

# **Decrypting Open Document Format (ODF) Files**

by

K. Udo Schuermann  
*udo.schuermann@gmail.com*

September 7, 2009



# Table of Contents

|       |  |    |
|-------|--|----|
| 1     | Introduction.....                            | 5  |
| 1.1   | Audience.....                                | 5  |
| 1.2   | Open Document Format Encryption.....         | 6  |
| 1.3   | Concepts.....                                | 6  |
| 1.3.1 | Stronger Keys: Salt and Algorithms.....      | 6  |
| 1.3.2 | Initialization Vectors.....                  | 7  |
| 1.3.3 | The Open Document Format (ODF) Manifest..... | 7  |
| 2     | Decryption Code.....                         | 9  |
|       | APPENDIX A: The PBKDF2 Algorithm.....        | 11 |
|       | APPENDIX B: Revisions.....                   | 13 |



# 1 Introduction

I've spent months, off and on, beating my head against the wall that is the standard Java crypto library's limitation in the PBEKeySpec class, trying to figure out first why it didn't work on OpenOffice.org documents and then trying to bend it to my will and coming to realize that there is simply no way to rely solely on the Java libraries to decrypt OpenOffice.org documents: OpenOffice.org transforms a user-entered password first via SHA1 digest into binary form which Java's PBEKeySpec cannot accept because the PBEKeySpec wants a UTF-8 encoded password in a char[] and there is simply no one-to-one transformation possible between arbitrary bytes and UTF-8 encoded characters.

Steven Elliot<sup>1</sup> kindly pointed me to his website where he had made available a substantial amount of information on OpenOffice.org's encryption and storage related details; further research led me to a clean-room implementation of PBKDF2 by Matthias Gärtner<sup>2</sup> which took a byte array rather than text for its password input, thereby completing the picture. I would like to thank both of these individuals (and you should, too!) for the work they've done.

What follows herein is all you need to know to decrypt OpenOffice.org documents with nothing but Java code. It is the fruit of many months of head scratching, countless hours of research, and the construction of source code; all of this comes to you, O breathless reader, merely for the cost of reading carefully.

Enjoy!

## 1.1 Audience

This document is aimed at software engineers and programmers, specifically those workign with Java. It is likely that the information in this document is of significant help to anyone working in other programming languages, too.

If you are so misfortunate to have lost/forgotten the password to an Open Document Format file, it is not likely that my document will be of much help to you, except in writing yourself the necessary software to attempt a brute-force attack on your document: Your lost password was cryptographically enhanced and transformed into a 128 bit key and then used by the Blowfish cipher to produce the encrypted version of the document. Without the password, you'll likely need decades or even centuries of time on a supercomputer to perform the necessary cryptanalysis on your document. Chances are that you don't have that kind of computer power at your disposal, so your best bet is to think back to the time when you chose your lost password, and try to recall the state of mind which led you to picking whatever password you cannot remember.

---

1 See Steven Elliot's page at <http://selliot.org/encryption/openoffice>

2 See Matthias Gärtner's page at <http://rtner.de/software/PBKDF2.html>

## 1.2 Open Document Format Encryption

I am not familiar enough with implementations of the OASIS Open Document Format standard to say for certain what variations in the encryption process exist and to what degree they are guaranteed to be interoperable between different implementations of the standard; for what it's worth, the following applies to OpenOffice.org 2.0 and later.

As with the reversal of any process, it helps to understand the original. The decryption process is aided by understanding how OpenOffice.org encrypts documents. This is the essential process:

1. Key Generation
  - a) The user-supplied password is transformed into a 20-byte (160 bit) digest using the SHA1 digest algorithm,
  - b) Using a random *salt* this password hash is then *strengthened* using the PBKDF2 algorithm with 1024 iterations of the SHA1 digest algorithm; the output is a 16-byte (128 bit) cryptographic key,
2. Data Compression and Encryption
  - a) Each sensitive document component is deflated (compressed) with the algorithm implemented by zlib (and used by gzip and similar zip tools); header and footer components part of the gzip file format are *not stored*<sup>3</sup>,
  - b) Each such deflated component is then encrypted with random *initialization vector* and the cryptographic key from step 1 above, using Bruce Schneier's Blowfish cipher in CFB (Cyclic Feedback) mode,
  - c) The name of the key strengthening algorithm and cryptographic cipher, as well as the value of the salt, initialization vector, and the SHA1/1K (SHA1 of the first 1024 bytes of encrypted data) is stored into the META-INF/manifest.xml file for the sensitive document component.

The code listed in this document essentially reverses these steps.

## 1.3 Concepts

Cryptography requires extreme care. I shall not presume to teach you all you need to know about it as I am far from expert in the subject, nor is it the focus of this document. I did learn a great deal, however, in my odyssey and gained a much clearer understanding. Many questions on Sun's cryptography forums are replete with even worse misconceptions than what I started out with so, in the interest of imparting a little knowledge along the way, here comes the clue train.

### 1.3.1 Stronger Keys: Salt and Algorithms

If the encrypted data, the *cipher text*, is to reveal no patterns to help an attacker analyze and derive from it the original cryptographic key, then this key cannot be merely based on a limited set of letters, such as the word "secret". First, this is rather too short to make a strong key and second, alphabetic letters are a small subset of all possible values in each byte of the key, and this could reveal patterns in the cyper text. Humans are notoriously bad, too, when it comes to producing really complex passwords, and even worse at remembering them, so this is where a key strengthening algorithm comes into play.

---

<sup>3</sup> gzip expects a 10-byte header (magic number, version, and time stamp) and 8-byte footer (original file's crc32 and size); the java.util.zip.Inflater class, when initialized with a 'true' parameter, uncompresses deflated data without this meta-data.

By using a digest algorithm, such as MD5 or SHA1, a cryptographic hash can be produced that has not just a fairly decent size (16 or 20 bytes respectively), but an apparently random spread of bits across all of these bytes. A single application of such an algorithm already tends to produce a “better” key. Strengthening algorithms such as PBKDF2<sup>4</sup> recommend at least 1000 iterations of SHA1 combined with something called *salt*, whose role is to make it infeasible for an attacker to pre-compute the cryptographic keys of an attack dictionary: for any given password, there are now  $2^{64}$  different keys, thus slowing the speed of brute force attacks.

### 1.3.2 Initialization Vectors

If the same key were reused on similar data, the two cipher texts could be analyzed for similarities and differences, thereby revealing something about the key and, therefore, the data. Initialization vectors effectively add a random value to the start of the data to be encrypted, which causes the cipher engine to produce significantly different cryptographic output regardless of the similarity of the data and the key used in the encryption process. Even a small amount of such random data *at the start* produces drastically different cipher text.

### 1.3.3 The Open Document Format (ODF) Manifest

OpenOffice.org implements the OASIS Open Document Format (ODF) standard. The styles, configuration, included images, and the data itself are all stored in what is effectively a “zip” file. You can verify this by running a command such as the following on the command line; the -l (lowercase L) argument merely lists, but does not extract, the contents:

```
unzip -l test.odt
```

One required component in the ODF container (zip) file is the manifest, which is found in a file named

```
META-INF/manifest.xml
```

If you were to extract and list the contents of this file, or dump it straight to the console with a command such as the following (the -c argument dumps the named file to the console) ...

```
unzip -c test.odt META-INF/manifest.xml
```

... you'll find that it lists numerous “file-entry” sections that give details on the document's components. Encrypted components will have additional information, namely in an XML element enclosed by the “file-entry” that is named “encryption-data”; one such file-entry is shown below, with the enclosed crypto information listed in **bold**:

```
<manifest:file-entry manifest:full-path="content.xml" manifest:media-type="text/xml" manifest:size="4836">
  <manifest:encryption-data manifest:checksum="LmjB93oYe9o0bLITQ020/QzVveM=" manifest:checksum-type="SHA1/1K">
    <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisation-vector="W9S51NUZ3JU=" />
    <manifest:key-derivation manifest:iteration-count="1024" manifest:key-derivation-name="PBKDF2"
      manifest:salt="QqaBc+3uA5TAahHBaWZ5ng==" />
  </manifest:encryption-data>
</manifest:file-entry>
```

Let's go over that XML fragment as the information is important. In top-down, left-to-right order, this is what you should be able to find in this file (the relevant elements above are underlined):

1. The name of the file is “content.xml” — This is the file that contains your data but is encrypted,
2. The original size of the file (before being compressed and encrypted) is 4836 bytes,
3. The file is encrypted (there is an “encryption-data” element),

<sup>4</sup> PBKDF2 = Password Based Key Derivation Function v2.

4. We have a checksum (it is Base64 encoded),
5. The checksum was produced by the SHA1/1K algorithm (it covers only the first 1024 bytes of the plain text),
6. The cipher is Blowfish in CFB mode (the fact that it uses no padding is not specified but derived by experimentation on my part); technically Blowfish/CFB/NoPadding,
7. The initialization vector for the cipher is also given in Base64 encoded form,
8. The key derivation uses an iteration count of 1024,
9. The algorithm for the key derivation is PBKDF2 (technically PBKDF2WithHmacSHA1),
10. The salt for the key derivation is also given in Base64 encoding.

Make sure you can find these pieces before you continue. It is not difficult, though admittedly laborious, to extract all these parts and decode the Base-64 encoded blocks by hand. A programmatic way is far less error prone and, if you plan to do this more than just once or twice, significantly easier. That work is, however, left to you as an exercise.

The main component of the ODF file (as shown in the XML fragment above) is “content.xml”; it is that file which we will need to decrypt to get at the body of the encrypted data, the stuff you'd generally edit on the screen; the same password will work for all other parts if you want to decrypt them, too, but each entry uses its own salt for the password and initialization vector for the cipher, of course.

## 2 Decryption Code

Transform the password using the SHA1 digest algorithm into a 20-byte (160 bit) hash; note that because the PBKDF2 algorithm performs such a transformation (along with the application of salt), this step can be considered rather redundant but this is what decrypting OpenOffice.org documents requires:

```
MessageDigest sha1 = MessageDigest.getInstance( "SHA1" );
sha1.update( password.getBytes() );
byte[] pwdHash = sha1.digest();
```

Now we strengthen the password hash (see Appendix 2: The PBKDF2 Algorithm): The salt comes from the manifest, as does the iteration count. The derived key length will be 16 bytes (128 bits); we convert it directly into one that the Java cryptographic library can use with its implementation of the Blowfish cipher:

```
byte[] dk = PBKDF2.deriveKey( pwdHash, salt, iterationCount, 16 );
Key key = new SecretKeySpec( dk, "PBKDF2WithHmacSHA1" );
```

Next, we build the Initialization Vector from the data obtained from the manifest:

```
IvParameterSpec iv = new IvParameterSpec( initvector );
```

And obtain the cipher, specifically the Blowfish cipher in CFB mode, and then initialize it with the initialization vector; we set it up for decryption:

```
Cipher cipher = Cipher.getInstance( "Blowfish/CFB/NoPadding" );
cipher.init( Cipher.DECRYPT_MODE, key, iv );
```

and now, we're ready to decrypt the cipher text, which is loaded directly from the ODF container; this would be the contents of the "content.xml" file, for example:

```
byte[] deflatedPlainText = cipher.doFinal( cipherText );
```

The result still needs to be inflated; the size is optional here but useful in pre-allocating the buffer, which reduces heap fragmentation. The size is provided by the manifest, too:

```
ByteArrayInputStream iStream = new ByteArrayInputStream( deflatedPlainText );
InflaterInputStream inflater = new InflaterInputStream( iStream,
                                                       new Inflater(true) );

ByteArrayOutputStream oStream = new ByteArrayOutputStream( size );
int inBuffer;
while( (inBuffer = inflater.read(buffer)) >= 0 )
{
    oStream.write( buffer, 0, inBuffer );
}
```

And the result of the decompression is easily obtained from the `ByteArrayOutputStream` object; it is the original plain text:

```
byte[] plainText = oStream.toByteArray();
```

And that's it!

## APPENDIX A: The PBKDF2 Algorithm

The following code belongs, in essence, to Matthias Gärtner. Some variables I have renamed; added some comments, reordered the arguments, and reformatted the whole to my liking. No functional changes were made, of course, as Matthias' code worked perfectly as is:

```
byte[] deriveKey( byte[] password,
                  byte[] salt,
                  int iterationCount,
                  int dkLen )
    throws NoSuchAlgorithmException,
           InvalidKeyException
{
    SecretKeySpec keyspec = new SecretKeySpec( password, "HmacSHA1" );
    Mac prf = Mac.getInstance( "HmacSHA1" );
    prf.init( keyspec );

    // Note: hLen, dkLen, l, r, T, F, etc. are horrible names for
    //       variables and functions in this day and age, but they
    //       reflect the terse symbols used in RFC 2898 to describe
    //       the PBKDF2 algorithm, which improves validation of the
    //       code vs. the RFC:

    int hLen = prf.getMacLength(); // 20 for SHA1
    int l = ceil( dkLen, hLen); // 1 for 128bit (16-byte) keys
    int r = dkLen - (l-1)*hLen; // 16 for 128bit (16-byte) keys

    byte T[] = new byte[l * hLen];
    int ti_offset = 0;
    for (int i = 1; i <= l; i++)
    {
        F( T, ti_offset, prf, salt, iterationCount, i );
        ti_offset += hLen;
    }
    if (r < hLen)
    {
        // Incomplete last block
        byte DK[] = new byte[dkLen];
        System.arraycopy(T, 0, DK, 0, dkLen);
        return DK;
    }
    return T;
}
```

Here is the function F as referenced in RFC 2898, the original source for the PBKDF2 algorithm from which Matthias implemented PBKDF2:

```
private static void F( byte[] dest,
                      int offset,
                      Mac prf,
                      byte[] S,
                      int c,
                      int blockIndex )
{
    final int hLen = prf.getMacLength();
    byte U_r[] = new byte[ hLen ];

    // U0 = S || INT (i);
    byte U_i[] = new byte[S.length + 4];
    System.arraycopy( S, 0, U_i, 0, S.length );
    INT( U_i, S.length, blockIndex );

    for( int i = 0; i < c; i++ )
    {
        U_i = prf.doFinal( U_i );
        xor( U_r, U_i );
    }
    System.arraycopy( U_r, 0, dest, offset, hLen );
}
```

And the function xor:

```
private static void xor( byte[] dest,
                       byte[] src )
{
    for( int i = 0; i < dest.length; i++ )
    {
        dest[i] ^= src[i];
    }
}
```

And last, the INT function:

```
private static void INT( byte[] dest,
                       int offset,
                       int i )
{
    dest[offset + 0] = (byte) (i / (256 * 256 * 256));
    dest[offset + 1] = (byte) (i / (256 * 256));
    dest[offset + 2] = (byte) (i / (256));
    dest[offset + 3] = (byte) (i);
}
```

## APPENDIX B: Revisions

This document has been revised since its original publication on November 30, 2008; in reverse-chronological order these revision are ...

September 7, 2009 — Improved the explanation on strengthening passwords using cryptography; also fixed a typographical error in that section.

July 29, 2009 — Renamed the document in an effort to clarify that the algorithms presented should apply to the OASIS Open Document Format (ODF) in general; OpenOffice.org is merely one package that implements this standard. Also, added a section describing the intended audience for this document.

December 3, 2008 — Fixed some typographical errors and glitches in my wording. Clarified some aspects that were not necessarily as clear as they could have been. No technical changes.