorithm as parameters. An instance of the `java.security.MessageDigest` object is created and stored as a member variable. The password isn't stored in the `DigestDefault` class. Instead we store the result of the `digest()` method. Out of curiosity, we send some info about this result to the `System.out`: its length and its value as a hexadecimal `String`.

When we test the password with the `checkPassword()` method, we use the same `digest()` method, and we compare the result with the digest that was stored. If the result is identical, the password was correct; otherwise it was wrong. How does this work? What is a digest algorithm?

1.2.2 What is a digest algorithm?

When we create a digest, we're using a cryptographic hash function to turn an arbitrary block of data into a fixed-size bit string. The block of data is often called the 'message', and the hash value is referred to as the 'message digest'. In the previous example, the message was a password. In the context of PDF documents, the block of data could be the byte range of a PDF file. The result of this hash function is always identical, provided the message hasn't been altered. Any accidental or intentional change to the data will result in a different hash value. It's also a one-way algorithm.

When you create a hash of a password, it shouldn't be possible to calculate the password based on the message digest. If a database is compromised, the hacker has a collection of message digests, but he can't convert this to a list of passwords.

> **NOTE:** If a hacker obtains a database containing hashes of passwords, he can still use brute force to recover passwords. Also some older hashing algorithms have flaws that can be exploited. Nowadays, at least salting the hash and applying multiple iterations is necessary for security, but that's outside the scope of this paper.

When making a hash, you can choose among different algorithms. Code sample 1.5 shows some of them.

Code sample 1.5: testing different Hashing algorithms

```
public static void testAll() {
    showTest("MD5");
    showTest("SHA-1");
    showTest("SHA-224");
    showTest("SHA-256");
    showTest("SHA-384");
    showTest("SHA-512");
    showTest("RIPEMD128");
    showTest("RIPEMD160");
    showTest("RIPEMD256");
}
```

MD5, SHA, and RIPEMD are implementations of different cryptographic hash algorithms.

- *MD5*— one in a series of message digest algorithms designed by Professor Ron Rivest of MIT. It was designed in 1991. The MD5 algorithm allows you to create a 128-bit (16-byte) digest of a message. In other words: there are 2^{128} possible hashes for an infinite possible number of messages, which means that two different documents can result in the same digest. This is known as 'hash collision'. The quality of a hash function is determined by how 'easy' it is to create a collision. MD5 is still widely used, but it's no longer considered secure.
- *SHA*— stands for Secure Hash Algorithm.
    - SHA-1 was designed by the US National Security Agency (NSA). The 160-bit (20 bytes) hash was considered safer than MD5, but in 2005 a number of security flaws were identified.
    - SHA-2 fixes these flaws. SHA-2 is a set of hash functions: SHA-224, SHA-256, SHA-384, and SHA-512. The length of the message digest in SHA-2 can be 224 bits (28 bytes), 256 bits (32 bytes), 384 bits (48 bytes), or 512 bits (64 bytes).
- *RIPEMD*— stands for RACE Integrity Primitives Evaluation Message Digest. It was developed at the Katholieke Universiteit Leuven[1]. In our example, we use a 128, 160 and 256 bit version (16, 20 and 32 bytes).

In section 2.1.1, we'll see that not all of these digest algorithms can be used in the context of PDF digital signatures, but let's not get ahead of ourselves.

[1] http://www.esat.kuleuven.be/

1.2.3 Java SDK's default MessageDigest implementation

If we look at the output of code sample 5, we discover that not all algorithms are supported by Oracle's implementation of the abstract `java.security.MessageDigest` class.

The output for "MD5" and "SHA-1" looks like this:

```
Digest using MD5: 16
Digest: 5f4dcc3b5aa765d61d8327deb882cf99
Is the password 'password'? true
Is the password 'secret'? false
Digest using SHA-1: 20
Digest: 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
Is the password 'password'? true
Is the password 'secret'? false
```

But the output for SHA-224 and the RIPEMD algorithms looks like this:

```
SHA-224 MessageDigest not available
RIPEMD128 MessageDigest not available
```

We can fix this by using another security provider.

1.2.4 The BouncyCastle library

Bouncy Castle is a collection of APIs used in cryptography[2]. It's available in both Java and C#. We could extend the class in code sample 1.4 by defining a `BouncyCastleProvider` instance (named "`BC`") as security provider. This is done in code sample 1.6.

Code sample 1.6: using a different crypto provider

```
public class DigestBC extends DigestDefault {
    public static final BouncyCastleProvider PROVIDER = new BouncyCastleProvider();
    static {
        Security.addProvider(PROVIDER);
    }
    protected DigestBC(String password, String algorithm)
        throws GeneralSecurityException {
        super(password, algorithm, PROVIDER.getName());
    }
    public static DigestBC getInstance(
        String password, String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException {
        return new DigestBC(password, algorithm);
    }
}
```

Note that we create a `BouncyCastleProvider` instance, and we add it to the `Security` class (in the `java.security` package). This class centralizes all security properties. One of its primary uses is to manage security providers.

Now the output for SHA-224 and RIPEMD128 looks like this:

```
Digest using SHA-224: 28
Digest: d63dc919e201d7bc4c825630d2cf25fdc93d4b2f0d46706d29038d01
Is the password 'password'? true
```

[2] http://www.bouncycastle.org/

```
Is the password 'secret'? false
Digest using RIPEMD128: 16
Digest: c9c6d316d6dc4d952a789fd4b8858ed7
Is the password 'password'? true
Is the password 'secret'? false
```

We've already mentioned that the message could be any block of bytes, including the bytes of a specific byte range of a PDF file. To detect whether or not a PDF file has been altered, we could create a message digest of those bytes and store it inside the PDF. Then when somebody changes the PDF, the message digest taken from the changed bytes in the byte range will no longer correspond with the message digest stored in the PDF. Does this solve our problems of data integrity, authenticity and non-repudiation?

Not yet. One could easily change the bytes of the PDF file, guess the digest algorithm that was used, and store a new message digest inside the PDF. To avoid this, we need to introduce the concept of encryption by means of asymmetric key algorithms.

1.3 Encrypting a message using a public-key encryption

Suppose that Bob wants to send Alice a private message[3], but Bob doesn't trust the channel used to transmit the message. If it falls in the wrong hands, Bob wants to avoid that the message meant for Alice's eyes only, can be read by anyone else.

Bob could use an algorithm to encrypt his message, and then Alice would need to use an algorithm to decrypt it. If the same key can be used for encryption as well as decryption (or if the key for decryption can be derived from the key used for encryption), Bob and Alice are using a symmetric key algorithm. The problem with such an algorithm is that Bob and Alice need to exchange that algorithm in a safe way first. Anybody with access to the key can decrypt your message.

To avoid this, Bob and Alice can choose an asymmetric key algorithm. In this case two different keys are used: one key to encrypt, the other one to decrypt the message. One key cannot be derived from the other. Now if Bob wants to send Alice a private message, Alice can send her public key to Bob. Anybody can use this key to encrypt messages meant for Alice. When Bob sends Alice such an encrypted message, Alice will use her private key to decrypt it. Suppose that Chuck intercepts the message, he won't be able to read it as long as he doesn't have access to Alice's private key.

The most commonly used public-key encryption system is RSA, named after Ron Rivest, Adi Shamir, and Leonard Adleman, its inventors. When you create a key pair, you can specify a key length in bits. The higher the number of bits, the stronger the encryption will be. There's also a downside: with every doubling of the RSA key length, decryption is 6 to 7 times slower.

Let's create such a public and private key pair.

1.3.1 Creating a key store

The Java SDK contains a tool named `keytool`. You can use this tool to create a key store. This key store will contain your private key, and a digital certificate containing information about you as well as your public key. See code sample 1.7 to find out how `keytool` is used.

[3] http://en.wikipedia.org/wiki/Bob_and_Alice

Code sample 1.7: creating a key store using keytool

```
$ keytool -genkey -alias demo -keyalg RSA -keysize 2048 -keystore ks
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Bruno Specimen
What is the name of your organizational unit?
  [Unknown]:  IT
What is the name of your organization?
  [Unknown]:  iText Software
What is the name of your City or Locality?
  [Unknown]:  Ghent
What is the name of your State or Province?
  [Unknown]:  OVL
What is the two-letter country code for this unit?
  [Unknown]:  BE
Is CN=Bruno Specimen, OU=IT, O=iText Software, L=Ghent, ST=OVL, C=BE correct?
  [no]:  yes
Enter key password for <demo>
        (RETURN if same as keystore password):
```

A key store can contain more than one private key, and we could define a password that is different from the key store password for every key. For the sake of simplicity, we used only one: "`password`". We can use this key store from our Java program to encrypt and decrypt messages.

1.3.2 Encrypting and decrypting messages

Code sample 1.8 shows a simple class containing an `encrypt()` and a `decrypt()` method:

Code sample 1.8: A simple class to encrypt and decrypt messages

```
public class EncryptDecrypt {
    protected KeyStore ks;

    public EncryptDecrypt(String keystore, String ks_pass)
        throws GeneralSecurityException, IOException {
        initKeyStore(keystore, ks_pass);
    }

    public void initKeyStore(String keystore, String ks_pass)
        throws GeneralSecurityException, IOException {
        ks = KeyStore.getInstance(KeyStore.getDefaultType());
        ks.load(new FileInputStream(keystore), ks_pass.toCharArray());
    }

    public X509Certificate getCertificate(String alias)
        throws KeyStoreException {
        return (X509Certificate) ks.getCertificate(alias);
    }

    public Key getPublicKey(String alias)
        throws GeneralSecurityException, IOException {
        return getCertificate(alias).getPublicKey();
    }

    public Key getPrivateKey(String alias, String pk_pass)
        throws GeneralSecurityException, IOException {
        return ks.getKey(alias, pk_pass.toCharArray());
```

```
    }

    public byte[] encrypt(Key key, String message)
        throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] cipherData = cipher.doFinal(message.getBytes());
        return cipherData;
    }

    public String decrypt(Key key, byte[] message)
        throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] cipherData = cipher.doFinal(message);
        return new String(cipherData);
    }

    public static void main(String[] args)
        throws GeneralSecurityException, IOException {
        EncryptDecrypt app =
            new EncryptDecrypt("src/main/resources/ks", "password");
        Key publicKey = app.getPublicKey("demo");
        Key privateKey = app.getPrivateKey("demo", "password");
        System.out.println("Let's encrypt 'secret message' with a public key");
        byte[] encrypted = app.encrypt(publicKey, "secret message");
        System.out.println("Encrypted message: "
            + new BigInteger(1, encrypted).toString(16));
        System.out.println("Let's decrypt it with the corresponding private key");
        String decrypted = app.decrypt(privateKey, encrypted);
        System.out.println(decrypted);
        System.out.println("You can also encrypt the message with a private key");
        encrypted = app.encrypt(privateKey, "secret message");
        System.out.println("Encrypted message: "
            + new BigInteger(1, encrypted).toString(16));
        System.out.println("Now you need the public key to decrypt it");
        decrypted = app.decrypt(publicKey, encrypted);
        System.out.println(decrypted);
    }
}
```

Feel free to experiment with this example, but don't panic if you get an `InvalidKeyException` saying that the key size is invalid. Due to import control restrictions by the governments of a few countries, the encryption libraries shipped by default with the Java SDK restrict the length, and as a result the strength, of encryption keys.

If you want to avoid this problem, you need to replace the default security JARs in your Java installation with the *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files*. These JARs are available for download from http://java.oracle.com/ in eligible countries.

1.3.3 Inspecting our self-signed certificate

As you can see, we create a `KeyStore` object in the constructor. From this key store, we can obtain the public certificate that contains the public key to be used to decrypt a message that was encrypted with the corresponding private key.

A public certificate also contains information about the owner of the key, its validity period, and so on. The information in this certificate is signed by an 'issuer'. In this case, we've created the certificate ourselves, so we're owner and issuer at the same time, and we refer to the certificate as being *self-signed*. We'll learn more about other certificates in chapter 3.

Code sample 1.9 shows the result of the `toString()` method of the `Certificate` object:

Code sample 1.9: an example of a public certificate

```
[
[
  Version: V3
  Subject: CN=Bruno Specimen, OU=IT, O=iText Software, L=Ghent, ST=OVL, C=BE
  Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

  Key:  Sun RSA public key, 2048 bits
  modulus:
27706646249437583578501322921252037659324960984454438650274096621513733947318221232
90092536075175589409888251417041849614639606544370595805501222639942552792696182924
19557917502293557528812483868420880765808333319067679184013346901221838396913865166
99015383461952441725262486245434952426855074038516834028858534816117097190264270919
71970499616689684012198665415564791592761123642686002605100319784405598279465396131
52730660815729426764990600604032553721917074418187300648866487699179740248069790221
86704383972995455717886346330217224211116969013795163606127880980836981725138593346
18582280371213412072225864232981019 3
  public exponent: 65537
  Validity: [From: Sat Aug 04 15:40:30 CEST 2012,
              To: Tue Nov 17 14:40:30 CET 2015]
  Issuer: CN=Bruno Specimen, OU=IT, O=iText Software, L=Ghent, ST=OVL, C=BE
  SerialNumber: [    501d264e]

]
  Algorithm: [SHA1withRSA]
  Signature:
0000: 12 ED EA 66 FE 6C 2C FC   0F F4 59 19 44 40 FE BF  ...f.l,...Y.D@..
0010: CF 9E 66 D3 DC 62 85 F1   D5 62 76 07 F6 F2 67 04  ..f..b...bv...g.
0020: E8 F6 61 42 02 F9 36 A9   8B 12 6F 8B 4B B6 14 9B  ..aB..6...o.K...
0030: 78 2F CA F0 53 76 41 F4   47 B7 5A 2B F7 A1 A9 73  x/..SvA.G.Z+...s
0040: 8E 44 55 31 14 D3 AB 3F   59 6C 53 E7 04 C4 2E 36  .DU1...?YlS....6
0050: DE 2C 1E F5 F9 E3 19 EF   7D 92 67 66 56 73 22 18  .,........gfVs".
0060: EC AF CB 86 22 B8 F0 D0   94 EC 37 97 D1 23 DA 43  ....".....7..#.C
0070: 98 8E 37 34 7E AD 76 78   99 63 21 0D 06 C3 1D 47  ..74..vx.c!....G
0080: 5D 21 0A A6 CD 57 70 C1   A4 23 5E 85 1E B9 80 DC  ]!...Wp..#^.....
0090: A1 BF 61 02 12 D1 3D 5F   D8 2E C5 5C 16 A2 6D 8D  ..a...=_...\..m.
00A0: E1 0B 3C 1F 22 CF 11 18   AA 2D CF 75 C1 F6 C2 E8  ..<."....-.u....
00B0: 40 C2 59 C1 19 8B 86 61   79 12 4F F2 3E EC 61 1B  @.Y....ay.O.>.a.
00C0: D3 CF FD 8C 3B F4 6D 1D   F2 25 C3 7F 69 B1 B7 D2  ....;.m..%..i...
00D0: 49 96 61 B2 50 2B 70 74   AA 8E DC 18 6A 22 FC 00  I.a.P+pt....j"..
00E0: 96 67 A4 0B 70 62 63 A8   49 D6 E1 36 92 60 FF 0C  .g..pbc.I..6.`..
00F0: 0E 72 55 0B A2 EC 3F CE   2E B4 BE E8 42 2C F0 73  .rU...?.....B,.s

]
```

In code sample 1.8, we obtain the public key from the `Certificate` object and we get the private key from the `KeyStore` object. The public and the private key objects can be used as parameters for the `encrypt()` and `decrypt()` method. These methods are almost identical. The only difference is the `Cipher` mode: we're using either `ENCRYPT_MODE` or `DECRYPT_MODE`.

1.3.4 Using a public-key algorithm for authentication and non-repudiation

When we run this class, we get the following output:

```
Let's encrypt 'secret message' with a public key
Encrypted message:
66e7a06a40e5092aa0e70b4d57f2dd139e1902ddc012aa6d75deb6ecc9727fb9219d9aae98c054ebd53
8bfb002d31314c9fa6990d25cb528cf7516bbe3c8923c6670ba1b3673eb40a908a12146369f98fba36c
6beee04479411af7a6226e122a04d119dd9648a5d1be5cb3c52584f42fdffc6719f4d4b1e593a85c377
97d025b60f88d1ad19ad985911768bbb511c732761adace642dc8634925a8e197265ccadf07dd3a6185
50ffaf8dff13b44c91a3f9063d51f33ebd6d896321a795c34c6905f0e227253f3f86da4ca1d5f0da818
010621e1a88d4dc6cc4567d0fbc5461dbd696799f89da26b8eebcb4b4ac72115f021c2519e7036a5902
74f52ddde8d57d
Let's decrypt it with the corresponding private key
secret message
You can also encrypt the message with a private key
Encrypted message:
14d5a453baeb584a5c979ebcc142535684ce9503d1db01aa0bed0a3991dd5c6818ddf6f89bca94e24d8
dd02038564c42168c764d4d1f18fcf5e98f8da92b721add388e1971b63db841c8ddec1ae27b2f100f43
6e1f0a5294a9d2fb641b03a9aed412a4257c46f8b71700255b98a8d406a6daeb65bf64b28f85c786a67
2c56c95fa7dd539c8bfffa960a4d75ec16166088362dc1dbf0cb11ca6e7eb9d2730d885cc28abcd7d9b
56b179350d1975de16ec6c2c9ac2978cf2baa5da4b7d5b650ad5195a7fc5437072c05af55ab106f1231
c75c1a498ed83c113e4d008ee5710f32120641ea9ce7895e9f8c304ef6e8d169a9b0e0b155c91ae5f90
7978cd88e6e0f8
Now you need the public key to decrypt it
secret message
```

Do you see what happens here? First you take a public key and use it to encrypt a message so that only somebody with the corresponding key can decrypt it. That's what encryption is about.

But in the next few lines, you do the inverse: you encrypt a message using your private key. Now when you share your public key —which is usually the intention— the whole world can decrypt your message. In this case, you're not using the encryption mechanism to make sure a third party can't read your message, but to make it absolutely clear that you're the author. If I can decrypt a message using your public key, I'm 100% sure that it was encrypted using your private key. As only you have access to your private key, I'm 100% sure that you're the author. That's why the concept of encryption using an asymmetric key algorithm can be used for digital signing.

Figure 1.5 shows the two different concepts. For encryption, the public key (the green key) is used to encrypt, the private key (the red key) to decrypt. For creating a signature, it's the other way round.

Figure 1.5: asymmetric key algorithms

Does this solve our problems of data integrity, authenticity and non-repudiation?

Yes, it does, because as soon as anybody alters an encrypted message sent from me to you, you won't be able to decrypt it using my public key. On the other hand: if you succeed in decrypting it, you're certain that I was the author. I can't claim that I didn't sign the message while also claiming that my private key isn't compromised.

However, that's not how it's done in practice: encrypting and decrypting large messages requires large resources. If my goal is not to protect the message from being read by anybody but you, we both could save valuable time and CPU on our machines if I encrypt only a digest of the message, and attach this 'signed digest' to the actual message. When you receive my message, you can decrypt the digest and hash the message for comparison. If there's a match, the message is genuine. If there isn't, the message was altered.

We haven't solved all possible problems, though. Many questions remain. For instance: how can you be sure that the public key you're using is mine, and not a public key from somebody pretending he's me? What happens if my private key is compromised? We'll answer these questions in the next chapters. First, we need to get acquainted with some acronyms and standards.

1.4 Overview of cryptography Acronyms and Standards

Understanding the terminology was one of the most important thresholds I encountered when I was first confronted with cryptography and digital signing. Here's a list of acronyms and standards, you'll meet in the next couple of chapters. You'll also need to understand their meaning when using the PDF library iText to add digital signatures to PDF documents.

1.4.1 Acronyms

PKI stands for Public Key Infrastructure. It's a set of hardware, software, people, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates.

ASN.1 stands for Abstract Syntax Notation One. It's a standard and flexible notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunications and computer networking.

BER stands for Basic Encoding Rules. They were the original rules laid out by the ASN.1 standard for encoding abstract information into a specific data stream.

DER stands for Distinguished Encoding Rules. DER is a subset of BER providing for exactly one way to encode an ASN.1 value. It's drawn from the constraints placed on BER encodings by X.509.

X.509 is a standard for a public key infrastructure (PKI) and privilege management infrastructure. It specifies, amongst other things, standard formats for public key certificates and certificate revocation lists.

IETF stands for the Internet Engineering Task Force[4]. They develop and promote internet standards, cooperating closely with the World Wide Web Consortium (W3C), the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

RFC stands for Request for Comments. An RFC is a memorandum published by the Internet Engineering Task Force (IETF) describing methods, behaviors, research, or innovations applicable to the working of the Internet and Internet-connected systems.

FIPS stands for Federal Information Processing Standard[5]. It's a publicly announced standardization developed by the US federal government for use in computer systems.

1.4.2 Public-Key Cryptography Standards (PKCS)

Apart from an encryption algorithm, RSA is also the name of an American computer and network security company. Just like the algorithm, it was named after the initials of its co-founders Rivest, Shamir and Adleman. The RSA Company has devised and published a group of public-key cryptography standards numbered from #1 to #15. Some of these standards are no longer in use. In the context of this paper, we're mostly interested in PKCS#7, the Cryptographic Message Syntax (CMS), but we'll also take a look at a couple of other standards.

PKCS#1: RSA Cryptography Standard

This standard is published as RFC 3447. It defines the mathematical properties and format of RSA public and private keys (ASN.1 encoded in clear text), and the basic algorithms and encoding/padding schemes for performing RSA encryption, decryption, and producing and verifying signatures. We'll encounter PKCS#1 briefly when talking about the sub filter `/adbe.x509.rsa_sha1`.

PKCS#7: Cryptographic Message Syntax Standard

This standard is published as RFC 2315. It's used to digitally sign, digest, authenticate or encrypt any form of digital data. The Cryptographic Message Syntax (CMS) is updated in RFC 5652. Its architecture is built around certificate-based key management, such as the profile defined by the PKIX (PKI X.509) working group.

PKCS#11: Cryptographic Token Interface

This standard is also known as "Cryptoki" which is an amalgamation of "cryptographic token interface" and is pronounced as "crypto-key". It's an API defining a generic interface to cryptographic tokens, such as Hardware Security Modules (HSM), USB keys and smart cards. We'll encounter PKCS#11 in chapter 4, where we'll sign a PDF using a Luna SA HSM and an iKey 4000 USB key.

PKCS#12: Personal Information Exchange Syntax Standard

This standard defines a file format used to store private keys with accompanying public key certificates, protected with a password-based symmetric key. In practice, these files will have the extension `.p12`. It's usable as format for the Java key store. Note that you may also encounter `.pfx` files. PFX is a predecessor to PKCS#12.

[4] http://www.ietf.org/
[5] http://www.itl.nist.gov/fipspubs/

PKCS#13: Elliptic Curve Cryptography Standard

This standard is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. We won't use PKCS#13 in the context of this paper, but we'll briefly mention it in section 2.1.4 when looking at the different encryption algorithms that are supported in PDF.

1.4.3 The PDF ISO Standard

The Portable Document Format used to be a specification that was proprietary to Adobe. The first version was published in 1993 and Adobe owned the copyright, but soon it was made open in the sense that Adobe allowed developers to use the specification to create a PDF writer (such as iText[6]) and/or a PDF viewer (such as JPedal[7]).

Since 2001, a couple of subsets of the PDF specification were already published by the International Organization for Standardization (ISO)[8] as ISO standards: ISO-15930 (aka PDF/X) for the prepress sector and ISO-19005 (aka PDF/A) which is a standard for archiving PDFs.

In 2007, Adobe decided to bring the PDF specification (at that time PDF 1.7) to the Enterprise Content Management Association (AIIM)[9], for the purpose of publication as an ISO standard. The standard was published on July 1st, 2008 as ISO-32000-1 (ISO, 2008).

Other PDF standards followed or will follow in the future, such as ISO 24517 (aka PDF/E) for engineering, ISO 16612 (aka PDF/VT) for Variable Data Printing and Transaction printing, and ISO-14289 (aka PDF/UA) for Universal Accessibility. In the meantime, we're looking forward to ISO-32000-2, the successor of ISO-32000-1. It's expected for 2013, at which point PDF 2.0 will be introduced. In this paper, we're already taking into account some of the changes that will be introduced in PDF 2.0.

1.4.4 CAdES, XAdES and PAdES

The European Telecommunications Standards Institute (ETSI)[10] is an independent, non-profit, standardization organization in the telecommunications industry (equipment makers and network operators). The organization publishes Technical Standards (TS) of which the ones explaining Advanced Electronic Signatures are most important to us in the context of this white paper.

An Advanced Electronic Signature is an electronic signature that is:

- Uniquely linked to the signatory,
- Capable of identifying the signatory,
- Created using means that the signatory can maintain under his sole control, and
- Linked to the data to which it relates so that any subsequent change of the data is detectable.

The ETSI published three sets of specifications.

[6] http://www.itextpdf.com/
[7] http://www.jpedal.org/
[8] http://www.iso.org/
[9] http://www.aiim.org/
[10] http://www.etsi.org/

CMS Advanced Electronic Signatures (CAdES)

CAdES is a set of extensions to CMS, making it suitable for advanced electronic signatures. It's described in ETSI TS 101 733. One important benefit from CAdES is that electronically signed documents can remain valid for long periods, even if underlying cryptographic algorithms are broken.

XML Advanced Electronic Signatures (XAdES)

XAdES is an extension to XML-DSig[11], a standard defining the XML syntax for digital signatures. It's described in ETSI TS 101 903. The use of XAdES is not supported in iText. I'm mentioning it here only because it's the technology used in PAdES Part 5.

PDF Advanced Electronic Signatures (PAdES)

PAdES is a set of restrictions and extensions to PDF and ISO-32000-1 making it suitable for advanced electronic signatures. It's described in TS 102 778 (ETSI, 2009), and it will be implemented in ISO-32000-2. PAdES consists of six parts:

- *Part 1*—the first part is an overview of support for signatures in PDF documents, and it lists the features of the PDF profiles in the other documents.
- *Part 2*—PAdES Basic is based on ISO-32000-1. If you want to know more about digital signatures in PDF, you should read this specification before starting to dig into the PDF reference. PAdES part 2 is supported in iText since version 5.0.0.
- *Part 3*—PAdES Enhanced describes profiles that are based on CAdES: PAdES Basic Electronic Signature (BES) and Explicit Policy Electronic Signature (EPES). PAdES part 3 is supported in iText since version 5.3.0.
- *Part 4*—PAdES Long-Term Validation (LTV) is about protecting data beyond the expiry of the user's signing certificate. This mechanism requires a Document Security Store (DSS). PAdES part 4 is supported in iText since version 5.1.3.
- *Part 5*—PAdES for XML content describes profiles for XAdES signatures. For instance, after filling an XFA form, which is XML content embedded in a PDF file, a user may sign selected parts of the form. This isn't supported in iText yet.
- *Part 6*—Visual representations of Electronic Signatures. This is supported in iText, but it also depends on other factors. For instance: does your certificate contain sufficient information?

We'll discuss PAdES in more detail in the following chapters of this white paper. Note that the code used in this paper will only work with iText version 5.3.4 and later.

1.5 Summary

In this chapter, we've explored the different aspects of digital signing. First we looked at a PDF file and discovered how we could forge an existing document. We've learned about hashing algorithms and encryption, and we found out how to combine these concepts to protect a PDF file. We ended with an overview of terms and standards.

Now we're ready to take a closer look at digital signatures inside a PDF file. In the next chapter, we'll put the theory into practice with a first set of examples.

[11] http://www.w3.org/TR/xmldsig-core/

2. PDF and digital signatures

In this paper, we'll combine the specifications found in ISO-32000-1 which is the PDF standard from the International Organization for Standardization (ISO), TS 102 778 (aka PAdES) from the European Telecommunications Standards Institute (ETSI), and ISO-32000-2, the successor of ISO-32000-1 which hasn't been officially released yet.

Some implementations explained in ISO-32000-1 will be deprecated in ISO-32000-2. There may also be room for interpretation when comparing the ISO standard with the ETSI standard, but whenever that's the case, we'll opt for what we believe is the most future-proof choice. Note that the principles explained in this paper are a *summary*. For more details, please consult the actual standards.

2.1 Digital signatures in PDF

In section 1.1, we've taken a look inside different PDF files. We've looked at the infrastructure that is needed to digitally sign a PDF document. Now let's focus on the actual signature.

2.1.1 The signature handler and the sub filters

When creating a digital signature for a PDF, you need to define a preferred signature handler (a `/Filter` entry). In iText, we'll always use the `/Adobe.PPKLite` filter. It's possible to adapt iText to use another filter, but there's very little need to do so: an interactive PDF processor can use any handler it prefers as long as the handler supports the specified `/SubFilter` format.

The sub filter refers to an encoding or a format that was used to create the signature. For instance: does it use PKCS#1, PKCS#7, or CAdES? Is part of the information (such as the public certificate) stored outside the signature, or is it embedded in the signature?

> **NOTE:** In PDF, we sometimes refer to a *detached signature*. According to Wikipedia, a detached signature is a type of digital signature that is kept 'separate from its signed data', as opposed to 'bundled together into a single file'. This definition isn't entirely correct in the context of PDF: the signature is enclosed in the PDF file, but the attributes of the signature are 'part of the signature', as opposed to 'stored in the signature dictionary'.

In versions predating iText 5.3.0, you'd sign a PDF document choosing one of the following parameters for the `setCrypto()` method:

- *`PdfSignatureAppearance.WINCER_SIGNED`*— this created a signature with the sub filter `/adbe.pkcs7.sha1`.
- *`PdfSignatureAppearance.SELF_SIGNED`*— this created a signature with the sub filter `/adbe.x509.rsa_sha1`.

These options have been removed in 5.3.0 for very specific reasons.

The `/adbe.pkcs7.sha1` sub filter will be deprecated in PDF 2.0. ISO-32000-2 recommends: *"To support backward compatibility, PDF readers should process this value for the `/SubFilter` key but PDF writers shall not use this value for that key."* iText is a PDF writer, and since iText 5.3.0, we no longer allow the creation of this type of signatures. Please don't sign any documents using this sub filter anymore.

As for `/adbe.x509.rsa_sha1`, it will still be available in PDF 2.0, but the underlying standard that is used (PKCS#1) is explicitly forbidden in PAdES[12]. That is: the value of the `/Contents` entry of a signature may not be a DER-encoded PKCS#1 binary data object. We've discontinued support for the creation of pure PKCS#1 signatures, so that iText-created signatures comply with PAdES.

> **NOTE:** The encrypted digests in a PKCS#7 container are stored as embedded PKCS#1 objects. This is, of course, perfectly OK and allowed in the PAdES standard.

With iText 5.3.0 (released in June 2012), we completely redesigned the digital signature functionality. From this version on, we'll use detached signatures by default, either `/adbe.pkcs7.detached` or `/ETSI.CAdES.detached`.

> **NOTE:** There's also `/ETSI.RFC3161`, but if you look up RFC 3161, you'll discover that it's an X509 PKI timestamp protocol. See section 5.3.

It goes without saying that iText is versatile enough to allow creating signatures with the sub filters `/adbe.x509.rsa_sha1` or `/adbe.pkcs7.sha1`, but we deliberately made it more difficult to do so in order to discourage their use. In the following sections and chapters, you can safely assume that we're talking about detached signatures, unless specified otherwise.

2.1.2 The byte range covered by the digital signature

Figure 2.1 shows a schematic view of a signed PDF.

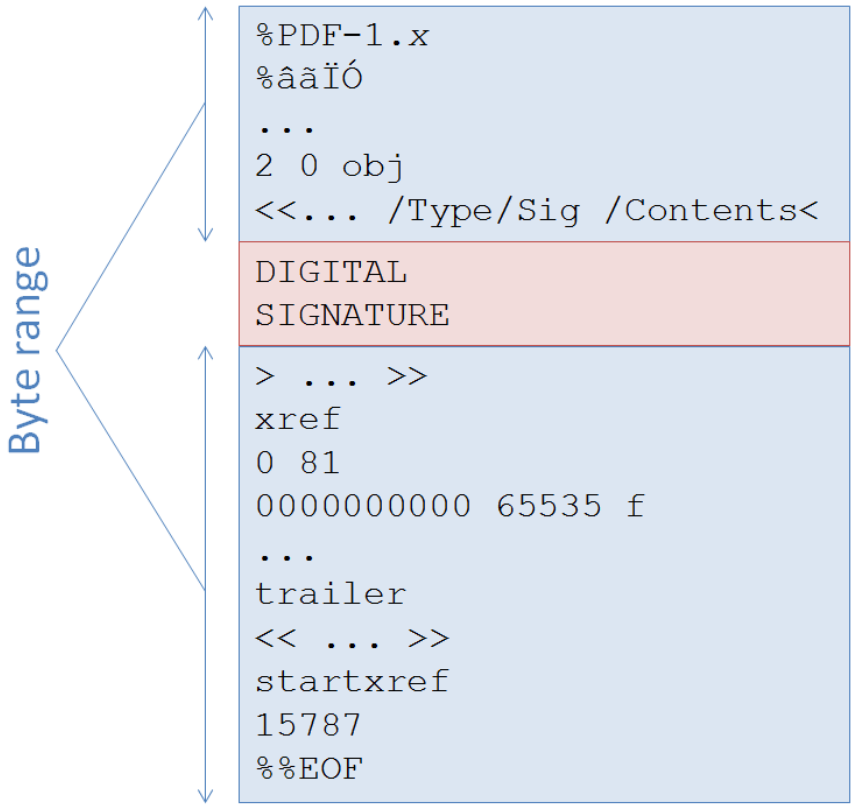

Figure 2.1: A signed PDF

[12] ETSI TS 102 778-2, section 5.1, Note 1

In section 1.1.3, we inspected the syntax of a signed PDF, and we saw that the signature dictionary contains a `/ByteRange` entry. If you read ISO-32000-1, you'll find out that this byte range may contain gaps: areas in the PDF that aren't covered by the signature. In theory, these gaps could make the PDF vulnerable[13].

PAdES introduces an extra restriction to the PDF digital signature specification that has been taken into account in ISO-32000-2 for ETSI sub filters: *"If* `/SubFilter` *is* `/ETSI.CAdES.detached` *or* `/ETSI.RFC3161`, *the* `/ByteRange` *shall cover the entire file, including the signature dictionary but excluding the* `/Contents` *value."* Moreover, recent versions of Adobe Acrobat/Reader have been rejecting signatures with greater or more holes in their byte range. That's why iText always takes the complete byte range of the PDF document, regardless of the sub filter that was chosen.

2.1.3 How to compose a signature

The elements needed to compose a signature are shown on one side in figure 2.2. The actual contents are shown on the other side.

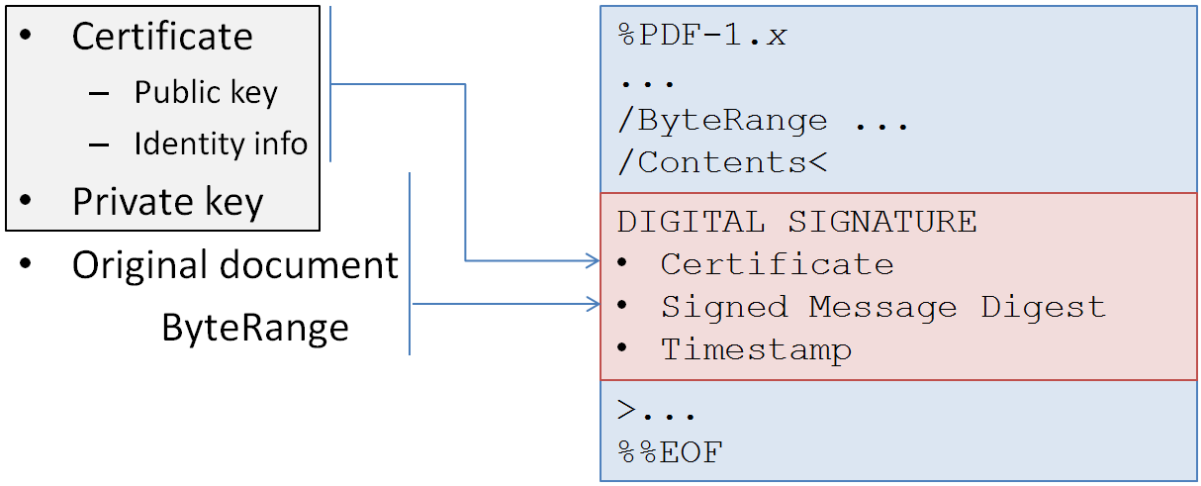

Figure 2.2: the contents of a digital signature

To the left, we have the elements of the digital identity of a person: his private key and a certificate containing his public key and identity info. Note that most of the times, there will be a chain of certificates. This will be explained in more detail in chapter 3. For now, we're working with a single self-signed certificate.

> **NOTE:** looking closely at figures 2.1 and 2.2, you see that the complete document is covered by the digital signature. It's not possible to sign specific pages. Some countries demand that every page is initialed for the signature to be valid so that there's proof that every page has been seen, but the concept to 'initial' the pages of a document doesn't exist in PDF.

In a pure PKCS#1 signature (which is no longer supported), the certificate is an entry in the signature dictionary. It isn't part of the actual signature. For CMS and CAdES based signatures, the certificate (or certificate chain) is embedded in the digital signature. The signature also contains a digest of the original document that was signed using the private key. Additionally, the signature can contain a timestamp.

[13] See the article about "Collisions in PDF Signatures" by Florian Zumbiehl http://pdfsig-collision.florz.de/

2.1.4 Algorithms supported in PDF

In section 1.2.2, we listed several digest algorithms we could use to create a message digest, but not all of the algorithms we've listed are supported in PDF. In section 1.3, we talked about the RSA encryption algorithm, but there are some other algorithms we can use for signing:

- *The Digital Signature Algorithm (DSA)* — this is a FIPS standard for digital signatures. RSA can be used for both encrypting and signing; DSA is used mainly for signing. DSA is faster in signing, but slower in verifying.
- *The Elliptic Curve Digital Signature Algorithm (ECDSA)* — this is a new standard (PKCS #13). It will be introduced for signing PDFs in PDF 2.0; it's supported in iText, but it hasn't been tested yet. At the time this paper was written, it wasn't supported in Adobe Reader yet.

Let's take a look at an overview of the digest and encryption algorithms that are supported based on the sub filter:

adbe.pkcs7.sha1

Supported message digests: SHA1 (other digests may be used to digest the signed-data field, but SHA1 is required to digest the PDF document data that is being signed).
RSA Algorithm: up to 1024 bits (since PDF 1.3), 2048 bits (since PDF 1.5), 4096 bits (since PDF 1.7).
DSA Algorithm: up to 4096 bits (since PDF 1.6).

Note that the use of this sub filter for signature creation will be deprecated in PDF 2.0 (ISO-32000-2). It's no longer supported in iText since version 5.3.0.

adbe.x509.rsa_sha1

Supported message digests: SHA1 (since PDF 1.3), SHA256 (since PDF 1.6), and SHA384, SHA512, RIPEMD160 (since 1.7).
RSA Algorithm: up to 1024 bits (since PDF 1.3), 2048 bits (since PDF 1.5), 4096 bits (since PDF 1.7).
DSA Algorithm: not supported.

Note that despite the reference to SHA1 in the name, other digest algorithms are supported. As pure PKCS#1 is forbidden in the PAdES standard, we no longer support this sub filter in iText (since 5.3.0).

adbe.pkcs7.detached, ETSI.CAdES.detached and ETSI.RFC3161

Supported message digests: SHA1 (since PDF 1.3), SHA256 (since PDF 1.6), and SHA384, SHA512, RIPEMD160 (since 1.7).
RSA Algorithm: up to 1024 bits (since PDF 1.3), 2048 bits (since PDF 1.5), 4096 bits (since PDF 1.7).
DSA Algorithm: up to 4096 bits (since PDF 1.6).
ECDSA: the Elliptic Curve Digital Signature Algorithm will be supported in PDF 2.0.

Detached signatures are fully supported by default in iText since version 5.3.0.

> **WARNING ABOUT HASHING:** The use of SHA-1 is being phased out in some countries. The use of stronger hashing algorithms is recommended.
> **WARNING ABOUT ENCRYPTION:** NIST (US National Institute of Standards and Technology) advises that 1024-bit RSA keys are no longer viable (since 2010) and advises moving to 2048-bit RSA keys. So do other agencies in other countries, for instance in Germany.

It's high time that we throw in an example, demonstrating how to sign a PDF document using iText.

2.2 The "Hello World" of digital signing using iText

Forget everything that was written in the first and second edition of "iText in Action" (Lowagie, 2011); forget all the code you wrote before upgrading to iText 5.3.0. Signing a PDF using iText has been completely redesigned. You'll soon find out that the changes are for the better.

2.2.1 A simple example adding a visible signature to a document

Let's start with the "Hello World" app of signing. How can we use iText to sign a file and get a result as is shown in figure 2.3?

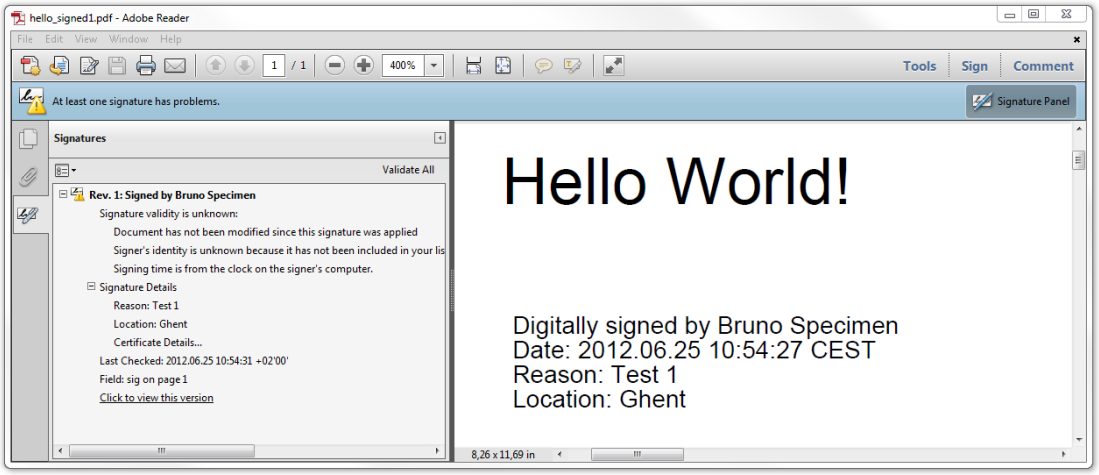

Figure 2.3: a signature appearance

With code sample 2.1, we can create a very simple digital signature using iText.

Code sample 2.1: the "Hello World" of signing with iText

```
public void sign(String src, String dest,
    Certificate[] chain, PrivateKey pk, String digestAlgorithm, String provider,
    CryptoStandard subfilter, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    // Creating the signature
    ExternalDigest digest = new BouncyCastleDigest();
    ExternalSignature signature =
        new PrivateKeySignature(pk, digestAlgorithm, provider);
    MakeSignature.signDetached(appearance, digest, signature, chain,
        null, null, null, 0, subfilter);
}
```

Please be aware that this is far from the 'definite code sample', but it's a start. Let's examine the example step by step.

First we create a `PdfReader` and a `PdfStamper` object. This is what you'll always do in iText when you want to manipulate an existing PDF document. The main difference is that you have to use the `createSignature()` method instead of the `PdfStamper` constructor.

> **NOTE:** If you need to sign a PDF/A file, you should use the `createSignature()` method available in the `PdfAStamper` stamper class. This class can be found in a separate itext-pdfa.jar starting with iText 5.3.4.

Secondly, we define the appearance using the `PdfSignatureAppearance` class. We'll go into more detail later on in this chapter, but for now we only set a reason for signing and a location. We also define a rectangle where the signature will be added (lower-left coordinate `[36,748]`; upper-right coordinate `[144,780]`), a page number (page 1), and a name for the signature field ("`sig`").

Furthermore, we need an implementation of the `ExternalDigest` interface to create a digest and of the `ExternalSignature` interface to create the signature, a process that involves hashing as well as encryption.

We can use Bouncy Castle as security provider for the digest by choosing an instance of the `BouncyCastleDigest` class. If you want another provider, use the `ProviderDigest` class. iText has only one implementation for the signing process: `PrivateKeySignature`; we'll use another implementation in chapter 4 when we sign a document using a smart card. The constructor of the `PrivateKeySignature` class needs a `PrivateKey` instance, a digest algorithm and a provider. For instance: we won't use Bouncy Castle as provider when we apply a signature using PKCS#11 in chapter 4.

> **NOTE:** Unless otherwise specified, iText uses the same digest algorithm for making the hash of the PDF bytes as defined for creating the signature. The encryption method ("RSA", "DSA" or "ECDSA") and the key size will be obtained from the private key object. If we use the key store we created in section 1.3.1, iText will use 2048-bit RSA encryption because those were the parameters we passed to the `keytool` utility.

We're using the `signDetached()` method, which means we're creating a detached signature, and we can choose between `adbe.pkcs7.detached` and `ETSI.CAdES.detached`. This is done with an `enum` named `CrypoStandard` in `MakeSignature`: use either `CMS` or `CADES`.

> **WARNING:** For the moment, we're passing plenty of `null` objects and one `0` value to the `signDetached()` method. It's important to understand that we're creating a signed PDF that only meets the minimum requirements of a digital signature. You'll need to replace at least two of those `null` values with actual objects if you want to create a signature that conforms to the best practices in signing. This will be discussed in chapter 3.

Code sample 2.2 shows how we can create a variety of signed Hello World files. First we create a `PrivateKey` instance and an array of `Certificate` objects based on the `keystore` we created in section 1.3.1. Then we invoke the method from code sample 2.1 using different digest algorithms, choosing between CMS (PKSC#7 as described in ISO-32000-1 and PAdES 2) and CAdES (as described in PAdES 3).

Code sample 2.2: signing a PDF using different algorithms and sub filters

```
public static final String KEYSTORE = "src/main/resources/ks";
public static final char[] PASSWORD = "password".toCharArray();
public static final String SRC = "src/main/resources/hello.pdf";
public static final String DEST = "results/chapter2/hello_signed%s.pdf";

public static void main(String[] args)
    throws GeneralSecurityException, IOException, DocumentException {
    BouncyCastleProvider provider = new BouncyCastleProvider();
    Security.addProvider(provider);
    KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
    ks.load(new FileInputStream(KEYSTORE), PASSWORD);
    String alias = (String)ks.aliases().nextElement();
    PrivateKey pk = (PrivateKey) ks.getKey(alias, PASSWORD);
    Certificate[] chain = ks.getCertificateChain(alias);
    SignHelloWorld app = new E04_SignHelloWorld();
    app.sign(SRC, String.format(DEST, 1), chain, pk, DigestAlgorithms.SHA256,
        provider.getName(), CryptoStandard.CMS, "Test 1", "Ghent");
    app.sign(SRC, String.format(DEST, 2), chain, pk, DigestAlgorithms.SHA512,
        provider.getName(), CryptoStandard.CMS, "Test 2", "Ghent");
    app.sign(SRC, String.format(DEST, 3), chain, pk, DigestAlgorithms.SHA256,
        provider.getName(), CryptoStandard.CADES, "Test 3", "Ghent");
    app.sign(SRC, String.format(DEST, 4), chain, pk, DigestAlgorithms.RIPEMD160,
        provider.getName(), CryptoStandard.CADES, "Test 4", "Ghent");
}
```

Figure 2.4 shows the different icons that indicate whether or not a signature is valid. As you can see, these icons have changed over the years depending on the version of Adobe Acrobat or Reader you're using.

Figure 2.4: Different icons used to recognize if a signature was validated

A red cross always means that your signature is broken: the content has been altered or corrupted, or one of the certificates isn't valid, and so on. In any case, you shouldn't trust the signature.

> **WARNING:** support for CAdES is very new. Don't expect versions older than Acrobat/Reader X to be able to validate CAdES signatures! Acrobat 9 only supports signatures as described in the specification for PDF 1.7, and CAdES is new in PDF 2.0.

In figure 2.3, we get a yellow triangle with a message *"At least one signature has problems"*. A yellow triangle (or a question mark in older versions of Adobe Reader) means that the signature can't be validated because some information is missing. In our case, Adobe Reader says: *"The signature validity is unknown."* There's no problem with the integrity because we see that the *"Document has not been modified since this signature was applied."* So what's missing?

In section 1.3.1, we created our own key store for Bruno Specimen. Anyone can create such a key store and pretend that he or she is Bruno Specimen. Adobe Reader is aware of this problem and tells us: *"Signer's identity is unknown because it has not been included in your list of trusted identities and none of its parent certificates are trusted identities."* Let's ignore the second part about the parent certificates for now, and focus on the list of trusted identities first.

2.2.2 Manage trusted identities

There are different ways to add a certificate to the list of trusted identities. One way is to look at the signature panel and to open the Certificate Viewer by clicking on Certificate Details under Certificate details. See figure 2.5.

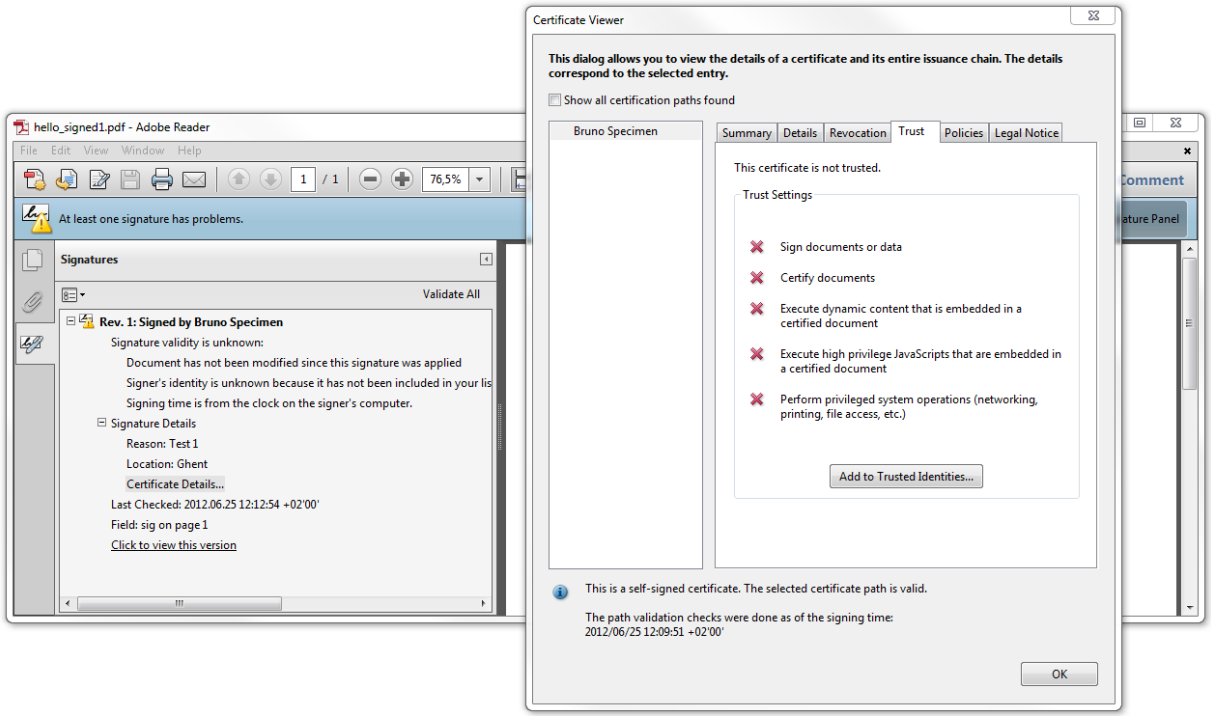

Figure 2.5: Certificate details viewer: add to trusted identities

There's a tab named Trust saying *"This is a self-signed certificate. The selected certificate path is valid."* However: *"The certificate is not trusted."* In the trust settings, there's a button with caption *"Add to trusted identities"*. If you click this button, you get another dialog, as shown in figure 2.6.

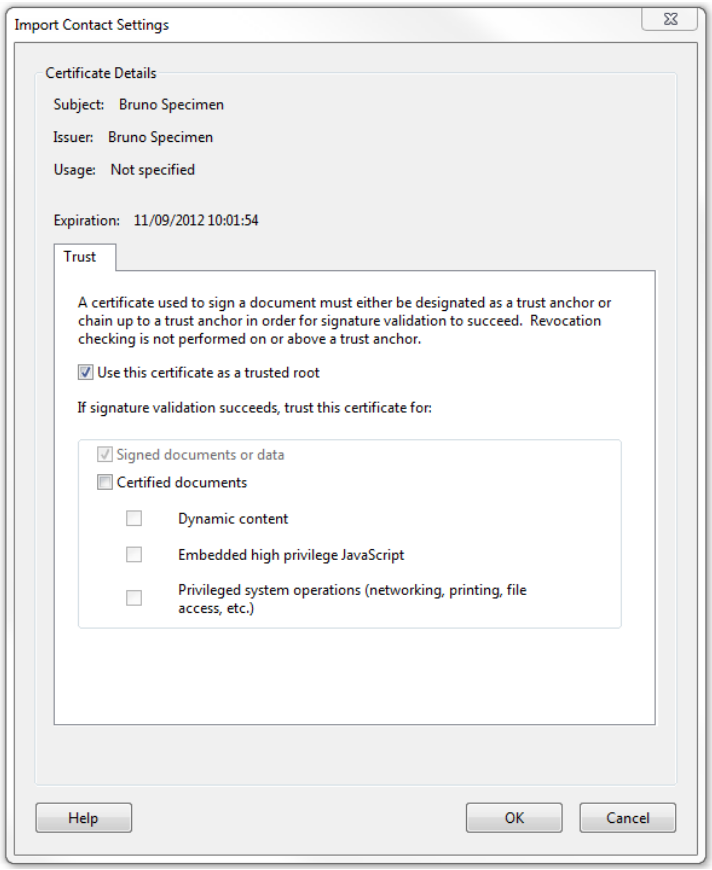

Figure 2.6: Import Contact Settings dialog

You could decide to use this certificate as a trusted root. Let's try this and see what happens. Go to *Edit > Protection > Manage Trusted Identities…* as shown in figure 2.7:

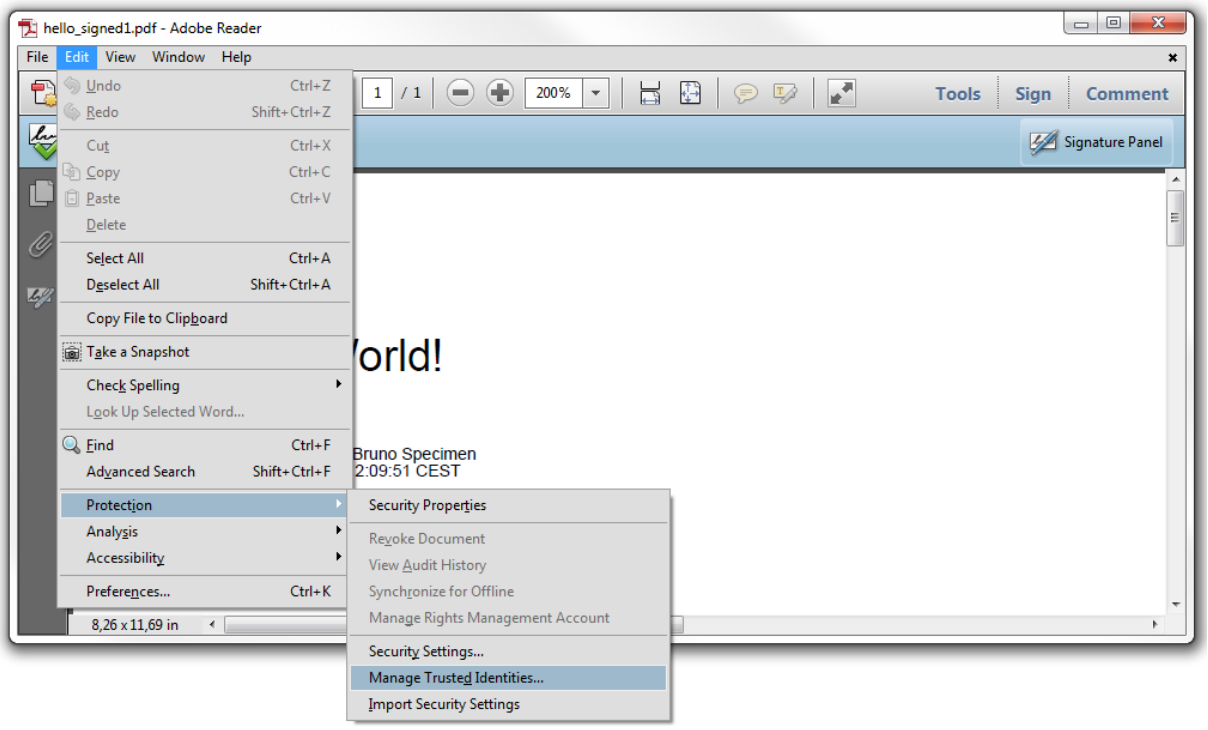

Figure 2.7: Import Contact Settings dialog

You'll find Bruno Specimen listed if you display the Certificates as is done in figure 2.8:

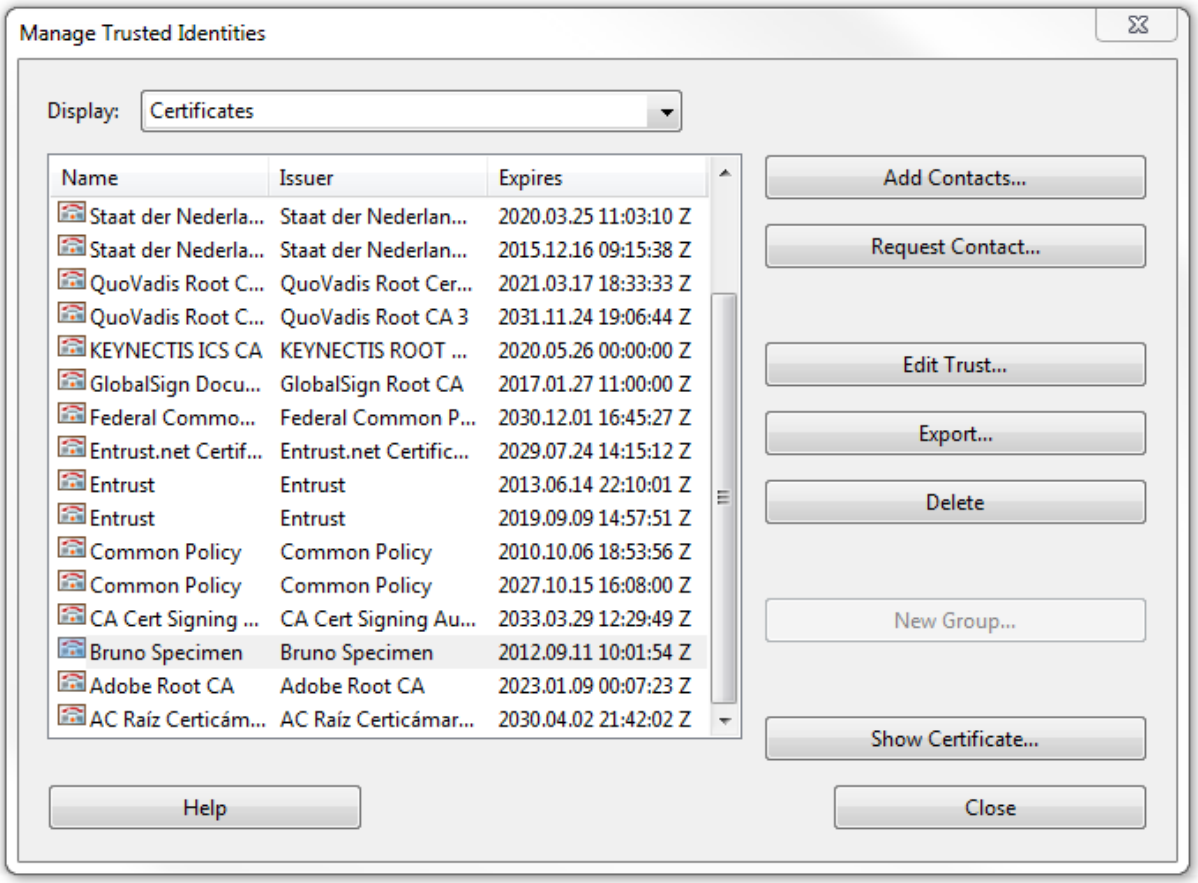

Figure 2.8: the Manage Trusted Identities dialog

From now on, you'll see a green check mark when opening the signed document. See figure 2.9.

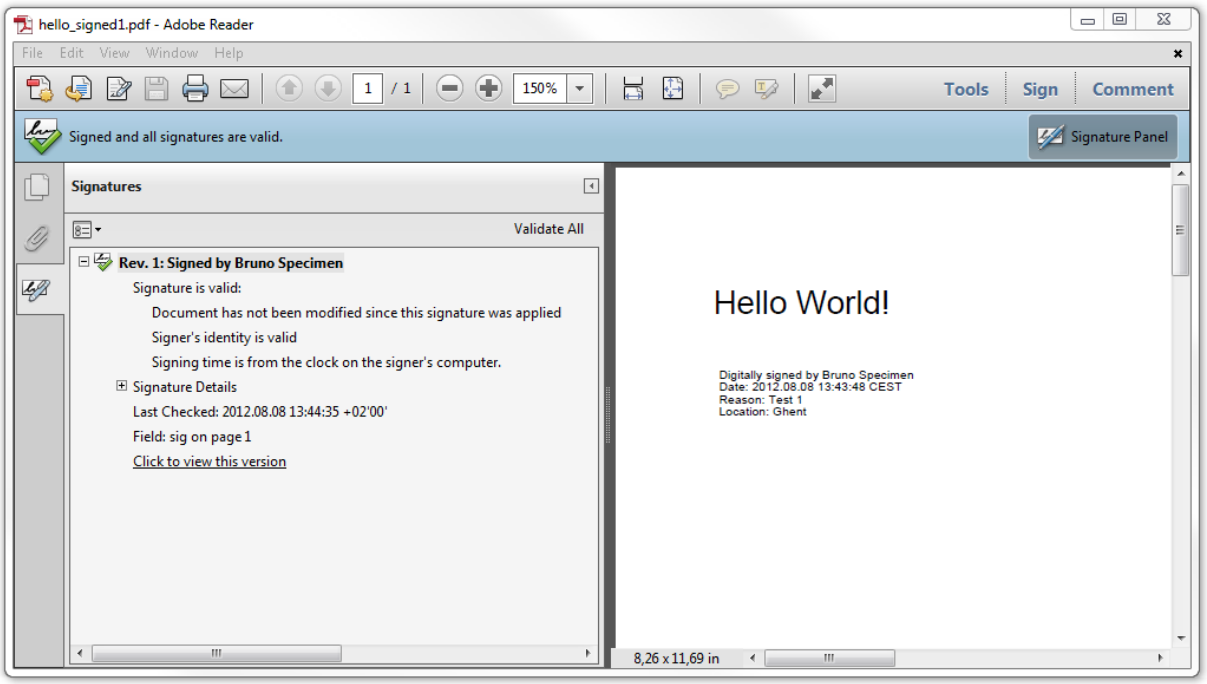

Figure 2.9: Signed document and the Signature is valid.

This is one way you can add the certificate to the trusted identities, but it defies the purpose of a digital signature. If you accept the validity of the signer's identity anyway, why go through the hassle of signing? The initial idea we thought of when we discussed authentication, was that Bruno Specimen would send you his public certificate, and that you would use it to identify him. So let's remove the certificate from that list of trusted identities, and try anew using a different approach.

2.2.3 Adding a certificate to the Contacts list in Adobe Reader

Bruno Specimen can extract his public certificate from his key store using the command shown in code sample 2.3.

Code sample 2.3: exporting a certificate from a key store

```
$ keytool -export -alias demo -file bruno.crt -keystore ks -storepass password
Certificate stored in file <bruno.crt>
```

The result is a file name that can be imported into Adobe Reader by clicking "Add Contacts" (see figure 2.8). Just browse for the file `bruno.crt` and click "Import" (see figure 2.10). The file will show up in the list of Trusted Identities, but don't forget to edit the trust settings! You need to explicitly trust the certificate (as is shown in figure 2.6) before you'll be able to validate Bruno Specimen's signature.

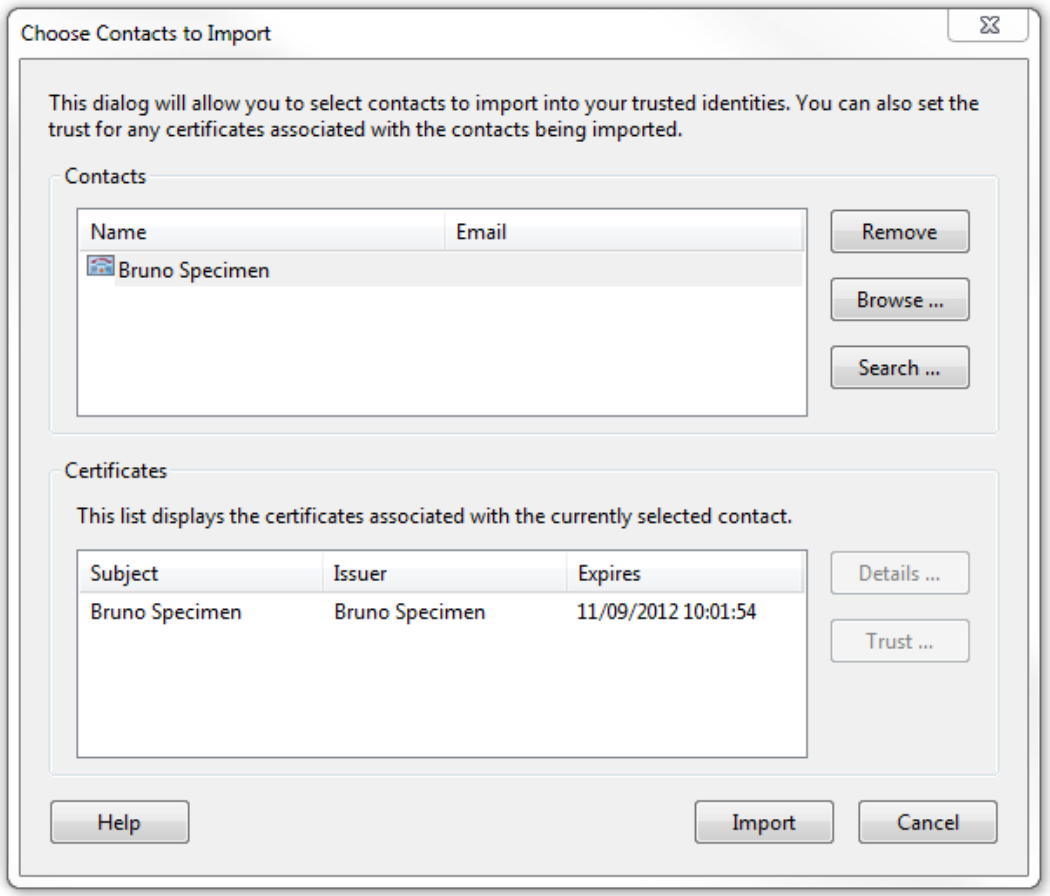

Figure 2.10: Import a Certificate into the Trusted Identities list

But wait! What if you don't really know Bruno Specimen? What if somebody else sends you a public key pretending he's Bruno Specimen? How can you know that the person who is sending you his public key is the person he or she says he or she is?

There are different ways and mechanisms that allow you to obtain a green check mark without managing trusted identities manually. In chapter 3, we'll discuss certificate authorities, and in section 3.4, we'll discover different options to get a green check mark requiring less (or even no) manual intervention. First let's sign our Hello World file once more, using a different method to create a `PdfStamper` object.

2.2.4 Signing large PDF files

When you use the `createSignature()` method as shown in code sample 2.1, iText will create a copy of the document you're about to sign in memory, leaving the bytes that are reserved for the signature blank. iText needs this copy so that it can provide the bytes that need to be hashed and signed. This can be problematic for files with a large file size: you risk `OutOfMemoryException`s.

Take a look at code sample 2.4 if you want to store the copy on disk instead of keeping it in memory:

Code sample 2.4: Signing a document using a temporary file

```
    public void sign(String src, String tmp, String dest,
        Certificate[] chain, PrivateKey pk, String digestAlgorithm,
        String provider, CryptoStandard subfilter, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper =
        PdfStamper.createSignature(reader, os, '\0', new File(tmp));
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, subfilter);
}
```

There's only one difference with code sample 2.1, we added a `File` object as an extra parameter to the `createSignature()` method. The `tmp` variable in this code sample can be a path to a specific file or to a directory. In case a directory is chosen, iText will create a file with a unique name in that directory.

> **NOTE:** if you use the `createSignature()` method with a temporary file, you can use an `OutputStream` that is `null`, in that case, the temporary file will serve as the actual destination file. This is good practice if your goal is to store a signed file on your file system. If the `OutputStream` is not `null`, iText will always try to delete the temporary file after the signing is done.

Make sure you use a path with sufficient writing permissions, and make sure you don't try to overwrite existing files if you're working in a multithreaded environment. As we're working with very simple PDF files in these examples, we'll continue using the method that tells iText to keep the bytes in memory.

We've already seen an example of a PDF with an invisible signature (see figure 1.3) and we've already seen PDF documents with a visible signature (see for instance figure 2.3). In code samples 10 and 13, we created a visible signature using a `Rectangle` object and absolute coordinates for the lower-left corner and the upper-right corner. If you define a rectangle of which either the width, or the height (or both) are zero, you're creating an invisible signature.

When creating a document of which you know it will have to be signed, you can choose the coordinates of the signature rectangle in advance.

2.3 Creating and signing signature fields

It's not always simple to determine the coordinates where the signature should be placed. That's why you may want to create a placeholder if you need a visible signature. Let's start by creating a PDF document with an empty signature field using Adobe Acrobat.

2.3.1 Adding a signature field using Adobe Acrobat

Open the document that needs to be signed in Adobe Acrobat and select *Forms > Add or Edit Fields* in the menu. Now select *Digital Signature* in the drop-down list box under *Add New Field*, and draw a rectangle. Acrobat will suggest a name for the field, for instance `Signature1`. See figure 2.11. You can change this name if necessary, but for us this name will do.

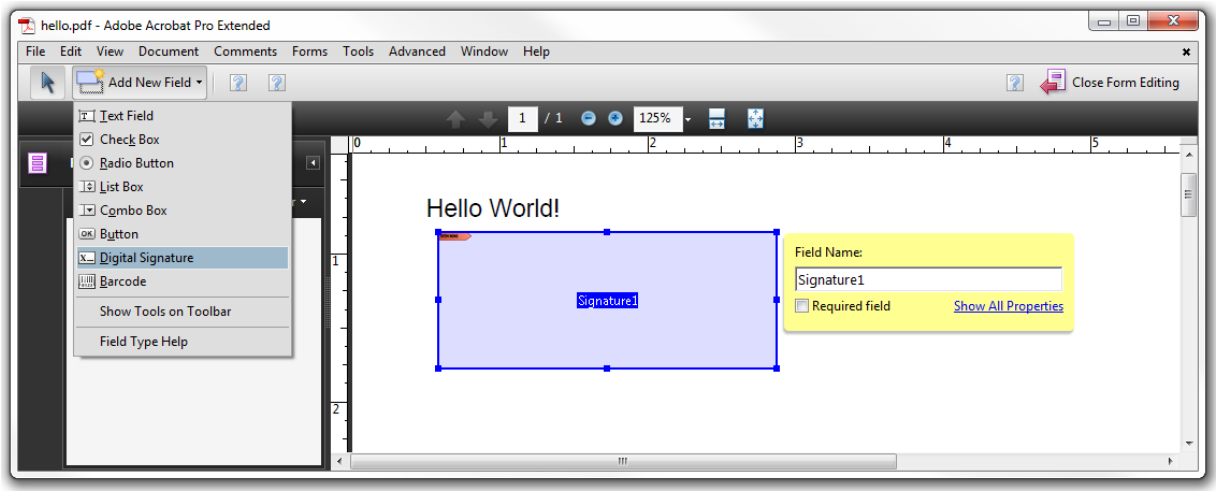

Figure 2.11: adding an empty signature field using Adobe Acrobat

Let's save this document, and fill it out using code sample 2.5.

Code sample 2.5: Signing a signature field

```
public void sign(String src, String name, String dest, Certificate[] chain,
        PrivateKey pk, String digestAlgorithm, String provider,
        CryptoStandard subfilter, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
```

```
    appearance.setVisibleSignature(name);
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, subfilter);
}
```

Again, there's only one difference with code sample 2.1. We no longer need to pass a rectangle and a page number; the name of the field is sufficient. This way, a developer no longer has to worry about finding the best coordinates to position the signature.

Now let's see if we can create an empty signature field using iText.

2.3.2 Creating a signature field programmatically using iText

If you know how to create AcroForm fields using iText, you know how to create an empty signature field as shown in figure 2.11. If you don't know anything about AcroForm fields, take a look at code sample 2.6.

Code sample 2.6: Creating a signature field

```
public void createPdf(String filename) throws IOException, DocumentException {
    // step 1: Create a Document
    Document document = new Document();
    // step 2: Create a PdfWriter
    PdfWriter writer = PdfWriter.getInstance(
        document, new FileOutputStream(filename));
    // step 3: Open the Document
    document.open();
    // step 4: Add content
    document.add(new Paragraph("Hello World!"));
    // create a signature form field
    PdfFormField field = PdfFormField.createSignature(writer);
    field.setFieldName(SIGNAME);
    // set the widget properties
    field.setPage();
    field.setWidget(
        new Rectangle(72, 732, 144, 780), PdfAnnotation.HIGHLIGHT_INVERT);
    field.setFlags(PdfAnnotation.FLAGS_PRINT);
    // add it as an annotation
    writer.addAnnotation(field);
    // maybe you want to define an appearance
    PdfAppearance tp = PdfAppearance.createAppearance(writer, 72, 48);
    tp.setColorStroke(BaseColor.BLUE);
    tp.setColorFill(BaseColor.LIGHT_GRAY);
    tp.rectangle(0.5f, 0.5f, 71.5f, 47.5f);
    tp.fillStroke();
    tp.setColorFill(BaseColor.BLUE);
    ColumnText.showTextAligned(tp, Element.ALIGN_CENTER,
        new Phrase("SIGN HERE"), 36, 24, 25);
    field.setAppearance(PdfAnnotation.APPEARANCE_NORMAL, tp);
    // step 5: Close the Document
    document.close();
}
```

When using iText, a PDF file is created from scratch in five steps: create a `Document` object, create a `PdfWriter`, open the `Document`, add content, and close the `Document`. We're interested in the fourth step: adding content.

> **NOTE:** iText has different convenience classes for text fields (`TextField`), push buttons (`PushbuttonField`), radio fields and checkboxes (`RadioCheckField`). For signatures, we use the generic class `PdfFormField` and we create the signature field using the convenience method `createSignature()`. We choose a name, and we set some other field properties if necessary.

In a PDF form based on AcroForm technology, each field corresponds with zero, one or more widget annotations. These annotations define the visual appearance of the field on the document. In this case, we use `setPage()` to indicate the signature field has to be added to the current page. We use `setWidget()` to define the position on the page as well as the behavior when somebody clicks on the field widget. There are four different types of behavior:

- `HIGHLIGHT_NONE`— no highlighting.
- `HIGHLIGHT_INVERT`— inverts the content of the annotation square.
- `HIGHLIGHT_OUTLINE`— inverts the annotation border.
- `HIGHLIGHT_PUSH`— displays the annotation as if it were being pushed below the surface of the page

If a field corresponds with a single widget annotation (as is the case here), the field properties and the annotation properties are usually merged into a single dictionary object. We can add the field and its visual representation to a document by adding the `PdfFormField` object to the `PdfWriter` using the `addAnnotation()` method.

> **NOTE:** Is it possible to have one signature correspond with more than one widget? I'm sorry, but that's not a good question. See the spec about digital signature appearances by Adobe: "*The location of a signature within a document can have a bearing on its legal meaning. For this reason, signatures never refer to more than one annotation. If more than one location is associated with a signature the meaning may become ambiguous.*"

In code sample 2.6, we create an appearance for the empty signature using the `PdfAppearance` class. This class extends the `PdfTemplate` class used in iText to create small patches of reusable content. It's not to be mistaken with the `PdfSignatureAppearance` class. With `PdfAppearance`, you define what the field looks like *before* it's signed, whereas `PdfSignatureAppearance` defines what the field looks like *after* signing.

> **NOTE:** Creating an appearance for an empty signature field is optional: if you omit this code, a valid signature field will be added, but the end user might not really notice it. He'll only see a small orange ribbon added by Adobe Reader marking the location of the field.

The original, unsigned document and the resulting, signed document for code sample 2.6 are shown next to each other in figure 2.12.

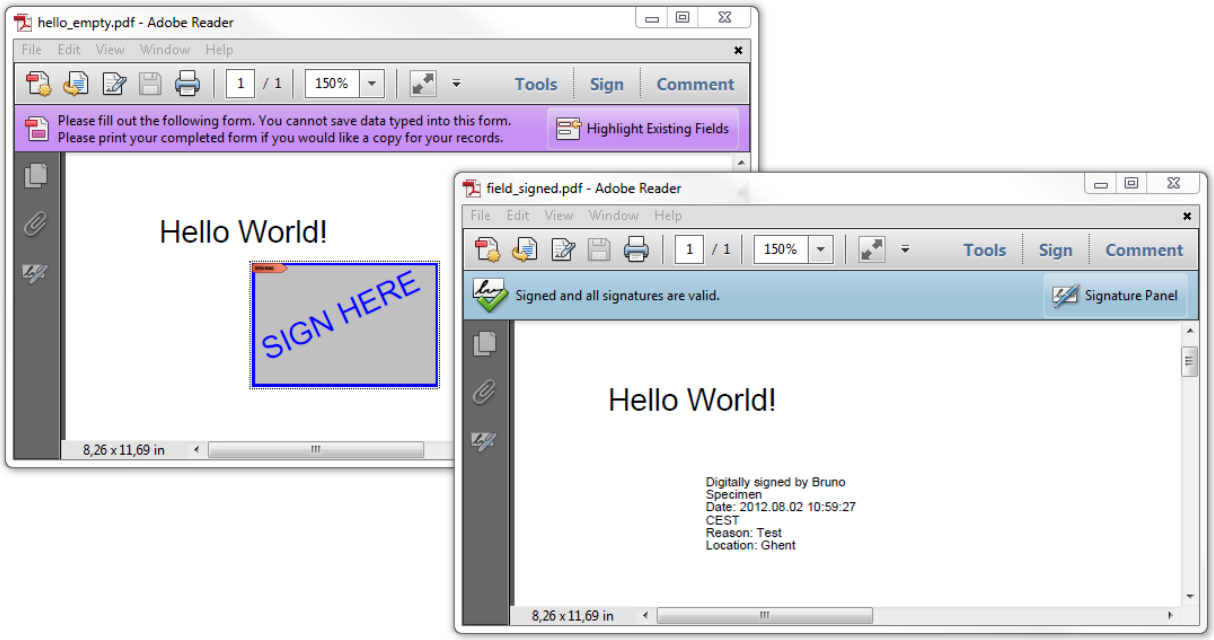

Figure 2.12: iText created document with empty signature field and the same document signed

If you want to add a signature field to specific page of an existing document, you can invoke the `addAnnotation()` method on a `PdfStamper` object passing the `PdfFormField` instance and a page number as parameters.

2.3.3 Adding an empty signature field to an existing document using iText

In code sample 2.7, we pass the page number as a parameter for the `addAnnotation()` method in `PdfStamper`. We don't need to define the page number on the level of the widget annotation.

Code sample 2.7: adding a signature field to an existing PDF

```
PdfReader reader = new PdfReader(src);
PdfStamper stamper = new PdfStamper(reader, new FileOutputStream(dest));
// create a signature form field
PdfFormField field = PdfFormField.createSignature(stamper.getWriter());
field.setFieldName(SIGNAME);
// set the widget properties
field.setWidget(new Rectangle(72, 732, 144, 780), PdfAnnotation.HIGHLIGHT_OUTLINE);
field.setFlags(PdfAnnotation.FLAGS_PRINT);
// add the annotation
stamper.addAnnotation(field, 1);
// close the stamper
stamper.close();
```

We can now reuse the method from code sample 2.5 to sign a document to which we've added a signature field using iText. Code sample 2.8 shouldn't have any secrets for you anymore.

Code sample 2.8: Signing a signature field created with iText

```
CreateEmptyField appCreate = new CreateEmptyField();
appCreate.createPdf(UNSIGNED);
BouncyCastleProvider provider = new BouncyCastleProvider();
Security.addProvider(provider);
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
ks.load(new FileInputStream(KEYSTORE), PASSWORD);
String alias = (String)ks.aliases().nextElement();
```

```
PrivateKey pk = (PrivateKey) ks.getKey(alias, PASSWORD);
Certificate[] chain = ks.getCertificateChain(alias);
SignEmptyField appSign = new SignEmptyField();
appSign.sign(UNSIGNED, SIGNAME, DEST, chain, pk, DigestAlgorithms.SHA256,
    provider.getName(), CryptoStandard.CMS, "Test", "Ghent");
```

The signature appearance as shown in the signed PDF of figure 2.12 is what a signature created by iText looks like by default. It contains the following information:

- *Who has signed the document?* — iText extracts this information from the certificate.
- *When was the document signed?* — If you provide a connection to a timestamp server, iText will use the date provided by the TSA (see chapter 3). Otherwise, iText will use the date and time of the computer that is used for signing, or a `Calendar` object provided in your code.
- *What was the reason for signing?* — The reason is provided by you or your program.
- *Where was the document signed?* — The location is provided by you or your program.

This information was chosen by the iText developers. It may not correspond with the way you want to present a signature, so let's find out how we can change this appearance.

2.4 Creating different signature appearances

Suppose your customer isn't used to digital signatures. Suppose he doesn't realize that the field marked with the text *"Digitally signed by…"* is the visual representation of a valid digital signature. Suppose that he wants to see an image of a wet ink signature instead of some plain text. That image as such wouldn't have any legal value whatsoever, but it can be reassuring on a psychological level. That's more or less what the recommendations in PAdES part 6 are about, and why iText provides different methods to create custom appearances for signatures.

2.4.1 Defining a custom PdfSignatureAppearance

In this section, I'm going to start by explaining something, and then I want you to completely forget all about it: in early versions of the PDF specification, a signature appearance consisted of five different layers that are drawn on top of each other.

These layers were numbered from n0 to n4:

- *n0*—Background layer.
- *n1*—Validity layer, used for the unknown and valid state.
- *n2*—Signature appearance, containing information about the signature.
- *n3*—Validity layer, used for the invalid state.
- *n4*—Text layer, for a text representation of the state of the signature

In old Acrobat versions, one would for instance create a graphic of a yellow question mark, and put that into layer n1. On top of this yellow question mark, in layer n2, you'd put the information about the signature. If the signature was made invalid, you'd see the content of layer n3, usually a red cross. Layers would be made visible or not, depending on the status of the signature.

Now please forget about these layers. Since Acrobat 6 (2003) the use of layers n1, n3 and n4 is no longer recommended. The only reason I mention them is to avoid questions about the following code snippet.

Code sample 2.9: Creating a custom appearance for the signature.

```
public void sign(String src, String name, String dest, Certificate[] chain,
    PrivateKey pk, String digestAlgorithm, String provider,
    CryptoStandard subfilter, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(name);
    // Creating the appearance for layer 0
    PdfTemplate n0 = appearance.getLayer(0);
    float x = n0.getBoundingBox().getLeft();
    float y = n0.getBoundingBox().getBottom();
    float width = n0.getBoundingBox().getWidth();
    float height = n0.getBoundingBox().getHeight();
    n0.setColorFill(BaseColor.LIGHT_GRAY);
    n0.rectangle(x, y, width, height);
    n0.fill();
    // Creating the appearance for layer 2
    PdfTemplate n2 = appearance.getLayer(2);
    ColumnText ct = new ColumnText(n2);
    ct.setSimpleColumn(n2.getBoundingBox());
    Paragraph p = new Paragraph("This document was signed by Bruno Specimen.");
    ct.addElement(p);
    ct.go();
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, subfilter);
}
```

In code sample 2.6, we created a `PdfAppearance` for the signature field before signing, but this appearance was lost and replaced with a default appearance chosen by iText. Code sample 2.9 now creates a custom appearance instead of the default one. The code is more complex than sample 2.1 and 2.5 because we use low-level methods to create a custom `PdfSignatureAppearance`. We use the `getLayer()` method to obtain specific layers, and we draw custom content for the background to layer 0, and information about the signature to layer 2.

> **NOTE:** If you've followed the advice I gave in the first line of this section, this is the point where you're supposed to ask: *Why are you only using layer 0 and layer 2? What happened to layer 1?* The answer is: there used to be a layer 1, 3 and 4 (and you'll find references to them in iText), but you should no longer use them. `PdfSignatureAppearance` will ignore all changes applied to these layers, unless you add the following line to your code: `appearance.setAcro6Layers(false);` (Again: this is not recommended!)

In code sample 2.9, we chose a gray rectangle as background and the text *"This document was signed by Bruno Specimen"* for the signature information. The result is shown in figure 2.13.

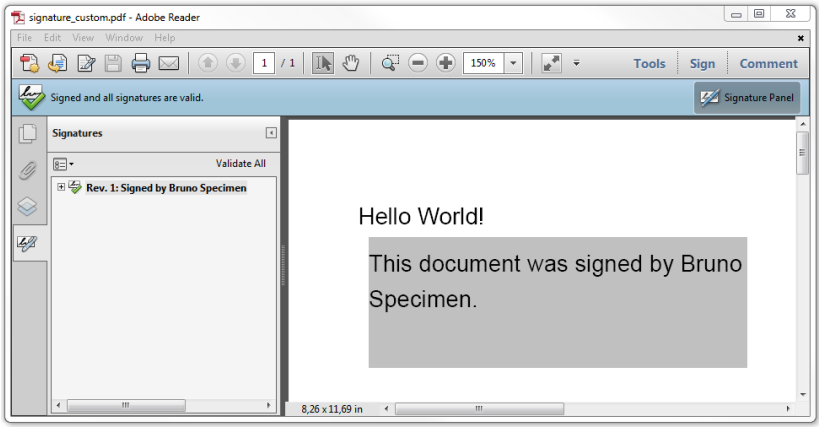

Figure 2.13: A signature with a custom appearance

Creating your own appearance is fun, but it demands more work. iText provides some convenience methods that allow you to create custom appearances in a more programmer-friendly way.

2.4.2 Creating signature appearances using convenience methods

Suppose you want to change the signature information, but you don't want to have to draw the text using `PdfContentByte` or `ColumnText` methods.

Custom text

Adding text is made easy using the `setLayer2Text()` and `setLayer2Font()` methods. There's even a `setRunDirection()` method if you want to add text that is written from right to left. Adding an image in the background of the text, can be done with the `setImage()` method.

These methods are used to create the signature widgets shown in figure 2.14.

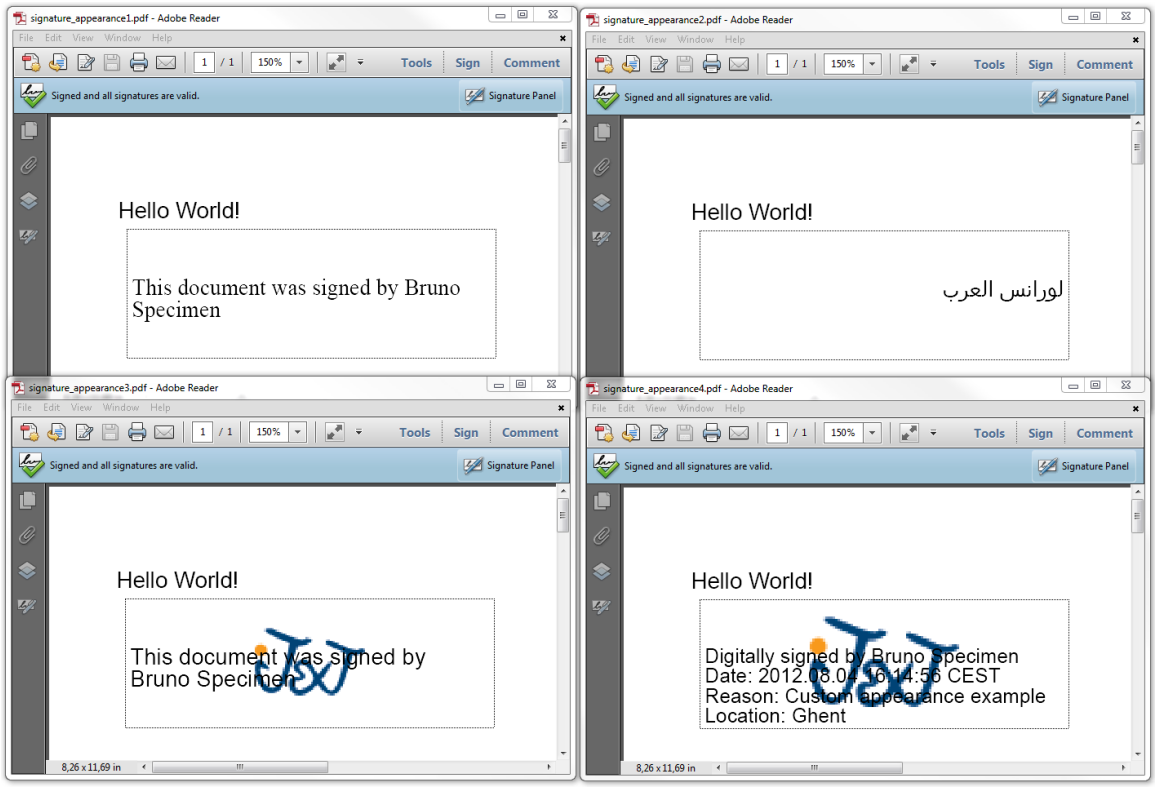

Figure 2.14: Signature appearances with custom text or background

Code sample 2.10 shows the relevant snippets of the complete source code for this example.

Code sample 2.10: Custom signature info and background image.

```
public void sign1(…)
    throws GeneralSecurityException, IOException, DocumentException {
    …
    // Custom text and custom font
    appearance.setLayer2Text("This document was signed by Bruno Specimen");
    appearance.setLayer2Font(new Font(FontFamily.TIMES_ROMAN));
    …
}
public void sign2(…)
    throws GeneralSecurityException, IOException, DocumentException {
    …
    // Custom text, custom font, and right-to-left writing
    appearance.setLayer2Text(
    "\u0644\u0648\u0631\u0627\u0646\u0633 \u0627\u0644\u0639\u0631\u0628");
    appearance.setRunDirection(PdfWriter.RUN_DIRECTION_RTL);
    appearance.setLayer2Font(
        new Font(BaseFont.createFont("C:/windows/fonts/arialuni.ttf",
        BaseFont.IDENTITY_H, BaseFont.EMBEDDED), 12));
    …
}
public void sign3(…)
    throws GeneralSecurityException, IOException, DocumentException {
    …
    // Custom text and background image
    appearance.setLayer2Text("This document was signed by Bruno Specimen");
    appearance.setImage(Image.getInstance(IMG));
    appearance.setImageScale(1);
    …
}
public void sign4(…)
    throws GeneralSecurityException, IOException, DocumentException {
    …
    // Default text and scaled background image
    appearance.setImage(Image.getInstance(IMG));
    appearance.setImageScale(-1);
    …
}
```

Observe that we use the run direction in the `sign2()` method to create the signature of Lawrence of Arabia in Arabic.

Custom images

In `sign3()`, we add an image using its original dimensions: the image scale is 100%. Change the scale value to `0.5f` if you want the image to be scaled to 50%, to `2f` if you want it scaled 200%.

If you omit the `setImageScale()` method, iText will scale the image to the absolute dimensions of the signature field, possibly changing the aspect ratio of the image.

If you want to avoid the distortion of the background image, you can pass a negative value as the image scale parameter as is done in the `sign4()` method.

Custom rendering mode

Apart from changing the signature info and the background, iText also allows you to choose from four different rendering modes:

- `RenderingMode.DESCRIPTION`—this is the default, it just shows whatever description was defined for layer 2.
- `RenderingMode.NAME_AND_DESCRIPTION`—this will split the signature field in two and add the name of the signer on one side, the description on the other side.
- `RenderingMode.GRAPHIC_AND_DESCRIPTION`—this will split the signature field in two and add an image on one side, the description on the other side.
- `RenderingMode.GRAPHIC`—the signature field will consist of an image only; no description will be shown.

Figure 2.15 shows an example of every rendering mode.

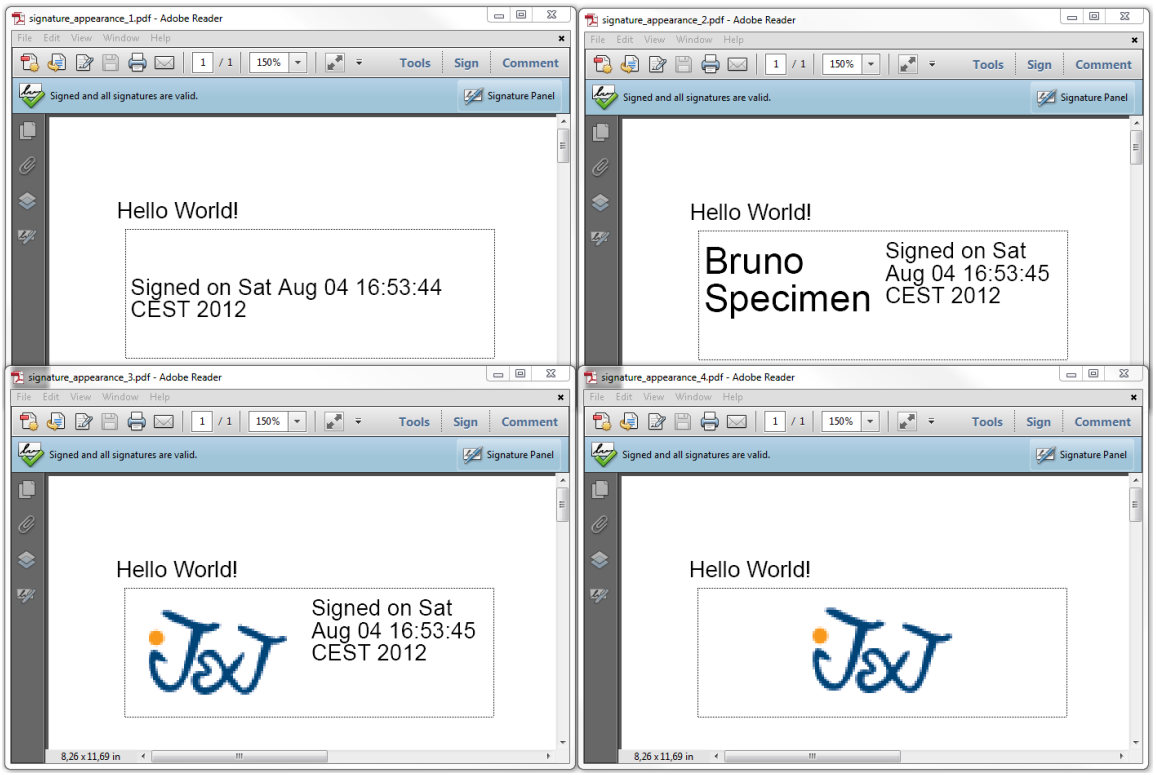

Figure 2.15: Signature appearances with different rendering modes

The rendering mode is set using the `setRenderingMode()` method. Make sure that you also define an image using the `setSignatureGraphic()` method if you choose the rendering mode `GRAPHIC_AND_DESCRIPTION` or `GRAPHIC`. See code sample 2.11.

Code sample 2.11: Changing the rendering mode and adding a signature graphic

```
PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
appearance.setReason(reason);
appearance.setLocation(location);
appearance.setVisibleSignature(name);
appearance.setLayer2Text("Signed on " + new Date().toString());
appearance.setRenderingMode(renderingMode);
appearance.setSignatureGraphic(image);
```

So far, we've always added a reason and a location to the signature appearance. This information is present in the signature dictionary, but this isn't the only metadata we can add.

2.4.3 Adding metadata to the signature dictionary

The PDF specification allows you to add the following metadata to the signature dictionary:

- *Name*— the name of the person or authority signing the document. This value should be used only when it's not possible to extract the name from the signature.
- *M*— the time of signing. Depending on the signature handler, this may be a normal unverified computer time or a time generated in a verifiable way from a secure server. ISO-32000-1 tells us *this should only be used when the time of signing isn't available in the signature*, but iText will always add this entry anyway — it doesn't hurt.
- *Location*— the CPU host name or the physical location of the signing.
- *Reason*— the reason for signing, such as *"I agree"*.
- *ContactInfo*— information provided by the signer to enable a recipient to contact the signer to verify the signature, for instance a phone number.

The name and the signature time are set by iText automatically, but you can override them. Let's take a look at code sample 2.12 to find out how this is done.

Code sample 2.12: Setting the metadata for the signature dictionary

```
PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
appearance.setReason(reason);
appearance.setLocation(location);
appearance.setVisibleSignature(name);
appearance.setContact(contact);
appearance.setSignDate(signDate);
appearance.setSignatureEvent(
    new SignatureEvent(){
        public void getSignatureDictionary(PdfDictionary sig) {
            sig.put(PdfName.NAME, new PdfString(fullName));
        }
    }
);
```

We change the signature time using the `setSignDate()` method. This is a dangerous method: if you choose a date that predates the certificate, you'll get a warning (a yellow triangle message) saying *"Signer's identity is invalid because it has expired or is not yet valid."* If you choose a date in the future, the signature will be broken (a red-cross icon) with the message *"There are errors in the formatting or information contained in this signature (Signature seems to be signed in future)."*

> **NOTE:** It's NOT a good idea to use the `setSignDate()` method. Apart from risking error messages in Adobe Reader, using the wrong date when signing a document may also be considered as fraud. Moreover, many instances won't accept a document that is signed without an official timestamp from a TSA. See chapter 3 to find out why a document that is signed without a timestamp could be considered as a useless document.

If you take a closer look to code sample 2.12, you see that there are methods to set the reason, location, and contact info, but suppose that you want to add a key for which there's no method available in iText.

In that case, you can use the `SignatureEvent` interface. Its `getSignatureDictionary()` method will be called right before the dictionary is written to the file, so you can use it to add extra keys or to override existing ones. In the code sample, I added a name "Bruno L. Specimen" which is slightly different from the name on the certificate.

> **NOTE:** This example is meant to explain a concept. Don't use it in your applications. Many of these metadata values aren't visible in Adobe Reader. For example: Adobe Reader will prefer the name obtained from the certificate instead of the name added in the signature dictionary (which should only be used in case the name can't be retrieved from the certificate).

You'll need to look inside the PDF document (for instance using iText RUPS) to see the information we added in code sample 2.12. Figure 2.16 shows such a look inside.

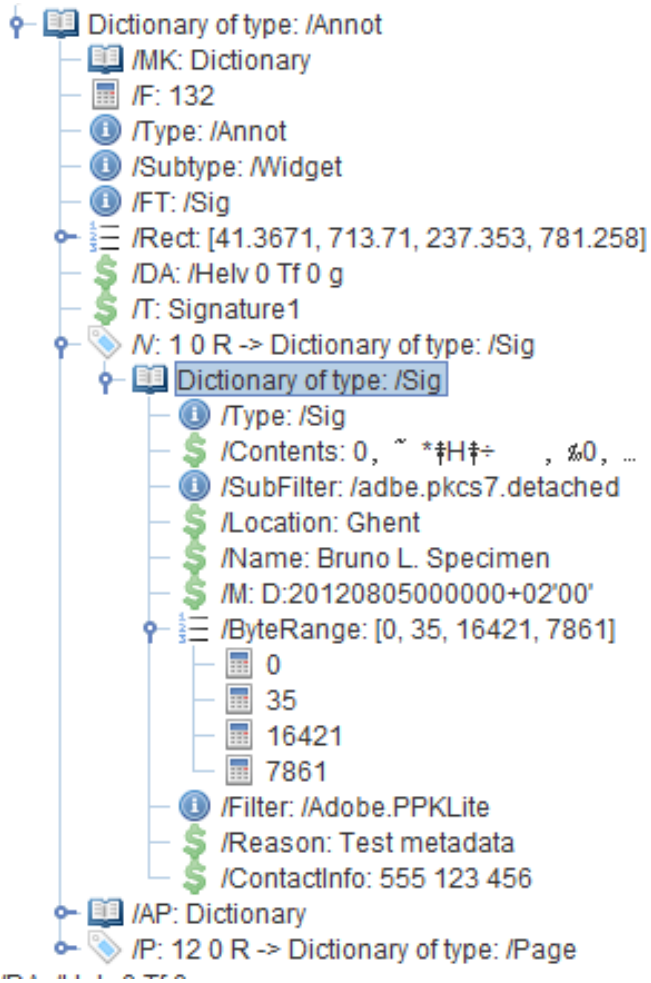

Figure 2.16: A look inside a signed PDF showing the keys of the Signature dictionary

There's one more method, actually a very important one, we haven't discussed yet: a method to set the certification level.

2.4.4 Ordinary and Certifying signatures

There are three different types of signatures. One of them isn't relevant in the context of this paper. Two of them are shown in figure 2.17.

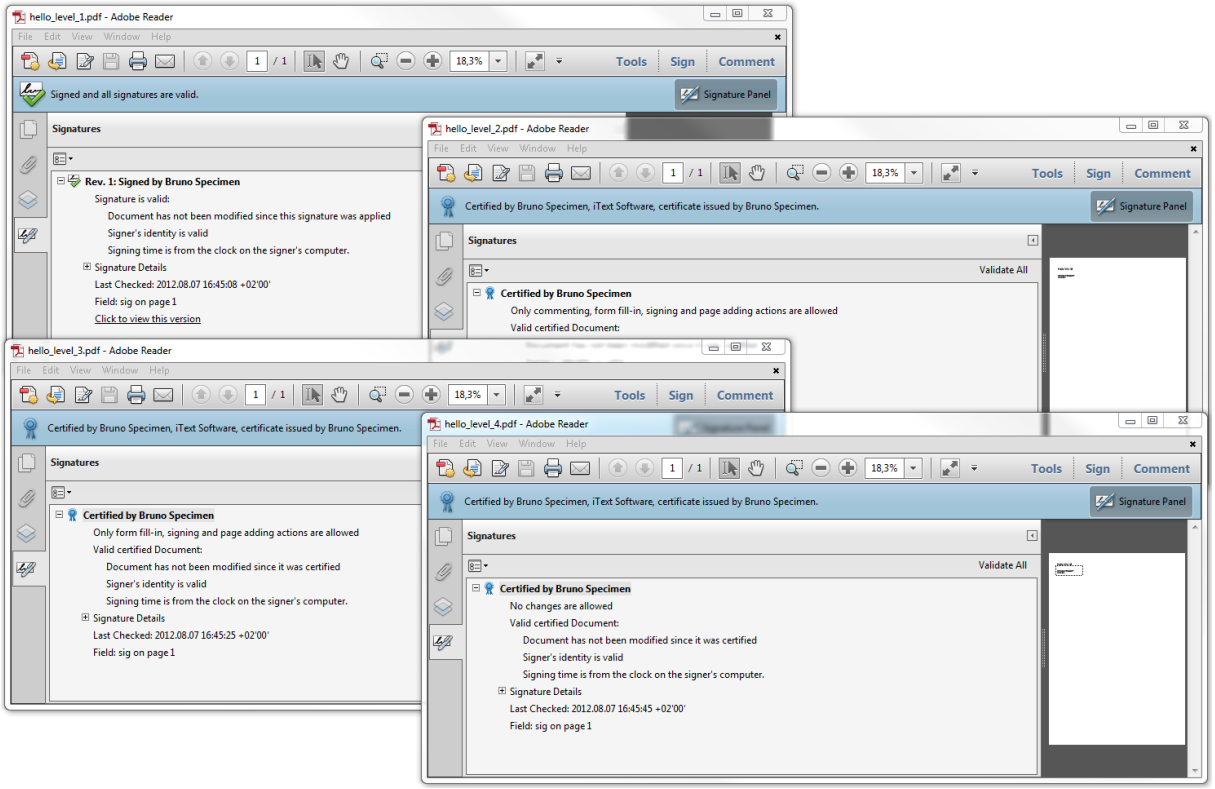

Figure 2.17: Ordinary (approval) and Certification (author) signature

ISO-32000-1 specifies that a PDF document may contain the following standard types of signatures:

- *One or more approval signatures*— these signatures appear in signature form fields. The upper left document in figure 2.17 is an example of such a signature. Adobe Reader shows a green check mark when a valid approval signature is present.
- *At most one certification signature*— this signature can define different permissions, as shown in figure 2.17. You'll see a blue ribbon when a document was certified.
- *At most two usage rights signatures*— these are signatures created using a private key that is proprietary for instance to Adobe. When a PDF document is signed with a usage rights signature using Adobe's key, the document is Reader enabled. Adobe Reader will unlock certain usage rights, for instance allowing you to save a filled out form locally. Other vendors can use their private key to enforce similar usage rights. This type of signature is outside the scope of this paper.

Let's recycle code sample 2.1 once more, and add one line:

Code sample 2.13: Setting the certification level

```
public void sign(String src, String dest, Certificate[] chain, PrivateKey pk,
    String digestAlgorithm, String provider, CryptoStandard subfilter,
    int certificationLevel, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
```

```
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    appearance.setCertificationLevel(certificationLevel);
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, subfilter);
}
```

In code sample 2.13, we pass a parameter named `certificationLevel`. This parameter can have one of the following values available as constants in the `PdfSignatureAppearance` class:

- *NOT_CERTIFIED*— creates an ordinary signature aka an approval or a recipient signature. A document can be signed for approval by one or more recipients.
- *CERTIFIED_NO_CHANGES_ALLOWED*— creates a certification signature aka an author signature. After the signature is applied, no changes to the document will be allowed.
- *CERTIFIED_FORM_FILLING*— creates a certification signature for the author of the document. Other people can still fill out form fields or add approval signatures without invalidating the signature.
- *CERTIFIED_FORM_FILLING_AND_ANNOTATIONS*— creates a certification signature. Other people can still fill out form fields- or add approval signatures as well as annotations without invalidating the signature.

Up until now, we've always created documents containing a single signature. In section 2.5, we'll describe different workflows adding multiple signatures to the same document.

2.4.5 Adding content after a document was signed

Unless otherwise specified, for instance using a signature lock dictionary changing the signatures *modification detection and prevention* (MDP) level (see section 2.5.5), you can always fill out fields, add extra signatures and annotations to a document that was signed using an ordinary signature, but make sure you do it correctly. Code sample 2.14 shows how NOT to do it:

Code sample 2.14: how NOT to add an annotation to a signed document

```
public void addWrongAnnotation(String src, String dest)
    throws IOException, DocumentException {
    PdfReader reader = new PdfReader(src);
    PdfStamper stamper = new PdfStamper(reader, new FileOutputStream(dest));
    PdfAnnotation comment = PdfAnnotation.createText(stamper.getWriter(),
        new Rectangle(200, 800, 250, 820), "Finally Signed!",
        "Bruno Specimen has finally signed the document", true, "Comment");
    stamper.addAnnotation(comment, 1);
    stamper.close();
}
```

This example will work perfectly for PDF documents without any signature. `PdfStamper` will take the objects that are read by `PdfReader`, add new objects that represent a text annotation, and reorganize the objects, thus creating a new document. The internal structure of this new PDF file can be quite different when compared to the original file. It goes without saying that this will break the signature.

Code sample 2.15 shows how to preserve the original bytes that were signed:

Code Sample 2.15: how to add an annotation to a signed document

```
public void addAnnotation(String src, String dest)
    throws IOException, DocumentException {
    PdfReader reader = new PdfReader(src);
    PdfStamper stamper =
        new PdfStamper(reader, new FileOutputStream(dest), '\0', true);
    PdfAnnotation comment = PdfAnnotation.createText(stamper.getWriter(),
        new Rectangle(200, 800, 250, 820), "Finally Signed!",
        "Bruno Specimen has finally signed the document", true, "Comment");
    stamper.addAnnotation(comment, 1);
    stamper.close();
}
```

Did you spot the difference? The code is almost identical. We only used a different constructor for `PdfStamper`. The zero byte means we don't want to change the version number of the PDF file. The Boolean value indicates whether or not we want to manipulate the file in *'append mode'*. This value is `false` by default. The original bytes aren't preserved. By changing this value to `true`, we tell iText not to change any of the original bytes.

When using `PdfStamper` in append mode, the extra objects defining the annotation will be added after the original `%%EOF` statement. Some other objects will be overridden, but the original file structure will be kept intact. As a result, an ordinary signature will not be broken when using code sample 2.15.

If we take a closer look at figure 2.17, we see messages such as *'only commenting, form fill-in, signing and page adding actions are allowed.'* That's a document signed with certification level 'form filling and annotations.' Code sample 2.15 won't invalidate such a signature either.

> **NOTE:** Be aware that '*page adding actions are allowed*' doesn't mean you can use the `insertPage()` method. This message refers to page template instantiation as described in the Adobe Acrobat JavaScript Reference Manual, which is out of scope in this paper.

There are two more PDFs with another message *'Only form fill-in, signing and page adding actions are allowed'* and *'No changes are allowed.'* Adding an annotation as is done in code sample 2.15 will invalidate those signatures.

This is shown in figure 2.18. The upper window shows a certifying signature with a certification level that allows adding a text annotation aka 'commenting'. It's still a *'Valid certified Document'*. The lower window shows a certifying signature with a certification level that doesn't allow any changes. As we've added an annotation, the *'Signature is invalid'*.

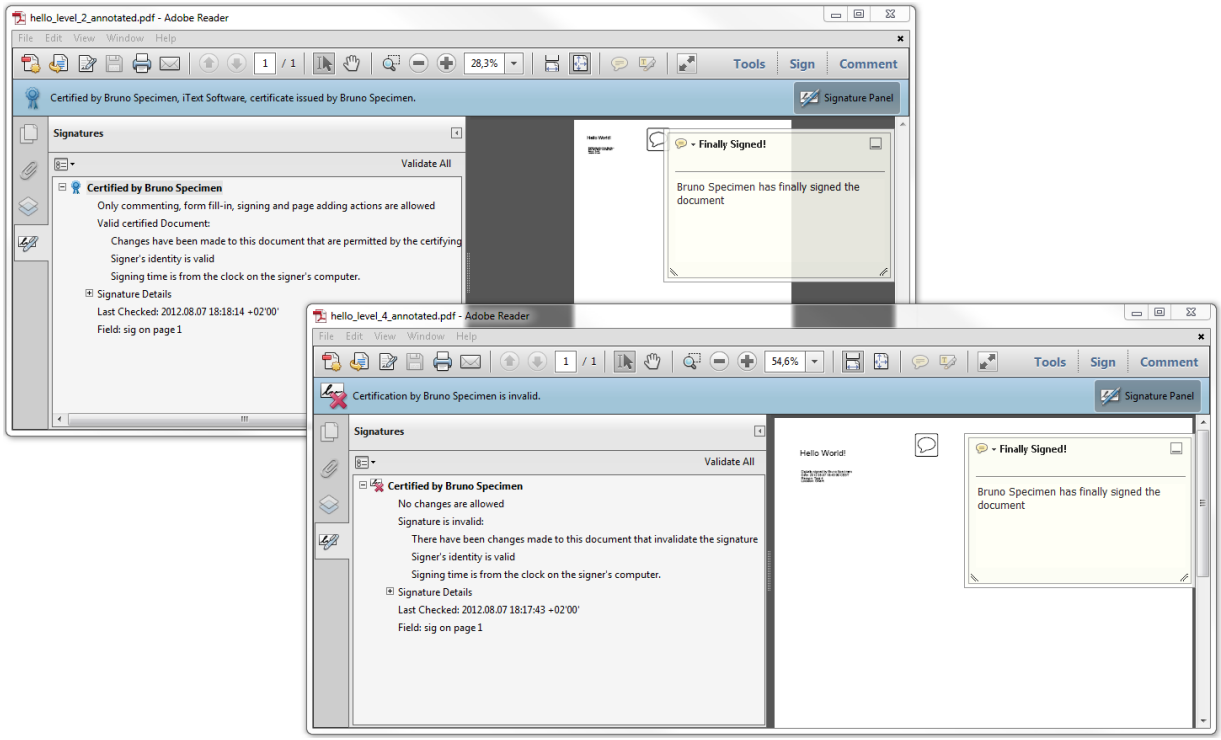

Figure 2.18: Preserving versus invalidating a signature

It's important to understand that adding or removing any other content will invalidate any type of signature. For instance: it's not possible to add some extra words to a page.

Code sample 2.16: Breaking a signature by adding text

```
public void addText(String src, String dest)
    throws IOException, DocumentException {
    PdfReader reader = new PdfReader(src);
    PdfStamper stamper =
        new PdfStamper(reader, new FileOutputStream(dest), '\0', true);
    ColumnText.showTextAligned(stamper.getOverContent(1),
        Element.ALIGN_LEFT, new Phrase("TOP SECRET"), 36, 820, 0);
    stamper.close();
}
```

Code sample 2.16 works for most ordinary PDFs, but it will always break the signature of a signed PDF, even if we create the `PdfStamper` object in append mode.

The code to add a second signature (and a third, and a fourth…) is similar to the code adding an annotation. Code sample 2.17 won't break existing signatures unless the modification detection and prevention level of one of the signatures is 'no changes allowed'.

Code sample 2.17: adding an extra signature

```
public void signAgain(String src, String dest, Certificate[] chain, PrivateKey pk,
    String digestAlgorithm, String provider, CryptoStandard subfilter,
    String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0', null, true);
```

```
        // Creating the appearance
        PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
        appearance.setReason(reason);
        appearance.setLocation(location);
        appearance.setVisibleSignature(
            new Rectangle(36, 700, 144, 732), 1, "Signature2");
        // Creating the signature
        ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
        ExternalDigest digest = new BouncyCastleDigest();
        MakeSignature.signDetached(appearance, digest, pks, chain,
            null, null, null, 0, subfilter);
    }
```

In this code snippet, Bruno Specimen is signing the same document with a second signature. This may seem strange, but in some cases it makes perfect sense. Suppose that Bruno Specimen owns two different companies. When signing an intercompany agreement, he needs to sign once as managing director of company A and once as managing director of company B.

Figure 2.19 shows two examples, in the upper window, we have a document that is signed twice by Bruno Specimen, once with a certification signature that allowed adding form fields and once with an approval signature. In the lower window, the first signature was a certification signature that doesn't allow a second signature.

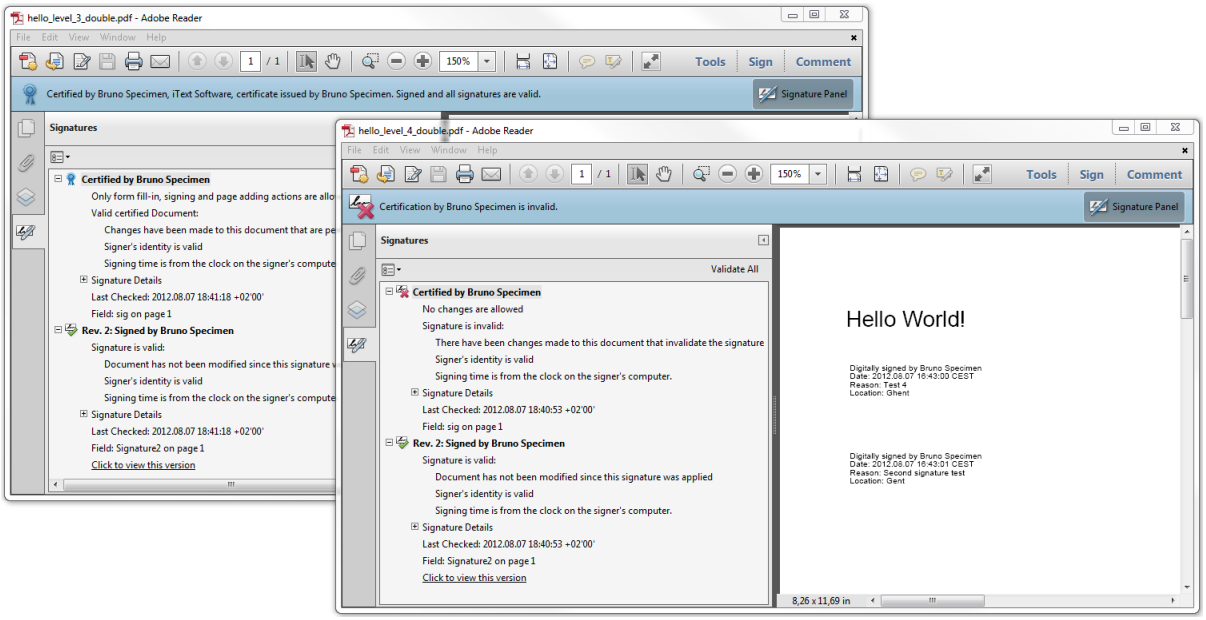

Figure 2.19: Documents that are signed twice

This functionality is very useful in a workflow where different people have to approve a document.

2.5 Signatures in PDF and workflow

Imagine a book publisher drawing up a contract for a book written by two authors. Such a book contract could contain one certification signature from the publisher, saying: *"I have written this contract. This is legally binding."* In addition, the authors could add their approval signature, saying: *"We agree to the terms of the contract."*

When I wrote my first books for Manning publications, I received a paper contract that I had to sign with a wet ink signature. I then had to mail it from Belgium to the US using snail mail. Wouldn't it be great if we could use digital signatures to achieve this?

2.5.1 Sequential signatures in PDF

Figure 2.20 shows what a PDF that is signed multiple times looks like on the inside.

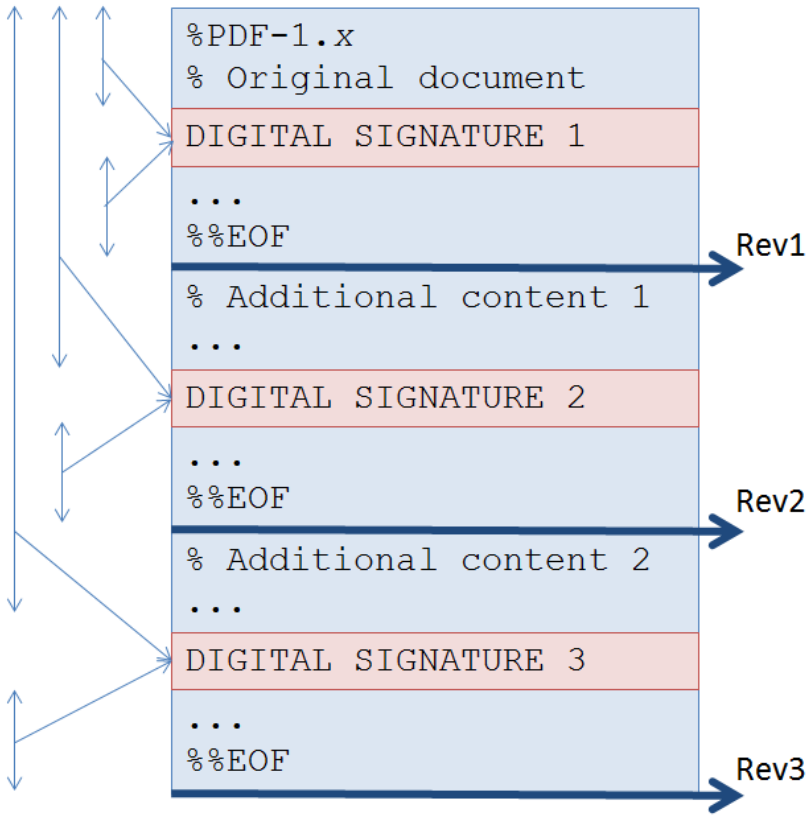

Figure 2.20: Schematic view of a PDF file that is signed trice

The part above the line marked with Rev1 is revision 1 of the document. It's identical to what we had in figure 2.1. When signing the document with a second signature, we don't change any of the bytes of revision 1. We add additional content (provided that the first signature allows this content), and we create a new signature. This new signature is based on a message digest that includes the entire byte array of revision 1. The result is revision 2. When signing the document with a third signature, the bytes of revision 2 are preserved.

> **NOTE:** Looking at figure 2.20, you see that the signatures have to be applied sequentially. It's not possible to sign in parallel. For instance: a publisher can't send his contract to two authors at the same time for approval, and then merge the signed documents afterwards if the signatures need to be in the same document (as opposed to bundled in a portfolio). One author always has to sign the contract first, then followed by the other author.

We've already seen an example of a document that was signed twice by Bruno Specimen. In the next section, we'll make some examples that need to be signed by different parties.

2.5.2 Creating a form with placeholders for multiple signatures

Figure 2.21 is a form with three empty signature fields. In code samples 2.6 and 2.7, we created such fields using the `PdfFormField` class and we used a hardcoded `Rectangle` to define its position.

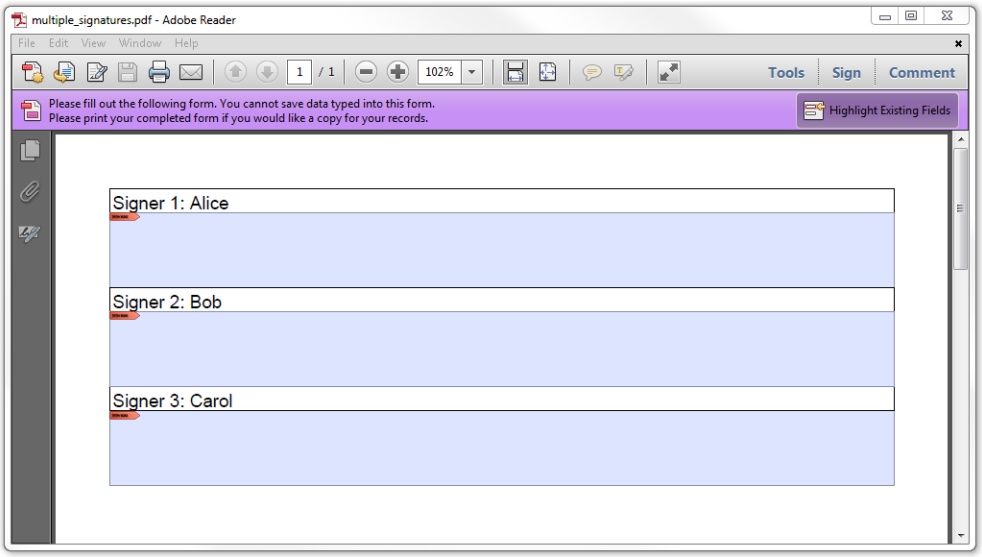

Figure 2.21: This form needs to be signed by Alice, Bob, and Carol

In code sample 2.18, we'll let iText define the position. We'll create a table with one column, and we'll use cell events to add the signature fields at the correct position in the table.

Code sample 2.18: Creating a form with empty fields

```
public void createForm() throws IOException, DocumentException {
    Document document = new Document();
    PdfWriter writer = PdfWriter.getInstance(document, new FileOutputStream(FORM));
    document.open();
    PdfPTable table = new PdfPTable(1);
    table.setWidthPercentage(100);
    table.addCell("Signer 1: Alice");
    table.addCell(createSignatureFieldCell(writer, "sig1"));
    table.addCell("Signer 2: Bob");
    table.addCell(createSignatureFieldCell(writer, "sig2"));
    table.addCell("Signer 3: Carol");
    table.addCell(createSignatureFieldCell(writer, "sig3"));
    document.add(table);
    document.close();
}
protected PdfPCell createSignatureFieldCell(PdfWriter writer, String name) {
    PdfPCell cell = new PdfPCell();
    cell.setMinimumHeight(50);
    PdfFormField field = PdfFormField.createSignature(writer);
    field.setFieldName(name);
    field.setFlags(PdfAnnotation.FLAGS_PRINT);
    cell.setCellEvent(new MySignatureFieldEvent(field));
    return cell;
}
public class MySignatureFieldEvent implements PdfPCellEvent {
    public PdfFormField field;
    public MySignatureFieldEvent(PdfFormField field) {
        this.field = field;
    }
```

```
    public void cellLayout(PdfPCell cell, Rectangle position,
            PdfContentByte[] canvases) {
        PdfWriter writer = canvases[0].getPdfWriter();
        field.setPage();
        field.setWidget(position, PdfAnnotation.HIGHLIGHT_INVERT);
        writer.addAnnotation(field);
    }
}
```

The `createForm()` method is pretty straightforward. We create a document from scratch in five steps. In step four, we create a table with one column; we define its width percentage. Then we add six cells, and we add the table to the document.

Three cells are created in a separate method. The `createSignatureFieldCell()` method creates a `PdfPCell` instance, we define a minimum width, and then we create a signature field. We don't define a widget yet. Instead we pass the field object to a `PdfPCellEvent` and we add an instance of this event to the cell.

What does this event do? After the cell is rendered to the page by iText, the `cellLayout()` method is invoked. In this method, we set the page and the position of the signature field, and we add the signature field to the `PdfWriter`.

Now let's sign this form three times.

2.5.3 Signing a document multiple times

Please read figure 2.22 from top to bottom. There's revision 1 signed by Alice using a certification signature. Two blank signature fields remain. Revision two is signed by Bob. Finally revision 3 is signed by Carol.

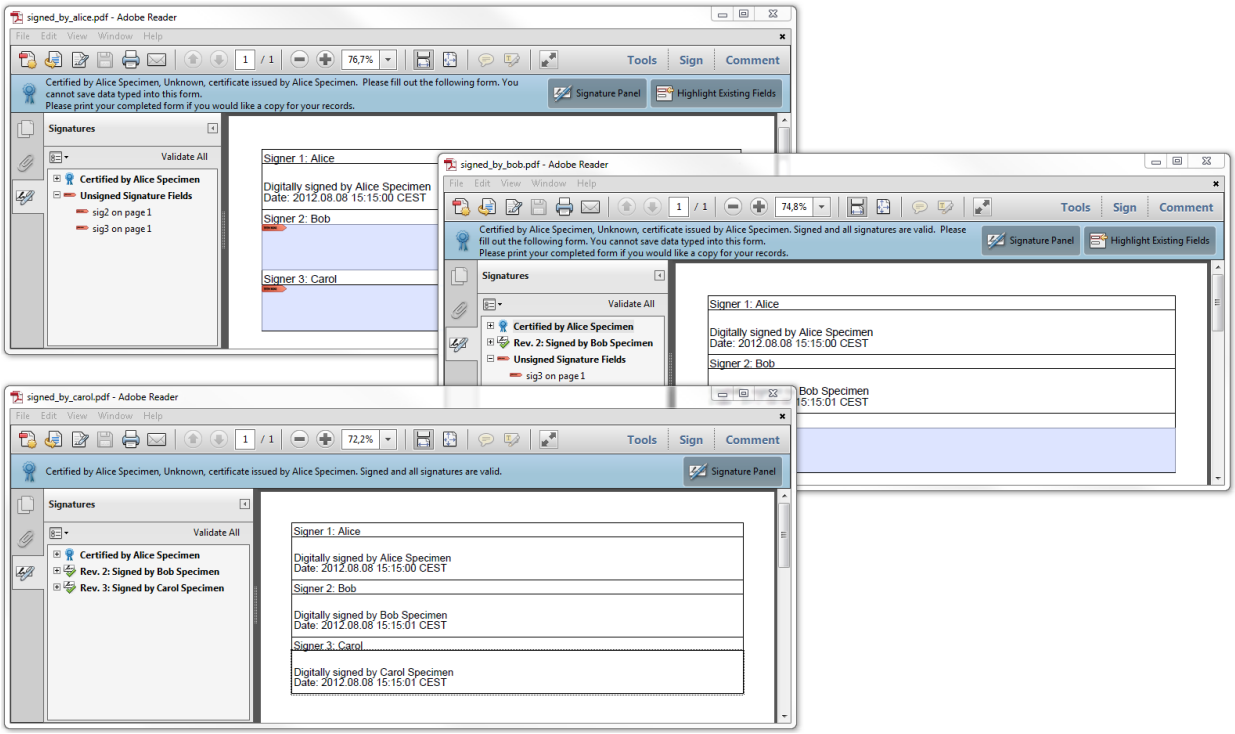

Figure 2.22: Signing a document thrice

They all signed their revision using code sample 2.19.

Code sample 2.19: Signing in append mode

```
public void sign(String keystore, int level,
    String src, String name, String dest)
    throws GeneralSecurityException, IOException, DocumentException {
    KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
    ks.load(new FileInputStream(keystore), PASSWORD);
    String alias = (String)ks.aliases().nextElement();
    PrivateKey pk = (PrivateKey) ks.getKey(alias, PASSWORD);
    Certificate[] chain = ks.getCertificateChain(alias);
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0', null, true);
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setVisibleSignature(name);
    appearance.setCertificationLevel(level);
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, "SHA-256", "BC");
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, CryptoStandard.CMS);
}
```

Alice, Bob and Carol signed the form in the order as defined in code sample 2.20. Note that the value of `DEST` in this example isn't a path (as was the case in previous examples), but a pattern for a path.

Code sample 2.20: Signing a document multiple times

```
app.sign(ALICE, PdfSignatureAppearance.CERTIFIED_FORM_FILLING, FORM,
    "sig1", String.format(DEST, "alice"));
app.sign(BOB, PdfSignatureAppearance.NOT_CERTIFIED, String.format(DEST, "alice"),
    "sig2", String.format(DEST, "bob"));
app.sign(CAROL, PdfSignatureAppearance.NOT_CERTIFIED, String.format(DEST, "bob"),
    "sig3", String.format(DEST, "carol"));
```

That is: first Alice added a certification signature, and then Bob and Carol added an approval signature. Suppose we switched the order. Suppose that Alice and Bob first approve the document, and that Carol certifies the document afterwards. Or suppose that both Alice and Carol try adding a certification signature, what will happen?

I never questioned this before I wrote this paper, because when discussing modification detection and prevention using certification signatures, ISO-32000-1 mentions the author of a document as *'the person applying the first signature'*[14].

Out of curiosity, I tried three different setups and I was surprised by the result. Apparently, the certification signature doesn't have to be the first signature in the document. I don't know if it makes sense, but Adobe Reader validates documents that are signed by approval signatures first, followed by a certification signature as correct, provided that the certification level allows form filling.

[14] See ISO-32000-1, section 12.8.8.2.1 §1

I was also surprised by the output if the final signature is a certification signature that doesn't allow form filling. If you look closely at figure 2.23, you can see that Carol's certification signature is valid, but the original signatures by Alice and Bob are invalid. Somehow this doesn't make sense, does it?

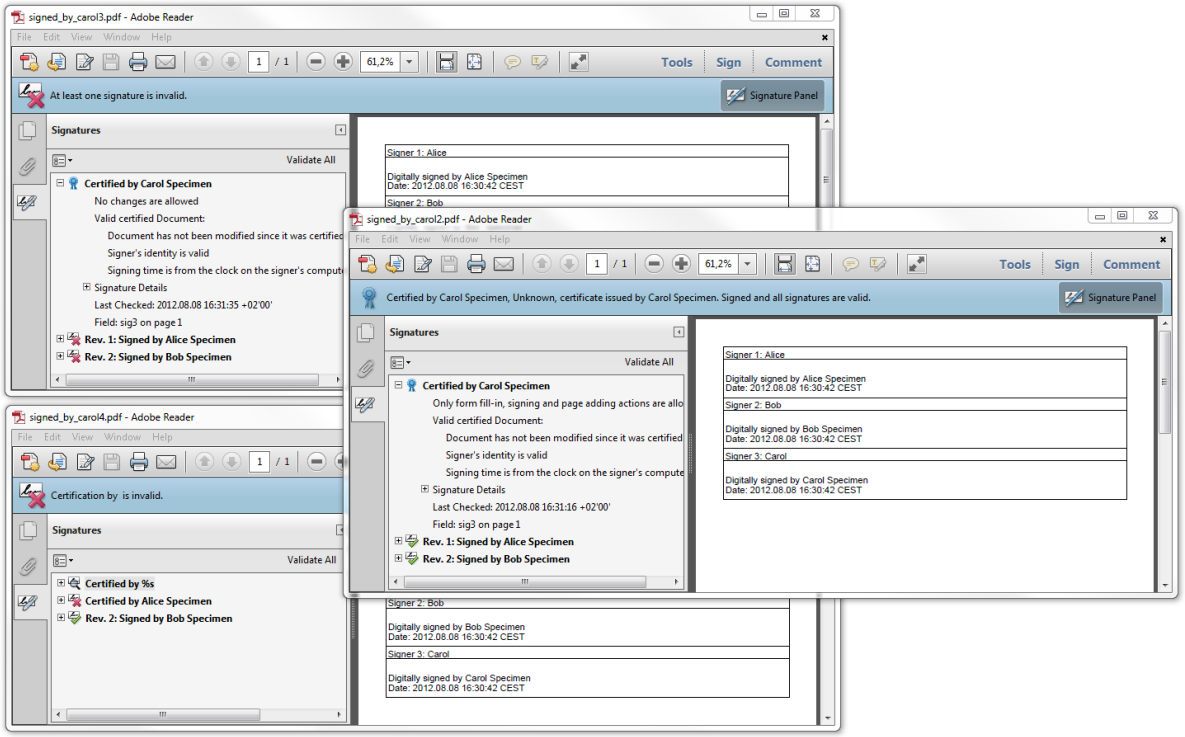

Figure 2.23: Strange results for strange signature operations

As expected, Alice's certification signature is invalid after Carol tried to sign the document using a second certification signature, but the output is kind of strange. In the signature panel, we see *'Certified by %s'* instead of *'Certified by Carol'*, and there's a looking glass instead of a red cross. These situations are rather exotic and should be avoided. Let's focus on real life examples, and combine signing with form filling.

2.5.4 Signing and filling out fields multiple times

Let's create another form that needs to be signed by four parties. See the form shown in figure 2.24.

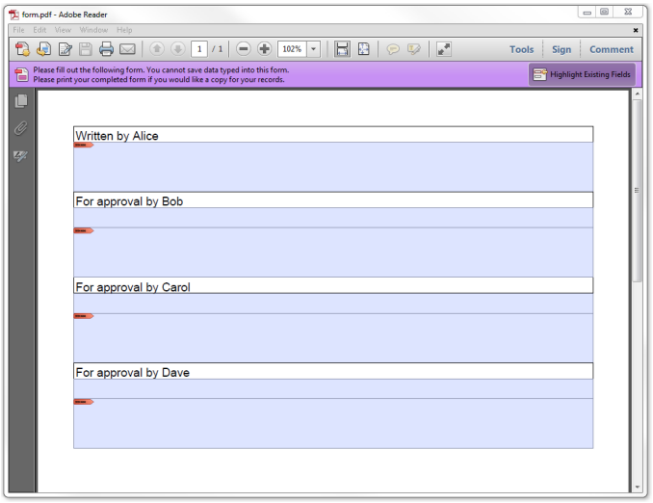

Figure 2.24: A form containing signature fields as well as text fields.

Again, Alice will be the author of the document. She'll sign first using a certification signature that allows form filling. Secondly, Bob will have to fill in a field, writing *"Read and approved by Bob"*, after which he signs the document with an approval signature. Thirdly, Carol needs to do the same thing. Finally, Dave approves the document as the fourth party in the workflow. Figures 2.25 to 2.30 show different steps in the process.

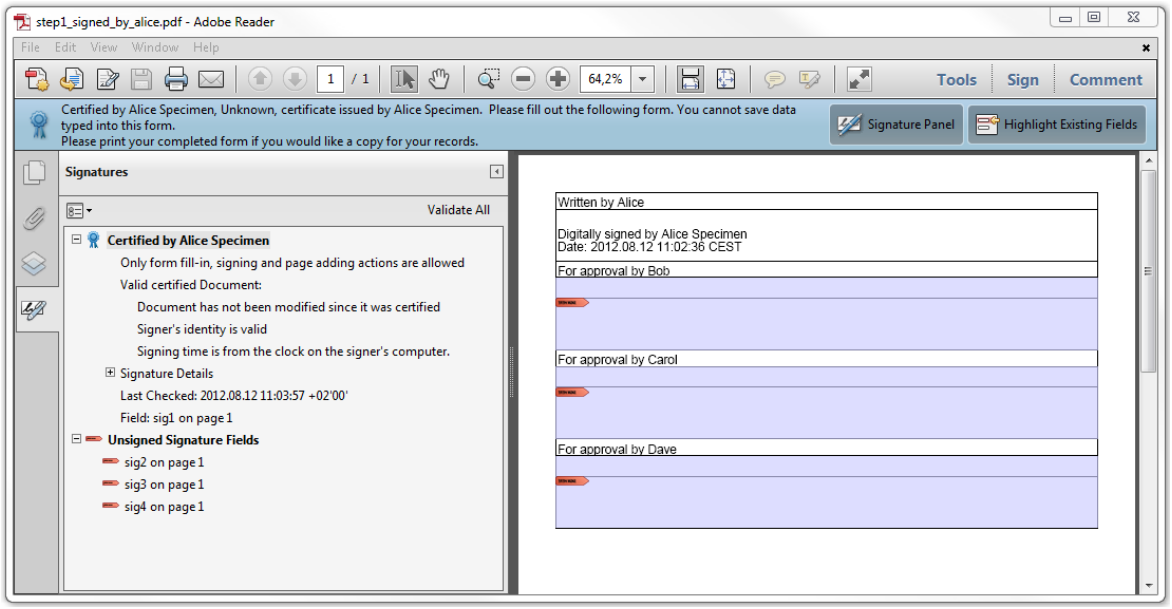

Figure 2.25: step 1, the document is certified by Alice

In figure 2.26, Bob has filled out the field named "approved_bob". The signature panel says that *'Changes have been made to this document that are permitted by the certifying party'*.

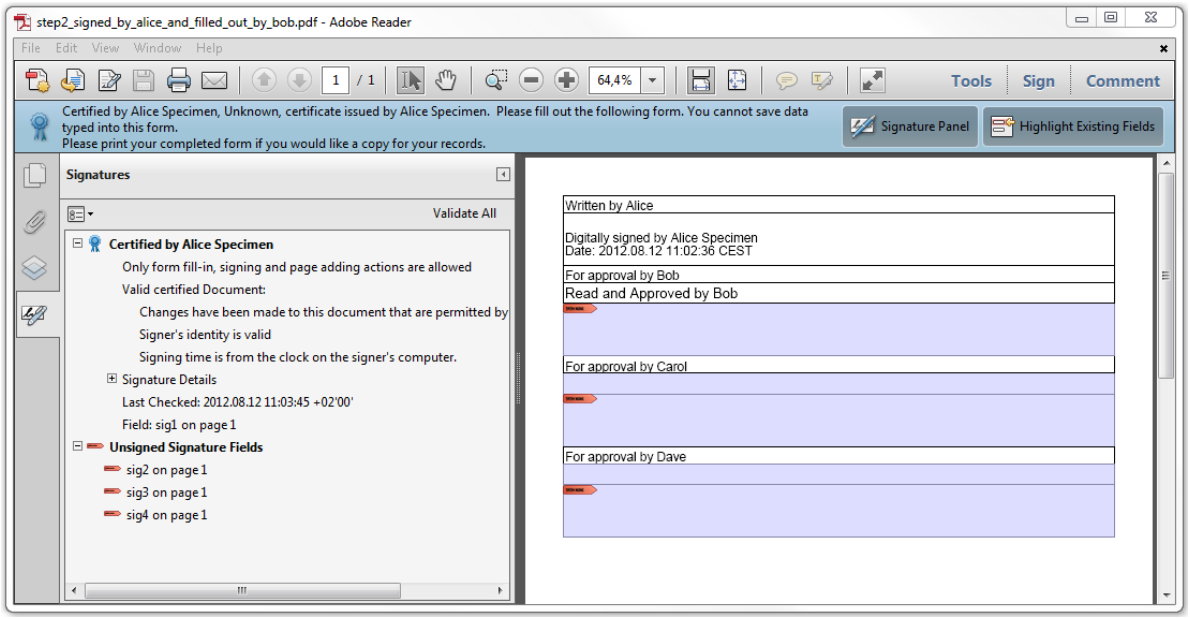

Figure 2.26: step 2, Bob says he has read and approved the document

Code sample 2.21 shows how you could fill out a form field in an interactive form (based on AcroForm technology) using iText.

Code sample 2.21: filling out a form using iText

```
public void fillOut(String src, String dest, String name, String value)
    throws IOException, DocumentException {
    PdfReader reader = new PdfReader(src);
    PdfStamper stamper =
        new PdfStamper(reader, new FileOutputStream(dest), '\0', true);
    AcroFields form = stamper.getAcroFields();
    form.setField(name, value);
    form.setFieldProperty(name, "setfflags", PdfFormField.FF_READ_ONLY, null);
    stamper.close();
}
```

As you can see, we're using `PdfStamper` in append mode to avoid breaking Alice's signature. The rest of the code is pretty standard in iText. You get an `AcroFields` object from the stamper, and you fill out fields using the `setField()` method, passing a key —the name as it was defined in the form— and a value.

We also set the field to read only so that somebody further down the workflow doesn't accidentally change the field. The method `setFieldProperty()` expects four parameters:

- *field*— the name of the field of which you want to change a property
- *action*— the action you want to perform ("`fflags`" will replace all field flags, "`setfflags`" will add field flags, "`clrfflags`" will remove all field flags).
- *value*— one or more properties set using field flags you want to change. See the constants starting with `FF_` in the `PdfFormField` class).
- *Inst[]*— an array of indexes to select specific field items. In this case, each field corresponds with a single item, so it's safe to use `null` (which will apply the property to *all* items).

In the next step, Bob signs with an approval signature. See figure 2.27:

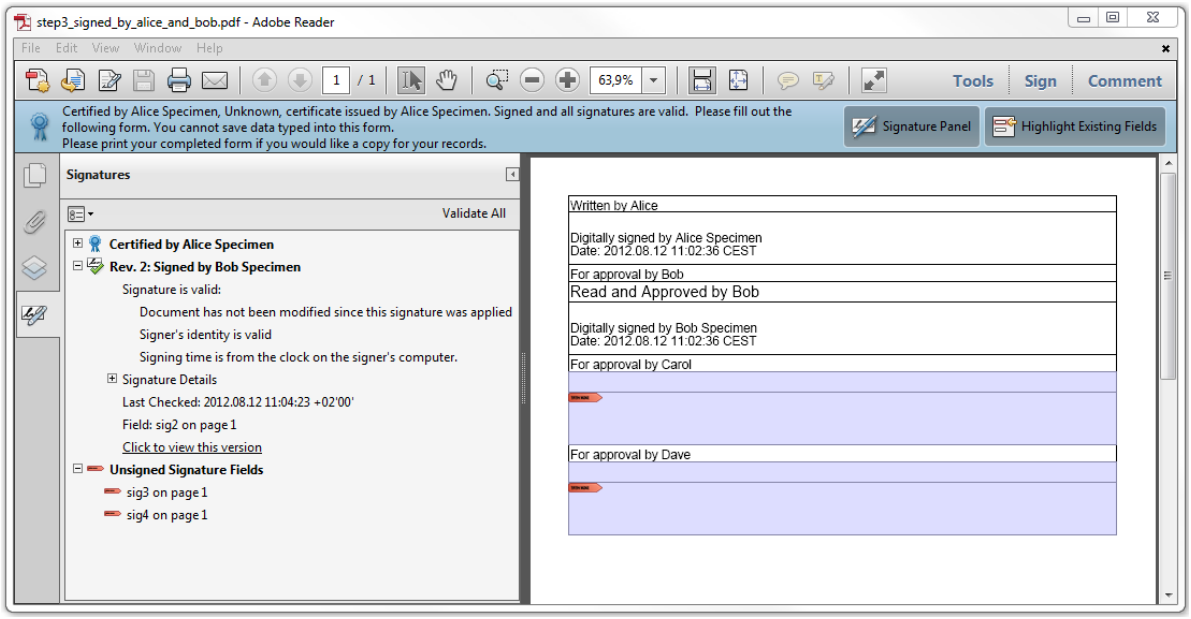

Figure 2.27: step 3, Bob has signed the document for approval

Now Carol repeats these two steps.

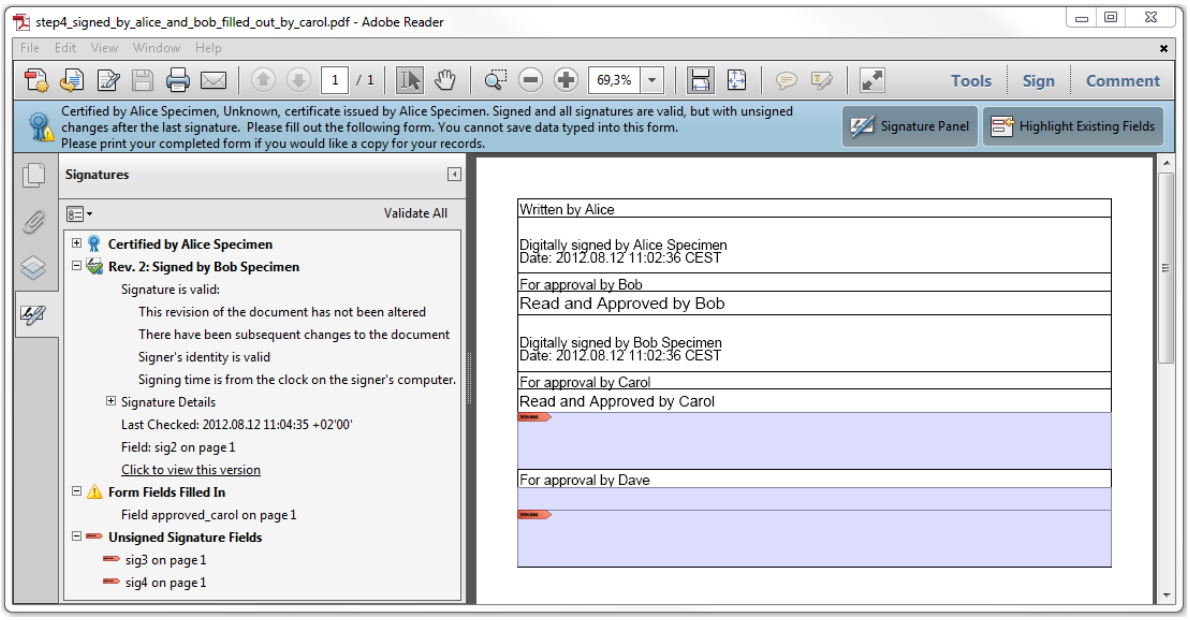

Figure 2.28: step 4, Carol says she has read and approved the document

Although filling out a field after signing for approval is allowed, we see a yellow question mark in figure 2.28. There's nothing to worry about: none of the signatures is broken. Filling out fields after a document was signed for approval is allowed, but we get a warning when the value of fields (in this case a field named 'approved_carol') has changed.

If we take a look at figure 2.29, we see that the yellow triangle disappears as soon as Carol signs the document.

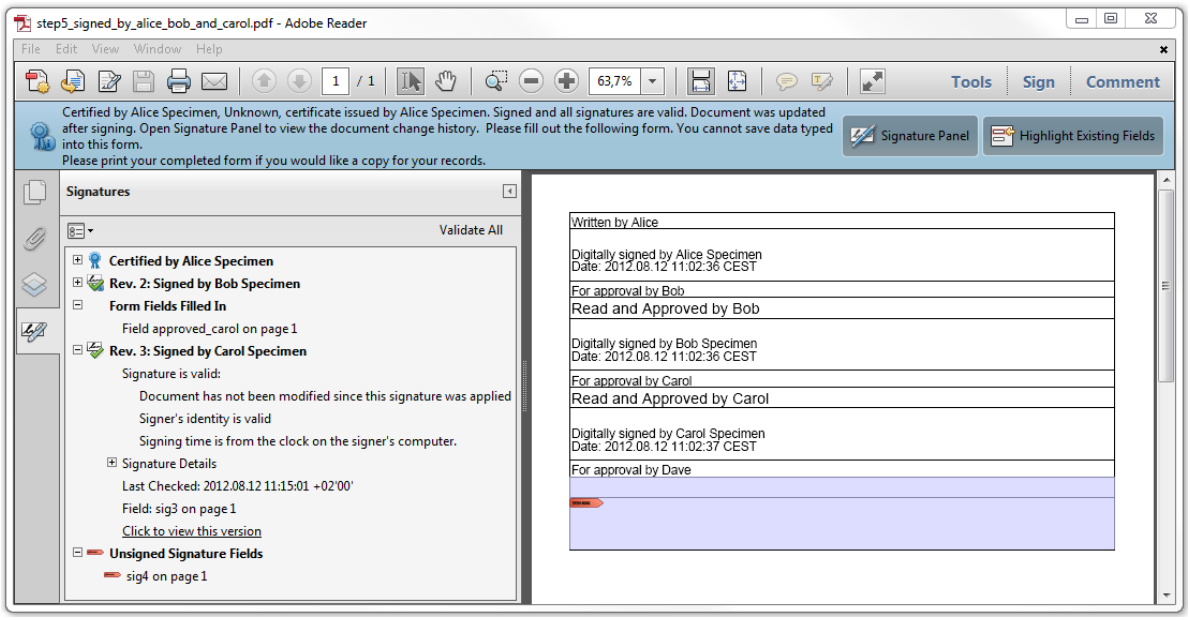

Figure 2.29: step 5, signed for approval by Carol

We still see an extra line in the signatures panel saying the field 'approved_carol' was filled out, but the yellow triangle previously shown in the upper blue bar (partly covered by the blue 'certification ribbon') has now been replaced with blue circle with an *i* inside. This informs us that a change was applied, but the change was allowed by the MDP settings of the signatures.

It's probably a good idea to skip the form filling step, and to fill out the form and sign it simultaneously. That's what we do in code sample 2.22.

Code sample 2.22: Filling and signing in one go

```
public void fillOutAndSign(String keystore,
    String src, String name, String fname, String value, String dest)
    throws GeneralSecurityException, IOException, DocumentException {
    KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
    ks.load(new FileInputStream(keystore), PASSWORD);
    String alias = (String)ks.aliases().nextElement();
    PrivateKey pk = (PrivateKey) ks.getKey(alias, PASSWORD);
    Certificate[] chain = ks.getCertificateChain(alias);
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0', null, true);
    AcroFields form = stamper.getAcroFields();
    form.setField(fname, value);
    form.setFieldProperty(fname, "setfflags", PdfFormField.FF_READ_ONLY, null);
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setVisibleSignature(name);
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, "SHA-256", "BC");
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        null, null, null, 0, CryptoStandard.CMS);
}
```

The final result is shown in figure 2.30:

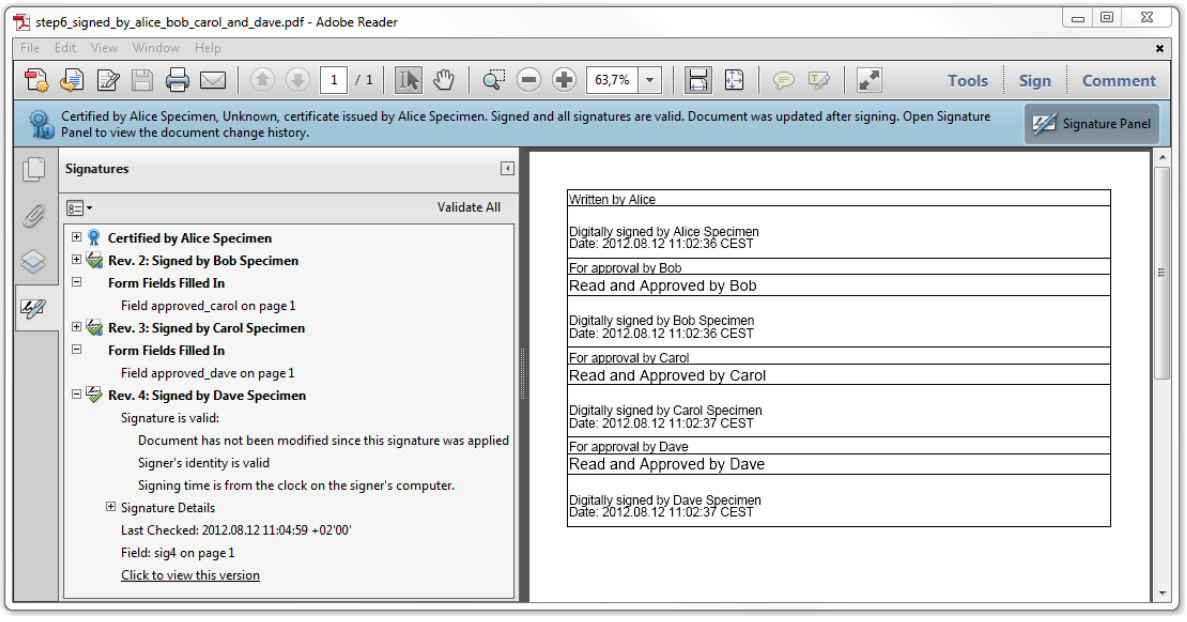

Figure 2.30: step 6, Dave said he approves the document and signed it

We have protected the fields against accidental change by other people down the workflow, but one could easily change the read-only status, and replace the value of the field in a later revision. Wouldn't it be nice if an approval signature could lock a field or even the whole document the way you can lock a document using a certification signature? Let's take a look at the different options.

2.5.5 Locking fields and documents after signing

Since PDF 1.5, it's possible to lock specific fields when signing a document. In PDF 2.0, there will be a new feature that allows you to define the document-level permissions that are already available for certification signatures, but now also for approval signatures.

Both types of permissions can be defined by adding a `/Lock` entry to the signature field with a `PdfSigLockDictionary` as value. Code sample 2.23 shows different flavors of this dictionary.

Code sample 2.23: creating PdfSigLockDictionary objects

```
table.addCell("For approval by Bob");
table.addCell(createTextFieldCell("approved_bob"));
PdfSigLockDictionary lock =
    new PdfSigLockDictionary(LockAction.INCLUDE, "sig1", "approved_bob", "sig2");
table.addCell(createSignatureFieldCell(writer, "sig2", lock));
table.addCell("For approval by Carol");
table.addCell(createTextFieldCell("approved_carol"));
lock =
    new PdfSigLockDictionary(LockAction.EXCLUDE, "approved_dave", "sig4");
table.addCell(createSignatureFieldCell(writer, "sig3", lock));
table.addCell("For approval by Dave");
table.addCell(createTextFieldCell("approved_dave"));
lock =
    new PdfSigLockDictionary(LockPermissions.NO_CHANGES_ALLOWED);
table.addCell(createSignatureFieldCell(writer, "sig4", lock));
```

The first `lock` instance defines that we want to lock `sig1` (Alice's signature), `approved_bob` (Bob's text field) and `sig2` (Bob's signature) as soon as Bob signs. The second `lock` defines that we want to lock all fields except `approved_dave` and `sig4` as soon as Carol signs. The final `lock` object defines that no changes whatsoever are allowed on the document as soon as Dave signs.

These are the values you can use as `LockAction` if you want to lock specific fields:

- *ALL*— lock all fields in the document.
- *INCLUDE*— lock all fields added to the signature lock dictionary.
- *EXCLUDE*— lock all fields except those added to the signature lock dictionary.

If you want to define a lock that covers the whole document, use the `PdfSigLockDictionary` constructor with one `LockPermissions` argument. iText will automatically select *`LockAction.ALL`*, and you'll need to choose one of the following `LockPermissions`:

- *NO_CHANGES_ALLOWED*— after approval, nobody can change the document without breaking the signature
- *FORM_FILLING*— after approval, form filling is allowed.
- *FORM_FILLING_AND_ANNOTATION*— after approval, form filling and adding annotations is allowed.

If a document already has specific permissions in place, you need to apply a more restrictive action. You can't use this functionality to remove restrictions. Let's take a look at code sample 2.24 and examine the `createSignatureFieldCell()` snippet that adds the lock to the signature field.

Code sample 2.24: locking fields

```
PdfFormField field = PdfFormField.createSignature(writer);
field.setFieldName(name);
if (lock != null)
    field.put(PdfName.LOCK, writer.addToBody(lock).getIndirectReference());
field.setFlags(PdfAnnotation.FLAGS_PRINT);
```

We don't add the `PdfSigLockDictionary` entry straight to the field object. Instead we add the dictionary to the `PdfWriter` instance first using the `addToBody()` method. This writes the object to the `OutputStream` creating a reference number. We use that number to create an indirect reference object with the `getIndirectReference()` method, and add it as value for the `/Lock` entry. Why? Because ISO-32000-1 demands that the entry *'shall be an indirect reference'*.

Let's take a look at figure 2.31 to see the form with all fields filled in and all signatures applied. The final line in the signature panel informs us that the document was locked by `sig4`.

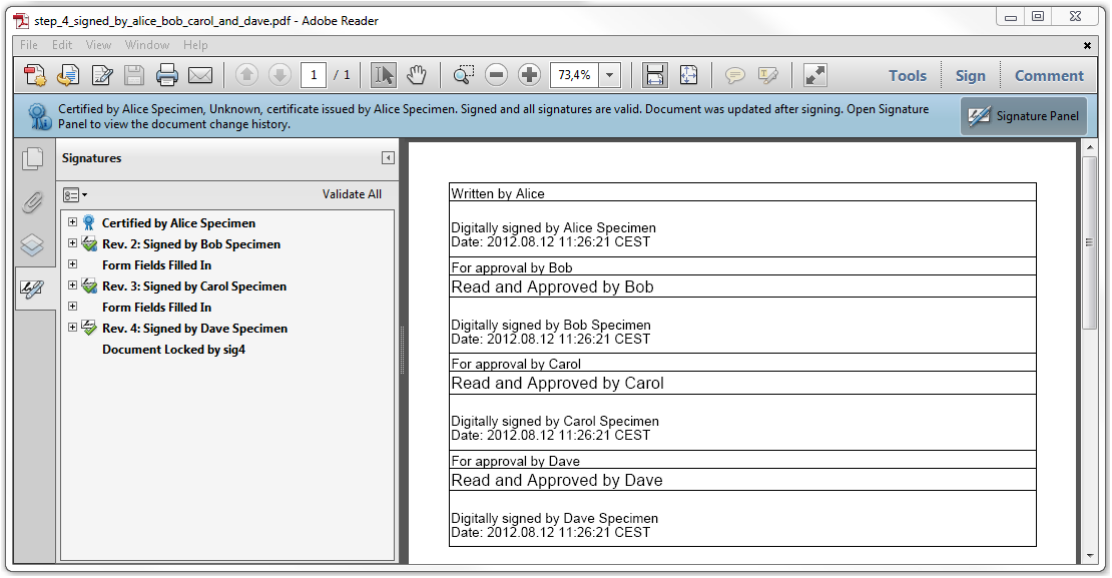

Figure 2.31: locked fields after final approval

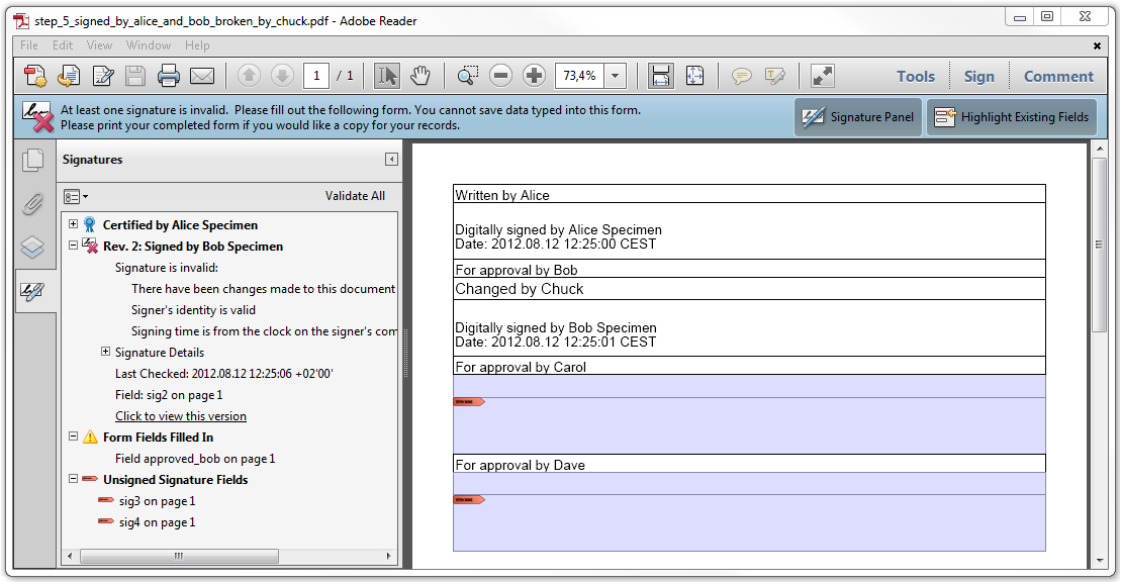

Figure 2.32: Bob's signature is invalidated by Chuck

Figure 2.32 shows what happens if Chuck changes Bob's text field after Bob signed the document. Bob's signature is invalidated because the `approved_bob` field was locked by `sig2`. Figure 2.33 shows what happens if Chuck changes `approved_carol` after Dave has signed.

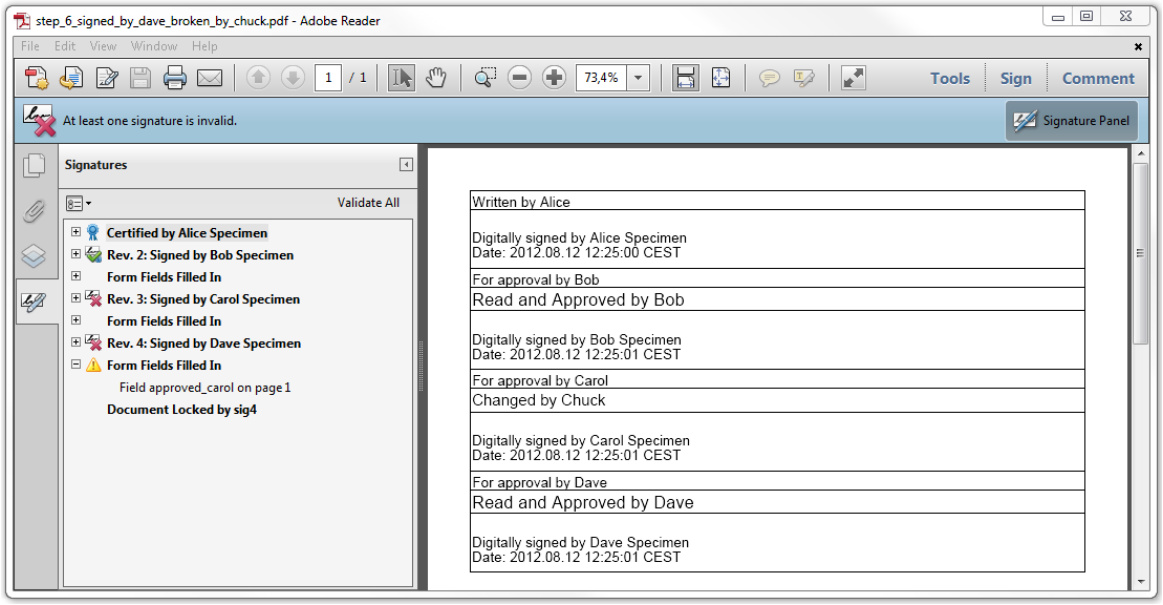

Figure 2.33: Carol's and Dave's signature are broken by Chuck

Carol's signature is invalid because `sig3` restricted the permissions on Carol's text field. Dave's signature is also invalid because `sig4` doesn't allow any further changes on the complete document.

Let's conclude this section with one final note about the `LockPermissions` functionality: this extra permission that can be set on the document level will only be introduced in ISO-32000-2 (PDF 2.0), but you can already use it in Adobe Acrobat and Adobe Reader X.

2.6 Summary

In this chapter, we started with a simple Hello World example showing how easy it is to sign a document using iText. In the examples that followed, we focused mainly on the cosmetic aspects of a signature.

We learned how to create a signature field, and we found out how to create a variety of appearances. We noticed that there are different types of signatures: the 'blue ribbon' and the 'green check mark' signatures. The former represent certification or author signatures, the latter are approval signatures. We ended this chapter by applying one certification and multiple approval signatures in a workflow.

All these examples were meant to get acquainted with using iText for signing PDF documents. The signatures we've created so far aren't created using the best practices. In the next chapter, we'll explain what's missing.

3 Certificate Authorities, certificate revocation and time stamping

You're reading this paper with a specific goal. This goal was summarized in the introduction using three bullets. You want to use digital signatures to ensure:

- *The integrity of the document*— assurance that the document hasn't been changed somewhere in the workflow.
- *The authenticity of the document*— assurance that the author of the document is who you think it is (and not somebody else)
- *Non-repudiation*— assurance that the author can't deny his or her authorship.

Did we reach those three goals? Not yet! By using the concepts of hashing and encryption, we can assure that a document hasn't been modified. Using the public/private key mechanism, we can authenticate the author and he can't deny the authorship, but so far, we've only been using self-signed certificates (see section 1.3.3). We've signed documents in the name of fictive personae named Alice, Bob, Carol, and Dave.

In the real world, we want documents to be signed by real human beings or real organizations, not by people we've made up. That's what this chapter is about. We want to make sure that we can trust the document, its signature and its signer.

3.1 Certificate authorities

Imagine two real-world people who have never met: Alice and Bob. Alice needs to send a signed PDF document to Bob. For Bob to be able to validate the signature in the document, he needs Alice's public certificate. That certificate is embedded in the signed PDF document, so he could add it to his list of Trusted Identities (2.2.2), but how does Bob know he can trust that certificate? Anyone can make a self-signed certificate using Alice's name.

This problem can be solved by involving a third person who knows both Alice and Bob. This person needs to be a 'trusted entity'; let's call him Trent. Trent knows Alice, and Trent is known by Bob. If somehow Trent marks Alice's public key to confirm its authenticity, Bob knows that Alice is who she claims to be. In the context of signatures, Trent does this by signing Alice's certificate. Bob can decrypt Alice's signed certificate using Trent's certificate, and as Bob trusts Trent's public certificate, he can now also trust Alice's certificate. See figure 3.1:

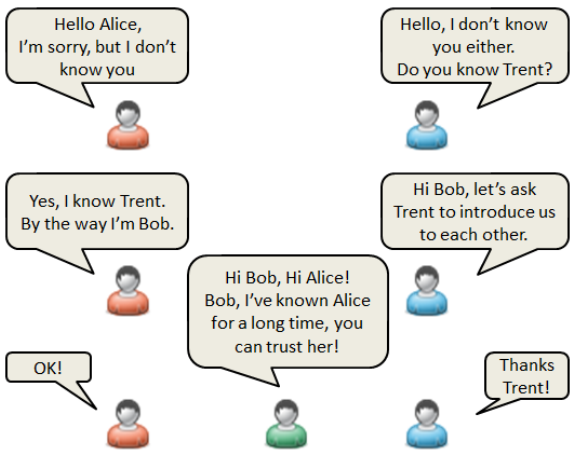

Figure 3.1: Bob trusts Alice, because he knows Trent and Trent trusts Alice

The more people we want to exchange messages with, the more trusted people we'll need to involve. This is called a "web of trust."

In practice, it's not very convenient to manage all these personal relationships. Usually Trent isn't an actual person, but a specialized, well-known organization: a certificate authority (CA). There are several commercial certificate authorities whose business it is to sell certificates. In one of the next examples, we'll learn how to sign a document using a key issued by GlobalSign[15], but first let's work with a private key from CAcert[16].

> **NOTE:** CAcert is a community effort that allows you to get a certificate for free. In order to get a certificate in your name, you need to accumulate sufficient points. You can earn points by meeting people in real life, showing them your driver's license or any other ID that proofs that you are who you claim you are.

I've registered at cacert.org, and I've created a public and private key pair that is stored in a file named `bruno.p12`. Remember that the 12 refers to PKCS#12, the Personal Information Exchange Standard that defines the format of the key store.

3.1.1 Signing a document with a p12 file from a Certificate Authority

Let's adapt the code from section 2.2.1 to sign a document with the CAcert key and make some changes resulting in code sample 3.1.

Code sample 3.1: Signing using a CAcert certificate

```
public static final String SRC = "src/main/resources/hello.pdf";
public static final String DEST = "results/chapter3/hello_cacert.pdf";

public static void main(String[] args)
    throws IOException, GeneralSecurityException, DocumentException {
    Properties properties = new Properties();
    properties.load(new FileInputStream("c:/home/blowagie/key.properties"));
    String path = properties.getProperty("PRIVATE");
    char[] pass = properties.getProperty("PASSWORD").toCharArray();
    BouncyCastleProvider provider = new BouncyCastleProvider();
    Security.addProvider(provider);
    KeyStore ks = KeyStore.getInstance("pkcs12", provider.getName());
    ks.load(new FileInputStream(path), pass);
    String alias = (String)ks.aliases().nextElement();
    PrivateKey pk = (PrivateKey) ks.getKey(alias, pass);
    Certificate[] chain = ks.getCertificateChain(alias);
    SignWithCAcert app = new SignWithCAcert();
    app.sign(SRC, DEST, chain, pk, DigestAlgorithms.SHA256, provider.getName(),
        CryptoStandard.CMS, "Test", "Ghent", null, null, null, 0);
}
```

I don't want to publish my password in a white paper that can be read by anyone. So I've stored it in a properties file that can be found in my home directory. If you want to run this example, you'll have to change the path `c:/home/blowagie/key.properties` to a path that points to a file containing at least two properties: a path to a `p12` file and a password. Observe that the sample code assumes that the key store password and the password for the private key are identical.

[15] https://www.globalsign.com/
[16] http://www.cacert.org/

We get an instance of the key store using two parameters:

- "`pkcs12`", because our keys and certificates are stored using PKCS#12 `.p12` file, and
- "`BC`" or `provider.getName()`; we're using the `BouncyCastleProvider`.

If you'd inspect the `Certificate` array named `chain` in code sample 3.1, you'd discover that it contains two certificates instead of just one as was the case in code sample 2.2. You can check this by taking a look at Certificate Viewer for the signed PDF document as is done in figure 3.2.

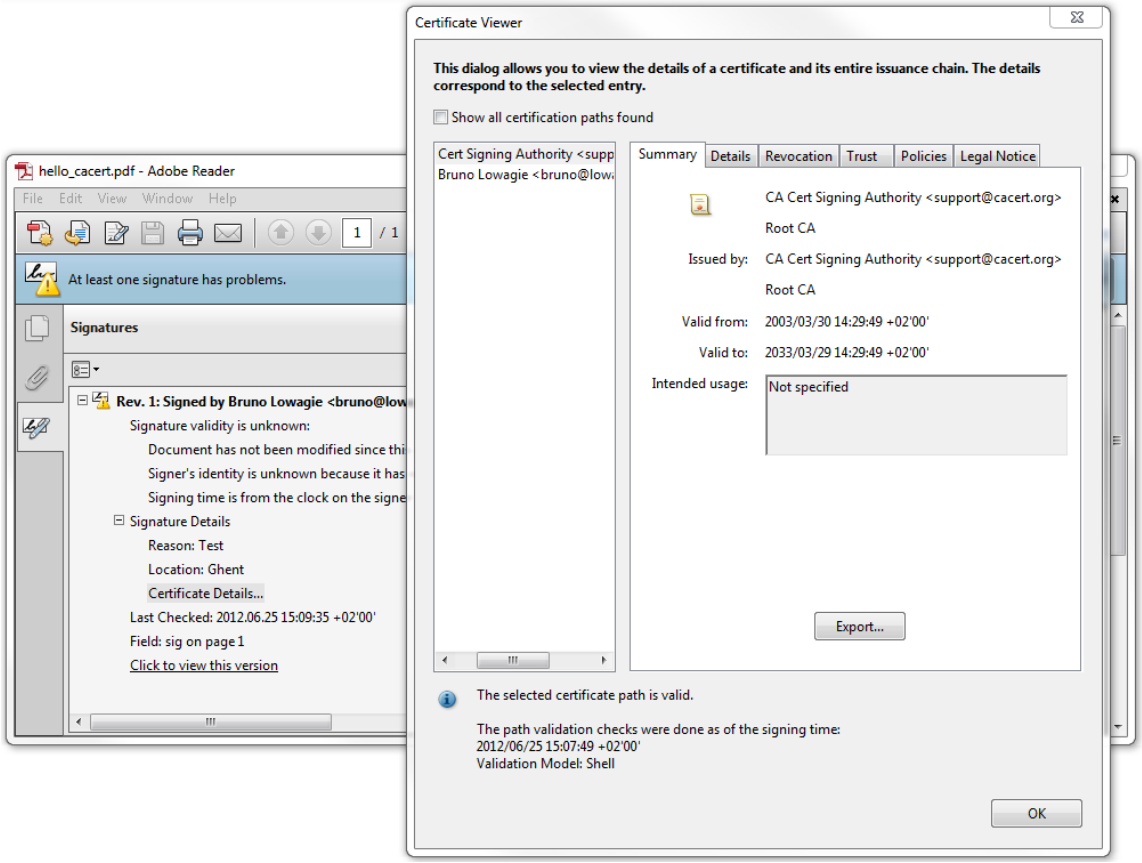

Figure 3.2: Certificate Viewer showing more than one Certificate

There's an important difference with what we saw in figure 2.5. The certificate is no longer self-signed! There are two certificates shown in the Certificate viewer. There's my own certificate (Bruno Lowagie) which is the *signing certificate* (always the first element in the chain), and there's the certificate of the CA Cert Signing Authority, the *root certificate* (the final element in the chain). The CA Cert Signing Authority is the Issuer of my certificate.

3.1.2 Trusting the root certificate of the Certificate Authority

CA Cert's root certificate corresponds with the private key that was used to sign all the public certificates that are issued to the members of the CA Cert Community. You can trust CA Cert's root certificate in the PDF the way we did in section 2.2.2, or you can download that root certificate from the cacert.org site and add it as a trusted identity as we did in section 2.2.3.

From now on, all the people who have a certificate issued by this CA will be trusted. You no longer have to trust each individual separately. This is shown in figures 3.3 and 3.4.

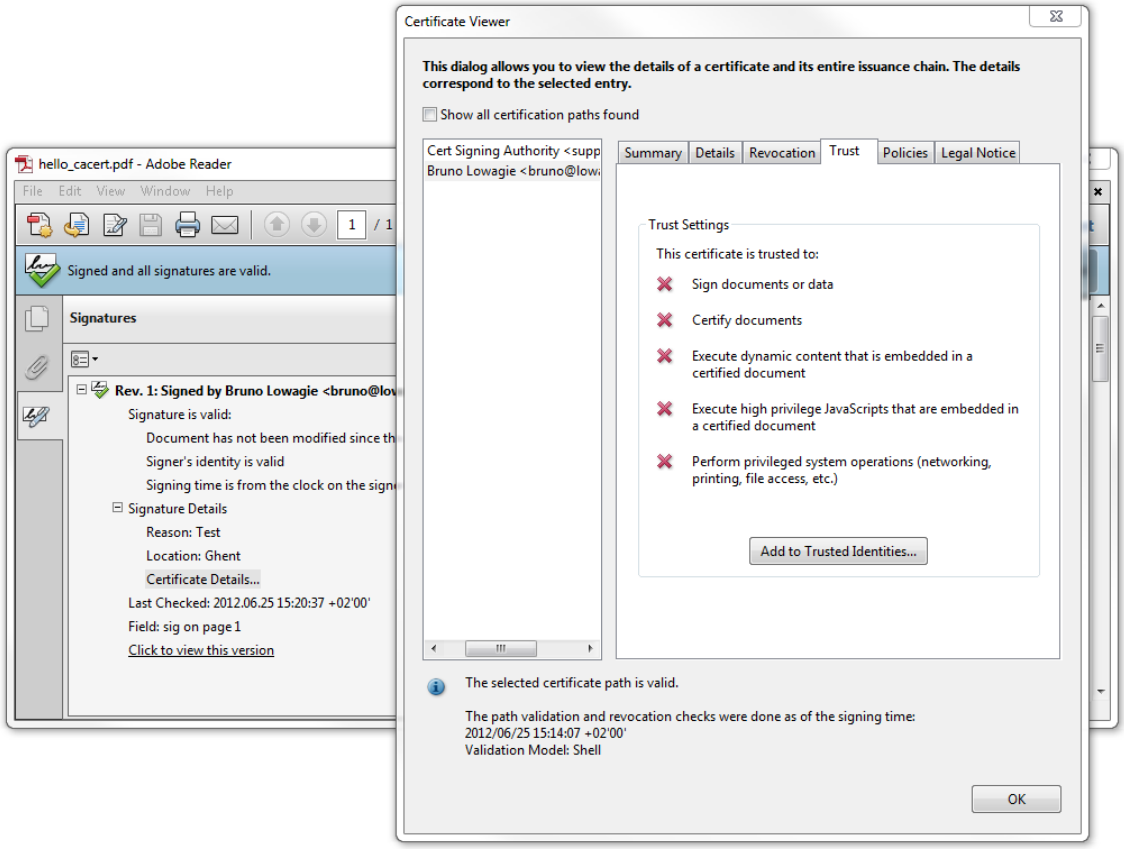

Figure 3.3: Individual signature not trusted, but Signer's identity is valid

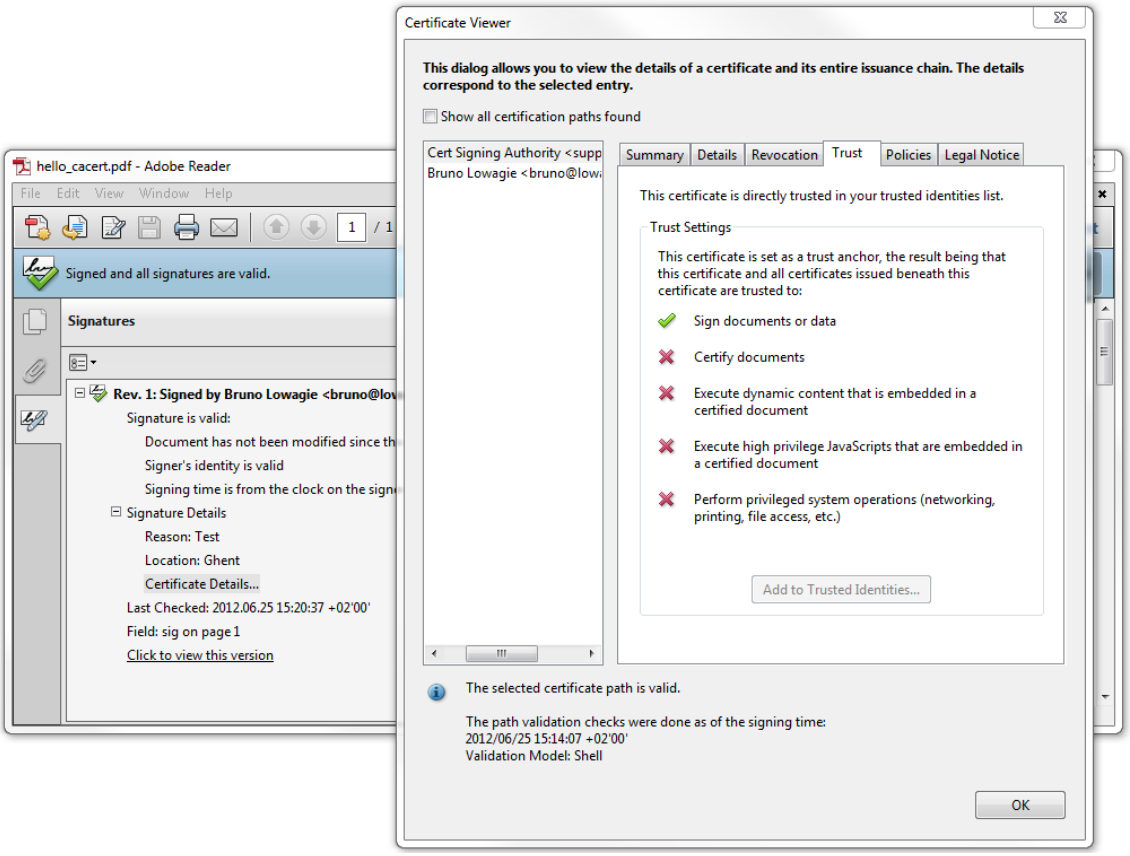

Figure 3.4: Signer's identity is valid, because root certificate is trusted

This answers the question: *"How can we be sure that the public key we're using is really from the person we think it's from?"* We make sure by relying on a trusted third party: a Certificate Authority.

However, there's another question we should ask: *"What happens if a private key is compromised?"* Suppose that I discover that somebody stole my certificate, and is planning to sign documents pretending to be me. If that happens, I should immediately notify the Certificate Authority, and ask the CA to add my certificate to the Certificate Revocation List (CRL).

Before we look at the concept of CRLs, let's update the `sign()` method we used in chapter 2.

3.1.3 Best practices in signing

When we signed documents in chapter 2, we always passed three `null` values and one zero to the `signDetached()` method. Let's take a closer look at the `sign()` method we used in code sample 3.2. It's an updated version of the method we used before.

Code sample 3.2: an updated sign() method

```
public void sign(String src, String dest, Certificate[] chain, PrivateKey pk,
    String digestAlgorithm, String provider, CryptoStandard subfilter,
    String reason, String location, Collection<CrlClient> crlList,
    OcspClient ocspClient, TSAClient tsaClient, int estimatedSize)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    // Creating the signature
    ExternalSignature pks = new PrivateKeySignature(pk, digestAlgorithm, provider);
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, pks, chain,
        crlList, ocspClient, tsaClient, estimatedSize, subfilter);
}
```

In code sample 3.2, the hardcoded `null` values and the `0`, are replaced by different objects: a `Collection` of `CrlClient`s, an `OcspClient` and a `TSAClient`. Finally, there's also an `estimatedSize` value.

Let's take a look at these objects one by one, starting with the ones regarding certificate revocation.

3.2 Adding Certificate Revocation information

There are different ways to prove that a certificate wasn't revoked at the moment it was signed. You can embed a Certificate Revocation List (CRL) into the PDF; you can specify an URL that can be used to check the revocation status of the certificate; or you can do both.

3.2.1 Finding the URL of a Certificate Revocation List

There can be a Certificate Revocation List for each certificate in the chain. To find the lists we need, we have to loop over the chain as is done in code sample 3.3. We can use the `getCRLURL()` method from iText's `CertificateUtil` convenience class to extract the URL of the CRL.

Code sample 3.3: getting the CRL URLs from a certificate chain

```
BouncyCastleProvider provider = new BouncyCastleProvider();
Security.addProvider(provider);
KeyStore ks = KeyStore.getInstance("pkcs12", provider.getName());
ks.load(new FileInputStream(path), pass.toCharArray());
String alias = (String)ks.aliases().nextElement();
Certificate[] chain = ks.getCertificateChain(alias);
for (int i = 0; i < chain.length; i++) {
    X509Certificate cert = (X509Certificate)chain[i];
    System.out.println(String.format("[%s] %s", i, cert.getSubjectDN()));
    System.out.println(CertificateUtil.getCRLURL(cert));
}
```

If I use this code on my CAcert certificate, I get the following result:

```
[0] CN=Bruno Lowagie,E=bruno@________.com
null
[1] O=Root CA,OU=http://www.cacert.org,CN=CA Cert Signing Authority,E=support@cacert.org
https://www.cacert.org/revoke.crl
```

In this case, my signing certificate doesn't contain any CRL, but the root certificate does. iText offers different ways to use this information.

3.2.2 Getting the CRL online

If you want to embed CRLs in a PDF, you need a `Collection` of `CrlClient` elements. Creating a `CrlClientOnline` instance is the easiest way to achieve this. See code sample 3.4.

Code sample 3.4: Using the default CrlClient implementation

```
LoggerFactory.getInstance().setLogger(new SysoLogger());
List<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(new CrlClientOnline());
SignWithCRLDefaultImp app = new SignWithCRLDefaultImp();
app.sign(SRC, DEST, chain, pk, DigestAlgorithms.SHA256, provider.getName(),
    CryptoStandard.CMS, "Test", "Ghent", crlList, null, null, 0);
```

The `MakeSignature` class will pass a single certificate to `CrlClientOnline`, starting with the signing certificate. As soon as the `CrlClient` finds an URL of a CRL, iText will stop looking for other CRLs. While this solution is easy and while it will work in many cases, it won't work for my CAcert certificate. Moreover, we'll soon discover it's not the best solution in general.

Trusting the root certificate in your Java Runtime Environment (JRE)

Let's start by explaining why it doesn't work for a CAcert certificate. The URL of the Certificate Revocation List as defined in the certificate starts with `https`. Suppose that you've installed a JRE or JDK from scratch, and you want to run the example shown in code sample 3.4, you'll hit the following error as soon as iText tries to fetch the CRL:

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException:
PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException
```

When seeing this message, you should immediately understand that this is not an iText problem. Clearly it isn't a programming error either. It's a matter of configuration.

iText uses the `java.net.URL` class to create a connection to the https-server at cacert.org. Https means that SSL is used, and that the Java classes try to do an SSL Handshake. For this handshake to succeed, the Java Virtual Machine (JVM) needs to trust the certificate on the cacert.org server, and that's what's going wrong here: there's a validator exception. The certificate isn't recognized in your Java Runtime Environment and as a result your JVM doesn't trust it.

You can solve this by downloading CAcert's root certificate and by adding it to the `cacerts` key store in the `lib/security` directory of your Java home directory. This is done in code sample 3.5:

Code sample 3.5: importing a root certificate into the Java cacerts file

```
$ keytool -import -keystore cacerts -file CAcertSigningAuthority.crt -alias cacert
Enter keystore password:
Owner: EMAILADDRESS=support@cacert.org, CN=CA Cert Signing Authority,
  OU=http://www.cacert.org, O=Root CA
Issuer: EMAILADDRESS=support@cacert.org, CN=CA Cert Signing Authority,
  OU=http://www.cacert.org, O=Root CA
Serial number: 0
Valid from: Sun Mar 30 14:29:49 CEST 2003 until: Tue Mar 29 14:29:49 CEST 2033
Certificate fingerprints:
         MD5:  A6:1B:37:5E:39:0D:9C:36:54:EE:BD:20:31:46:1F:6B
         SHA1: 13:5C:EC:36:F4:9C:B8:E9:3B:1A:B2:70:CD:80:88:46:76:CE:8F:33
         Signature algorithm name: MD5withRSA
         Version: 3
Extensions:
#1: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]
#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 16 B5 32 1B D4 C7 F3 E0   E6 8E F3 BD D2 B0 3A EE  ..2...........:.
0010: B2 39 18 D1                                        .9..
]
]
#3: ObjectId: 2.16.840.1.113730.1.8 Criticality=false
#4: ObjectId: 2.16.840.1.113730.1.4 Criticality=false
#5: ObjectId: 2.5.29.31 Criticality=false
CRLDistributionPoints [
  [DistributionPoint:
     [URIName: https://www.cacert.org/revoke.crl]
]]
#6: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 16 B5 32 1B D4 C7 F3 E0   E6 8E F3 BD D2 B0 3A EE  ..2...........:.
0010: B2 39 18 D1                                        .9..
]
[EMAILADDRESS=support@cacert.org, CN=CA Cert Signing Authority, OU=http://www.ca
cert.org, O=Root CA]
SerialNumber: [    00]
]
#7: ObjectId: 2.16.840.1.113730.1.13 Criticality=false
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Remember that the initial password of the `cacerts` key store when you install a JRE or a JDK is always `changeit`. If that doesn't work, somebody has changed the password on purpose.

Make sure you're updating the correct `cacerts` file. There can be more than one on your OS. When in doubt, use `System.getProperty("java.home")` to get the correct path, and check if you have the right permissions to change the file. I always forget to `chmod`, resulting in a `keytool` error: `java.io.FileNotFoundException: cacerts (Access is denied).`

What if somebody changes the CRL URL?

We overlooked one line in code sample 3.4. In the first line, we get an instance of iText's `LoggerFactory`, and we define a `SysoLogger` instance as logger. A `SysoLogger` writes whatever logging information iText produces to the `System.out`.

After we fixed the `SSLHandshakeException` by adding CAcert's root certificate to the cacerts key store of our JRE, we get the following logging information:

```
c.i.t.p.s.MakeSignature INFO  Processing com.itextpdf.text.pdf.security.CrlClientOnline
c.i.t.p.s.CrlClientOnline INFO  Looking for CRL for certificate CN=Bruno Lowagie,E=bruno@_.com
c.i.t.p.s.CrlClientOnline INFO  Skipped CRL url: null
c.i.t.p.s.MakeSignature INFO  Processing com.itextpdf.text.pdf.security.CrlClientOnline
c.i.t.p.s.CrlClientOnline INFO  Looking for CRL for certificate O=Root CA,
  OU=http://www.cacert.org,CN=CA Cert Signing Authority,E=support@cacert.org
c.i.t.p.s.CrlClientOnline INFO  Found CRL url: https://www.cacert.org/revoke.crl
c.i.t.p.s.CrlClientOnline INFO  Checking CRL: https://www.cacert.org/revoke.crl
c.i.t.p.s.CrlClientOnline INFO  Skipped CRL: Invalid HTTP response:
  302 for https://www.cacert.org/revoke.crl
```

The `MakeSignature` class processes `CrlClientOnline` a first time for the first certificate. As it doesn't find an URL for the CRL, it moves to the next certificate and processes `CrlClientOnline` a second time.

Now it finds this URL: `https://www.cacert.org/revoke.crl` which should be a valid URL, but when trying to fetch the CRL, the cacert.org server is returning status code 302, which means you're being redirected. iText doesn't trust this, and won't embed the CRL. How can we fix this?

Creating a CrlClient using a specific URL

If you go to the CAcert site, you'll see that there are two URLs to fetch the Certificate Revocation List, either http://crl.cacert.org/revoke.crl or https://crl.cacert.org/revoke.crl. You can create an instance of `CrlClientOnline` using either one. The former solves the SSL handshake problem; the latter solves the redirect problem if you've added CAcert's root certificate to your JRE's `cacerts` file.

Code sample 3.6: Creating a CRL list for a CAcert certificate

```
CrlClient crlClient = new CrlClientOnline("https://crl.cacert.org/revoke.crl");
List<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(crlClient);
```

Code sample 3.6 is added to show how to work around some problems inherent to certificates obtained from CAcert.

Creating a CrlClient using the Certificate chain

A better way to embed CRLs that are fetched online is shown in code sample 3.7.

Code sample 3.7: Creating a CRL list for all the certificates in a chain

```
List<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(new CrlClientOnline(chain));
```

This example is much better in general because you don't need to hardcode the URLs.

iText will loop over every certificate in the chain as we did in sample 3.3, and will fetch every CRL it encounters, not just the first one. This is important because if you buy a certificate from an established CA, there will be more certificates in the chain, and you'll want to embed the CRLs for each of those certificates.

Now that we finally succeeded in embedding a CRL into a PDF, let's take a look at the certificate details in the Certificate Viewer. See figure 3.5.

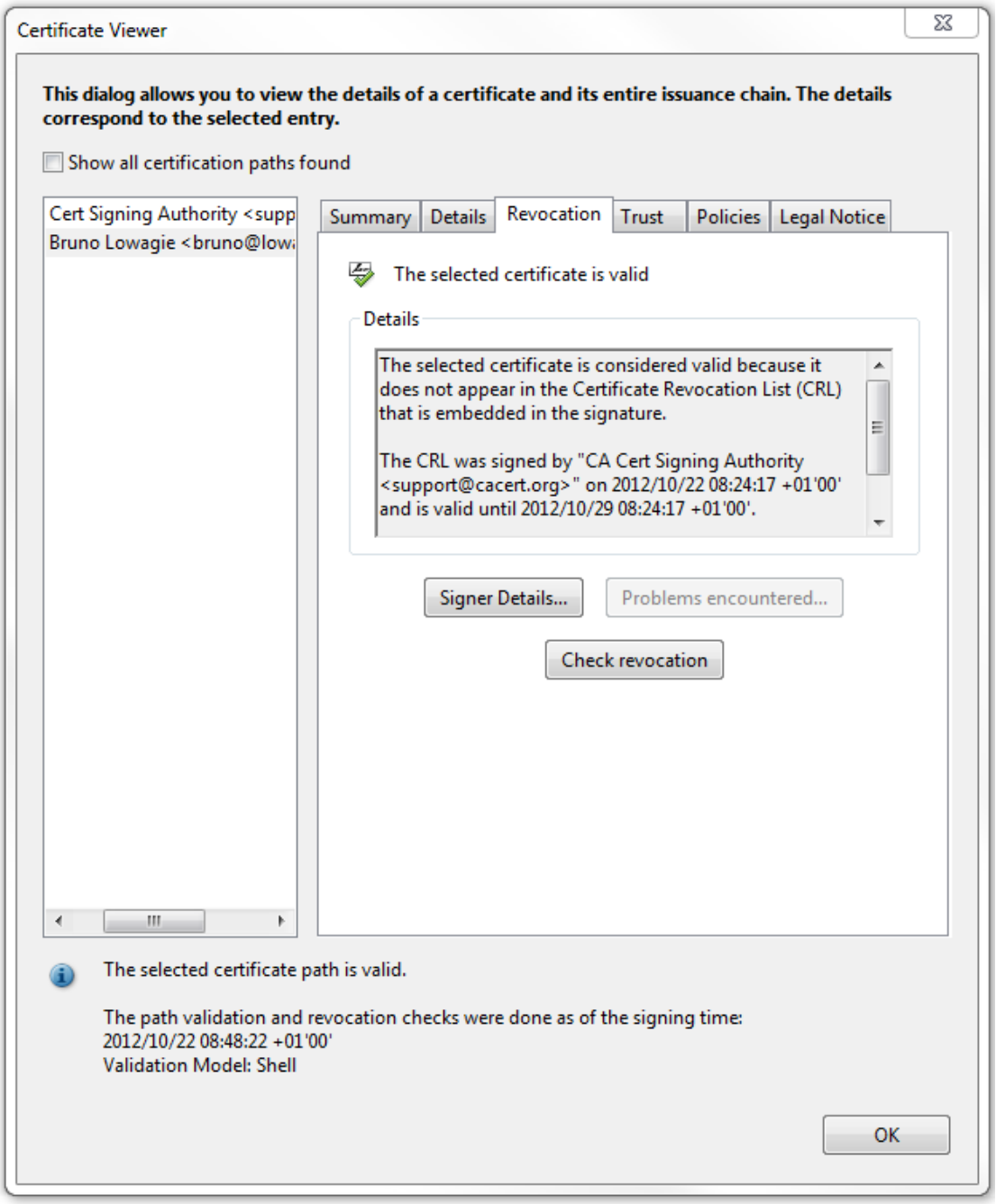

Figure 3.5: Valid certificate because the CRL is embedded

It reads *'The selected certificate is considered valid because it does not appear in the Certificate Revocation List (CRL) that is embedded in the signature.'* If you receive a file like this, you're now certain that the signer is who he claims he is, and you're also certain that the signer's didn't revoke his certificate prior to signing the document: it wasn't on the CA's certificate revocation list at the moment the PDF was signed.

As for the root certificate, that's *'either a trust anchor or a certificate above the trust anchor in the certificate chain.'* See figure 3.6. Remember that the root certificate is the certificate from CAcert that we added to the Trusted Identities manually.

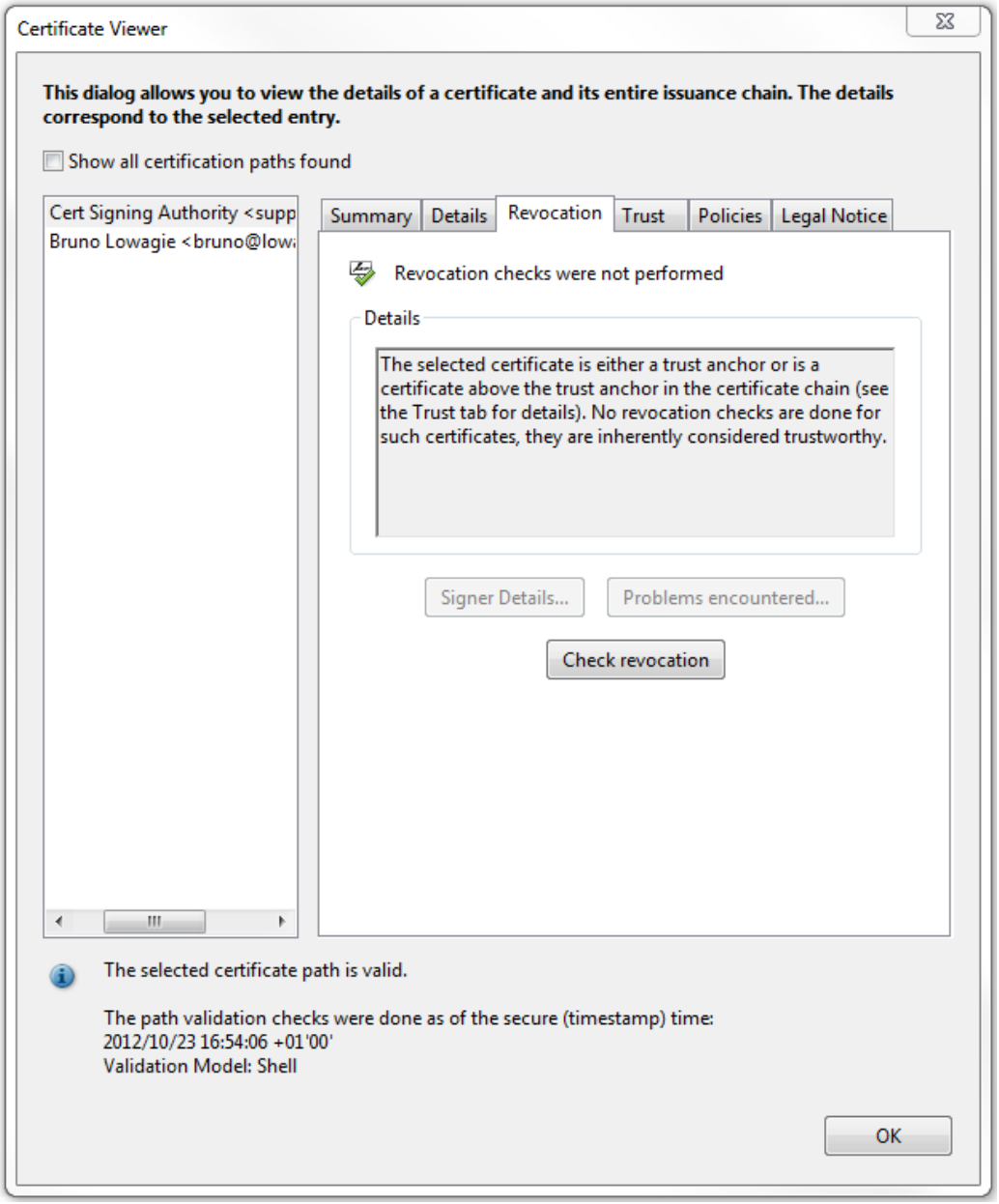

Figure 3.6: No CRL checking for the trusted root certificate

If you fetch the CRL from an URL, you always have the most recent certification revocation list. That's a good thing, but it isn't very practical if you're about to sign a thousand documents. Opening a connection for every separate document, downloading the same bytes over and over again, isn't such a good idea. That's why we also have a `CrlClient` implementation that allows you to use an offline copy of the CRL.

3.2.3 Creating a CrlClient using an offline copy of the CRL

Now that we know how to find the URL of a certificate revocation list, we can also use it in a browser to download the CRL to our hard disk. Or even better, we can write a `cron` job or a service that automatically downloads the CRL on a regular basis.

Figure 3.7 shows what you see when you double-click a CRL file.

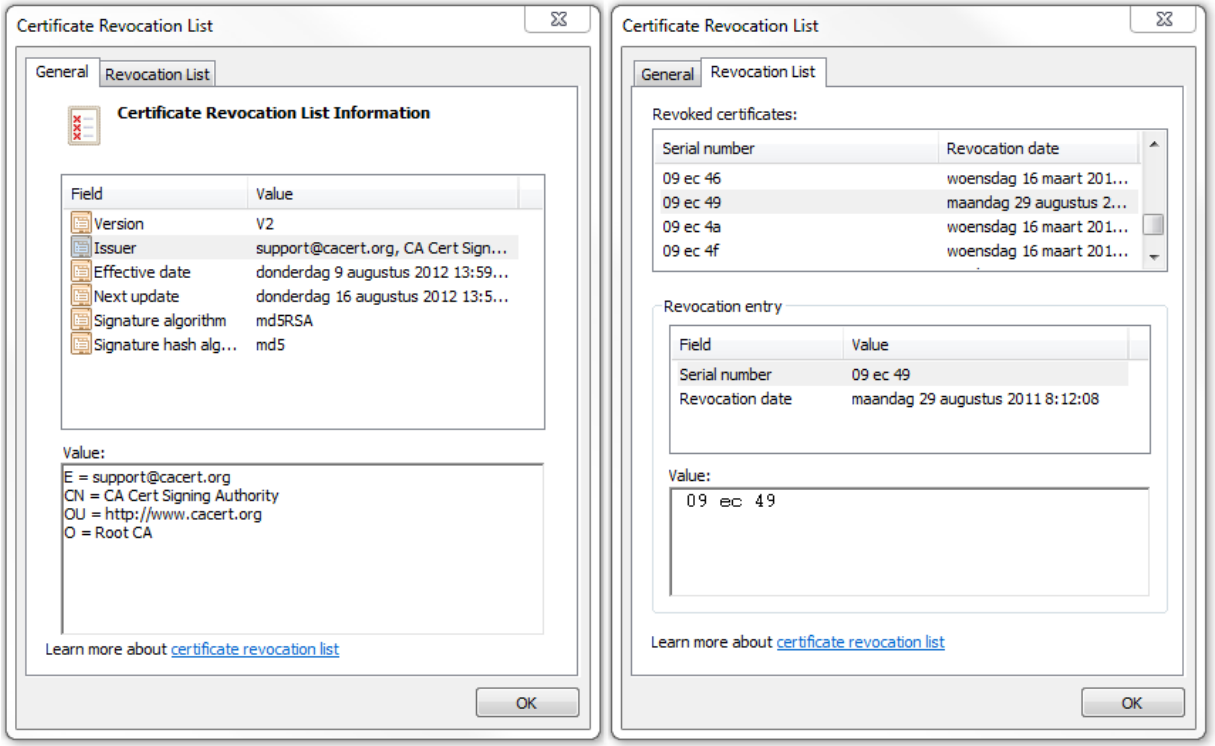

Figure 3.7: A CAcert Certificate Revocation List

Instead of downloading the list from our code, we can now create a `CrlClientOffline` object. This object has two constructors. One that accepts a `byte[]` and one that accepts a `CRL` object.

Code sample 3.8: Creating an offline CrlClient

```
FileInputStream is = new FileInputStream(CRL);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
byte[] buf = new byte[1024];
while (is.read(buf) != -1) baos.write(buf);
CrlClient crlClient = new CrlClientOffline(baos.toByteArray());
List<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(crlClient);
```

In code sample 3.8, we read a `FileInputStream` into a `ByteArrayOutputStream` and we use those bytes to create a `CrlClientOffline` object.

> **IMPORTANT:** iText doesn't check the validity of the CRL; iText just adds the bytes as-is without checking if the CRL is expired, nor if the certificate that is used has been revoked.

Code sample 3.9 shows how to create a `CRL` object from the stored CRL bytes. With this object, you can check the expiration date of the list, and check if the signing certificate is revoked (if `true`, your PDF will have an invalid signature).

Code sample 3.9: Creating a CRL object from a file

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509CRL crl = (X509CRL)cf.generateCRL(new FileInputStream(CRL));
System.out.println("CRL valid until: " + crl.getNextUpdate());
System.out.println("Certificate revoked: " + crl.isRevoked(chain[0]));
```

You can use this `CRL` object instead of the `byte[]` to create a `CrlClientOffline` instance.

WARNING: A downloaded CRL is valid for only a limited period. It usually expires after seven days. If it didn't expire, one could first download a CRL, then steal a certificate, then sign a document using the stolen certificate. As a matter of fact, this is always possible if the signature is applied within the expiration period. That's why it's good practice to cache a CRL only for a limited period. A CA normally updates its CRLs every 30 minutes. If possible, you should create a caching system that downloads a new CRL every half hour.

Finally, you may have noticed one of the major downsides of embedding CRLs: the file size can get really big if the Certificate Authority doesn't really bother about the size of the CRL (which is the case with CAcert). Later on in this chapter, we'll see how GlobalSign solves this problem. We can solve the problem ourselves by using the Online Certificate Status Protocol (OCSP) instead of CRLs.

3.2.4 Using the Online Certificate Status Protocol (OCSP)

OCSP is an internet protocol for obtaining the revocation status of a certificate online. You can post a request to check the status of a certificate over http, and the CA's OCSP server will send you a response. You no longer need to parse and embed long CRLs. An OCSP response is small and constant in size, and it can be easily embedded in a digital signature.

Use code sample 3.10 if you want to know if your certificate supports OCSP:

Code sample 3.10: fetching the OCSP URL from a certificate

```
BouncyCastleProvider provider = new BouncyCastleProvider();
Security.addProvider(provider);
KeyStore ks = KeyStore.getInstance("pkcs12", provider.getName());
ks.load(new FileInputStream(path), pass.toCharArray());
String alias = (String)ks.aliases().nextElement();
Certificate[] chain = ks.getCertificateChain(alias);
for (int i = 0; i < chain.length; i++) {
    X509Certificate cert = (X509Certificate)chain[i];
    System.out.println(String.format("[%s] %s", i, cert.getSubjectDN()));
    System.out.println(CertificateUtil.getOCSPURL(cert));
}
```

When I use this code on my CAcert certificate, I get the following output:

```
[0] CN=Bruno Lowagie,E=bruno@______.com
http://ocsp.cacert.org
[1] O=Root CA,OU=http://www.cacert.org,CN=CA Cert Signing Authority,E=support@cacert.org
null
```

When signing a document, iText needs a way to send a post to the url http://ocsp.cacert.org, and to receive a response regarding the validity of my certificate. This is done by an implementation of the `OcspClient` interface, for instance iText's `OcspClientBouncyCastle` class that uses the Bouncy Castle library.

Code sample 3.11: Signing with OCSP

```
OcspClient ocspClient = new OcspClientBouncyCastle();
SignWithOCSP app = new SignWithOCSP();
app.sign(pk, chain, SRC, DEST, provider.getName(), "Test", "Ghent",
    DigestAlgorithms.SHA256, CryptoStandard.CMS, null, ocspClient, null, 0);
```

If you look at code sample 3.11, you'll see that we don't need to pass any URL or certificate; iText will pass the certificate containing the OCSP URL to the `OcspClient` implementation.

Let's take a look at the Certificate Viewer of the resulting PDF. Figure 3.8 demonstrates that *'the selected certificate is considered valid because it has not been revoked as verified using the Online Certificate Status Protocol (OCSP) response that was embedded in the signature.'*

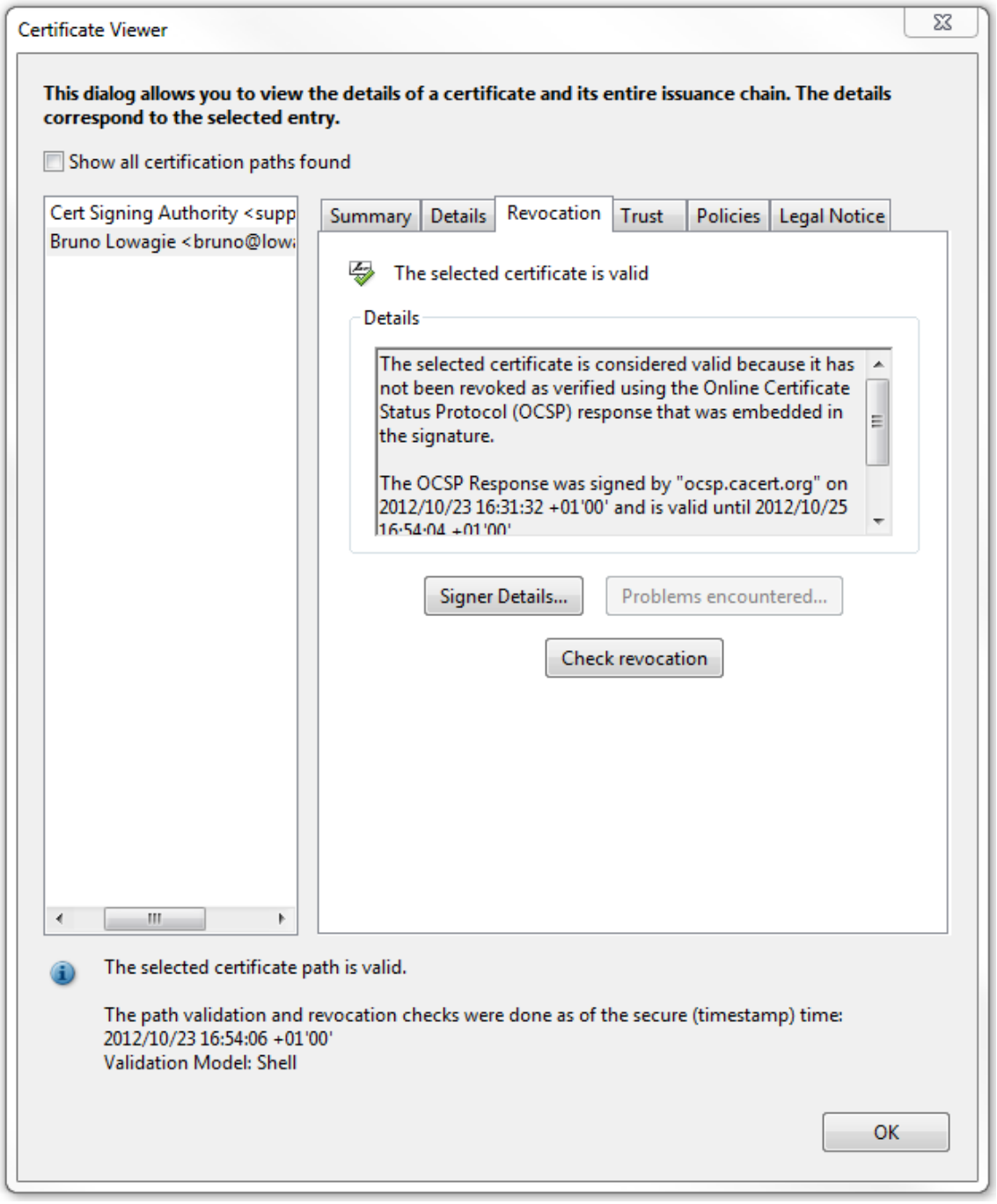

Figure 3.8: A digital signature with an embedded OCSP response

It's important to understand that the choice between using CRLs and using OCSP isn't an *'or this, or that'* question. For the sake of the consumer of the signed document, it's always best to use both because that offers more certainty than using only one type of certificate revocation checking.

> **NOTE:** Adobe Acrobat / Reader 8 had a preference for OCSP and would always do an OCSP lookup if there wasn't any embedded OCSP response. If it couldn't make a connection to the

internet, it looked at the embedded CRLs (if any). Adobe Acrobat / Reader 9 and X however, have a preference for CRLs. They look for the CRL first, and if it isn't found, they'll fall back on OCSP.

In many cases, especially when using software certificates like the one from CAcert, using OCSP is the better solution, but that's not a general rule. There's always a tradeoff.

3.2.5 Which is better: embedding CRLs or an OCSP response?

Three aspects are important when comparing embedding CRLs with embedding an OCSP response: the file size of the signed document, the performance when creating the signature, and the law.

Comparing the file size

If you compare the file size of a file that has CAcert's CRL embedded to the same file signed with the OCSP information from CAcert, you'll see that the difference is huge. The one with the CRL is almost 10 MByte, the one with the OCSP is only 28 Kbyte. Rest assure: the difference isn't always this terrifying. It all depends on the CA you've chosen. Some certificate authorities (such as CAcert) keep the revocation information of *all* their certificates in only one file; others don't maintain their revocation list: for instance, they don't remove certificates from the list that have expired anyway.

> **NOTE:** A CA that cares about the file size of the CRLs, will create a hierarchy of certificates, splitting regular certificates, USB tokens and Hardware Security Modules (HSM) to keep the CRLs small. For instance, you can't extract the private key from an HSM, and stealing an HSM is rather impractical, so a CRL used only for HSM certificates can be kept small. Embedding the CRL of a HSM could result in smaller files than when using OCSP.

Conclusion: you should choose your CA wisely if you need to embed CRLs into your documents.

Differences in performance

Even when the CRLs are bigger than the OCSP response, the larger file size may be an acceptable cost for improved performance. In the context of ISO-32000-1 and PAdES 2, revocation information is a signed attribute in a PDF document, which implies that the signing software (in this case iText) must capture the revocation information before signing. The OCSP connection to check the status can take time, whereas using CRLs that are cached on the file system can save a significant amount of time when you need to sign thousands of documents in batch.

Legal requirements

In some countries, not only non-revocation of the certificate, but also existence has to be proven when verifying qualified signatures. In Germany for instance, embedding a Certificate Revocation List won't be sufficient, because 'not appearing on the CRL' doesn't mean the certificate actually exists.

Now let's look at another problem that might arise. Suppose somebody has a CRL that dates from a year ago. He now wants to predate a document by signing it with signature that says the document was signed a year ago. He could do so by changing the clock on his computer.

The only way we can prevent this from happening, is to demand that every signed document has a timestamp obtained from a Time Stamping Authority (TSA).

3.3 Adding a timestamp

Before we jump into an example, let's try to understand why it's important to know when a document was signed. We already know that certificates can be revoked, but certificates also have an expiration date. What happens if a certificate expires?

3.3.1 Dealing with expiration and revocation dates

Suppose that Alice has acquired a private key with a certificate that is valid until 2014. As long as the expiration date hasn't passed, Alice can sign documents and these documents will be validated by Adobe Reader.

After the expiration date, Alice can no longer use her private key to sign documents (which is to be expected), but there's more! Adobe Reader will inspect the certificate and see that it has expired. If there's no embedded CRL, no embedded OCSP response, and no timestamp, Adobe Reader won't show a green check mark anymore, but it will give a warning saying: I can't trust this signature anymore because the expiration date is in the past. See figure 3.9.

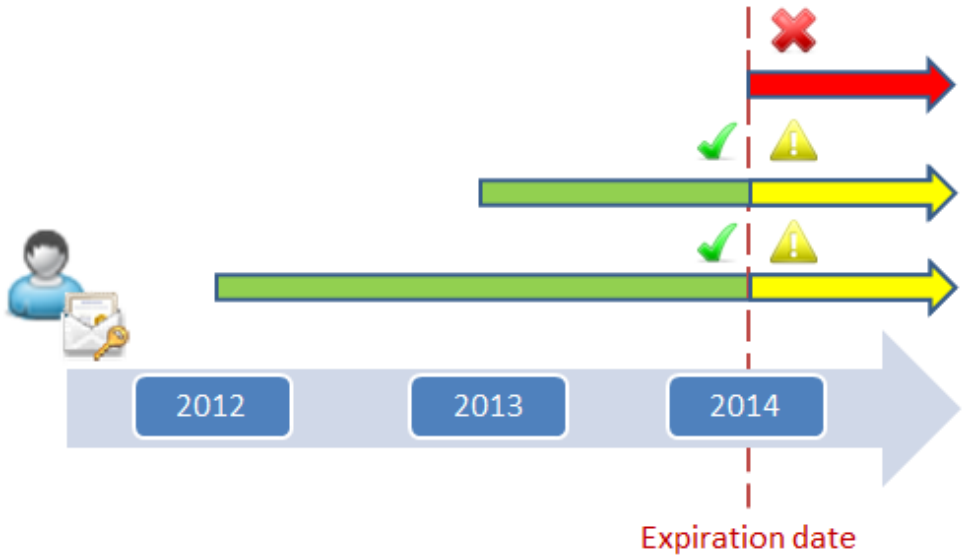

Figure 3.9: Alice signs a document without CRL and without timestamp

It gets even worse if the certificate is revoked. This doesn't always mean that Alice's key was stolen. It's also possible that Alice has left the company she used to work for. She had a key she could use to sign company documents until 2014, but if she leaves the company in 2013, the company will most probably want to revoke that key. Figure 3.10 shows the consequences if that ever happens:

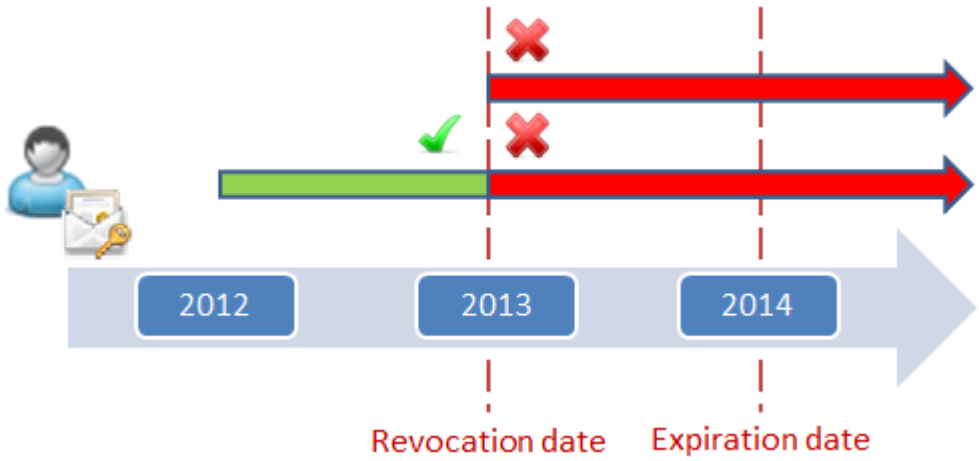

Figure 3.10: Alice leaves the company in 2013, her key is revoked

Apart from the fact that Alice can no longer sign documents from 2013 on, all signatures she created before that date can no longer be trusted if no CRL, OCSP response or timestamp was added. There's no way for the consumer of the document to verify if the document is valid. The certificate was revoked, and there's no certainty about the date the document was signed. After all, Alice could have changed the clock on her computer.

What we need, is a situation as shown in figure 3.11:

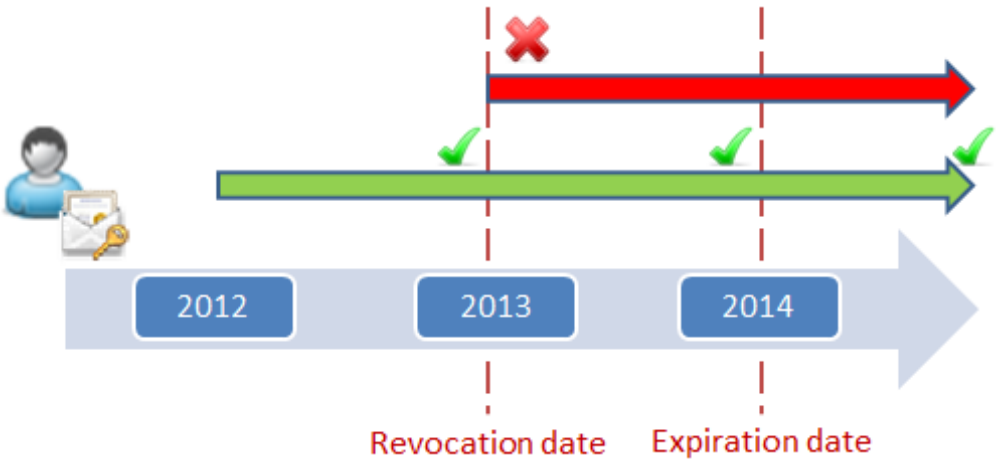

Figure 3.11: Alice leaves the company; all documents she signed in the past remain valid

Can we achieve this merely by adding a CRL or an OCSP response? No, because Alice could have cached the information about the certificate revocation before leaving the company. The only way we can assure that a document remains valid, is by also adding a timestamp.

A document that was signed by Alice in 2012 will contain revocation information dating from 2012 saying that her certificate wasn't revoked at that time; the timestamp will assure that the document was certainly signed in 2012. This signature will survive the revocation and expiration date.

But how do we add a timestamp to a digital signature in a PDF document?

3.3.2 Connecting to a timestamp server

To solve this problem, we need to involve another third party: a *Time Stamping Authority* (TSA). A TSA provides an online service, signing signature bytes and concatenating a timestamp to it. This is done on a timestamp server that is contacted during the signing process. The timestamp server will return a hash and authenticated attributes that are signed using the private key of the TSA.

> **NOTE:** You need to be online to create a signature with a timestamp. Connecting to a timestamp server sending and receiving the hash takes time. If you need to sign thousands of documents in batch, you could ask the TSA to provide a time stamping certificate. This certificate will usually be stored on a Hardware Security Module.

You can subscribe to a time stamping service, in which case you get an URL and account information (a username and password). Or you can use a certificate that contains an URL of the time stamping server. With code sample 3.12, we try to find out if CAcert also offers timestamping services.

Code sample 3.12: Extracting the TSA URL from a certificate

```
BouncyCastleProvider provider = new BouncyCastleProvider();
Security.addProvider(provider);
KeyStore ks = KeyStore.getInstance("pkcs12", provider.getName());
ks.load(new FileInputStream(path), pass.toCharArray());
String alias = (String)ks.aliases().nextElement();
Certificate[] chain = ks.getCertificateChain(alias);
for (int i = 0; i < chain.length; i++) {
    X509Certificate cert = (X509Certificate)chain[i];
    System.out.println(String.format("[%s] %s", i, cert.getSubjectDN()));
    System.out.println(CertificateUtil.getTSAURL(cert));
}
```

This code returns `null` for both certificates, meaning CAcert probably doesn't offer any TSA services. Checking their web site confirms this.

Fortunately, I received an account for a timestamp server at GlobalSign. For some accounts you need an URL, a username and a password. In that case, you can create a `TSAClient` instance as is done in code sample 3.13.

Code sample 3.13: Creating a TSAClient

```
TSAClient tsaClient = new TSAClientBouncyCastle(tsaUrl, tsaUser, tsaPass);
```

Observe that we're again using an implementation that uses the Bouncy Castle library, just like we did for the `OcspClient`. Sometimes you only need an URL containing a token identifying the user. In this case you can drop the parameters `tsaUser` and `tsaPass`. There's also a constructor that accepts an estimated size and a digest algorithm. We'll learn more about the estimated size in the final section of this chapter.

How to recognize a document that has a timestamp

Let's take a look at the result. In figure 3.12, we've opened the document, and we're looking at the Date/Time information in the Signature properties. We've also opened the Certificate viewer for the timestamp certificate.

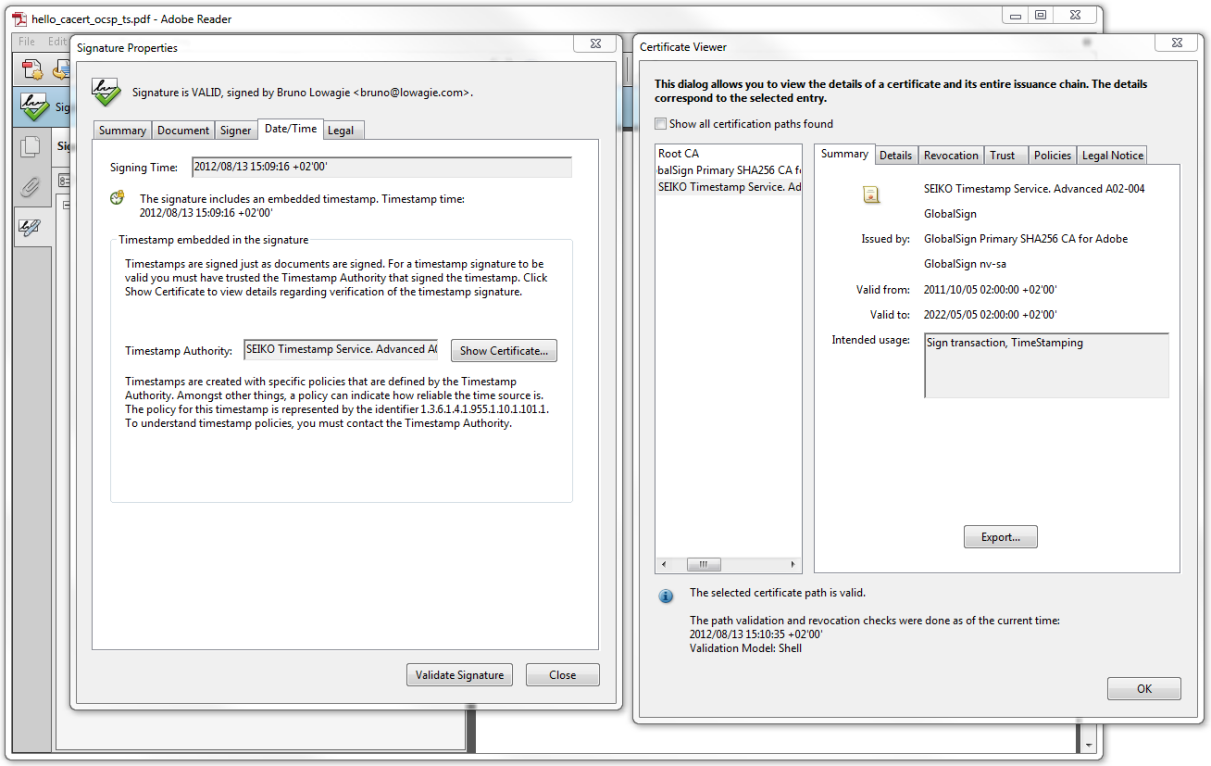

Figure 3.12: Looking at the timestamp info

Instead of saying *"Signing time is from the clock on the signer's computer"* as was the case with all previous documents we signed, we can now read *"The signature includes an embedded timestamp"* along with the exact time the document was signed and the certificate of the TSA, in this case the *"SEIKO Timestamp Service. Advanced A02-004"* from GlobalSign.

Retrieving TSA information during the time stamping process

We can follow the signing process, by adding a `Logger` instance to the `LoggerFactory`. We can see which CRLs iText is fetching. iText will also send a message to the `Logger` when a document gets its timestamp, but we may want to store that info somewhere. For instance: we may want to store the exact time returned from the TSA in a database. Let's take a look at code sample 3.14:

Code sample 3.14: Adding an event to a TSAClientBouncyCastle instance

```
TSAClientBouncyCastle tsaClient =
    new TSAClientBouncyCastle(tsaUrl, tsaUser, tsaPass);
tsaClient.setTSAInfo(
    new TSAInfoBouncyCastle() {
        public void inspectTimeStampTokenInfo(TimeStampTokenInfo info) {
            System.out.println(info.getGenTime());
        }
    }
);
```

In this example, we implement the `TSAInfoBouncyCastle` interface. As soon as iText receives a response from a TSA, it will trigger the `inspectTimeStampTokenInfo()` method. In this case, we write the generation time (a `Date` object) to the `System.out`.

> **NOTE:** Check out the API of Bouncy Castle's `TimeStampTokenInfo` class, and you'll discover that there's much more information you can retrieve. Instead of just writing this

info to a console, you could easily pass it to other objects to create your own logs in the form of a file or records in a database.

We've finally created a PDF document that complies with the best practices. There's only one thing that still bothers us: why do we have to add the root certificate of the CA to the Trusted Identities manually? Why can't Adobe Reader show a green check mark now that we've followed the rules by the book? Aren't there easier ways to trust root certificates?

3.4 How to get a green check mark

One of the most Frequently Asked Questions by iText customers sounds like this: *"We have signed a document correctly, and it shows a green check mark when we open it on our machines, but our end users complain that they get a message saying that the validity of the signer is unknown. How can we avoid this?"*

The answer they don't like to hear is: *"You have to ask the end users to trust the root certificate of the Certificate Authority."* They don't like this answer because there are far too many unknown and dangerous words in one and the same sentence. When you break up the sentence and provide a tutorial similar to what we discussed in section 2.2.3, they don't like the answer because *"There are too many difficult steps in that tutorial."* Fortunately, there are several other options.

3.4.1 Trust certificates stored elsewhere

A lot of software ships with a root store (for instance the cacerts file of the JDK), a lot of hardware ships with a root store (for instance: each smartphone has a root store). These root stores contain the root certificates of all the major CAs: GlobalSign, VeriSign, Entrust, and others.

There's also a root store in Acrobat/Reader, but by default, it only contains Adobe's root certificate. Since Adobe 9, an extra set of certificates approved by Adobe is downloaded automatically from an Adobe web site (see Section 3.4.2). As a result only signatures with one of these certificates in the root of the certificate chain get a 'green check mark' automatically, but you can change Adobe Reader's security settings.

Using the Windows Certificate Store

Chances are that the root certificate of that CA is already shipped with your Windows OS, or that you've already added it to your Windows Certificate Store for other purposes. You don't need to add and trust each of the certificates that are already present in the Windows Certificate Store manually. Instead, you can configure Adobe reader to search the operating system's certificate store when verifying signatures.

By default, this option is turned off. Go to *Edit > Preferences > Security > Advanced Preferences* and you'll see the dialog shown in figure 3.13. Just checkmark the options *Enable searching the Windows Certificate Store for the following operations*, choose which operations you approve, and you're all set. From now on, Adobe will trust all the certificates you trust in your OS.

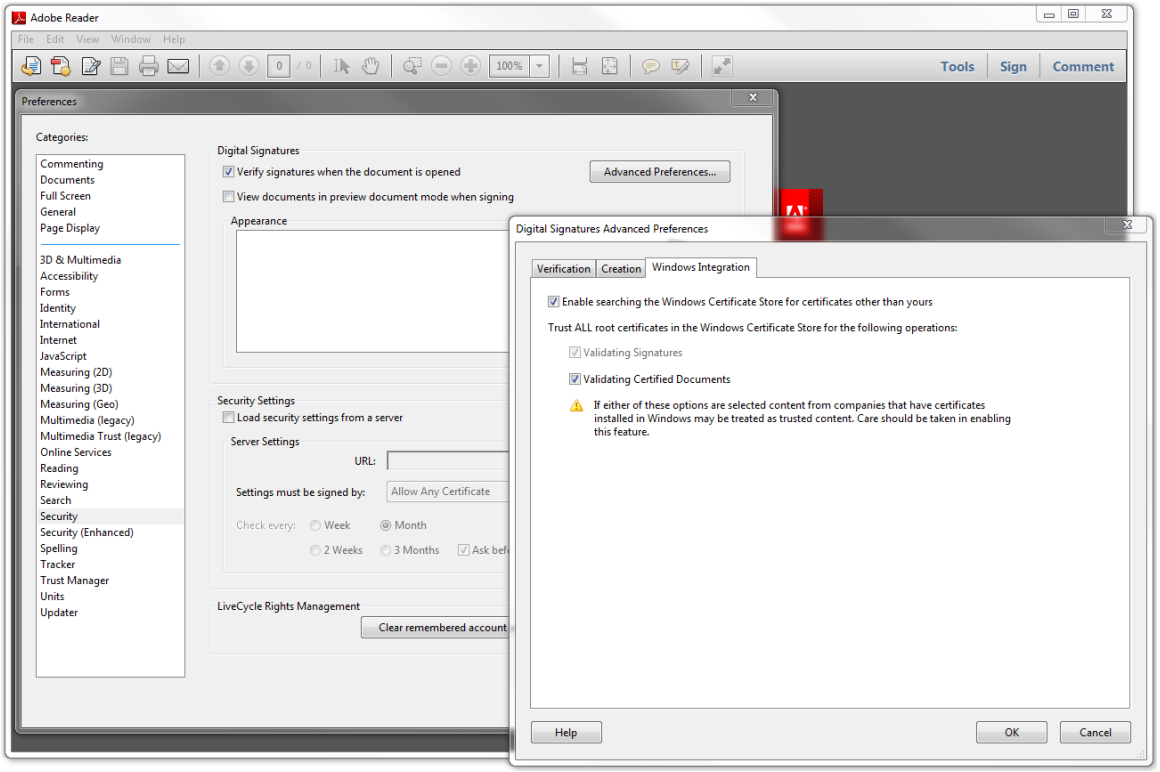

Figure 3.13: Changing the Security settings regarding certificates

You can also see another option in figure 3.13: you can load the Security settings from a server. In this case, a company's Sysadmin will have a tool to manage root certificates, and he can distribute a list of trusted certificates among the employees of the company.

This is also what Adobe does with its Approved Trust List.

3.4.2 Adobe Approved Trust List (AATL)

If the end user is outside the zone of your influence, the previous solutions won't work: you don't know which certificates he or she has in his certificate store, nor can you make him or her connect to a server. That's why Adobe introduced the Adobe Approved Trust List, a certificate trust program introduced with Adobe Reader 9[17]. Recent versions of Acrobat and Reader have been programmed to reach out to a web page to periodically download a list of "trusted root" digital certificates.

> **QUOTE:** Certificate authorities (CAs) as well as governments and businesses that provide certificates to their citizens and employees can apply to Adobe to join the AATL program by submitting application materials and their root certificates (or another qualifying certificate). After verifying that the applicant's services and credentials meet the assurance levels imposed by the AATL technical requirements, Adobe adds the certificate(s) to the Trust List itself, digitally signs the Trust List with an Adobe corporate digital ID that is linked to the Adobe Root certificate embedded in Adobe products, and then posts the list to a website hosted by Adobe.

Any digital signature created using a certificate that chains up to one of the high-assurance, trustworthy certificates on the list, is trusted automatically by Acrobat / Reader versions 9 and X. So

[17] http://www.adobe.com/security/approved-trust-list.html

if you buy a certificate, ask your CA if he's a partner in the AATL program, or check the list on the Adobe website[18].

But what if you also have to support version 8? Adobe has stopped support for Acrobat 8 in 2011[19], but there are still people out there with Adobe Reader 8. Should you ask them to upgrade? Yes, most certainly! Can you force them to upgrade? Probably not, so if you really need to support Reader 8, you need to use the predecessor of AATL: Adobe Certified Document Services.

Adobe Certified Document Services (CDS).

The Adobe Certified Document Services (CDS) system was introduced in 2005. With CDS, the signing certificate is chained up to Adobe's own root certificate. This certificate from Adobe is inherently trusted by Adobe Reader (including versions 9 and X). This way, CDS certified documents are validated without any action required by the user.

Adobe wants to make sure that certificates signed with its root certificate (or a certificate trusted in the AATL program) are as safe as possible. If a CA wants to issue certificates that are immediately trusted, he needs to follow Adobe's certificate policies. Both the AATL and CDS policies discuss topics such as: How are certificates issued? How are they created? Can they be easily compromised?

> **QUOTE:** Subscriber key pairs must be generated in a manner that ensures that the private key is not known by anybody other than the Subscriber or a Subscriber's authorized representative. Subscriber key pairs must be generated in a medium that prevents exportation or duplication and that meets or exceed FIPS 140-1 Level 2[20] certification standards.

The CDS program has been discontinued by Adobe, but a small number of vendors that were licensed to issue CDS certificates can still offer CDS certificates. In any case, both the AATL and the CDS program require that the key store is protected by a Hardware Security Module (HSM; see figure 3.14) or an USB token (see figure 3.15).

Figure 3.14: a Hardware Security Module (HSM) from SafeNet

[18] http://helpx.adobe.com/acrobat/kb/approved-trust-list1.html
[19] http://blogs.adobe.com/adobereader/2011/09/adobe-reader-and-acrobat-version-8-end-of-support.html
[20] http://csrc.nist.gov/publications/fips/fips1401.htm#sec1.2

A key store on a hardware device can be used to create a signature, but the private key can't be extracted. This means that we can no longer create a `KeyStore` object by passing an `InputStream` as we did in all the previous examples. What do we need to do instead?

Let's start with an example that uses the Windows Certificate Store to get access to an USB token.

3.4.3 Signing a document using a USB token (part 1: MSCAPI)

For this example, we'll use an iKey 4000 from SafeNet[21] (see figure 3.15).

Figure 3.15: an iKey 4000 from SafeNet

SafeNet only provides Windows drivers for this type of key. This means that we'll only be able to sign documents on a Windows machine. Check out the company's web site if you need a key that also works on other OSs.

Merely the iKey 4000 isn't enough for signing. We also need to put a private key on the USB token. I'm using a certificate issued by GlobalSign. In this white paper, we're using a certificate-based USB token for signing, but they can also be used for other purposes, such as authentication. You can use them to enable secure access, network logon, and so on.[22]

Inspecting the Certificates available on Windows

After I install the Windows drivers for the iKey 4000, my OS detects the USB token as soon as I plug it into a USB port. I can look for my certificate by opening the list of certificates available on my Windows installation. In figure 3.16, I used *MS Internet Explorer > Internet Options > Content* to get this information.

[21] http://www.safenet-inc.com/
[22] http://www.safenet-inc.com/data-protection/authentication/pki-authentication/

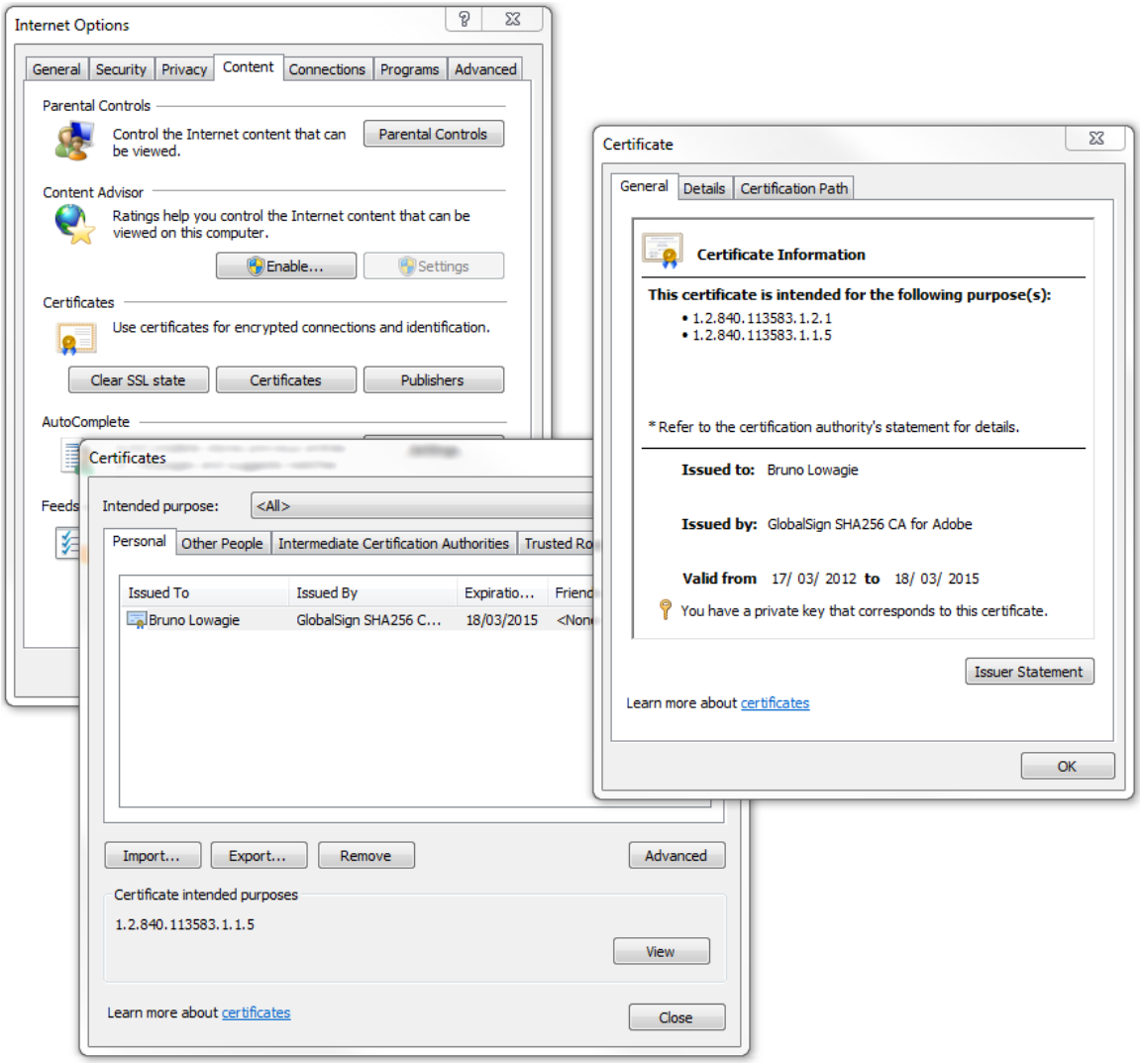

Figure 3.16: Internet Options > Content > Certificates

My GlobalSign certificate is present in my "`Windows-MY`" key store as long as I don't remove the USB token from my machine. I can address it through the Microsoft CryptoAPI (MSCAPI).

Signing a document using MSCAPI

When using Java, I need the `SunMSCAPI` provider, a class in the `sun.security.mscapi` package. I'm working on Windows 7, and as I was using the 64-bit version of Java 6, I had to upgrade to Java 7 because the `SunMSCAPI` provider wasn't available in the 64-bit version of Java 6.

> **NOTE:** if you need to upgrade to make this example work on your OS, you may want to avoid the 64-bit version of Java 7, and opt for the 32-bit version. The `SunPKCS11` provider is missing in the 64-bit version of Java 6 as well as Java 7. We'll need that provider in the next chapter. There's no ETA as to when Oracle will fix this problem.

We'll still use Bouncy Castle for dealing with the timestamp and doing the hashing, but we'll need to use the `SunMSCAPI` class as the provider for signing. If you use the Bouncy Castle provider, you'll get a *"Supplied key (*`sun.security.mscapi.RSAPrivateKey`*) is not a* `RSAPrivateKey` *instance"* exception.

In code sample 3.15, I'm explicitly adding an instance of `SunMSCAPI`. This isn't really necessary; that is: with the correct JDK it isn't. I'm adding it to make really sure the class is present in my JVM. I'll get a compile error if I try compiling the code using the 64-bit version of Java 6.

Code sample 3.15: Signing with a USB token using MSCAPI

```
public static void main(String[] args)
    throws IOException, GeneralSecurityException, DocumentException {
    LoggerFactory.getInstance().setLogger(new SysoLogger());
    BouncyCastleProvider providerBC = new BouncyCastleProvider();
    Security.addProvider(providerBC);
    SunMSCAPI providerMSCAPI = new SunMSCAPI();
    Security.addProvider(providerMSCAPI);
    KeyStore ks = KeyStore.getInstance("Windows-MY");
    ks.load(null, null);
    String alias = (String)ks.aliases().nextElement();
    PrivateKey pk = (PrivateKey)ks.getKey(alias, null);
    Certificate[] chain = ks.getCertificateChain(alias);
    OcspClient ocspClient = new OcspClientBouncyCastle();
    TSAClient tsaClient = null;
    for (int i = 0; i < chain.length; i++) {
        X509Certificate cert = (X509Certificate)chain[i];
        String tsaUrl = CertificateUtil.getTSAURL(cert);
        if (tsaUrl != null) {
            tsaClient = new TSAClientBouncyCastle(tsaUrl);
            break;
        }
    }
    List<CrlClient> crlList = new ArrayList<CrlClient>();
    crlList.add(new CrlClientOnline(chain));
    SignWithToken app = new SignWithToken();
    app.sign(SRC, DEST, chain, pk, DigestAlgorithms.SHA256,
        providerMSCAPI.getName(), CryptoStandard.CMS, "Test", "Ghent",
        crlList, ocspClient, tsaClient, 0);
}
```

Observe that we're now creating a `Keystore` instance using the parameter "`Windows-MY`". We load the `KeyStore` without passing an `InputStream` or a password. We just use `null`.

The same goes for the password for the `PrivateKey` object. All the signing operations are passed to the Microsoft CryptoAPI. As soon as the `MakeSignature` class wants to use the `PrivateKey` instance to create the signature, the MS CryptoAPI will address the device and its drivers. A dialog box will be opened (see figure 3.17), and as soon as I provide the correct passphrase the message digest will be sent to the device where it will be signed.

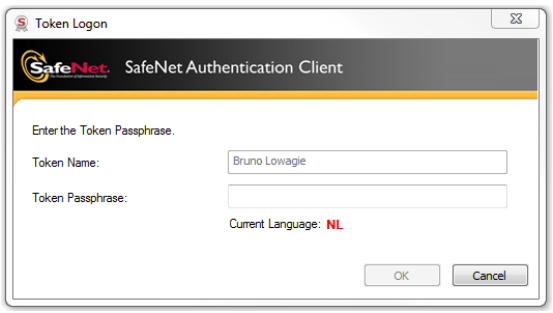

Figure 3.17: Password dialog for the iKey 4000

This is the first example where the actual signing isn't done on the JVM, but externally, on a separate device. We'll see some more examples in chapter 4 dedicated entirely to this type of signatures.

Inspecting the Certificate chain

When I open the resulting PDF, it is validated automatically. I immediately have a green check mark; I don't need to trust a certificate from a CA manually.

Let's take a look at the Certificate Viewer shown in figure 3.18 to find out why.

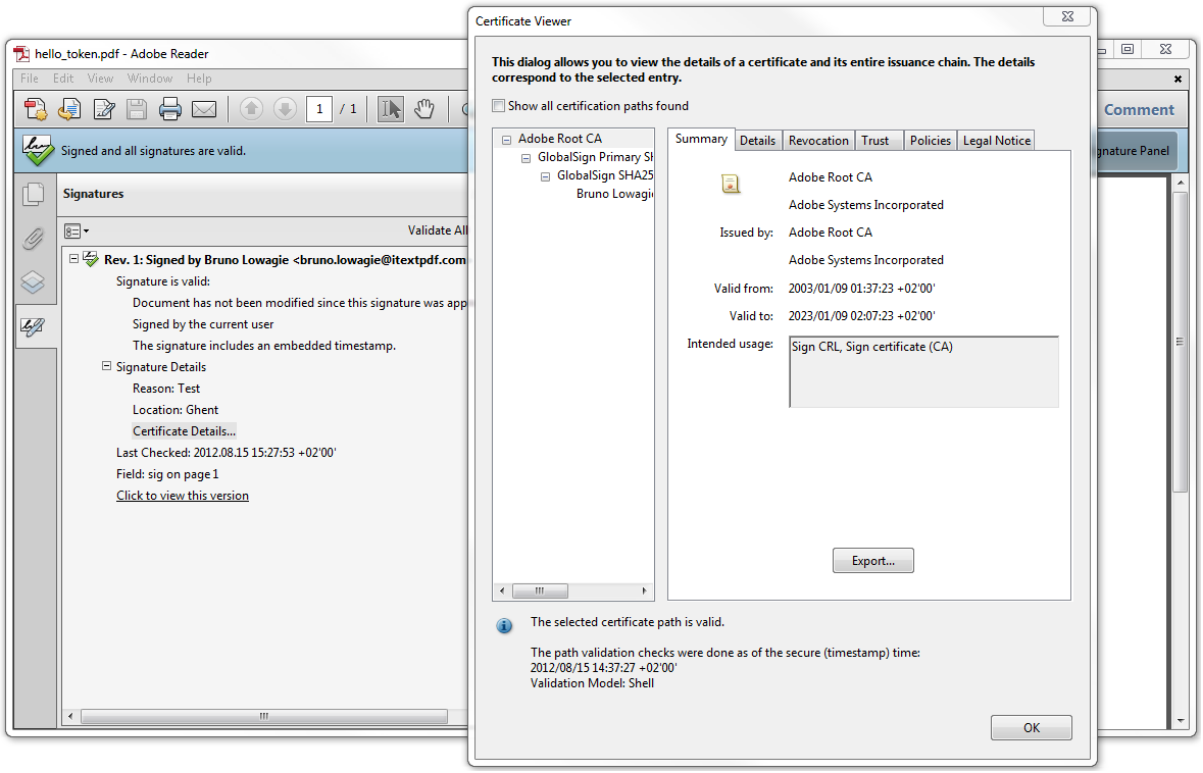

Figure 3.18: The Certificate Viewer showing the Adobe Root CA Certificate

We see that the certificate chain now has four entries. The signing certificate is my own. It was signed by a *'GlobalSign SHA256 CA for Adobe'* certificate. That certificate was in turn signed by a *'GlobalSign Primary SHA256 CA for Adobe'* certificate. At the top-level, there's the *'Adobe Root CA'* certificate. GlobalSign is a partner in Adobe's Certified Document Services program, and they are allowed to issue certificates with Adobe's root certificate. The presence of this certificate explains why I didn't need to add GlobalSign's root certificate one way or another to get a green check mark.

This would be a good time for you to ask why GlobalSign added two extra certificates in the chain. I'd answer that we already discussed this in section 3.2. When talking about CRLs, I made a remark that good CAs do extra efforts to keep their CRLs as small as possible. Using intermediate certificates helps them to achieve this goal.

Introducing intermediate certificates

On figure 3.19, you can see that GlobalSign has an intermediate certificate *'GlobalSign Primary SHA256 CA for Adobe'*, used to issue certificates for Hardware Security Modules (HSM).

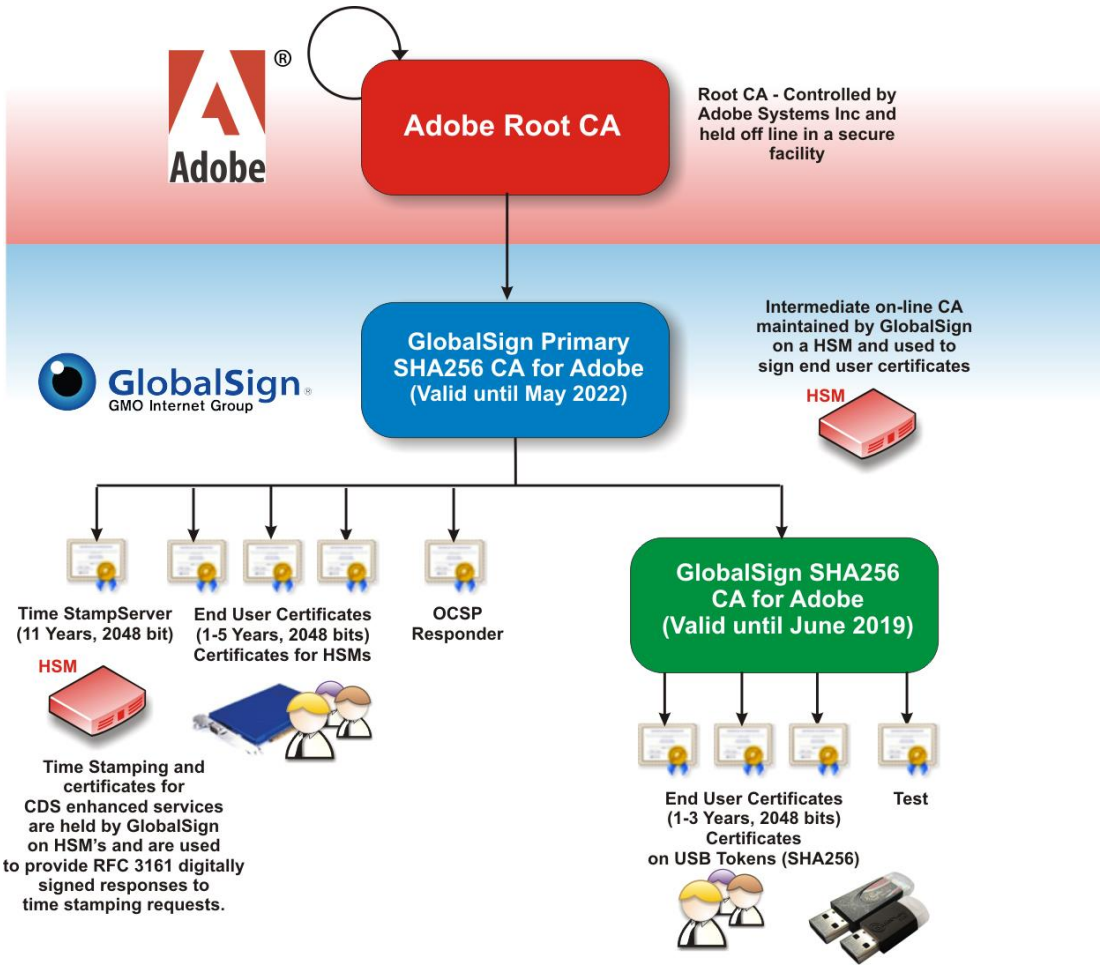

Figure 3.19: The Adobe Root CA, the intermediate GlobalSign and the end-user certificates

Also using its Primary certificate, GlobalSign issued another intermediate certificate *'GlobalSign SHA256 CA for Adobe'*. With this certificate, they issue end user certificates for USB tokens. We already saw these certificates in figure 3.18 when we looked at the certificate chain in the document we signed with the iKey 4000.

Why the need for an extra intermediate certificate? As you saw in figure 3.17, I have to enter my passphrase when I use my USB token. If I enter the wrong password a couple of times, my private key will be wiped from the device, and I'll have to revoke it. Also: a USB token can easily be lost or stolen, in which case I'll also have to revoke it. If all USB tokens would be issued immediately under the Adobe Root certificate, the CRL for the Adobe Root would be huge. By building in intermediate levels, the respective CRLs can be kept small.

> **NOTE:** It's less trivial to lose or steal an HSM from a server rack. Certificates stored on an HSM aren't very likely to be revoked. That's why there's no need for more than one level between the certificate on the HSM and the Adobe Root certificate.

In the next chapter, we'll create some more signatures on hardware, even on a smart card, but there's one more parameter in the signing process we should discuss before we can close this chapter.

3.5 Estimating the size of the signature content

Do you remember when we first looked at the concept of a digitally signed PDF document? In figure 2.1, I drew a schema of a document with content in blue and a signature in light red. I explained that iText either keeps the blue bytes in memory, or stores them in a temporary file (section 2.2.4) reserving a 'hole' that can only be filled once the blue bytes are hashed and signed.

How does iText know in advance which size the signature will have? How does it determine the size of the hole? The answer is: iText doesn't know. iText makes an educated guess.

There's a parameter named `estimatedSize` in code sample 3.2. Up until now, we've always passed `0` as its value. In this case, iText will start with a value of 8192 bytes. It will add the number of bytes of the CRLs, and an extra 10 bytes for every CRL. 4192 bytes will be added if you defined an `OcspClient`, another 4192 if you created a `TSAClient` (unless you've defined another size in the class that implements the time stamping interface). Normally, this educated guess creates a hole that is too big. If you look inside the PDF, you'll see plenty of zeros at the end of the signature bytes. So be it. It's better to provide too much space, than not enough.

Suppose you have an algorithm that can make a better estimation that is lower than iText's guess. In that case, you can pass your estimated size to the `MakeSignature` class. In the case of code sample 3.16, iText would reserve 16,579 bytes: 8,192 (the minimum guess) + 4,192 (for the OCSP) + 4,192 (for the TSA). But let's start with an estimated size of 13,000, and see what happens. If 13,000 bytes aren't enough, we'll add another 50, and we'll keep doing so until the signature fits.

Code sample 3.16: Making a hit-and-miss estimation of the signature size

```
boolean succeeded = false;
int estimatedSize = 10300;
while (!succeeded) {
    try {
        System.out.println("Attempt: " + estimatedSize + " bytes");
        app.sign(SRC, DEST, chain, pk, DigestAlgorithms.SHA256, provider.getName(),
            CryptoStandard.CMS, "Test", "Ghent", null, ocspClient, tsaClient,
            estimatedSize);
        succeeded = true;
        System.out.println("Succeeded!");
    }
    catch(IOException ioe) {
        System.out.println("Not succeeded: " + ioe.getMessage());
        estimatedSize += 50;
    }
}
```

Let's take a look at the result:

```
Attempt: 10300 bytes
Not succeeded: Not enough space
Attempt: 10350 bytes
Not succeeded: Not enough space
Attempt: 10400 bytes
Succeeded!
```

The actual size of the signature was somewhere between 10,350 and 10,400 bytes. This means that iText reserved at least 6,176 bytes that weren't necessary for the signature.

Is that a high price to avoid an `IOException`? Maybe, but I don't think so. In some situations, you have to enter a PIN for each signing operation. You don't want to repeat that action for each try. Some smart cards take more than a second to create a signature. In our hit-and-miss example, we've made three connections to a timestamp server. Is that acceptable? Does that cost more than having a file with 6 extra Kbytes? Maybe, maybe not; it's up to you to make a choice.

3.6 Summary

In this chapter, we've learned about the best practices for signing PDF documents. First we were introduced to certificate authorities. We found out that they act as trusted entities that issue certificates to end users.

One could argue that all the signatures we created in the previous chapter were useless and of no value: they didn't contain any certificate revocation information, nor did they have a timestamp. We experimented with different ways to embed certificate revocation information into a PDF: we worked with online and offline Certificate Revocation Lists and we used the Online Certificate Status Protocol.

Then we discovered the need for a Time Stamping Authority, and we connected to a timestamp server. All of this didn't automatically result in a PDF that showed us a green check mark automatically, but then we created a document that was signed using an USB token and a signing certificate that was chained up to the Adobe root.

Finally, we created a small example that explained why documents that are signed using iText are slightly bigger than strictly necessary. In the next chapter, we'll start by rewriting the example with the iKey 4000, but we'll no longer depend on the Microsoft CryptoAPI, instead we'll use PKCS#11.

4 Creating signatures externally

In the previous chapter, we've created one document that wasn't signed in the JVM of the machine where iText was running. The signature was created on an external device, more specifically, a USB token. Figure 4.1 shows the architecture of such an example.

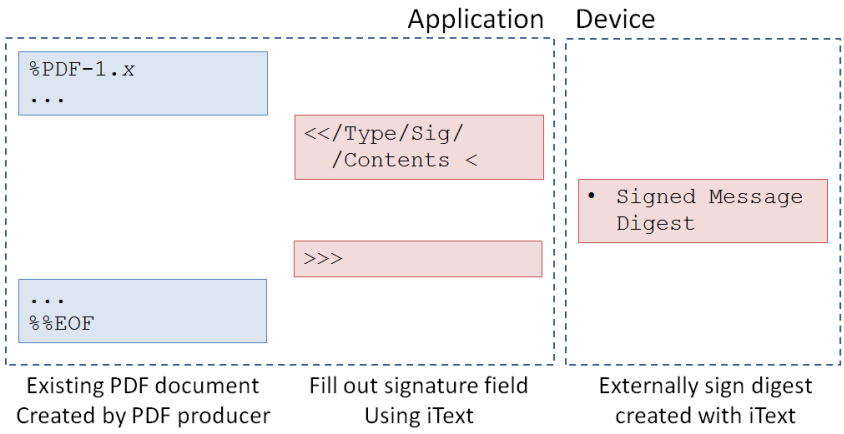

Figure 4.1: Architecture of a solution where a hash is signed externally

To the left, we have an application using iText; to the right, we have an application creating the signature without the help of iText. In this chapter, we'll look at some other examples where this architecture applies. We'll start with some PKCS#11 examples, and then study a smart card solution. We'll finish by discussing different client/server solutions.

4.1 Signing a document using PKCS#11

The example we discussed in the previous chapter only worked on Windows because we were relying on the Windows Certificate Store. Can't we change the example so that it also works on other OSs? Another disadvantage of that MSCAPI example was that the signer had to enter his password every time he wanted to sign a document. That may be a requirement when you're at a notary signing important documents —you don't want to enter your PIN and discover you've signed more documents than you intended to—, but you don't want to see a popup asking for a passphrase when documents need to be signed in unattended mode on a server.

4.1.1 Signing a document using an HSM

Suppose that your company needs to send out a huge amount of signed documents. You could assign an employee as the person responsible for signing documents, and have him use a company certificate to sign each document one by one. That's madness of course. Imagine the CEO of a company with thousands of customers having to sign each contract manually. Typically you'll want an automated solution. You'll want to sign those documents on a server, using the company's HSM.

In that case, I have good news for you: you don't need to change a single thing to the `sign()` method we've used before! You can copy and paste code sample 3.2, and half of your work is done. Just like with MSCAPI, you only need to change the security provider.

Code sample 4.1: Signing a document using PKCS#11

```
LoggerFactory.getInstance().setLogger(new SysoLogger());

Properties properties = new Properties();
properties.load(new FileInputStream(PROPS));
```

```
char[] pass = properties.getProperty("PASSWORD").toCharArray();
String pkcs11cfg = properties.getProperty("PKCS11CFG");

BouncyCastleProvider providerBC = new BouncyCastleProvider();
Security.addProvider(providerBC);
FileInputStream fis = new FileInputStream(pkcs11cfg);
Provider providerPKCS11 = new SunPKCS11(fis);
Security.addProvider(providerPKCS11);

KeyStore ks = KeyStore.getInstance("PKCS11");
ks.load(null, pass);
String alias = (String)ks.aliases().nextElement();
PrivateKey pk = (PrivateKey)ks.getKey(alias, pass);
Certificate[] chain = ks.getCertificateChain(alias);

OcspClient ocspClient = new OcspClientBouncyCastle();
TSAClient tsaClient = null;
for (int i = 0; i < chain.length; i++) {
    X509Certificate cert = (X509Certificate)chain[i];
    String tsaUrl = CertificateUtil.getTSAURL(cert);
    if (tsaUrl != null) {
        tsaClient = new TSAClientBouncyCastle(tsaUrl);
        break;
    }
}
List<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(new CrlClientOnline(chain));
SignWithPKCS11HSM app = new SignWithPKCS11HSM();
app.sign(SRC, DEST, chain, pk, DigestAlgorithms.SHA256, providerPKCS11.getName(),
    CryptoStandard.CMS, "HSM test", "Ghent", crlList, ocspClient, tsaClient, 0);
```

It's not really necessary to create a `Logger`, but it helps you to see what happens during the signing process. The first difference is that we don't load a key store file and a password, but a configuration file and a password. In my case, the content of the configuration file looks like this:

```
Name = Luna
library = /usr/lunasa/lib/libCryptoki2_64.so
slot = 1
```

I'm working on a Luna SA from SafeNet[23]. It's connected to a Linux machine, hence the name `Luna` and the reference to an Executable and Linkable Format (ELF) file. The HSM itself is installed in a 'slot' inside the Luna SA. In my case, I use slot `1`.

Again we'll add the Bouncy Castle provider because we rely on it for the timestamp, and we reuse the `sign()` method that uses Bouncy Castle to create the digest of the PDF bytes, but now we also create an instance of a `SunPKCS11` provider by passing an `InputStream` that allows it to read the configuration file. In our case, the name of the provider will be "`SunPKCS11-Luna`".

Now we create a PKCS#11 `KeyStore` using the parameter "`PKCS11`", and we load it using the password for the HSM. From that moment on, all the code is almost identical to what we had before. The only change you need to make is the provider used for signing. You can still use the `PrivateKeySignature` class, but don't forget that it now expects "`SunPKCS11-Luna`" instead of "`BC`".

[23] http://www.safenet-inc.com/products/data-protection/hardware-security-modules/luna-sa/

IMPORTANT: on Windows, the SunPKCS11 provider is missing the the 64-bit version of the JDK. You need to use the 32-bit version if you want the PKCS#11 examples to work.

Only one question remains: *What to put in the configuration file?* The answer: *It depends.* Ask the vendor of the device and he'll help you out. Or let's try another example ourselves.

4.1.2 Signing a document using a USB token (part 2: PKCS#11)

We could easily reuse the source from code sample 4.1 and use it to sign a document on Windows with our iKey 4000. The only difference would be the configuration file. I can pick any name, so let's choose "`ikey4000`" (you can choose any other name you want). I'm working on Windows now, so I need a path to a DLL instead of to an ELF file. Finally, I need to know which slot to use.

We'll create the configuration file dynamically as shown in code sample 4.2, and use a separate method to determine the number of the slot to use.

Code sample 4.2: creating a configuration file dynamically

```
public static final String DLL = "c:/windows/system32/dkck201.dll";

String config = "name=ikey4000\n"
    + "library=" + DLL + "\n"
    + "slotListIndex = " + getSlotsWithTokens(DLL)[0];
ByteArrayInputStream bais = new ByteArrayInputStream(config.getBytes());
Provider providerPKCS11 = new SunPKCS11(bais);
Security.addProvider(providerPKCS11);
```

Make you have the correct path to the CRYPTOKI (PKCS#11) DLL. If you want to make this work on Linux, you'll need a path to a `.so`-file such as `libCryptoki2_64.so`, `libpkcs11.so`…

Determining the slot index

Code sample 4.3 shows a method named `getSlotsWithTokens()` to find the index of the slot.

Code sample 4.3: getting the slot used for the iKey 4000

```
public static long[] getSlotsWithTokens(String libraryPath) throws IOException{
    CK_C_INITIALIZE_ARGS initArgs = new CK_C_INITIALIZE_ARGS();
    String functionList = "C_GetFunctionList";
    initArgs.flags = 0;
    PKCS11 tmpPKCS11 = null;
    long[] slotList = null;
    try {
        try {
            tmpPKCS11 =
                PKCS11.getInstance(libraryPath, functionList, initArgs, false);
        } catch (IOException ex) {
            ex.printStackTrace();
            throw ex;
        }
    } catch (PKCS11Exception e) {
        try {
            initArgs = null;
            tmpPKCS11 =
                PKCS11.getInstance(libraryPath, functionList, initArgs, true);
        } catch (IOException ex) {
             ex.printStackTrace();
        } catch (PKCS11Exception ex) {
```

```
                ex.printStackTrace();
            }
        }
        try {
            slotList = tmpPKCS11.C_GetSlotList(true);
            for (long slot : slotList){
                CK_TOKEN_INFO tokenInfo = tmpPKCS11.C_GetTokenInfo(slot);
                System.out.println("slot: "+slot+"\nmanufacturerID: "
                    + String.valueOf(tokenInfo.manufacturerID) + "\nmodel: "
                    + String.valueOf(tokenInfo.model));
            }
        } catch (PKCS11Exception ex) {
            ex.printStackTrace();
        } catch (Throwable t) {
            t.printStackTrace();
        }
        return slotList;
    }
```

In my case, I'm using only one USB token, and the method writes the following info to the console:

```
slot: 2
manufacturerID: ©SafeNet, Inc.
model: Model 400
```

The method returns an array containing only one `int` value, so it's OK to use the first (and only) item in the array. Now that I know that the value is `2`, I could easily reuse the code from sample 4.1 changing nothing but the configuration file. It doesn't matter if we're signing on an HSM or on a USB token. It doesn't matter if we're signing in a Windows or on a Linux machine. We're using PKCS#11.

Possible use case

Imagine a situation of a large corporation, for instance an international car rental company.

For each car that is rented, a contract needs to be signed by the party that owns the car as well as the party that will use the car. For the first signature, a choice needs to be made. The contract can be signed using a company certificate on a Hardware Security Module; or the contract can be signed by an employee using his personal company USB token.

If the contract is signed by the customer first, you could counter-sign it on a server, and immediately archive all contracts in a central repository. If the contract is signed by an employee using his personal USB token, you'll always have 100% certainty regarding the identity of the person who is responsible for the contract, but maintaining certificates for each individual employee will be a significant cost if you have many.

No matter which solution you choose, there will always be advantages as well as disadvantages, and we mustn't forget about another important question: *how will the end user sign?* Usually the end user doesn't own a public/private key pair, does he?

As a matter of fact, he could; in many European countries, he does. I live in Belgium, and every grown-up citizen in my country owns at least two key stores. That is: every citizen who possesses a valid identity card.

4.1.3 Signing a document using a smart card

Although I'm 40+, I sometimes have to show my identity card as proof of my age in the US. The last time this happened was in a grocery store where I wanted to buy some beer. I was kind of flattered: *Do I still look like I'm under 21?*

Suppose that I was only 20 years old, I wouldn't have been allowed to make the purchase, but watching movies has taught me that it's relatively easy for teenagers to make a fake ID. Whether that is truth or fiction, I don't know. In any case: there are better solutions to verify one's identity, especially for matters more important than buying alcoholic beverages.

A Belgian use case: identity cards

Upon birth, each Belgian citizen is assigned a unique national number. He or she is registered in the National Register, which is a central database containing key attributes about Belgian citizens. Foreigners are also registered and issued with a national number at their first encounter with the Belgian social security administration, for instance a foreign employee working in Belgium or a tourist who needs medical assistance. This national number is used on an identity card.

An identity card is mandatory for every Belgian citizen who has reached the age of twelve. Children between six and twelve years old are issued a kid's card. A foreigner's card is issued to non-Belgians, for instance persons with a residence permit of five years. Figure 4.2 shows two identity cards: my own, personal identity card, and a card that is available for testing purposes (in the name of Alice SPECIMEN).

Figure 4.2: Belgian identity cards and a smart card reader

As you can see, the Belgian identity card contains a chip: it's a smart card. In Belgium, we refer to it as our eID card (or simply eID). It's valid for five years, and it has four functions. [24]

1. *It shows information printed on the card*— it shows the name, birth date, gender, nationality, and some numbers. It also shows the card holder's picture for physical identification of the holder. On the back, some of the info is repeated in machine readable form.
2. *It has information stored on the chip*— the identity information, e.g. the national number, can be digitally captured. The pass photo can be extracted in JPEG-format.
3. *It can be used for authentication*— the creation of authentication signatures allows the card holder to prove his identity. He can use his eID to file taxes online, to get a registered mail at the post office, and so on. An authentication signature is accompanied with the citizen's authentication certificate. This certificate contains privacy-sensitive information, such as the national number of the citizen, which in turn also discloses gender and birth date.
4. *It can be used for non-repudiation*— the creation of non-repudiation signatures enables the card holder to generate signatures that are legally binding. These are also known as qualified electronic signatures, which are defined by the European Directive on electronic signatures. This directive implies that non-repudiation signatures, as produced by an eID card, are equivalent to handwritten signatures. This distinguishes them from authentication signatures in the sense that a judge is forced to accept their legal validity should they be at issue.

The first function is visual identification. The other three functions require a smart card reader as the one shown in the picture. Two functions involve the cryptographic capabilities of the card.

Signing a document using a smart card on Windows

If I connect the smart card reader to my computer and insert the cards, they show up in my Windows Certificate Store. See figure 4.3.

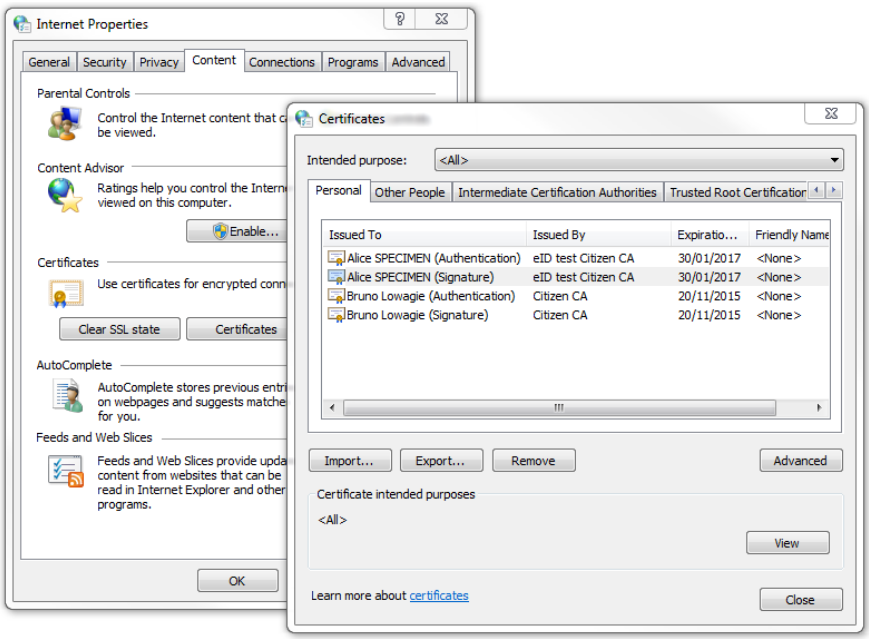

Figure 4.3: Smart cards listed in the Windows Certificate Store

[24] The following paragraphs are taken from http://www.cosic.esat.kuleuven.be/publications/article-1160.pdf (De Cock, Danny; Simoens, Koen; Preneel, Bart, 2008)

As you can see, there are two certificates for each card: one for Authentication, one for signing. We can now run the code from section 3.4.3 (code sample 3.15) using the "Windows-MY" `KeyStore` and the `SunMSCAPI` provider. Windows will get a request for signing, and select the appropriate certificate. As each eID is protected with a PIN code, a dialog will open as shown in figure 4.4.

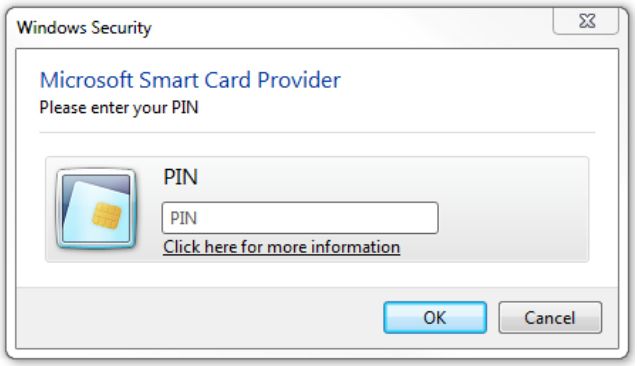

Figure 4.4: the Microsoft Smart Card Provider PIN dialog

The MSCAPI example works perfectly without having to change anything to the code, but what about the PKCS#11 examples?

Signing a document using a smart card and PKCS#11

First of all, we need a DLL that can be used to create a `SunPKCS11` instance (or a `.so`-file if you're working on Linux). As there's more than one key store on the card, we'll also need to choose the appropriate alias. There's a snippet in code sample 4.4 that lists the available aliases.

Code sample 4.4: Signing a document with a smart card using PKCS#11

```
public static final String DLL = "c:/windows/system32/beidpkcs11.dll";

public static void main(String[] args)
    throws IOException, GeneralSecurityException, DocumentException {
    LoggerFactory.getInstance().setLogger(new SysoLogger());
    String config = "name=beid\n" +
        "library=" + DLL + "\n" +
        "slotListIndex = " + getSlotsWithTokens(DLL)[0];
    ByteArrayInputStream bais = new ByteArrayInputStream(config.getBytes());
    Provider providerPKCS11 = new SunPKCS11(bais);
    Security.addProvider(providerPKCS11);
    BouncyCastleProvider providerBC = new BouncyCastleProvider();
    Security.addProvider(providerBC);
    KeyStore ks = KeyStore.getInstance("PKCS11");
    ks.load(null, null);
    Enumeration<String> aliases = ks.aliases();
        while (aliases.hasMoreElements()) {
        System.out.println(aliases.nextElement());
    }
    smartcardsign(providerPKCS11.getName(), ks, "Authentication");
    smartcardsign(providerPKCS11.getName(), ks, "Signature");
}
public static void smartcardsign(String provider, KeyStore ks, String alias)
    throws GeneralSecurityException, IOException, DocumentException {
    PrivateKey pk = (PrivateKey)ks.getKey(alias, null);
    Certificate[] chain = ks.getCertificateChain(alias);
    OcspClient ocspClient = new OcspClientBouncyCastle();
```

```
    List<CrlClient> crlList = new ArrayList<CrlClient>();
    crlList.add(new CrlClientOnline(chain));
    SignWithPKCS11SC app = new SignWithPKCS11SC();
    app.sign(SRC, String.format(DEST, alias), chain, pk,
        DigestAlgorithms.SHA256, provider, CryptoStandard.CMS, "Test", "Ghent",
        crlList, ocspClient, null, 0);
}
```

First we see that we now need a DLL named `beidpkcs11.dll`. This file is installed automatically when you install the drivers for the Belgian eID. If we look at the output produced by the method that gets the slot index, we see.

```
slot: 0
manufacturerID: Belgium Government
model: Belgium eID
```

Note that when I tried this with a different card reader, I got a different slot index.

In previous examples, we always used the first alias we encountered in the key store. This time, there's more than one key store on the device. If we list the aliases, we get the following output.

```
Root
CA
Authentication
Signature
```

The `Root` and `CA` key store are to be used by the government only, for instance to sign the information on your eID. Typically, you don't know the full password to use them, otherwise you'd be able to change your name, gender, address yourself.

Let's use the `Authentication` and the `Signature` alias, and create two files. We'll use `DEST` as a pattern so that we can distinguish which key was used. Again, we'll have to enter our PIN. Figure 4.5 shows what happens the first time we sign, using the key for authentication. We don't get a dialog generated by the OS, but a dialog created by the `beidpkcs11.dll`.

Figure 4.5: Signing with the key for authentication

It says: *Enter your PIN to authenticate yourself.* Although you can digitally sign a document using the key for authentication, you're not creating a legally binding signature. For a signature to be legally valid, you need to use the key for non-repudiation. Figure 4.6 explains the consequences when you enter our PIN code.

Figure 4.6: Signing with the key for non-repudiation

Let me translate from Dutch: *"Watch out: You're about to place a legally binding electronic signature using your identity card. Enter your PIN to continue. Warning: If you only want to log in on a web site or a server, DO NOT enter your PIN."* This sounds ominous, but using the non-repudiation key is the only way to use your eID to create a signature that is legally binding!

Is the eID card safe?

The administration knows its citizens through the information contained in its National Register. This is the authentic source of the citizen's basic set of identity information, including his or her official address, marital status, children, ID card, passport and driving license. Identity cards can only be issued to persons who are referenced in the National Register. [25]

Passive authentication

The issuing authority digitally signs all the information on the eID card. The card contains all certificates needed to verify these signatures. It's up to the verifier to check whether the root certificates are genuine. This *passive authentication* permits the detection of manipulated identity data. It doesn't prevent copying the data files to another chip; neither does it prevent chip substitution, such as placing the chip of one eID into another one.

Active authentication

The goal of *active authentication* is to confirm that the chip is genuine. To this end, a terminal can initiate a challenge-response protocol with the chip. The terminal sends a random challenge to the chip which in turn signs the challenge using its private chip authentication key. The terminal retrieves the resulting signature. The terminal then obtains a genuine copy of the public key needed to verify the signature, either from the chip itself or retrieves it from its infrastructure. If the signature is correct, then a genuine chip was involved in the process.

Active authentication permits detection of card cloning. Unfortunately, this does not protect against chip-in-the-middle attacks, whereby the challenged chip forwards the challenge to a genuine chip and relays the reply from the genuine chip to the terminal. Nor does it prevent replay of challenge attacks, where an attacker sends random challenges to a genuine passport and builds a data base of challenge-response pairs for later use.

[25] The following paragraphs are taken from http://www.cosic.esat.kuleuven.be/publications/article-1160.pdf (De Cock, Danny; Simoens, Koen; Preneel, Bart, 2008)

Can a third party sign in your name using your eID?

A major issue with the eID card is the lack of a strong link between the logical identity on the card and the physical identity of the card holder. Although the eID could contain several biometrics, only the card holder's photo and handwritten signature are currently included. This implies that it's impossible to unquestionably link a record in the Register with a physical person. Biometric verification of the requester of an eID would prevent identity theft.

For the moment, the PIN serves as a link between the signature and the genuine card holder, but this is a weak link. A correct signature only implies that someone was able to make the eID card create a signature. The actual card user is not necessarily the genuine card holder. If somebody steals your eID and knows your PIN, he or she can sign legally binding documents in your name.

This explains why it's important that:

- If an eID card is lost or stolen before it expires, the validity of the card's certificates is blocked by revoking these certificates.
- You should be very careful with your PIN code. So far, we've signed documents using a smart card reader similar to the one in figure 4.2. We had to enter the PIN using an ordinary keyboard, and our PIN could have been captured by software. Figure 4.7 shows some smart card readers that allow you to enter your PIN on a pin-pad. These devices are more secure.

In figure 4.7, we see (from left to right), the VASCO Digipass 920, the VASCO Digipass 865, and the VASCO Digipass 855. For more info about these devices, please visit the VASCO web site[26].

Figure 4.7: Smart card readers with a pin-pad

[26] https://www.vasco.com/

When we run the MSCAPI and the PKCS#11 examples using one of these devices, we'll enter the pin code on the smart card reader's PIN pad, not on the keyboard. The PIN is verified on the smart card: it never leaves the device. It can't be captured by any other device or machine. Just make sure nobody is watching over your shoulder or filming you while you're entering your PIN.

We'll discuss these smart card readers in more detail in the next section, where we'll explore yet another way not only to sign a document using a smart card, but also how to extract information from the Belgian eID.

4.2 Signing a document with a Smart Card using javax.smartcardio

In Java 6, a new package was introduced in the JDK: `javax.smartcardio`. This package allows Java applications to interact with a smart card as described in ISO/IEC 7816-4.

A smart card is a plastic card with a chip (see figure 4.2). The chip is a mini computer with an operating system. It has a processor, memory, a file system and it can run its own applications. In this section, we'll use the smart card package from the JDK to retrieve information from an eID. We'll also run an application: we'll sign a byte array on the smart card using one of its private keys.

But let's start by taking a closer look at the `javax.smartcardio` API.

4.2.1 Overview of the most important classes in javax.smartcardio

First we need the `TerminalFactory` to create a `CardTerminals` object. This object contains a list of `CardTerminal` instances. These are the terminals connected to your computer. If one of those devices contains a smart card, you can create a `Card` object using the `connect()` method.

You can communicate with the card using Application Protocol Data Unit (APDU) commands and responses. You'll create a `CardChannel`, use the `transmit()` method to send a `CommandAPDU` to the card, and you'll get a `ResponseAPDU` in return. The structure of the bytes that are to be sent and received is shown in figure 4.8.

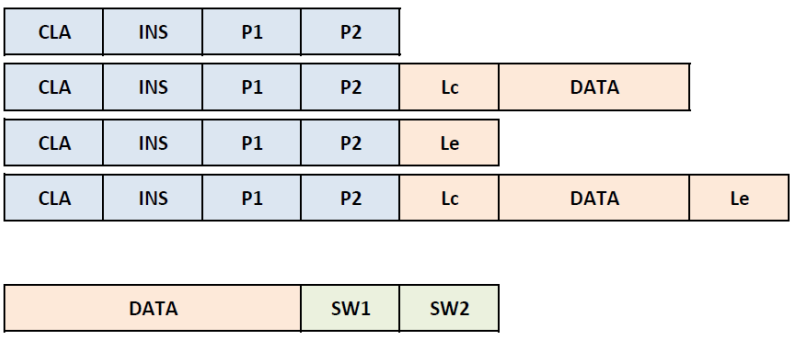

Figure 4.8: Command and response APDUs

Figure 4.8 shows four possible command APDUs. There's always an instruction class (`CLA`), an instruction code (`INS`), and two parameters (`P1` and `P2`). The APDU can also contain a byte array (`DATA`) of a certain length (`Lc`). The command can expect a response of a certain length (`Le`). The response can consist of a block of `DATA`, followed by two status bytes (`SW1` and `SW2`).

> **NOTE:** `CLA`, `INS`, `P1`, `P2`, `SW1`, and `SW2` each take one byte. Their possible values are listed in part 4 of ISO/IEC 7816. `Lc` can consist of 0, 1 or 3 bytes, `Le` of 0, 1, 2, or 3 bytes.

The Card object also has a `transmitControlCommand()` method. This triggers an action on the smart card reader. For example, you can send a series of commands to verify the PIN code of a smart card on the reader's PIN pad.

We can use the `javax.smartcardio` API to access any smart card, but unfortunately it's usually not that simple. Every card has its own file structure, its own applications, its own data definitions, and so on. Apart from the ISO/IEC specification, you'll always need the specifications of the card you want to use. I've written a small library that wraps the generic Java API classes in objects such as `CardReaders`, `SmartCard` and `SmartCardWithKey`. I wrote a special class for the Belgian eID, named `BeIDCard`. You can find this library on SourceForge[27]. It's meant as a light-weight library that supports only the basic functionality.

There are other, more elaborate libraries available[28], but I chose to do a rewrite because most of the libraries are found are hard to read, whereas I introduced a class named `IsoIec7816`, containing constants such as `INS_READ_BINARY`, `P1_DIGITAL_SIGNATURE`, `SW1_WARNING` so that it's easier for you to read the code. The other libraries work with the actual values of those constants such as `0xB0`, `0x9E`, `0x63`, which makes it hard to understand what the code is supposed to do.

4.2.2 Extracting data from the Belgian eID using smartcardsign

Each eID contains three data files, namely the citizen's pass photo in JPEG format, his or her identity data, and his or her official address. The citizen's identity data file contains name, given names, national number, nationality, birth place and date, gender, title, special status (for example: 'white cane' for blind people) and a cryptographic hash of the photo. Other information in the file refers to the card and the chip: card number, validity period, document type (for example: Belgian citizen or EU citizen), and the issuing municipality.

The format of these files is proprietary. It will be different for different countries. I've created three POJOs for each type of file on the Belgian eID: `AddressPojo`, `IdentityPojo`, and `PhotoPojo`. The *smartcardsign* library hides all the complexity to get the info.

Code sample 4.5: Reading the contents from a Belgian eID

```
CardReaders readers = new CardReaders();
for (CardTerminal terminal : readers.getReaders()) {
    System.out.println(terminal.getName());
}
for (CardTerminal terminal : readers.getReadersWithCard()) {
    System.out.println(terminal.getName());
    SmartCard card = new SmartCard(terminal);
    IdentityPojo id = BeIDFileFactory.getIdentity(card);
    System.out.println(id.toString());
    AddressPojo address = BeIDFileFactory.getAddress(card);
    System.out.println(address);
    PhotoPojo photo = BeIDFileFactory.getPhoto(card);
    FileOutputStream fos = new FileOutputStream(PHOTO);
    fos.write(photo.getPhoto());
    fos.flush();
    fos.close();
}
```

[27] https://sourceforge.net/projects/smartcardsign/
[28] https://code.google.com/p/eid-applet/

In code sample 4.5, we start by listing all the card terminals. `CardReaders` is a class in my own library, `CardTerminal` is a `javax.smartcardio` class. We're only interested in the card readers that have a smart card inserted. We create a `SmartCard` object and we can pass it to the `BeIDFileFactory` class. The methods in this factory know how the data files are organized on the Belgian eID. They return POJOs containing information fields in a format that is easy for us to use.

If we run this code with the test card, we get the following output:

```
card number: 000000157220
card valid from: Wed Jan 25 00:00:00 CET 2012 until Wed Jan 25 00:00:00 CET 2017
issued: Antwerpen
national number: 71715100070
Name: SPECIMEN, Alice Geldigekaart A
Nationality: BELG
Birth Location: Hamont-Achel
Birth Date: 01 JAN 1971
Sex: V
Noble condition:
Document type: 1
Special status: 0
Street + number: Meirplaats 1 bus 1
Zip: 2000
Municipality: Antwerpen
```

There's more info on the card, but this is what I implemented in the POJOs. We've also stored the pass photo in the results directory. In Belgium, it's always a black and white photo of about 3 Kbytes.

Now let's return to the scope of this paper, and find out how to sign a document.

4.2.3 Signing data for the purpose of authentication

The section 4.1.3, we found out that the Belgian eID contains two key stores that can be used by the card holder: one for authentication and one for non-repudiation. Code sample 4.6 shows us how to sign a simple message such as "ABCD" for authentication.

Code sample 4.6: Signing a String for authentication

```
CardReaders readers = new CardReaders();
for (CardTerminal terminal : readers.getReadersWithCard()) {
    SmartCardWithKey card = new SmartCardWithKey(terminal,
        BeIDCertificates.AUTHENTICATION_KEY_ID, "RSA");
    card.setPinProvider(new PinDialog(4));
    byte[] signed = card.sign("ABCD".getBytes(), "SHA-256");
    System.out.println(new String(signed));
    X509Certificate cert =
        card.readCertificate(BeIDCertificates.AUTHN_CERT_FILE_ID);
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE, cert.getPublicKey());
    System.out.println(new String(cipher.doFinal(signed)));
}
```

Instead of a `SmartCard` instance, we now create a `SmartCardWithKey`. We want to use the key store for authentication, so we pass the appropriate file id (as defined in the specs for the eID). The encryption algorithm used on the Belgian eID is "RSA".

If you're not using a smart card reader with a PIN pad, you'll need a way to ask the user for his pin. This is done with the `setPinProvider()` method. This method expects a `PinProvider`, which is an interface with only one method: `getPin()`.

I wrote an implementation named `PinDialog`. This class will show a `JOptionPane` confirm dialog as shown in figure 4.9.

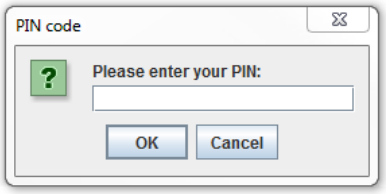

Figure 4.9: a custom PinDialog

The `sign()` method of the `SmartCardWithKey` object will do all the heavy-lifting, and return a series of signed bytes. We could now use the `readCertificate()` method to read the public certificate that corresponds with the private key that was used to sign the data. When we decrypt the signed bytes using the key in that certificate, we'll recognize our original "`ABCD`" message.

Don't worry if all of this sounds too complex. The *smartcardsign* package has a `BeIDCard` class and an `EidSignature` implementation of the `ExternSignature` interface that makes things much easier if you want to sign a document.

4.2.4 Signing a document with the Belgian eID

In code sample 4.7, we create a `BeIDCard` instance, which is a subclass of `SmartCardWithKey`. This way, we can't make the mistake choosing the key for authentication instead of the key for signing. We also don't have to worry about the encryption algorithm; the `BeIDCard` knows that RSA is used for the Belgian eID.

Code sample 4.7: Creating a BeIDCard instance for signing

```
CardReaders readers = new CardReaders();
SmartCardWithKey card = new BeIDCard(readers.getReadersWithCard().get(0));
card.setSecure(true);
Certificate[] chain = BeIDCertificates.getSignCertificateChain(card);
Collection<CrlClient> crlList = new ArrayList<CrlClient>();
crlList.add(new CrlClientOnline(chain));
OcspClient ocspClient = new OcspClientBouncyCastle();
SignWithBEID app = new SignWithBEID();
app.sign(SRC, DEST, card, chain, CryptoStandard.CMS,
    "Test", "Ghent", crlList, ocspClient, null, 0);
```

We don't pass a `PinProvider` in this example. Instead we've used the `setSecure()` method to instruct the library to allow secure signing only. This means this example will only work with a smart card reader that has a PIN pad. If you use an ordinary smart card reader, an exception will be thrown.

> **IMPORTANT:** If you're working on Windows, you may experience a problem with the smart card readers with a PIN pad. When you plug such a device in for the first time, Windows immediately installs its own Microsoft USB CCID smartcard reader (WUDF) driver. This driver discards all commands sent with the `transmitControlCommand()` method, which disables features such as verifying the PIN code on the PIN pad.

You can solve this problem by installing the VASCO Digipass CCID filter driver. It filters all I/O control commands before they are processed by the driver and re-maps the commands to proprietary APDU calls that can be interpreted by a VASCO Digipass card reader[29].

We're using the `BeIDCertificates` convenience class to compose the certificate chain. The first certificate in the chain is the non-repudiation certificate of the card holder. Just like the authentication certificate, it's issued by the Citizen Certificate Authority (Citizen CA). The Citizen CA certificate is the next one in the chain. The top certificate is the self-signed Belgian Root CA certificate (see figure 4.10). The `getSignCertificateChain()` method returns an array containing the three certificates involved in the signing process in the correct order.

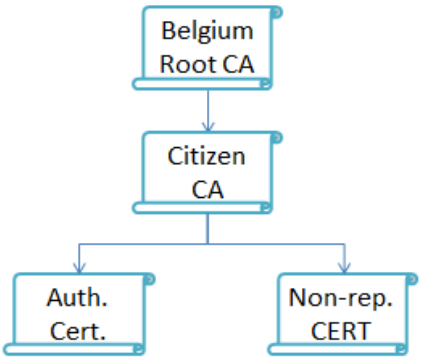

Figure 4.10: eID Certificate hiërarchy

The rest of the code sample creates the objects that are needed to get the CRL and the OCSP response. The Belgian government doesn't provide a time stamping server for public use.

Now let's take a look at the code that signs the PDF document in code sample 4.8.

Code sample 4.8: Signing a document using the EidSignature class

```
public void sign(String src, String dest,
    SmartCardWithKey card, Certificate[] chain, CryptoStandard subfilter,
    String reason, String location, Collection<CrlClient> crlList,
    OcspClient ocspClient, TSAClient tsaClient, int estimatedSize)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    // Creating the signature
    ExternalSignature eid = new EidSignature(card, "SHA256", "BC");
    ExternalDigest digest = new BouncyCastleDigest();
    MakeSignature.signDetached(appearance, digest, eid, chain,
        crlList, ocspClient, tsaClient, estimatedSize, subfilter);
}
```

[29] See http://lowagie.com/smartcardreaders

There's only one difference with what we had before. Instead of using a `PrivateKeySignature`, we're now defining an `EidSignature`. It expects a `SmartCardWithKey` instance, a hashing algorithm and a provider. Internally it will use the same `sign()` method we've already used in code sample 4.6, but now with the key for non-repudiation.

In the next section, we'll write our own implementation of the `ExternalSignature` interface, because we'll be signing a document in one process (for instance on the client), but we'll create the actual signature in another process (for instance on the server).

4.3 Client/server architectures for signing

I'm currently working on a project with the following use case. A university has an SAP system that allows them to create renting contracts for student homes. These contracts are Reader-enabled forms with two signature fields. The form is sent to a student who signs the document with Adobe Reader and the student's eID. The signed form is then sent to the administration where it needs to be countersigned by the university. This is done using a signing server. A document containing one signature can be sent to this server from a limited number of IP-addresses, and the server returns the document signed with two signatures: the original signature from the student and the university's signature. One possible way to implement this requirement, is by creating a signing service.

4.3.1 Building a simple signature server

I've created a key store in the name of an Unknown person working for an Unknown unit in an Unknown company. I'll use that key store and the self-signed certificate to provide a signing service on this URL: http://demo.itextsupport.com/SigningApp/ (see figure 4.11).

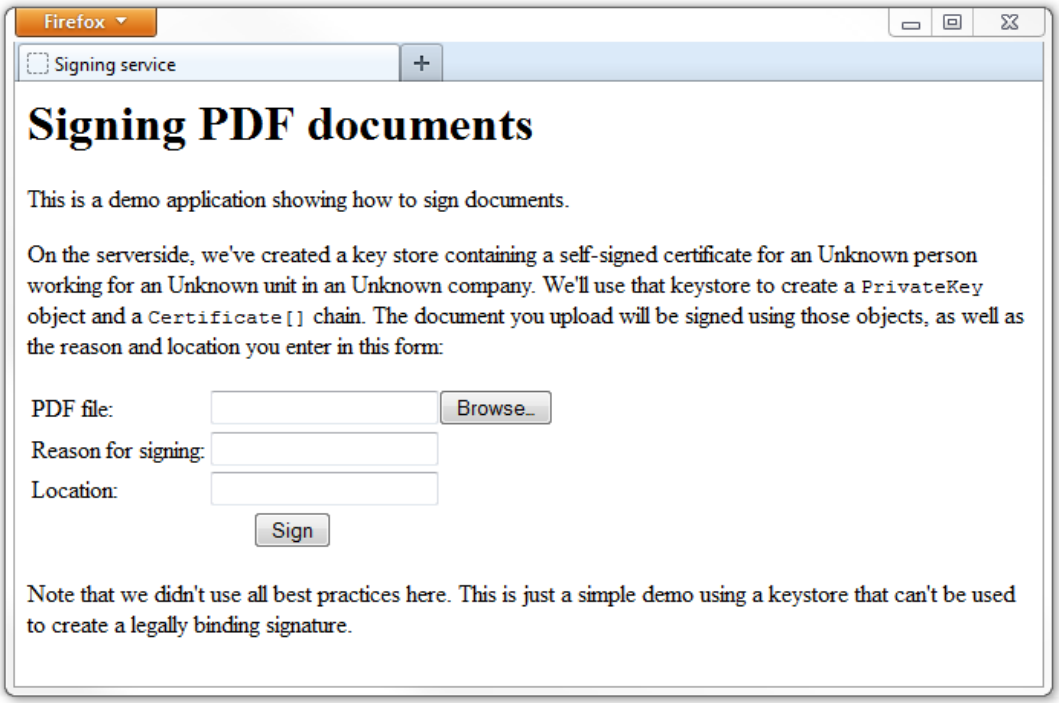

Figure 4.11: Simple signing service

In a real-world example, we'd use an HSM, and we'd add plenty of security to avoid that anybody can access the server and sign in our name. This example was created for demo purposes only. You can try it, but you'll have to adapt the code if you want to use it in a production environment.

A signing Servlet

Assume we have a simple HTML page that allows you to select a PDF on your local machine, add a reason for signing and a location, and submit the file to sign it on the server. Code sample 4.9 shows the `web.xml` for this web app.

Code sample 4.9: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
    xmlns=http://java.sun.com/xml/ns/j2ee
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>SigningApp</display-name>
  <servlet>
    <description>Signs a PDF document</description>
    <display-name>SignServlet</display-name>
    <servlet-name>SignServlet</servlet-name>
    <servlet-class>com.itextpdf.sign.SignServlet</servlet-class>
    <init-param>
      <description>Keystore containing the public/private key</description>
      <param-name>keystorepath</param-name>
      <param-value>/WEB-INF/.keystore</param-value>
    </init-param>
    <init-param>
      <description>Keystore type</description>
      <param-name>keystoretype</param-name>
      <param-value>jks</param-value>
    </init-param>
    <init-param>
      <description>Password for the keystore</description>
      <param-name>storepass</param-name>
      <param-value>f00b4r</param-value>
    </init-param>
    <init-param>
      <description>Alias for the key pair</description>
      <param-name>alias</param-name>
      <param-value>itextpdf</param-value>
    </init-param>
    <init-param>
      <description>Password for the key pair</description>
      <param-name>keypass</param-name>
      <param-value>f1lmf3st</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>SignServlet</servlet-name>
    <url-pattern>/sign</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

As you can see, we've created a `SigningServlet` to which we're passing parameters such as the location of the key store, its password, the alias for the keys we'll use, and the password for the private key. Code sample 4.10 shows the code for that Servlet.

Code Sample 4.10: the SignServlet class

```
public class SignServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        String keystorepath = getServletConfig().getInitParameter("keystorepath");
        String keystoretype = getServletConfig().getInitParameter("keystoretype");
        String alias = getServletConfig().getInitParameter("alias");
        String storepass = getServletConfig().getInitParameter("storepass");
        String keypass = getServletConfig().getInitParameter("keypass");
        try {
            InputStream is = getServletContext().getResourceAsStream(keystorepath);
            KeyStore ks = KeyStore.getInstance(keystoretype);
            ks.load(is, storepass.toCharArray());
            PrivateKey pk = (PrivateKey)ks.getKey(alias, keypass.toCharArray());
            Certificate[] chain = ks.getCertificateChain(alias);
            SignImp signImp = new SignImp(pk, chain);
            config.getServletContext().setAttribute("signImp", signImp);
        } catch (GeneralSecurityException e) {
            throw new ServletException(e);
        } catch (IOException e) {
            throw new ServletException(e);
        }
    }
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setHeader("Expires", "0");
        resp.setHeader("Cache-Control",
            "must-revalidate, post-check=0, pre-check=0");
        resp.setHeader("Pragma", "public");
        resp.setContentType("application/pdf");
        OutputStream os = resp.getOutputStream();
        try {
            InputStream is = null;
            String reason = "";
            String location = "";
            FileItemFactory factory = new DiskFileItemFactory();
            ServletFileUpload upload = new ServletFileUpload(factory);
            List<FileItem> items = upload.parseRequest(req);
            for (FileItem item : items) {
                if ("pdf".equals(item.getFieldName())) {
                    is = item.getInputStream();
                }
                else if ("reason".equals(item.getFieldName())) {
                    reason = item.getString();
                }
                else if ("location".equals(item.getFieldName())) {
                    location = item.getString();
                }
            }
            SignImp signImp =
                (SignImp) getServletContext().getAttribute("signImp");
            byte[] result = signImp.signDoc(is, reason, location);
            resp.setContentLength(result.length);
            for (int i = 0; i < result.length; i++) {
                os.write(result[i]);
            }
        } catch (Exception e) {
            throw new ExceptionConverter(e);
```

```
        }
        os.flush();
        os.close();
    }
}
```

In the `init()` method, we get the parameters from the `web.xml`, we create a `PrivateKey` object and a `Certificate` array, and we use these classes to construct a `SignImp` instance. We store this instance in the Servlet context. We'll reuse it every time somebody wants to sign a file. That is: every time somebody triggers the `doPost()` method. In this method, we use the Apache Commons library for file upload[30] to get the parameters: a reason, a location, and an `InputStream` for the PDF that is being uploaded. We pass these parameters to our `signImp` instance retrieved from the servlet context. Its `signDoc()` method returns a `byte[]` that can be sent to the `OutputStream` of the `HttpResponse` object.

The signing implementation

The `SignImp` class isn't much different from all the examples we've written before. When I write Servlets that involve the use of iText, I always start by writing a short standalone application with a `main()` method. This way I can test the code before integrating it into a web application.

In code sample 4.11, I've removed the `main()` method for brevity.

Code sample 4.11: the SignImp class

```
public class SignImp {
    private PrivateKey pk;
    private Certificate[] chain;
    public SignImp(PrivateKey pk, Certificate[] chain) {
        this.pk = pk;
        this.chain = chain;
    }
    public byte[] signDoc(InputStream pdf, String reason, String location)
        throws GeneralSecurityException, IOException, DocumentException {
        // Creating the reader and the stamper
        PdfReader reader = new PdfReader(pdf, null);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PdfStamper stamper =
            PdfStamper.createSignature(reader, baos, '\0', null, true);
        // Creating the appearance
        PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
        appearance.setReason(reason);
        appearance.setLocation(location);
        // Creating the signature
        ExternalSignature pks =
            new PrivateKeySignature(pk, DigestAlgorithms.SHA256, null);
        ExternalDigest digest = new BouncyCastleDigest();
        MakeSignature.signDetached(appearance, digest, pks, chain,
            null, null, null, 0, CryptoStandard.CMS);
        return baos.toByteArray();
    }
}
```

[30] http://commons.apache.org/fileupload/

You should adapt this example according to the best practices described in chapter 3. As this is a sample available online for everybody to test, I omitted all the extras on purpose; I don't want anybody to mistake a PDF with a test signature with an officially signed PDF.

BouncyCastle-related problems

If you look closely, you'll notice one peculiar difference with previous examples: I'm not using Bouncy Castle anymore as the security provider. We noticed a problem that is caused by running two different web apps in the same JVM that require a different version of Bouncy Castle. One application adds the `BouncyCastleProvider` version 1.46 to the `Security` class; the other tries to add the `BouncyCastleProvider` version 1.47. This leads to all kinds of exceptions.

I would strongly advise against sharing a JVM when deploying an application that signs documents. It opens the way for security liabilities: you don't want a rogue web application to have access to your private key and credentials. I'm mentioning this problem because I experienced it on a test server, and I thought it would be interesting to share that experience. It's the kind of problem you run into if you're an early adopter of new versions of software.

Suppose that you're an early adopter too, and that you always want to use the most recent version of iText. In that case, you'll have to upgrade often. But is this in line with your company's policy? Maybe your boss doesn't want you to upgrade the code that runs on a machine with a Hardware Security Module a couple of times a year. If that is the case, you could separate the PDF functionality from the signing functionality.

4.3.2 Signing a document on the client using a signature created on the server

In section 4.1 and 4.2, we've signed a PDF using iText and a standalone Java program, but the actual signature was created on another device: a Hardware Security Module, a USB token, a smart card. Now we're going to look at an example where we'll sign a PDF in a standalone application, but we'll create the signature on a server. Let's take a look at the client side first (code sample 4.12).

Code sample 4.12: The client-side application

```
public static final String SRC = "src/main/resources/hello.pdf";
public static final String DEST = "results/chapter4/hello_server.pdf";
public static final String CERT =
    "http://demo.itextsupport.com/SigningApp/itextpdf.cer";

public static void main(String[] args)
    throws GeneralSecurityException, IOException, DocumentException {
    CertificateFactory factory = CertificateFactory.getInstance("X.509");
    URL certUrl = new URL(CERT);
    Certificate[] chain = new Certificate[1];
    chain[0] = factory.generateCertificate(certUrl.openStream());
    ClientServerSigning app = new ClientServerSigning();
    app.sign(SRC, DEST, chain, CryptoStandard.CMS, "Test", "Ghent");
}
public void sign(String src, String dest, Certificate[] chain,
    CryptoStandard subfilter, String reason, String location)
    throws GeneralSecurityException, IOException, DocumentException {
    // Creating the reader and the stamper
    PdfReader reader = new PdfReader(src);
    FileOutputStream os = new FileOutputStream(dest);
    PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');
```

```
    // Creating the appearance
    PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
    appearance.setReason(reason);
    appearance.setLocation(location);
    appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
    // Creating the signature
    ExternalDigest digest = new BouncyCastleDigest();
    ExternalSignature signature = new ServerSignature();
    MakeSignature.signDetached(appearance, digest, signature, chain,
        null, null, null, 0, subfilter);
}
```

We start by retrieving the certificate(s) from the server. We're reusing the self-signed certificate created for the previous example. We could easily keep a local copy of the certificates in the chain as these certificates are public anyway. In the `sign()` method, we now use a custom implementation of the `ExternalSignature` interface. The source of this class is shown in code sample 4.13.

Code sample 4.13: The ServerSignature class

```
public class ServerSignature implements ExternalSignature {
    public static final String SIGN =
        "http://demo.itextsupport.com/SigningApp/signbytes";

    public String getHashAlgorithm() {
        return DigestAlgorithms.SHA256;
    }

    public String getEncryptionAlgorithm() {
        return "RSA";
    }

    public byte[] sign(byte[] message) throws GeneralSecurityException {
        try {
            URL url = new URL(SIGN);
            HttpURLConnection conn = (HttpURLConnection)url.openConnection();
            conn.setDoOutput(true);
            conn.setRequestMethod("POST");
            conn.connect();
            OutputStream os = conn.getOutputStream();
            os.write(message);
            os.flush();
            os.close();
            InputStream is = conn.getInputStream();
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            byte[] b = new byte[1];
            int read;
            while ((read = is.read(b)) != -1) {
                baos.write(b, 0, read);
            }
            is.close();
            return baos.toByteArray();
        } catch (IOException e) {
            throw new ExceptionConverter(e);
        }
    }
}
```

Let's assume that we'll always sign using SHA-256 with RSA on the server. As soon as the `sign()` method is triggered from within iText, a HTTP Post connection is opened to the server. We upload a `byte[]` containing the hash of a byte range in the PDF along with authenticated attributes. We receive an array with the signed bytes.

What happens on the server side? That's shown in code sample 4.14.

Code sample 4.14: The Servlet that signs the hash

```
public class SignBytes extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        String keystorepath = getServletConfig().getInitParameter("keystorepath");
        String keystoretype = getServletConfig().getInitParameter("keystoretype");
        String alias = getServletConfig().getInitParameter("alias");
        String storepass = getServletConfig().getInitParameter("storepass");
        String keypass = getServletConfig().getInitParameter("keypass");
        try {
            InputStream is = getServletContext().getResourceAsStream(keystorepath);
            KeyStore ks = KeyStore.getInstance(keystoretype);
            ks.load(is, storepass.toCharArray());
            PrivateKey pk = (PrivateKey)ks.getKey(alias, keypass.toCharArray());
            config.getServletContext().setAttribute("pk", pk);
        } catch (GeneralSecurityException e) {
            throw new ServletException(e);
        } catch (IOException e) {
            throw new ServletException(e);
        }
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setContentType("application/octet-stream");
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        InputStream is = req.getInputStream();
        int read;
        byte[] data = new byte[256];
        while ((read = is.read(data, 0, data.length)) != -1) {
            baos.write(data, 0, read);
        }
        data = baos.toByteArray();
        try {
            Signature sig = Signature.getInstance("SHA256withRSA");
            PrivateKey pk = (PrivateKey)getServletContext().getAttribute("pk");
            sig.initSign(pk);
            sig.update(data);
            data = sig.sign();
            OutputStream os = resp.getOutputStream();
            os.write(data, 0, data.length);
            os.flush();
            os.close();
        } catch (GeneralSecurityException e) {
            e.printStackTrace();
        }
    }
}
```

Again we read all the parameters from the `web.xml`, but now we store a `PrivateKey` object in the Servlet context. Whenever the `sign()` method is triggered in the `ServerSignature` class, the `SignBytes` Servlet accepts those bytes in the `doPost()` method. The bytes are signed using the `java.security.Signature` class.

> **NOTE:** iText isn't involved in the server-side process. The signed bytes are sent to the client through the `OutputStream` of the `HttpResponse`. The client accepts the signed bytes, and iText puts them in the PDF.

The reverse can also be done: suppose you have a document on the server, and you want to create a signature on the client, but you don't want to use iText in the client-side application.

4.3.3 Signing a document on the server using a signature created on the client

This example is more complex because we need two requests from the client:

- *Pre-signing*— the client asks the server for a hash.
- *Post-signing*— the client sends the signed bytes to the server.

We'll create a Proof-of-Concept that makes a lot of assumptions. Building a robust application would lead us too far astray. Code sample 4.15 shows the first part of the code on the client side.

Code sample 4.15: Calling the PreSign Servlet from the client

```
// we make a connection to a PreSign servlet
URL url = new URL("http://demo.itextsupport.com/SigningApp/presign";);
HttpURLConnection conn = (HttpURLConnection)url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.connect();
// we upload our self-signed certificate
OutputStream os = conn.getOutputStream();
FileInputStream fis = new FileInputStream(CERT);
int read;
byte[] data = new byte[256];
while ((read = fis.read(data, 0, data.length)) != -1) {
    os.write(data, 0, read);
}
os.flush();
os.close();
// we use cookies to maintain a session
List<String> cookies = conn.getHeaderFields().get("Set-Cookie");
// we receive a hash that needs to be signed
InputStream is = conn.getInputStream();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
data = new byte[256];
while ((read = is.read(data)) != -1) {
    baos.write(data, 0, read);
}
is.close();
byte[] hash = baos.toByteArray();
```

We open a connection to a Servlet and we post our self-signed certificate (`CERT`) so that it can be used to prepare a signed document on the server. We receive a `byte[]` that needs to be signed on the client. Code sample 4.16 shows how this `byte[]` was created.

Code sample 4.16: the PreSign servlet

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    resp.setContentType("application/octet-stream");
    try {
        // We get the self-signed certificate from the client
        CertificateFactory factory = CertificateFactory.getInstance("X.509");
        Certificate[] chain = new Certificate[1];
        chain[0] = factory.generateCertificate(req.getInputStream());
        // we create a reader and a stamper
        PdfReader reader = new PdfReader(hello);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PdfStamper stamper = PdfStamper.createSignature(reader, baos, '\0');
        // we create the signature appearance
        PdfSignatureAppearance sap = stamper.getSignatureAppearance();
        sap.setReason("Test");
        sap.setLocation("On a server!");
        sap.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
        sap.setCertificate(chain[0]);
        // we create the signature infrastructure
        PdfSignature dic = new PdfSignature(
            PdfName.ADOBE_PPKLITE,  PdfName.ADBE_PKCS7_DETACHED);
        dic.setReason(sap.getReason());
        dic.setLocation(sap.getLocation());
        dic.setContact(sap.getContact());
        dic.setDate(new PdfDate(sap.getSignDate()));
        sap.setCryptoDictionary(dic);
        HashMap<PdfName, Integer> exc = new HashMap<PdfName, Integer>();
        exc.put(PdfName.CONTENTS, new Integer(8192 * 2 + 2));
        sap.preClose(exc);
        ExternalDigest externalDigest = new ExternalDigest() {
            public MessageDigest getMessageDigest(String hashAlgorithm)
                throws GeneralSecurityException {
                return DigestAlgorithms.getMessageDigest(hashAlgorithm, null);
            }
        };
        PdfPKCS7 sgn = new PdfPKCS7(null, chain, "SHA256",
            null, externalDigest, false);
        InputStream data = sap.getRangeStream();
        byte hash[] = DigestAlgorithms.digest(data,
            externalDigest.getMessageDigest("SHA256"));
        Calendar cal = Calendar.getInstance();
        byte[] sh = sgn.getAuthenticatedAttributeBytes(hash, cal,
            null, null, CryptoStandard.CMS);
        // We store the objects we'll need for post signing in a session
        HttpSession session = req.getSession(true);
        session.setAttribute("sgn", sgn);
        session.setAttribute("hash", hash);
        session.setAttribute("cal", cal);
        session.setAttribute("sap", sap);
        session.setAttribute("baos", baos);
        // we write the hash that needs to be signed to the HttpResponse output
        OutputStream os = resp.getOutputStream();
        os.write(sh, 0, sh.length);
        os.flush();
        os.close();
    } catch (DocumentException e) {
        throw new IOException(e);
```

```
    } catch (GeneralSecurityException e) {
        throw new IOException(e);
    }
}
```

It's immediately clear that this is only a Proof of Concept. We start by creating a certificate chain that consists of only one certificate; in reality, the chain will consist of more certificates. We create a reader (for a *Hello World* file), a stamper, and an appearance.

Instead of using `MakeSignature`, we have to do all the nitty-gritty work of creating the signature infrastructure ourselves. I won't go into the details, but this part of the code gives you an idea of what iText looks like on the inside.

> **NOTE:** If you want to create a real-world application where the PDF remains on the server, and the signature is created on the client, you'll probably want to create an implementation of the `ExternalSignature` interface similar to what we've done in section 4.3.2.
> You probably won't keep all the information in memory, but serialize all the objects involved as early as possible.

The Servlet responds with a `byte[]` that needs to be signed on the client. Once we receive the the bytes, we'll need to shape them correctly and insert them in the PDF. We store the objects needed to achieve this in an `HttpSession`. Code sample 4.17 shows the second part of the client application.

Code sample 4.17: Calling the PostSign Servlet from the client

```
// we load our private key from the key store
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
ks.load(new FileInputStream(KEYSTORE), PASSWORD);
String alias = (String)ks.aliases().nextElement();
PrivateKey pk = (PrivateKey) ks.getKey(alias, PASSWORD);
// we sign the bytes received from the server
Signature sig = Signature.getInstance("SHA256withRSA");
sig.initSign(pk);
sig.update(hash);
data = sig.sign();
// we make a connection to the PostSign Servlet
url = new URL("http://demo.itextsupport.com/SigningApp/postsign");
conn = (HttpURLConnection)url.openConnection();
for (String cookie : cookies) {
    conn.addRequestProperty("Cookie", cookie.split(";", 2)[0]);
}
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.connect();
// we upload the signed bytes
os = conn.getOutputStream();
os.write(data);
os.flush();
os.close();
```

We're not using iText at all on the client. We create a `KeyStore` object from which we obtain a `PrivateKey`. We sign the bytes returned by the `PreSign` Servlet using the `Signature` class in the `java.security` package. We connect to the `PostSign` Servlet and send the signed bytes to the server. Code sample 4.18 shows what happens with these bytes.

Code sample 4.18: the PostSign Servlet

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    resp.setContentType("application/octet-stream");
    // we get the objects we need for postsigning from the session
    HttpSession session = req.getSession(false);
    PdfPKCS7 sgn = (PdfPKCS7) session.getAttribute("sgn");
    byte[] hash = (byte[]) session.getAttribute("hash");
    Calendar cal = (Calendar) session.getAttribute("cal");
    PdfSignatureAppearance sap =
        (PdfSignatureAppearance) session.getAttribute("sap");
    ByteArrayOutputStream os =
        (ByteArrayOutputStream) session.getAttribute("baos");
    session.invalidate();
    // we read the signed bytes
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    InputStream is = req.getInputStream();
    int read;
    byte[] data = new byte[256];
    while ((read = is.read(data, 0, data.length)) != -1) {
        baos.write(data, 0, read);
    }
    // we complete the PDF signing process
    sgn.setExternalDigest(baos.toByteArray(), null, "RSA");
    byte[] encodedSig = sgn.getEncodedPKCS7(hash, cal, null,
        null, null, CryptoStandard.CMS);
    byte[] paddedSig = new byte[8192];
    System.arraycopy(encodedSig, 0, paddedSig, 0, encodedSig.length);
    PdfDictionary dic2 = new PdfDictionary();
    dic2.put(PdfName.CONTENTS, new PdfString(paddedSig).setHexWriting(true));
    try {
        sap.close(dic2);
    } catch (DocumentException e) {
        throw new IOException(e);
    }
    // we write the signed document to the HttpResponse output stream
    byte[] pdf = os.toByteArray();
    OutputStream sos = resp.getOutputStream();
    sos.write(pdf, 0, pdf.length);
    sos.flush();
    sos.close();
}
```

We start by retrieving the objects created in the `PreSign` Servlet from the `HttpSession`, we read the bytes sent by the client, and we continue doing the work that is normally done by the `MakeSignature` class. For this Proof of Concept, we send the resulting PDF to the client.

Writing this example, was fun, I can't help asking myself: *Does this make sense?* Let's find out.

4.3.4 Choosing the right architecture

Based on the examples we've made in previous sections and chapters, we can choose between four major configurations: one where the application runs entirely on the client, one where the application runs on the server, and two different client-server architectures.

The application runs entirely on the client

Suppose you're writing an offline, standalone Desktop application that allows people to sign a PDF document. In that case, you're using one if the architectures described in figure 4.12.

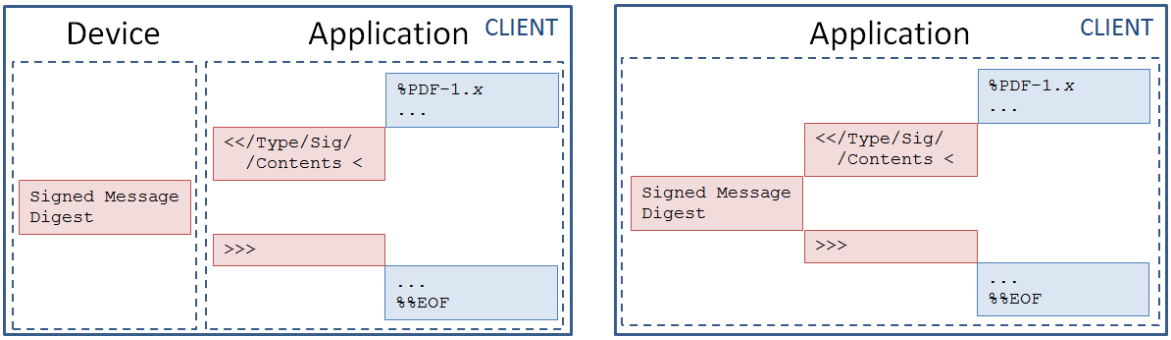

Figure 4.12: Client-side solutions

Most of the examples we've made in this white paper were standalone applications in which iText created the syntactical infrastructure to add a digital signature to a PDF document. In the schema shown to the left, the signature was created on a device, for instance a USB token or a smart card. In the situation shown to the left, the signature was created by iText using a software certificate.

What you see is what you sign (WYSIWYS); you select a document on your own machine. If you accidentally pick the wrong document, there's no harm done: it remains on your desktop, and you can easily delete documents you didn't mean to sign. The situation where you sign with a software certificate is less safe than the situation where you sign with a USB token or a smart card, because software certificates can be compromised more easily.

The application runs entirely on the server

In figure 4.13, we see the architecture we used when we built our signature server. We've used a software certificate —a self-signed certificate for testing purposes only. We've already explained the advantages of using a HSM in the previous chapter.

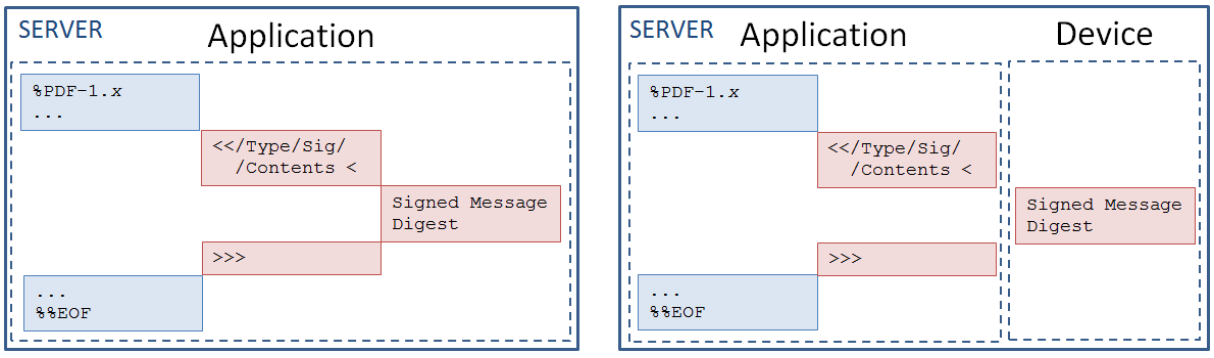

Figure 4.13: Server-side solutions

This is the architecture that will be used when documents need to be signed by companies, instead of individual people: contracts, official documents from the government, and so on. In a production environment —be it using a web interface, a cloud service, or another network solution— signature servers should be secured using different mechanisms. You'll only be able to reach these servers from a restricted number of IP addresses; you'll need credentials to access them; and so on.

Typically the documents will be archived on the server, so that the company or organization responsible for the signature can keep track of the number and the content of the documents that were signed.

Some companies use this architecture to offer signing services in the cloud. They manage your documents and sign them with their own private key. In this case, you aren't the signer. You entitle a third party to sign as your legal representative. Other companies create a key store for you on their server, and allow you to sign documents with what they claim is your signature. That's not entirely correct. Your personal signature requires a key that is private, and a private key isn't private if a third party has access to it, is it?

> **IMPORTANT:** It goes without saying that you should never write an application that sends a private key store over a network. Keep your private key in a safe place!

A company offering a signing server in the cloud will store IP-addresses, timestamps, login information, and so on. From a technical point of view, WYSIWYS isn't guaranteed. The signer can never be sure which document(s) he or she is signing because he can view a copy of the document, but the actual bits and bytes that are signed remain on the server. A Common Criteria (ISO/IEC 15408) certification can introduce sufficient trust to make such a solution work.

> **NOTE:** The international *"Common Criteria for Information Technology Security Evaluation"* standard defines a framework in which computer system users can specify their security functional and assurance requirements, vendors can then implement and make claims about the security attributes of their products, and testing laboratories can evaluate the products to determine if they actually meet the claims. This provides assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner.

Even if you trust a third party to keep your key safe from hackers and if you trust that company never to sign a document in your name without you knowing, there's always a chance that your credentials to use the online signing service get compromised. Granted, this sounds more than a tad paranoid, but when discussing security hazards, it's your duty to be paranoid.

The application runs on the client, the signature is created on the server

Figure 4.14 describes an architecture that can be useful in an intranet. For instance: an organization has a Hardware Security Module on a secured server, running an application that signs messages. This application is agnostic regarding the nature of the messages: it could be a hash of PDF byte array, a hashed XML file, a digest of some plain text… It shouldn't matter. The client could then be an application running on another server hosting an iText application.

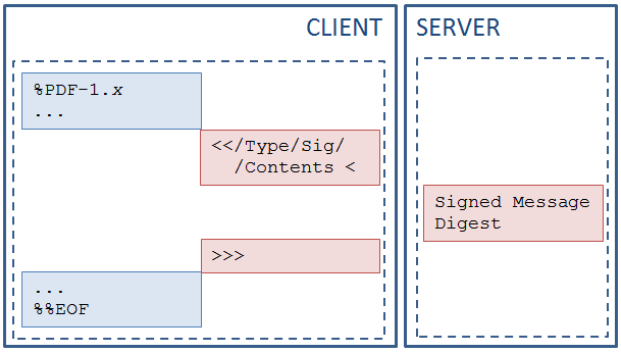

Figure 4.14: Client-server solution with iText only on the client

Another possible use case would be a situation where local dealers have a Desktop application that allows them to create contracts in PDF. These PDFs are signed using a central certificate of the mother company.

Although it's technically possible to build such a solution, this design has some disadvantages. For example: the mother company only receives digests of documents, and can't archive the content of what was signed. The company depends on the local dealers to archive the signed documents, for instance by uploading them to a central repository.

The application runs on the server, the signature is created on the client

The context of figure 4.15 could be a Document or Enterprise Content Management System (DMS, ECM…) that allows people to sign documents that reside (and remain) on the server using their private key on the client-side. This solution isn't WYSIWYS; the signer receives a hash of a document, but when he or she signs it, the signer has no idea which document the hash belongs to. The signer will have to trust the application. At least, this architecture doesn't bring the private key in jeopardy.

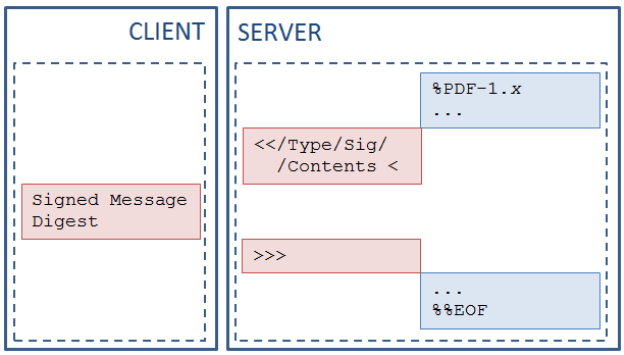

Figure 4.15: Client-Server solution with iText only on the server

It isn't trivial to build a system like the one in figure 4.15. Usually, you'll download the document from the DMS or ECM, view it, sign it, and upload it.

I've once heard of an implementation by a bank (I didn't catch its name). The bank made its customers believe they were signing documents locally using their eID. In reality, the bank had created different key stores on the server for every customer. The eID was used only for authentication, and authenticated customers signed documents using their key store on the server.

Customers who weren't tech-savvy believed that they had signed the document using their eID, and they didn't understand why their documents always indicated that the identity of the signer couldn't

be verified. Their documents couldn't be validated with the Belgian Root CA Certificate because another key store was used.

This example demonstrates that you should choose the architecture of your signing solution with great care.

4.4 Summary

We started this chapter with the PKCS#11-way to sign PDF documents using a signature that is created on hardware. We signed documents using a Hardware Security Module, using a USB token, and using a smart card. We learned about the eID, an identity card in the form of a smart card. First we signed documents with a test card using MSCAPI and PKCS#11. Then we used the `javax.smartcardio` API to retrieve information from the card, and to sign messages on secure smart card readers.

After having written many small standalone applications, we wrote a Servlet as a Proof of Concept for an online signing service. We also created some experimental client-server test cases. Finally, we looked at an overview of different architectures. The conclusions that were made for some of these architectures were at times controversial; many companies have built a business using a pragmatic approach to tackle issues that are difficult to solve in a cloud environment. Please understand that I wrote that final section in 'paranoid'-mode. A less optimal solution can still be valuable in many situations, but you should be aware that there's always a tradeoff.

In the next chapter, we'll learn how to verify signatures automatically, without having to open the document in Adobe Reader. We'll also find out how we can extend the life of a signed document.

5. Validation of signed documents

When you receive a signed PDF, you can open it in Adobe Reader and check if the signature is valid by opening the signature panel, or —in the case of a visible signature— by clicking the signature's widget annotation. That's easy. It's a different story when you receive thousands of PDFs. In that case, opening the documents manually one by one isn't an option; you'll want to test the integrity of the documents and the validity of the signatures in an automated process.

5.1 Checking a document's integrity

When we discussed the goal of signing documents using a digital signature, we listed three reasons: to ensure the integrity of the document, to get assurance about the identity of the signer, to make sure the signer can't deny he has signed the document. Let's start by looking at the integrity of the document.

5.1.1 Listing the signatures in a document

Code sample 5.1 shows a method that accepts the path to a PDF file, and that loops over the signatures that are available in that document. This is done using the `getSignatureNames()` method in the `AcroFields` class.

Code sample 5.1: Listing the signatures in a document

```
public void verifySignatures(String path)
    throws IOException, GeneralSecurityException {
    System.out.println(path);
    PdfReader reader = new PdfReader(path);
    AcroFields fields = reader.getAcroFields();
    ArrayList<String> names = fields.getSignatureNames();
        for (String name : names) {
            System.out.println("===== " + name + " =====");
            verifySignature(fields, name);
        }
    System.out.println();
}
```

Let's execute this code on some of the PDFs we've created in chapter 2, more specifically on:

- *The result of code sample 2.14*— how NOT to add an annotation to a signed document,
- *The PDF shown in figure 2.31*— a document signed by four different people in a workflow,
- *The PDF shown in figure 2.33*— the document from figure 2.31, broken by a fifth person.

What does iText tell us about the integrity of these documents?

5.1.2 Checking the integrity of a revision

When you look at figure 2.31, you see that the signature panel lists different revisions. The document was certified by Alice Specimen. That's revision one. The next line shows "Rev. 2: Signed by Bob Specimen"; revision 3 is signed by Carol; revision 4 is signed by Dave. Code sample 5.2 shows how to find out more about the integrity of each revision.

Code sample 5.2: Checking the integrity of a revision based on a digital signature

```
public PdfPKCS7 verifySignature(AcroFields fields, String name)
    throws GeneralSecurityException, IOException {
    System.out.println("Signature covers whole document: "
        + fields.signatureCoversWholeDocument(name));
```

```
    System.out.println("Document revision: " + fields.getRevision(name)
        + " of " + fields.getTotalRevisions());
    PdfPKCS7 pkcs7 = fields.verifySignature(name);
    System.out.println("Integrity check OK? " + pkcs7.verify());
    return pkcs7;
}
```

The `signatureCoversWholeDocument()` method is self-explanatory. The `getRevision()` method tells you which revision is covered. The `getTotalRevisions()` method will typically give you the number of signatures in the document, or that number plus one if changes were applied after the last time the document was signed.

We create a `PdfPKCS7` object to verify the integrity of the document with the `verify()` method. We'll use this object later on, to find more information about the signature as well as the signer.

Verifying an invalid signature

When we execute this code on a broken PDF, we get the following result:

```
results/chapter2/hello_level_1_annotated_wrong.pdf
===== sig =====
Signature covers whole document: false
Document revision: 1 of 2
Integrity check OK? false
```

There was only one signature in this document. Hence, there's only one revision, but iText tells you there are two because an annotation was added after the document was signed by Bruno Specimen. The total number of revisions returned by iText may be misleading, but it was a choice we made to make sure you notice that something is wrong. The annotation was added incorrectly, breaking the integrity of the revision signed by Bruno Specimen. As a result, the *verify()* method returns `false`.

Verifying a series of valid signatures

When we execute the code on the PDF signed by four people, we get:

```
results/chapter2/step_4_signed_by_alice_bob_carol_and_dave.pdf
===== sig1 =====
Signature covers whole document: false
Document revision: 1 of 4
Integrity check OK? true
===== sig2 =====
Signature covers whole document: false
Document revision: 2 of 4
Integrity check OK? true
===== sig3 =====
Signature covers whole document: false
Document revision: 3 of 4
Integrity check OK? true
===== sig4 =====
Signature covers whole document: true
Document revision: 4 of 4
Integrity check OK? true
```

The result of the `signatureCoversWholeDocument()` method is `false` until we reach the final signature. We see four revisions, and the integrity of each revision is OK.

Verifying signatures that break DMP settings

However, when we verify the PDF that was invalidated by Chuck after it was signed by Dave, we get:

```
results/chapter2/step_6_signed_by_dave_broken_by_chuck.pdf
===== sig1 =====
Signature covers whole document: false
Document revision: 1 of 5
Integrity check OK? true
===== sig2 =====
Signature covers whole document: false
Document revision: 2 of 5
Integrity check OK? true
===== sig3 =====
Signature covers whole document: false
Document revision: 3 of 5
Integrity check OK? true
===== sig4 =====
Signature covers whole document: false
Document revision: 4 of 5
Integrity check OK? true
```

Revisions 1 to 4 all pass the integrity check, but we know from figure 2.33 that two signatures are invalidated. The signatures aren't broken, but they are invalidated by content of *later* revisions that don't adhere to limitations imposed in *earlier* revisions.

Limitation of the current iText version

iText can't verify the 'integrity' of revision 5 because there's no signature involved. There's no hash in revision 5 that can be decrypted and compared with the PDF bytes. iText can detect broken signatures, but doesn't detect later invalidation (yet).

What we'd need to do, is to find out all the permissions that are involved, and then look at the changes that were applied after a specific revision. The former is easy to achieve; we'll do that in the next section. The latter is more difficult; we'd need to check which content is added in-between revisions and after the final revision. All content other than form fields and annotations invalidate the signature of the previous revision. If form fields or annotations are added, these actions need to be compared with the MDP settings of the previous signatures. This isn't trivial. We've put it on our development roadmap, but we can't give an ETA because other projects have a higher priority for the time being. Now let's see what more information we can get from a signature.

5.2 Retrieving information from a signature

Using the `verifySignature()` method from code sample 5.2, we checked the integrity of signed documents, but we didn't retrieve much information about the signatures.

5.2.1 Overview of the information stored in a signature field and dictionary

Code sample 5.3 uses this method, but also returns useful info about the signature field, the signature field's widget annotation, the signature dictionary, the signature, and the signer.

Code sample 5.3: Retrieving information from a signature

```
public SignaturePermissions inspectSignature(
    AcroFields fields, String name, SignaturePermissions perms)
    throws GeneralSecurityException, IOException {
    List<FieldPosition> fps = fields.getFieldPositions(name);
```

```
    if (fps != null && fps.size() > 0) {
        FieldPosition fp = fps.get(0);
        Rectangle pos = fp.position;
        if (pos.getWidth() == 0 || pos.getHeight() == 0) {
            System.out.println("Invisible signature");
        }
        else {
            System.out.println(String.format(
                "Field on page %s; llx: %s, lly: %s, urx: %s; ury: %s", fp.page,
                pos.getLeft(), pos.getBottom(), pos.getRight(), pos.getTop()));
        }
    }
    PdfPKCS7 pkcs7 = super.verifySignature(fields, name);
    System.out.println("Digest algorithm: " + pkcs7.getHashAlgorithm());
    System.out.println("Encryption algorithm: " + pkcs7.getEncryptionAlgorithm());
    System.out.println("Filter subtype: " + pkcs7.getFilterSubtype());
    X509Certificate cert = (X509Certificate) pkcs7.getSigningCertificate();
    System.out.println("Name of the signer: " +
        CertificateInfo.getSubjectFields(cert).getField("CN"));
    if (pkcs7.getSignName() != null)
        System.out.println("Alternative name of the signer: "
            + pkcs7.getSignName());
    SimpleDateFormat date_format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SS");
    System.out.println("Signed on: " +
        date_format.format(pkcs7.getSignDate().getTime()));
    if (pkcs7.getTimeStampDate() != null) {
        System.out.println("TimeStamp: " +
            date_format.format(pkcs7.getTimeStampDate().getTime()));
        TimeStampToken ts = pkcs7.getTimeStampToken();
        System.out.println("TimeStamp service: " + ts.getTimeStampInfo().getTsa());
        System.out.println("TimeStamp verified? "+ pkcs7.verifyTimestampImprint());
    }
    System.out.println("Location: " + pkcs7.getLocation());
    System.out.println("Reason: " + pkcs7.getReason());
    PdfDictionary sigDict = fields.getSignatureDictionary(name);
    PdfString contact = sigDict.getAsString(PdfName.CONTACTINFO);
    if (contact != null)
        System.out.println("Contact info: " + contact);
    perms = new SignaturePermissions(sigDict, perms);
    System.out.println("Signature type: " +
        (perms.isCertification() ? "certification" : "approval"));
    System.out.println("Filling out fields allowed: " +
        perms.isFillInAllowed());
    System.out.println("Adding annotations allowed: " +
        perms.isAnnotationsAllowed());
    for (FieldLock lock : perms.getFieldLocks()) {
        System.out.println("Lock: " + lock.toString());
    }
    return perms;
}
```

We start by checking the visibility of the signature annotation. A signature is represented using a form field. You can use the `getFieldPositions()` method to get the page number that contains the widget annotation and the coordinates on that page representing the field —assuming there is such a page. As defined in the PDF specification, an invisible signature field will have a widget annotation with a zero width or height. A signature field will also be invisible if the Hidden of NoView flag of the widget annotation are set.

Next, we use the `PdfPKCS7` object and two of its methods —`getHashAlgorithm()` and `getEncryptionAlgorithm()`— to find out how the message digest of the PDF bytes was created and how these bytes and additional attributes were signed. The `getFilterSubtype()` will tell you how the signed bytes are stored in the PDF (see section 2.1.1).

We need the signing certificate if we want to know the name of the signer. We cast the result of the `getSigningCertificate()` method to an `X509Certificate` object, and we can use the `CertificateInfo` helper class to get more info, for instance: the subject fields. In this example, we only retrieve the Common Name ("`CN`"). If you look at code sample 1.9 showing the contents of a public certificate, you see that we could also ask for information such as the Organization ("`O`"), the Organizational Unit ("`OU`"), and so on. We'll do much more with certificates in the next section.

> **NOTE:** The `getSignName()` usually doesn't return any information. That specific field in the signature dictionary should only be used when somebody wants to sign a document and his name can't be retrieved from the certificate.

We can get the time on the clock of the computer of the signer using the `getSignDate()` method. If a timestamp was applied, we need the `getTimeStampDate()` method. You can check if the timestamp covers the document with the `verifyTimestampImprint()` method. As we're using BouncyCastle to verify signatures, we can also retrieve a `TimeStampToken` object. This object gives you access to the `TimeStampInfo` which reveals plenty of information on the TSA. This is the object you need if you want to check the TSA's certificate.

The `PdfPKCS7` object has methods for the most common entries in the signature dictionary, such as `getLocation()` and `getReason()`. If you want less common entries, such as the contact info, you should use the `getSignatureDictionary()` method and get the properties by name.

We also need the signature dictionary if we want to create a `SignaturePermissions` object. Every new signature can add more restrictions to a document, but it can't take away previous restrictions. That explains why you need to pass the `SignaturePermissions` instance of the previous signature, or `null` if there was none. We can use this class to check whether the signature was meant for certification, or for approval. We can also check the MDP settings: is form filling and adding extra signatures allowed? Is adding annotations allowed? Finally, we can also list the locks that are set on the fields.

5.2.2 Inspecting signatures

If we return to the PDF from figure 2.31, and inspect the signatures, we get the following output:

```
results/chapter2/step_4_signed_by_alice_bob_carol_and_dave.pdf
===== sig1 =====
Field on page 1; llx: 36.0, lly: 740.0, urx: 559.0; ury: 790.0
Signature covers whole document: false
Document revision: 1 of 4
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Alice Specimen
Signed on: 2012-09-17 18:03:44.00
```

```
Location:
Reason:
Contact info:
Signature type: certification
Filling out fields allowed: true
Adding annotations allowed: false
===== sig2 =====
Field on page 1; llx: 36.0, lly: 654.0, urx: 559.0; ury: 704.0
Signature covers whole document: false
Document revision: 2 of 4
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Bob Specimen
Signed on: 2012-09-17 18:03:44.00
Location:
Reason:
Contact info:
Signature type: approval
Filling out fields allowed: true
Adding annotations allowed: false
Lock: /Include[sig1, approved_bob, sig2]
===== sig3 =====
Field on page 1; llx: 36.0, lly: 568.0, urx: 559.0; ury: 618.0
Signature covers whole document: false
Document revision: 3 of 4
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Carol Specimen
Signed on: 2012-09-17 18:03:44.00
Location:
Reason:
Contact info:
Signature type: approval
Filling out fields allowed: true
Adding annotations allowed: false
Lock: /Include[sig1, approved_bob, sig2]
Lock: /Exclude[approved_dave, sig4]
===== sig4 =====
Field on page 1; llx: 36.0, lly: 482.0, urx: 559.0; ury: 532.0
Signature covers whole document: true
Document revision: 4 of 4
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Dave Specimen
Signed on: 2012-09-17 18:03:44.00
Location:
Reason:
Contact info:
Signature type: approval
Filling out fields allowed: false
Adding annotations allowed: false
Lock: /Include[sig1, approved_bob, sig2]
```

```
Lock: /Exclude[approved_dave, sig4]
Lock: /All
```

All revisions were signed using SHA256 with RSA, and the `/adbe.pkcs7.detached` sub filter. We didn't add any location, reason, or contact info, but we gradually added restrictions. The first signature was a certification signature that didn't allow adding annotations. The subsequent signatures were approval signatures for which a field lock dictionary was defined. Approval signatures allow adding annotations, but as this was already restricted for the first revision, the restriction remains valid for the revisions that follow. The document is completely locked with the fourth revision.

In the next example, we're inspecting a document that was timestamped:

```
results/chapter3/hello_token.pdf
===== sig =====
Field on page 1; llx: 36.0, lly: 748.0, urx: 144.0; ury: 780.0
Signature covers whole document: true
Document revision: 1 of 1
Integrity check OK? true
Digest algorithm: SHA384
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Bruno Lowagie
Signed on: 2012-09-17 17:34:43.00
TimeStamp: 2012-09-17 17:34:54.392
TimeStamp service: 4:
  OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,
  CN=SEIKO Timestamp Service. Advanced AW02-001
Timestamp verified? true
Location: Ghent
Reason: Test
Contact info:
Signature type: approval
Filling out fields allowed: true
Adding annotations allowed: true
```

Observe that the time of the TSA is more accurate than the computer time. You can even get the accuracy of the timestamp from the `TimeStampInfo` object. There's a lot more to explore than what is revealed in this simple example.

Finally, this is the output of the example we made using code sample 2.12:

```
results/chapter2/field_metadata.pdf
===== Signature1 =====
Field on page 1; llx: 41.3671, lly: 713.71, urx: 237.353; ury: 781.258
Signature covers whole document: true
Document revision: 1 of 1
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Bruno Specimen
Alternative name of the signer: Bruno L. Specimen
Signed on: 2012-08-05 00:00:00.00
Location: Ghent
Reason: Test metadata
```

```
Contact info: 555 123 456
Signature type: approval
Filling out fields allowed: true
Adding annotations allowed: true
```

In section 2.4.3, we added metadata such as a phone number and an alternative name for Bruno Specimen. With code sample 5.3, we are able to retrieve that data.

Now that we have more information about the signature, we already have an idea about who signed the document, but we need to verify the certificates with the CA who issued them to find out if the certificate were valid at the moment of signing. That's what the next section is about.

5.3 Validating the certificates of a signature

In section 2.2.2, we added public certificates from individual people to the Trusted Identities. In 3.1.1, we added the root certificate of a CA. In section 3.4.1, we learned that a lot of software and hardware ships with root stores. In Java, you'll typically use the `cacerts` file as your root store.

5.3.1 Creating your own root store

In code sample 5.4, we create our own root store, and we'll add three certificates: the Adobe Root CA certificate, the CA Cert root certificate, and the public certificate for Bruno Specimen we created in code sample 2.3.

Code sample 5.4: Creating a Keystore containing public certificates

```
KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
ks.load(null, null);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
ks.setCertificateEntry("cacert",
    cf.generateCertificate(new FileInputStream(CACERT)));
ks.setCertificateEntry("adobe",
    cf.generateCertificate(new FileInputStream(ADOBE)));
ks.setCertificateEntry("bruno",
    cf.generateCertificate(new FileInputStream(BRUNO)));
```

We'll use this root store to verify the certificates in four PDFs we've signed before.

- A PDF signed with a CAcert certificate,
- A PDF signed with a token from GlobalSign (CDS)
- A PDF signed with a self-signed certificate,
- A PDF signed with an eID for testing purposes

Normally, we should be able to verify the certificates of the first three PDFs against the root store we created. As we didn't add any of the certificates in the certificate chain of the PDF signed with the smart card, the verification should fail for the fourth PDF.

Let's give it a try.

5.3.2 Verifying a signature against a key store

In code sample 5.5, we reuse the `PdfPKCS7` object and we get the certificate chain using the `getSignCertificateChain()` method. We also use the `getSignDate()` method to get the date and time if present in the signature dictionary.

Now we can use the `verifyCertificates()` method to check if the certificate chain present in the PDF can be verified against the key store we created.

Code sample 5.5: Verifying certificates

```
Certificate[] certs = pkcs7.getSignCertificateChain();
Calendar cal = pkcs7.getSignDate();
List<VerificationException> errors =
    CertificateVerification.verifyCertificates(certs, ks, cal);
if (errors.size() == 0)
    System.out.println("Certificates verified against the KeyStore");
else
    System.out.println(errors);
```

If the `errors` list is empty, the code shows:

```
Certificates verified against the KeyStore
```

This is indeed the case for the first three PDFs. We trusted three certificates by adding them to the key store `ks`, and we trust all the PDFs that have one of these certificates in their certificate chain.

This isn't the case for the PDF signed with the test eID. That's why we get the following message:

```
[com.itextpdf.text.pdf.security.VerificationException:
    Certificate C=BE,CN=eID test Root CA failed:
    Cannot be verified against the KeyStore or the certificate chain]
```

The square brakets indicate that we're printing a `List` object to the `System.out`. There's one element in the `List`: a `VerificationException` indicating that the certificate of the root certificate of my test eID card can't be verified. To verify the PDF signed with the eID, we need to add the certificate of the `eID test Root CA` to our key store, but maybe there are alternatives.

Let's start by inspecting the certificates and by extracting some extra information regarding the validity of a certificate on the signing date or even on the current date.

5.3.3 Extracting information from certificates

Code sample 5.6 shows a method we can use to find out if a certificate was valid on the `signDate`, and if it's still valid today. It also shows the Distinguished Name ("`DN`") of the issuer of the certificate and its owner.

Code Sample 5.6: Checking if a certificate was valid on a specific day

```
public void showCertificateInfo(X509Certificate cert, Date signDate) {
    System.out.println("Issuer: " + cert.getIssuerDN());
    System.out.println("Subject: " + cert.getSubjectDN());
    SimpleDateFormat date_format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SS");
    System.out.println("Valid from: " + date_format.format(cert.getNotBefore()));
    System.out.println("Valid to: " + date_format.format(cert.getNotAfter()));
    try {
        cert.checkValidity(signDate);
        System.out.println("The certificate was valid at the time of signing.");
    } catch (CertificateExpiredException e) {
        System.out.println("The certificate was expired at the time of signing.");
    } catch (CertificateNotYetValidException e) {
        System.out.println(
            "The certificate wasn't valid yet at the time of signing.");
    }
```

```
    try {
        cert.checkValidity();
        System.out.println("The certificate is still valid.");
    } catch (CertificateExpiredException e) {
        System.out.println("The certificate has expired.");
    } catch (CertificateNotYetValidException e) {
        System.out.println("The certificate isn't valid yet.");
    }
}
```

This is an example of the output we get when we check the file signed with the test eID:

```
=== Certificate 0 ===
Issuer: C=BE,CN=eID test Citizen CA
Subject: C=BE,CN=Alice SPECIMEN (Signature),SURNAME=SPECIMEN,GIVENNAME=Alice
Geldigekaart,SERIALNUMBER=71715100070
Valid from: 2012-01-30 10:20:51.00
Valid to: 2017-01-30 10:20:51.00
The certificate was valid at the time of signing.
The certificate is still valid.
=== Certificate 1 ===
Issuer: C=BE,CN=eID test Root CA
Subject: C=BE,CN=eID test Citizen CA
Valid from: 2009-02-25 18:26:09.00
Valid to: 2059-02-25 18:26:03.00
The certificate was valid at the time of signing.
The certificate is still valid.
=== Certificate 2 ===
Issuer: C=BE,CN=eID test Root CA
Subject: C=BE,CN=eID test Root CA
Valid from: 2009-02-25 18:26:04.00
Valid to: 2059-02-25 18:26:04.00
The certificate was valid at the time of signing.
The certificate is still valid.
```

Alice's certificate is valid until 2017. The `eID test Citizen` and `test Root` certificate are valid until 2059. This is just an example of some basic information we can extract from a certificate.

In chapter 3, we've also retrieved URLs for CRLs and OCSP servers from the certificate. Instead of adding the `eID test Root` to a key store with trusted anchors, we could use a CRL or an OCSP response to check the validity of a certificate.

5.3.4 Checking if the certificate was revoked using CRLs and OCSP

In code sample 5.7, we take the signing certificate and the certificate of the issuer of that certificate (or `null` if it was self-signed). We'll use these certificates as parameters for the `checkRevocation()` method.

Code Sample 5.7: Checking the revocation status of the signing certificate

```
X509Certificate signCert = (X509Certificate)certs[0];
X509Certificate issuerCert = (certs.length > 1 ? (X509Certificate)certs[1] : null);
System.out.println(
    "=== Checking validity of the document at the time of signing ===");
checkRevocation(pkcs7, signCert, issuerCert, cal.getTime());
System.out.println(
    "=== Checking validity of the document today ===");
checkRevocation(pkcs7, signCert, issuerCert, new Date());
```

Code sample 5.8 shows that the `checkRevocation()` method first tries to get the OCSP response stored in the document with the `getOcsp()` method. If this method returns a `BasicOCSPResp` object that isn't `null`, we add it to a `List`. With this list, we create an `OCSPVerifier`. This implementation of the `CertificateVerifier` class will check if the OCSP responses in the list were valid for the certificate on a specific date —in code sample 5.7 on the signing date or today. If not, the verifier will try to look for a valid OCSP response online. The `verify()` method return a List of `VerificationOK` objects. If that list is empty, we can't verify using OCSP, and we need to look for CRLs.

With the `getCRL()` method, we get the CRLs that were stored in the signature. We cast the `CRL` objects to `X509CRL` objects, and we pass the list to a `CRLVerifier`. This verifier will check if the CRLs in the list were valid on a specific date. If the certificate was revoked an exception will be thrown. If no valid CRL was found, the verifier will try to fetch a CRL online. If we still didn't get any `VerificationOK` objects, the certificate couldn't be verified. Otherwise, we get a list of the checks that made us accept the certificate.

Code Sample 5.8: Checking the validity of a certificate using OCSP

```
public void checkRevocation(PdfPKCS7 pkcs7,
    X509Certificate signCert, X509Certificate issuerCert, Date date)
    throws GeneralSecurityException, IOException {
    List<BasicOCSPResp> ocsps = new ArrayList<BasicOCSPResp>();
    if (pkcs7.getOcsp() != null)
        ocsps.add(pkcs7.getOcsp());
    OCSPVerifier ocspVerifier = new OCSPVerifier(null, ocsps);
    List<VerificationOK> verification =
        ocspVerifier.verify(signCert, issuerCert, date);
    if (verification.size() == 0) {
        List<X509CRL> crls = new ArrayList<X509CRL>();
        if (pkcs7.getCRLs() != null) {
            for (CRL crl : pkcs7.getCRLs())
            crls.add((X509CRL)crl);
        }
        CRLVerifier crlVerifier = new CRLVerifier(null, crls);
        verification.addAll(crlVerifier.verify(signCert, issuerCert, date));
    }
    if (verification.size() == 0) {
        System.out.println("The signing certificate couldn't be verified");
    }
    else {
        for (VerificationOK v : verification)
            System.out.println(v);
    }
}
```

The first parameter in the `OCSPVerifier` and `CRLVerifier` constructor is always null in this example. You can chain both verifiers like this: `new CRLVerifier(ocspVerifier, crls);`

Now, when you use the `verify()` method on the `CRLVerifier`, it will first look for CRLs, and then call the `verify()` method of the `OCSPVerifier`, even if a CRL was found. If there was a valid Certificate Revocation List as well as a valid OCSP response, two different `VerificationOK` instances will be returned.

Let's look at the output when running this method for the PDF file signed with my CAcert certificate, for which I included an OCSP response in the signature:

```
=== Checking validity of the document at the time of signing ===
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 1
CN=Bruno Lowagie,E=bruno@___________ verified with
    com.itextpdf.text.pdf.security.OCSPVerifier: Valid OCSPs Found: 1
```

When we check on the date the document was signed, we added an OCSP response. This response was found by the `getOcsp()` method, and it was valid on the signing date.

```
=== Checking validity of the document today ===
c.i.t.p.s.OCSPVerifier INFO  OCSP no longer valid: Sun Oct 07 16:27:16 CEST 2012
after Thu Sep 13 10:18:04 CEST 2012
c.i.t.p.s.OcspClientBouncyCastle INFO  Getting OCSP from http://ocsp.cacert.org
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 1
CN=Bruno Lowagie,E=bruno@___________ verified with
com.itextpdf.text.pdf.security.OCSPVerifier: Valid OCSPs Found: 1 (online)
```

When we do the same check on October 7th, we discover that the OCSP response is no longer valid: it expired on September 13th. The `OCSPVerifier` will ask the `OcspClientBouncyCastle` to get a new OCSPResponse from `http://ocsp.cacert.org` and the `VerificationOK` object will tell you that one valid OCSP response was found online.

Now let's look at the document signed with an USB token:

```
=== Checking validity of the document at the time of signing ===
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 0
c.i.t.p.s.CRLVerifier INFO  Valid CRLs found: 1
C=BE,L=Sint-Amandsberg,E=bruno@___________,O=iText Software BVBA,CN=Bruno Lowagie
    verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
=== Checking validity of the document today ===
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 0
c.i.t.p.s.CRLVerifier INFO  Getting CRL from
    http://crl.globalsign.com/gs/gssha2adobe.crl
c.i.t.p.s.CRLVerifier INFO  Valid CRLs found: 1
C=BE,L=Sint-Amandsberg,E=bruno@___________,O=iText Software BVBA,CN=Bruno Lowagie
    verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
(online)
```

In this case, there was no OCSP service available, so we didn't embed an OCSP response. Instead, we embedded a CRL. This CRL was valid in September, but it was no longer valid in October. We get a new CRL from `http://crl.globalsign.com/gs/gssha2adobe.crl`

These are cases were we were able to verify the signatures. You won't get any positive results when verifying the documents we signed in chapter 2, for instance with Bruno Specimen's self-signed key:

```
=== Checking validity of the document at the time of signing ===
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 0
c.i.t.p.s.CRLVerifier INFO  Valid CRLs found: 0
The signing certificate couldn't be verified
=== Checking validity of the document today ===
c.i.t.p.s.OCSPVerifier INFO  Valid OCSPs found: 0
c.i.t.p.s.CRLVerifier INFO  Valid CRLs found: 0
The signing certificate couldn't be verified
```

This demonstrates the need for a Certificate Authority. It also demonstrates the need for the best practices described in chapter 3. Suppose that we receive a document without any revocation information. We'll be able to check the validity as long as the certificate hasn't been revoked and as long as it hasn't expired.

But what if we want to be able to keep the document valid long after those dates? Suppose that we want to make sure that the signature can be verified long after the certificate has expired. Part 4 of the PAdES standard and ISO-32000-2 provide us with a solution.

5.4 PAdES-4: Long-Term Validation (LTV)

If you receive a signed document, and the signatures need to be verifiable long after the signing certificate has expired, the original validation data may no longer be available, or there may be uncertainty as to what validation data was used when the document was first verified. Or maybe you've received a document that was signed offline, in which case no recent revocation information could be added. You may want to add the missing information before archiving the document.

Furthermore, the cryptographic algorithms used to apply the signature in the past may no longer conform to industry standards in the future. In the past, we used to accept SHA-1 as digest algorithm and RSA with key length 1024. Nowadays, we need to use at least SHA-256 and a key length of at least 2048, but there is no guarantee that this will still be sufficient in the future.

Starting with PDF 2.0 (ISO-32000-2), we can solve these issues by introducing a Document Security Store (DSS) and a Document-level Timestamp.

5.4.1 Adding a Document Security Store (DSS) and a Document-Level Timestamp

If we have a document that is signed as is done in figure 5.1, we can't verify the certificate off-line because there's no certificate revocation information available in the PDF.

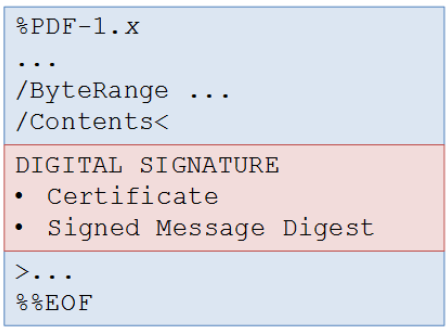

Figure 5.1: A signed PDF, no CRL and no OCSP

When we receive such a document, we can extend it by adding a DSS and a timestamp as shown in figure 5.2. In the DSS, we can store Validation Related Information (VRI). The DSS contains references to certificates, and we can add references to OCSP responses and CRLs that can be used to re-verify the certificates. To make sure that the DSS won't be altered, we sign the document including an authoritative timestamp. We've already mentioned the sub filter used for this type of signature in section 2.1.1: `/ETSI.RFC3161`.

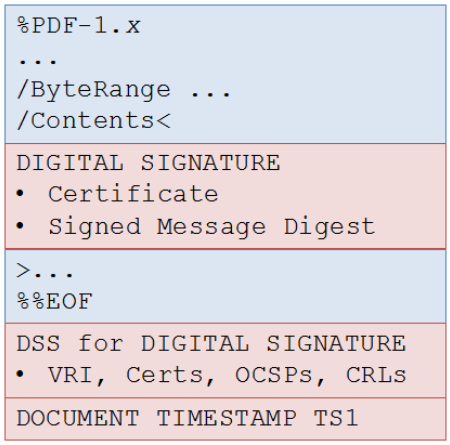

Figure 5.2: A signed PDF, with a DSS and a Document-Level Timestamp

The timestamp is itself signed, and so it's possible for the timestamp's own validation data to expire. If we want to extend the life of the signed document beyond the expiration date of the certificate used by the TSA, we need to add another DSS and document-level timestamp containing a CRL or OCSP response for the most recent TSA certificate as shown in figure 5.3.

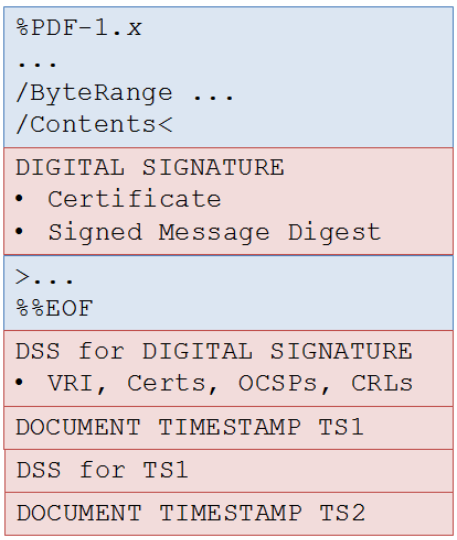

Figure 5.3: A signed PDF with different DSSs and Timestamps

Let's take a look at code sample 5.9 to see how this is done using iText code.

Code sample 5.9: Adding a DSS and a Document-level Timestamp

```
public void addLtv(String src, String dest, OcspClient ocsp,
    CrlClient crl, TSAClient tsa)
    throws IOException, DocumentException, GeneralSecurityException {
    PdfReader r = new PdfReader(src);
    FileOutputStream fos = new FileOutputStream(dest);
    PdfStamper stp = PdfStamper.createSignature(r, fos, '\0', null, true);
    LtvVerification v = stp.getLtvVerification();
    AcroFields fields = stp.getAcroFields();
    List<String> names = fields.getSignatureNames();
    String sigName = names.get(names.size() - 1);
    PdfPKCS7 pkcs7 = fields.verifySignature(sigName);
    if (pkcs7.isTsp())
        v.addVerification(sigName, ocsp, crl,
```

```
            LtvVerification.CertificateOption.SIGNING_CERTIFICATE,
            LtvVerification.Level.OCSP_CRL,
            LtvVerification.CertificateInclusion.NO);
    else {
        for (String name : names) {
            v.addVerification(name, ocsp, crl,
                LtvVerification.CertificateOption.WHOLE_CHAIN,
                LtvVerification.Level.OCSP_CRL,
                LtvVerification.CertificateInclusion.NO);
        }
    }
    PdfSignatureAppearance sap = stp.getSignatureAppearance();
    LtvTimestamp.timestamp(sap, tsa, null);
}
```

We recognize the parameters passed to this method from the previous chapters. We can use an instance of `OcspClientBouncyCastle`, an instance of `CrlClientOnline`, and an instance of `TSAClientBouncyCastle`.

Just like when creating a normal signature, we need a `PdfStamper` instance, but instead of getting a `PdfSignatureAppearance`, we first get an `LtvVerification` object using the `getLtvVerification()` method. Now we can add verification info for every signature in the document.

> **NOTE:** You can add a DSS days, weeks or ever years after the signed document has been created, but you should always add it before the revocation information is no longer available or valid. If you need a signed document to have a life-span of 75 years, you'll need to keep track of the expiration dates of the timestamps, and add a new DSS and time-stamp on a regular basis.

We look at the final signature that was added to the document. The `isTsp()` method will return true if this signature a document-level timestamp. In this case, we'll add a DSS with certificate revocation information for that specific signature, assuming that the certificate revocation information is already present in a previous DSS.

If the final signature isn't a document-level timestamp, we add certificate revocation information for all the signatures in the document.

5.4.2 Selecting which verification information needs to be added

You can choose between different options when creating the DSS:

- *SIGNING_CERTIFICATE*— just add the extra verification info for the signing certificate.
- *WHOLE_CHAIN*— add the extra verification for every certificate in the chain.

Regarding the choice between CRLs or OCSP, you have the following options:

- *OCSP*—include only OCSP responses (if available).
- *CRL*—include only Certificate Revocation Lists only (if available).
- *OCSP_CRL*—include both OCSP responses and CRLs (if available).
- *OCSP_OPTIONAL_CRL*—include only OCSP responses if available; otherwise try adding CRLs (if available).

Finally, you can also decide whether or not you want to include the certificates in the DSS:

- *YES*— the certificates will be added to the DSS and VRI dictionaries.
- *NO*— the certificates won't be added to the DSS and VRI dictionaries.

Once the content of the DSS is defined, we create a `PdfSignatureAppearance`, and we create the document-level timestamp with the `timestamp()` method. In our example, we pass `null` for the name of the signature, and iText will choose a name for you. In figure 5.4, two PDFs are shown. To the left, you see a PDF with one DSS and one document-level timestamp; to the right an extra DSS and an extra document-level timestamp were added.

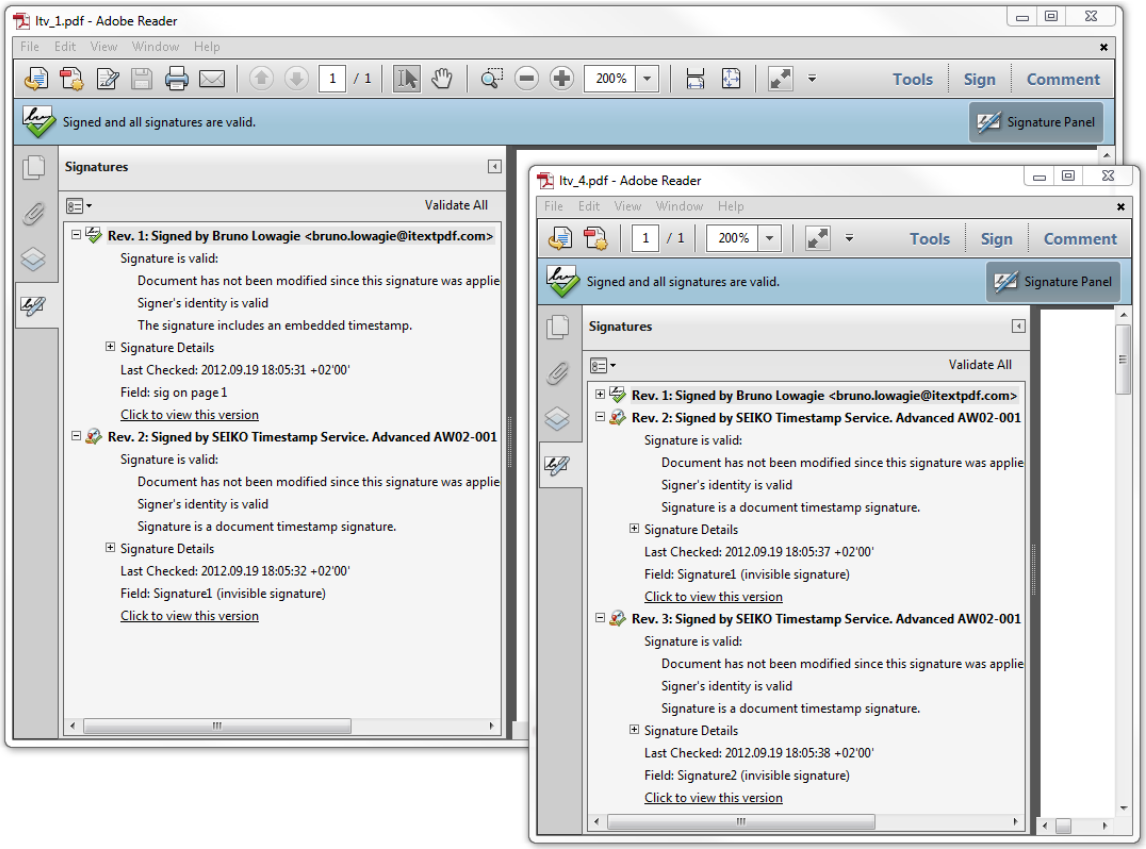

Figure 5.4: Sample documents with LTV information

As you can see, the difference between the two document-level timestamps is only a second. This doesn't make much sense. Normally, you'll put the documents in a Document Management System, and add an extra DSS and document-level timestamp either when the final timestamp in the document is about to expire, or when you want to add a signature that uses stronger hashing and encryption algorithms.

> **NOTE:** According to the ETSI standard, LTV can only be applied to documents that are signed using signatures described in PAdES-1 (CMS), PAdES-2 (CAdES), and PAdES-5 (XAdES). You shouldn't expect this to work with signatures that use the sub filters `/adbe.pkcs7.sha1` or `/adbe.x509.rsa_sha1` (PKCS#1).

Let me repeat that it's important to add a new DSS and timestamp before the certificate of the final revision has expired. Gaps aren't allowed! You'll need a good DMS to manage this.

5.4.3 Checking the integrity of documents with a Document-Level Timestamp

If we check the integrity of the different signatures and extract information from the certificates using the methods we've seen in section 5.1 and 5.2, we get the following output:

```
results/chapter5/ltv_1.pdf
===== sig =====
Signature covers whole document: false
Document revision: 1 of 2
Integrity check OK? true
Digest algorithm: SHA384
Encryption algorithm: RSA
Filter subtype: /adbe.pkcs7.detached
Name of the signer: Bruno Lowagie
===== Signature1 =====
Signature covers whole document: true
Document revision: 2 of 2
Integrity check OK? true
Digest algorithm: SHA256
Encryption algorithm: RSA
Filter subtype: /ETSI.RFC3161
Name of the signer: SEIKO Timestamp Service. Advanced AW02-001
```

Note that we created the TSAClient for the document-level timestamp like this:

```
TSAClient tsa = new TSAClientBouncyCastle(
    tsaUrl, tsaUser, tsaPass, 6500, "SHA512");
```

In this code snippet, we chose an estimated size of 6500 bytes for the signature, and we chose a digest algorithm that is stronger than the original SHA-384. The document bytes will be hashed with SHA-512, but the TSA will also apply hashing and in this case SHA-256 is used. The encryption algorithm and the key-length depend entirely on the private key used by the TSA.

> **Note**: that you can recognize a document-level timestamp by its filter sub type: `/ETSI.RFC3161`. In this example, we only extract the name of the signer from the TSA's certificate, but you could easily use code sample 5.7 to check the expiration date of that certificate, and decide to add a new timestamp if the expiration date is near.

If we consult the PAdES 4 specification, we notice that the validation methods described in the previous sections of this chapter aren't sufficient.

5.4.4 Validating an LTV document

PAdES 4 section 4.3 gives the following recommendations with respect to the validation process:

1. The "latest" document Time-stamp should be validated at current time with validation data collected at the current time.
2. The "inner" document Time-stamp should be validated at previous document Time-stamp time with the validation data present (and time-stamped for the successive enveloping time-stamps) in the previous DSS.
3. The signature and the signature Time-stamp should be validated at the latest innermost LTV document Time-stamp time using the validation data stored in the DSS and time-stamped (by the successive enveloping time-stamps)

Code sample 5.10 shows how it's implemented in iText.

Code sample 5.10: LTV Validation

```
public void validate(PdfReader reader)
    throws IOException, GeneralSecurityException  {
    KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
    ks.load(null, null);
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    ks.setCertificateEntry("adobe",
        cf.generateCertificate(new FileInputStream(ADOBE)));

    CertificateVerifier custom = new CertificateVerifier(null) {
        public List<VerificationOK> verify(
            X509Certificate signCert, X509Certificate issuerCert, Date signDate)
            throws GeneralSecurityException, IOException {
            System.out.println(signCert.getSubjectDN().getName() +
                ": ALL VERIFICATIONS DONE");
            return new ArrayList<VerificationOK>();
        }
    };
    LtvVerifier data = new LtvVerifier(reader);
    data.setRootStore(ks);
    data.setCertificateOption(CertificateOption.WHOLE_CHAIN);
    data.setVerifier(custom);
    data.setOnlineCheckingAllowed(false);
    data.setVerifyRootCertificate(false);
    List<VerificationOK> list = new ArrayList<VerificationOK>();
    try {
        data.verify(list);
    }
    catch(GeneralSecurityException e) {
        System.err.println(e.getMessage());
    }
    System.out.println();
    if (list.size() == 0) {
        System.out.println("The document can't be verified");
    }
    for (VerificationOK v : list)
        System.out.println(v.toString());
}
```

What happens in this snippet? First we create a key store with the Adobe root certificate. When we added a document-level timestamp in code sample 5.9, we used a timestamp from a CA that is a CDS partner of Adobe. We'll be able to validate the outer timestamp against the key store with the Adobe CA root certificate.

We create an `LtvVerifier` instance passing a `PdfReader` instance for the document we want to verify. We add the key store with the Adobe Root CA certificate, and we tell the verifier that we want to check the whole chain of each signature, not just the signing certificate.

> **NOTE:** in the current implementation (iText 5.3.4), the valitidy of the timestamps of the innermost documents isn't checked yet; only the validity of the document-level timestamps is currently checked. We'll try to fix this soon to conform to bullet 3 of PAdES-4 section 4.3.

iText will verify certificates against OCSP responses, CRLs and a root store, using a chain containing an `OCSPVerifier`, `CRLVerifier` and `RootStoreVerifier` (in that order). If you want to add your custom verifier to the chain, you can do so with the `setVerifier()` method.

For demonstration purposes I created a custom `CertificateVerifier` that always returns an empty list, but that writes *"ALL VERIFICATIONS DONE"* to the `System.out`. This verifier will be called after all the other verifiers have done verifying.

For a document signed with one document-level timestamp that extended the life of a document signed with my USB token, the output looks like this:

```
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW01-001:
ALL VERIFICATIONS DONE
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe:
ALL VERIFICATIONS DONE
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA:
ALL VERIFICATIONS DONE
C=BE,L=Sint-Amandsberg,E=bruno@___________,O=iText Software BVBA,CN=Bruno Lowagie:
ALL VERIFICATIONS DONE
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign SHA256 CA for Adobe:
ALL VERIFICATIONS DONE
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe:
ALL VERIFICATIONS DONE
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA:
ALL VERIFICATIONS DONE
```

You could use a custom verifier to log information about all the certificates that are encountered. In this case we verified seven certificates, but some certificates were checked twice because they were used in different certificate chains and different revisions.

To verify the outermost document-level timestamp, iText will get the most recent revocation information online. By default, iText will also do that for the certificates of the inner documents. Maybe you don't want that. Maybe you require *all* the revocation information to be inside the document. In that case, you have to tell iText that online verification of those certificate isn't allowed with the `setOnlineCheckingAllowed()` method.

iText will always try to verify the self-signed root certificates in the certificate chain, except for the root certificate of the "latest" document Time-stamp. This could cause problems for some documents. For instance: for the document signed with my CAcert certificate, we only included OCSPs, no CRLs. There is no OCSP response for CAcert's root certificate, and as there's no CRL either, we can't possibly validate that certificate unless it's present in our root store. I avoided this problem by setting the `verifyRootCertificate` variable to `false`. This may or may not be in compliance with your requirements.

Once we've set all the parameters, we create an empty `VerificationOK` list and we execute the `verify()` method. If an exception is thrown, we get a `GeneralSecurityException`, probably a `VerificationException` unless some elements of the signature were malformed.

Even if no exception was thrown, it's possible that the document couldn't be verified. As we're curious about the certificates that were successfully verified, regardless whether or not an exception was thrown, we print the `VerificationOK` objects to the `System.out`:

Let's take a look at the output of some of the documents we signed:

```
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW01-001 verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid
CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root
store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
C=BE,L=Sint-Amandsberg,E=bruno@___________,O=iText Software BVBA,CN=Bruno Lowagie verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign SHA256 CA for Adobe verified with
com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root
store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
```

We see the same seven certificates we had before, but some certificates pass more than one verification method. The outer timestamp is verified using a CRL that is found online. So is its issuer, but it can also be verified against the root store that contains the self-signed Adobe Root CA certificate.

My personal certificate can be verified agains an embedded CRL, and so can its issuers. There's no need to go online: all the CRLs are embedded. Now the Adobe Root CA can be verified against an embedded CRL as well as against the root store.

If we look at the output when verifying the document signed with a smart card, the lines for the document-level time-stamp are identical to what we had before, but now we get some info about OCSP responses for the inner document:

```
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW02-001 verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid
CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root
store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
C=BE,CN=Alice SPECIMEN,SURNAME=SPECIMEN,GIVENNAME=Alice Geldigekaart,SERIALNUMBER=71715100070
verified with com.itextpdf.text.pdf.security.OCSPVerifier: Valid OCSPs Found: 1
C=BE,CN=Alice SPECIMEN,SURNAME=SPECIMEN,GIVENNAME=Alice Geldigekaart,SERIALNUMBER=71715100070
verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,CN=eID test Citizen CA verified with com.itextpdf.text.pdf.security.CRLVerifier:
Valid CRLs found: 1
C=BE,CN=eID test Root CA verified with com.itextpdf.text.pdf.security.CRLVerifier:
Valid CRLs found: 1
```

We find an OCSP response as well as a CRL for Alice Specimen. We find CRLs for the issuers. This isn't the case when we verify the document signed with my CAcert key:

```
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW01-001 verified with com.itextpdf.text.pdf.security.CRLVerifier: Valid
CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root
store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
CN=Bruno Lowagie,E=bruno@__________ verified with com.itextpdf.text.pdf.security.OCSPVerifier:
Valid OCSPs Found: 1
O=Root CA,OU=http://www.cacert.org,CN=CA Cert Signing Authority,E=support@cacert.org verified
with com.itextpdf.text.pdf.security.LtvVerifier: Root certificate passed without checking
```

Now we find a valid OCSP response embedded in the document for my own certificate, but the root certificate passed without checking. If we hadn't told iText that it wasn't necessary to verify root certificates, a `VerificationException` would have been thrown.

Finally, to prove that we've also taken care of bullet 2 of PAdES-4 section 4.3, let me copy/paste the output of a document with two document-level time-stamps:

```
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW01-001 verified with com.itextpdf.text.pdf.security.CRLVerifier:
Valid CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1 (online)
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier:
Certificate verified against root store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
OU=SEIKO Precision Inc.,E=timestampinfo@globalsign.com,O=GlobalSign,CN=SEIKO Timestamp
Service. Advanced AW01-001 verified with com.itextpdf.text.pdf.security.CRLVerifier:
Valid CRLs found: 2
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier:
Certificate verified against root store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
C=BE,L=Sint-Amandsberg,E=bruno@___________,O=iText Software BVBA,CN=Bruno Lowagie verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign SHA256 CA for Adobe verified with
com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=BE,O=GlobalSign nv-sa,OU=GlobalSign CDS,CN=GlobalSign Primary SHA256 CA for Adobe verified
with com.itextpdf.text.pdf.security.RootStoreVerifier:
Certificate verified against root store.
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.CRLVerifier: Valid CRLs found: 1
C=US,O=Adobe Systems Incorporated,OU=Adobe Trust Services,CN=Adobe Root CA verified with
com.itextpdf.text.pdf.security.RootStoreVerifier: Certificate verified against root store.
```

This concludes the section about verifying LTV documents.

5.5 Summary

In this final chapter, we've taken a closer look at the documents we've signed in previous chapters. We've checked the integrity of the different revisions in a signed document, and we've retrieved information about the signer, the time the document was signed, the validity period of the certificates, and so on.

We've verified the certificates used for signing in different ways. We've created a root store and compared the certificates against the certificates in that root store. We've also checked if the certificates appear on a Certificate Revocation List, and we've tried to get an OCSP response from the CA issuing the certificate.

Finally, we've appended a Document Security Store to documents that lacked recent certificate revocation information, and we learned how to extend the life of a signed document by adding document-level timestamps.

Afterword

This white paper was written to promote the use of digital signatures. It's intended for developers who want to know more about signing PDF documents.

The code samples are written in Java and they all involve using iText version 5.3.4 or higher. I'm confident that the Java examples are also easy to understand for C# developers who are using iTextSharp instead of iText. Choosing iText and Java allowed me to introduce an abundance of examples and real-world use cases. I hope I succeeded in conveying my expertise with respect to digital signatures in general.

I combine the jobs of CEO and CTO at the iText Software Group. This group consists of different technology companies in Europe and the US. We provide software that can be used to create PDF documents from scratch, to post-process existing PDF documents, to fill out forms in a desktop or a web application. You can also use iText to digitally sign PDF documents.

PDF is our core business. We offer professional services, but we usually don't do projects from A to Z. For projects that go beyond PDF, we prefer working with an integrator as our partner. Currently we don't offer SaaS services, nor do we act as a CA or TSA. If you need a SaaS service or a certificate, we'll be happy to refer you to one of our customers or partners.

Although I've often referred to the legislation in different countries, please take into account that IANAL (I Am Not A Lawyer). When in doubt about the legal value of signatures, please consult an attorney or a specialized law firm. Be aware that the legislation can be very different in different countries. Don't assume that the concept of "Electronic Signatures" as it's defined in the USA, is also valid in Europe. I started writing an appendix comparing different laws, but I dropped the idea because that would have been a never-ending story. Did I already mention that IANAL?

One of the most surprising discoveries I personally made doing research for this white paper, is the fact that something I used to take for granted —the fact that every citizen of the country I live in has an eID— is a taboo in many countries, including the USA. When trying to reason about this subject, I was confronted with many sophisms and prejudices. Eventually, it wasn't always clear to me which arguments were valid and which arguments were based on irrational feelings.

That's why I decided to stick to what I know: the technical aspects of digital signatures. I'm sure you'll agree that this was a good decision.

Bibliography

Adobe. (1993). The Portable Document Format Reference. Addison Wesley.

De Cock, Danny; Simoens, Koen; Preneel, Bart. (2008, June 2). Insights on Identity Documents based on the Belgian Case Study. Leuven: Katholieke Universiteit Leuven, Department of Electrical Engineering ESAT/SCD–COSIC.

ETSI. (2009). Electronic Signatures and Infrastructures (ESI); PDF Advanced Electronic Signature Profiles. *PAdES (ETSI TS 102 778)*. European Telecommunications Standards Institute.

ISO. (2008). Document management — Portable Document Format — Part 1. *ISO-32000-1*. International Organization for Standardization.

Lowagie, B. (2011). iText in Action. Manning Publications Co.

Warnock, J. (1991). The Camelot Paper.

Links

This is a list of links that may be useful for people wanting to know more about the topics discussed in this white paper:

Software

iText Software:

http://itextpdf.com (for more info about the software used in this white paper)
http://lowagie.com (for more info about the author of this white paper)

BouncyCastle:

http://www.bouncycastle.org/ (when working on Android, look for SpongyCastle!)

Adobe:

http://adobe.com/

Oracle:

http://java.oracle.com/

- Look for the 32-bit version of Java 7: Java 7 for MSCAPI support; the 32-bit version for PKCS#11 support on Windows. When using Linux, using the 64-bit version is OK.
- Also look for the *"Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files"*

Certificate Authorities

GlobalSign:

https://www.globalsign.com/
Get your trial key at: https://www.globalsign.com/pdf-signing/trial.html

CAcert:

https://www.cacert.org

Services

AuthentiDate:

http://www.authentidate.de/en.html

Hardware

Vasco:

https://www.vasco.com/

SafeNet:

http://www.safenet-inc.com/

Research & Development

Katholieke Universiteit Leuven:

http://www.esat.kuleuven.be/scd/

http://www.cosic.esat.kuleuven.be/publications/article-1160.pdf

US Office of Legislative Counsel:

http://www.mnhs.org/preserve/records/legislativerecords/docs_pdfs/CA_Authentication_WhitePaper_Dec2011.pdf

Index

C

D

E

F

P

R

S

T

U

V

W

X