DI Management Using Padding in Encryption

Introduction

The most common query we get about cryptography concerns padding with a block cipher.

If you have to encrypt some data to send to another user and everything else works OK except the last few bytes, your problem is probably one of padding. Read on.

This article describes the mechanics of padding with examples using common block encryption algorithms like DES, Triple DES, Blowfish and AES. It considers where you should use padding and where you don't need to.

What is padding?

Block cipher algorithms like DES and Blowfish in Electronic Code Book (ECB) and Cipher Block Chaining (CBC) mode require their input to be an exact multiple of the block size. If the plaintext to be encrypted is not an exact multiple, you need to pad before encrypting by adding a *padding string*. When decrypting, the receiving party needs to know how to remove the padding in an unambiguous manner.

Example using DES in ECB mode

The plain text is the ASCII code for "`Now is the time for`". We are encrypting using DES in ECB mode with the cryptographic key `0x0123456789ABCDEF`. To encrypt, we break up the plaintext into blocks of 8 bytes (Note we are using 8 in this example because the block size for DES is 64 bits or 8 bytes; if this were AES we'd be using 16 bytes).

```
1234567812345678123
Now is the time for
```

breaks up into

```
|Now_is_t|he_time_|for?????|
|block 1-|block 2-|block 3-|
```

In ECB mode, each 8-byte block is encrypted independently.

First block:

```
DES INPUT BLOCK  = N  o  w   _   i  s   _   t
(IN HEX)           4E 6F 77 20 69 73 20 74
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = 3F A4 0E 8A 98 4D 43 15
```

Second block:

```
DES INPUT BLOCK  = h  e   _   t  i  m  e   _
(IN HEX)           68 65 20 74 69 6D 65 20
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = 6A 27 17 87 AB 88 83 F9
```

So far, so good. But what do we do with the odd characters at the end of our plaintext?

```
DES INPUT BLOCK  = f  o  r  ?? ?? ?? ?? ??
(IN HEX)           66 6F 72 ?? ?? ?? ?? ??
```

We need to pad the block with padding bytes to make it up to the required length. There are at least five common conventions:-

1. Pad with bytes all of the same value as the number of padding bytes
2. Pad with 0x80 followed by zero bytes
3. Pad with zeroes except make the last byte equal to the number of padding bytes
4. Pad with zero (null) characters
5. Pad with space characters

Method one is the method described in PKCS#5, PKCS#7 and RFC 3852 Section 6.3 (formerly RFC 3369 and RFC 2630). It is the most commonly used, and the one we recommend in the absence of any other considerations.

Method 1 - Pad with bytes all of the same value as the number of padding bytes

This is the method recommended in [PKCS5], [PKCS7], and [CMS].

Pad the input with a padding string of between 1 and 8 bytes to make the total length an exact multiple of 8 bytes. The value of each byte of the padding string is set to the number of bytes added - i.e. 8 bytes of value 0x08, 7 bytes of value 0x07, ..., 2 bytes of 0x02, or one byte of value 0x01.

Our fourth block is padded with 5 bytes of value 0x05:

```
DES INPUT BLOCK  = f  o  r
(IN HEX)           66 6F 72 05 05 05 05 05
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = FD 29 85 C9 E8 DF 41 40
```

After decrypting, read the last character decrypted and strip off that many bytes.

This method can be used with any plaintext, ASCII or binary. Don't forget to check first that the number of characters to be stripped is between one and eight. This also gives you an extra check that the decryption has been carried out correctly (well, if its value is not between 1 and 8 you know for sure that is hasn't, but if it is, it doesn't necessarily mean it has). As an additional check, check all of the padding bytes.

Method 2 - Pad with 0x80 followed by zero (null) bytes

Add a single padding byte of value 0x80 and then pad the balance with enough bytes of value zero to make the total length an exact multiple of 8 bytes. If the single 0x80 byte makes the total length an exact multiple then do not add any zero bytes.

Our fourth block is padded with 0x80 followed by 4 bytes of value 0x00:

```
DES INPUT BLOCK  = f  o  r  _  _  _  _  _
(IN HEX)           66 6F 72 80 00 00 00 00
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = BE 62 5D 9F F3 C6 C8 40
```

After decrypting, strip off all trailing zero bytes and the 0x80 byte.

This method can be used with any plaintext, ASCII or binary. Again, you have a check for incorrect decryption if you don't find the expected padding bytes at the end. Cryptographers who work with smart cards seem to prefer this method, see [RANK]. It is recommended in NIST 800-38a [BCMO]. In their latest book [FERG], Niels Ferguson and Bruce Schneier recommend either this method or method 1.

Method 3 - Pad with zeroes except make the last byte equal to the number of padding bytes

Our fourth block is padded with 4 null characters (0x00) followed by a byte with value 0x05:

```
DES INPUT BLOCK  = f  o  r  _  _  _  _  _
(IN HEX)           66 6f 72 00 00 00 00 05
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = 91 19 2C 64 B5 5C 5D B8
```

This is a variant of method 1 described in [SCHN]. The convention with this method is usually *always* to add a padding string, even if the original plaintext was already an exact multiple of 8 bytes. The final byte could therefore have a value between 0x01 and 0x08.

After decrypting, read the last character decrypted and strip off that many bytes (checking first that its value is between one and eight).

This method can be used with any plaintext, ASCII or binary.

Method 4 - Pad with zero (null) characters

Our fourth block is padded with 5 null characters (0x00):

```
DES INPUT BLOCK  = f  o  r  _  _  _  _  _
(IN HEX)           66 6f 72 00 00 00 00 00
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = 9E 14 FB 96 C5 FE EB 75
```

After decrypting, trim all null characters found at the end until you find a non-null character. You cannot use this method when the plaintext could contain a null value. This is not a problem if you are dealing with ASCII text, but would be if encrypting binary data like an EXE file.

Method 5 - Pad with spaces

Our fourth block is padded with 5 space characters (0x20):

```
DES INPUT BLOCK  = f  o  r  _  _  _  _  _
(IN HEX)           66 6f 72 20 20 20 20 20
KEY              = 01 23 45 67 89 AB CD EF
DES OUTPUT BLOCK = E3 FF EC E5 21 1F 35 25
```

You will find this method used in EDI applications where the plaintext is simple ASCII text and does not have trailing spaces at the end. The convention is usually that if the original plaintext is already an exact multiple of eight, then no padding is added, otherwise the last block is padded with space characters up to the next multiple of eight. See, for example, [NZEDI]

After decrypting, just trim any trailing space characters, if any

The resulting ciphertext from these five methods will be as follows:

```
   |-- block 1 ---| |-- block 2 ---| |-- block 3 ---|
1. 3FA40E8A984D4315 6A271787AB8883F9 FD2985C9E8DF4140
2. 3FA40E8A984D4315 6A271787AB8883F9 BE625D9FF3C6C840
3. 3FA40E8A984D4315 6A271787AB8883F9 91192C64B55C5DB8
4. 3FA40E8A984D4315 6A271787AB8883F9 9E14FB96C5FEEB75
5. 3FA40E8A984D4315 6A271787AB8883F9 E3FFECE5211F3525
```

Note how different the last blocks are.

Example using Blowfish in CBC mode

This example is from Eric Young's set of test vectors for Blowfish. Note that the plaintext data includes a trailing null character (0x00) after the final space character (0x20).

```
key[16]  = 0123456789ABCDEFF0E1D2C3B4A59687
iv[8]    = FEDCBA9876543210
data[29] = "7654321 Now is the time for " (includes trailing '\0')
data[29] = 37363534333231204E6F7720697320746865207 4696D6520666F722000

Method 1 PKCS#5/RFC3369 method
IN:  37363534333231204E6F7720697320746865207 4696D6520666F722000030303
OUT: 6B77B4D63006DEE605B156E27403979358DEB9E7154616D9749DECBEC05D264B
                                                ^^^^^^^^^^^^^^^^
Method 2 Pad with 0x80 + nulls
```

```
IN:  37363534333231204E6F77206973207468652074696D6520666F722000800000
OUT: 6B77B4D63006DEE605B156E27403979358DEB9E7154616D9BB3F8B9254003C40
                                                    ^^^^^^^^^^^^^^^^

Method 3 Pad with nulls + # bytes
IN:  37363534333231204E6F77206973207468652074696D6520666F722000000003
OUT: 6B77B4D63006DEE605B156E27403979358DEB9E7154616D9A078DBB46155E4AC
                                                    ^^^^^^^^^^^^^^^^

Method 4 Pad with nulls (this is the method used in the test vectors)
IN:  37363534333231204E6F77206973207468652074696D6520666F722000000000
OUT: 6B77B4D63006DEE605B156E27403979358DEB9E7154616D959F1652BD5FF92CC
                                                    ^^^^^^^^^^^^^^^^

Method 5 Pad with spaces
IN:  37363534333231204E6F77206973207468652074696D6520666F722000202020
OUT: 6B77B4D63006DEE605B156E27403979358DEB9E7154616D93322238999980E694
                                                    ^^^^^^^^^^^^^^^^
```

Note that the last 8 bytes of ciphertext are all completely different. For this example, because the plaintext contains trailing space characters and nulls, the padding in methods 4 and 5 cannot be removed unabiguously.

What about ciphers like AES that have larger block sizes?

More modern block ciphers like the Advanced Encryption Algorithm (AES) have larger block sizes than DES and Blowfish. All the arguments above still apply, except you should replace the number 8 with the appropriate block size in *bytes*. For example, if using AES with a 128-bit block, pad to the next multiple of 16. Note, too, that it is the *block* size that matters, not the size of the key.

Example using AES

This example uses AES-128 to encrypt a short message "Hello" using the 128-bit key `0xA456B7A422C5145ABCF2B3CB206579A8`. The block size of 128 bits is equal to 128/8 = 16 bytes, so we need to pad our 5-byte plaintext with an 11-byte padding string. The ASCII characters "Hello" are 48656C6C6F in hexadecimal and 11 decimal = 0x0B.

```
Method 1 PKCS#7/RFC3369 method
AES INPUT BLOCK  = 48656C6C6F0B0B0B0B0B0B0B0B0B0B0B
AES OUTPUT BLOCK = 42FF63CC06D53DA93F24389723E1611A

Method 2 Pad with 0x80 + nulls
AES INPUT BLOCK  = 48656C6C6F80000000000000000000000
AES OUTPUT BLOCK = 97AFA1455DA9E2E1B821275997CF4DC5

Method 3 Pad with nulls + # bytes
AES INPUT BLOCK  = 48656C6C6F0000000000000000000000B
AES OUTPUT BLOCK = 6024D07C11283639425E3A33D99F32BA

Method 4 Pad with nulls
AES INPUT BLOCK  = 48656C6C6F00000000000000000000000
AES OUTPUT BLOCK = 3FC6C30A64CD6E2970803871C7068998

Method 5 Pad with spaces
```

```
AES INPUT BLOCK  = 48656C6C6F20202020202020202020
AES OUTPUT BLOCK = 7FB72E7BE929223D001E3129DFFA20A0
```

Note that, in this instance where the message is shorter than the block size, the resulting output is completely different in each case.

Cipher-Feedback (CFB), Output-Feedback (OFB) and counter (CTR) modes

If you encrypt using Cipher-Feedback (CFB) or Output-Feedback (OFB) or counter (CTR) modes then the ciphertext will be the same size as the plaintext and so padding is not required. Be careful, though, when using these modes, because the initialisation vectors (IV) *must* be unique.

Similarly, encrypting using a stream cipher like RC4 or PC1 does not require padding.

When don't I need to use padding?

If your plaintext is *always* an exact multiple of the block length *and* both the sender and recipient agree, then you don't need to use padding.

Example

You need to communicate a packet of information that is always the same format:-

Field	Length	Example
Customer reference number:	5 digits	12345
Credit card number:	15 digits	376066666655555
Expiry date:	4 digits	1205
Total:	24 digits	

This is always 24 bytes long, so if we are encrypting with DES or Triple DES or Blowfish, our data is always a fixed multiple of the block size. There is no need to add any padding to this before encrypting, provided that you and the recipient agree not to.

Example using Triple DES with key
`0x0123456789ABCDEFFEDCBA987654321089ABCDEF01234567`

```
ASCII PLAINTEXT:   123453760666666555551205
3DES INPUT BLOCKS: 12345376 06666665 55551205
INPUT IN HEX: 3132333435333736 3036363636363635 3535353531323035
3DES OUTPUT:  7ADE45981580DB32 421E3D90B5B47D5B 1175FA3DD8B932D7
```

Random padding

If you are transmitting messages that are shorter than the block length and encrypting using ECB mode, your ciphertext will be identical for identical messages. One solution is to use a variant of method 3 but use random padding bytes instead of nulls.

Alternatively, if the size of your encrypted message might give some information away, add a random number of random padding bytes to the message.

Using random padding with identical messages

Our plaintext message is frequently just "buy". If we encrypt using DES in ECB mode with the key `0xfedcba9876543210` using the RFC3369 method of padding we will always end up with the same result:

```
ACSII PLAINTEXT: buy
(IN HEX):        627579
INPUT BLOCK:     6275790505050505
KEY:             FEDCBA9876543210
OUTPUT BLOCK:    C2666697D381CA11
```

We could use a variant of our method 3 above using random padding bytes instead of nulls. The final byte tells us how many bytes to strip after deciphering. The following two examples use four random padding bytes plus 0x05 and produce completely different results.

```
ACSII PLAINTEXT: buy
(IN HEX):        627579
INPUT BLOCK:     62757958B3989B05
KEY:             FEDCBA9876543210
OUTPUT BLOCK:    061FF118B96F4EE8

ACSII PLAINTEXT: buy
(IN HEX):        627579
INPUT BLOCK:     62757934297CC805
KEY:             FEDCBA9876543210
OUTPUT BLOCK:    AAE42B7527A9078A
```

A receiving party with the correct key could derive the original plaintext from any such message. An attacker is now faced with solving many more alternatives.

Note that this will only work for a short message. You can achieve the same security by using CBC mode with a unique Initialisation Vector (IV), but you then have the overhead of having to transmit the IV as well as the ciphertext.

Using random padding to disguise the size of the message

As a very contrived example, let's say we might be sending two alternative messages that read:

```
Offer $90000.00
OR
Offer $1000000.00
```

Let's say we use the PKCS#5/RFC3369 method of padding when we encrypt these two messages using Blowfish in CBC mode with 128-bit key `0x0123456789ABCDEFF0E1D2C3B4A59687` and IV `0xFEDCBA9876543210`.

```
ASCII PT: Offer $90000.00
(IN HEX): 4F666665722024393030302E3030
INPUT:    4F666665722024393030302E303001
OUTPUT:   33BEF550BADE4798DDA5C960E2C70EB9

ASCII PT: Offer $1000000.00
```

```
(IN HEX): 4F666665722024313030303030302E3030
INPUT:    4F666665722024313030303030302E303007070707070707
OUTPUT:   A4B8D1BF3020DB24CDD459BAB6A7BA7B02AC39EE7C1BF090
```

An eavesdropper who knows that the message may contain either $90,000 or $1,000,000 could work out the answer just from the length of the ciphertext.

One solution is to add a random number of random bytes as padding, and to indicate the number of random bytes added by the last byte in padded input block.

```
ASCII PT: Offer $90000.00
(IN HEX): 4F666665722024393030303030302E3030
INPUT:    4F666665722024393030303030302E303012441C0D5E2C60147DF54910B6A6445311
OUTPUT:   33BEF550BADE4798B164164E571A5266B0D488FAD934D6386494FAF528C8ED82

ASCII PT: Offer $1000000.00
(IN HEX): 4F666665722024313030303030302E3030
INPUT:    4F666665722024313030303030302E3030CEF8302A84BA07
OUTPUT:   A4B8D1BF3020DB24CDD459BAB6A7BA7BC01DF3FCC3B7DC1B
```

This particular convention of using only the last byte limits you to 255 random padding bytes. It may be more practical to encode your plaintext with a header that indicates the number of valid bytes that follow plus a much larger random number of random padding bytes and then encrypt that.

So when do I use padding and when don't I?

- As a general rule, if you as a programmer need to encrypt data of variable length that you can't control, use padding with CBC mode. To avoid any ambiguity, make it standard practice *always* to add padding.
- If your plaintext data is always a fixed length, you can avoid using padding.
- If you use CFB or OFB modes of encryption, or use a stream cipher like RC4 or PC1, then you don't use padding.

Note that Bruce Schneier in [SCHN] recommends that best way to encrypt files is generally to use the CBC mode of enciphering - see section 9.11. In their more recent book [FERG], Ferguson and Schneier prefer the counter (CTR) mode.

References

- **[BCMO]** NIST Special Publication 800-38A *Recommendations for Block Cipher Modes of Operation, Methods and Techniques*, Morris Dworkin, December 2001.
- **[CMS]** RFC 3852 *Cryptographic Message Syntax (CMS)*, R. Housley, July 2004 (obsoletes RFC 3369 and RFC 2630).
- **[FERG]** Niels Ferguson and Bruce Schneier, *Practical Cryptography*, John Wiley, 2003.
- **[FIPS46]** Federal Information Processing Standard (FIPS) 46-3, *Data Encryption Standard (DES)*, U.S. Department Of Commerce/National Institute of Standards and Technology, 25 October 1999.
- **[FIPS74]** Federal Information Processing Standard 74 (FIPS PUB 74), *Guidelines for Implementing and Using the NBS Data Encryption Standard*, U.S. Department Of Commerce/National Institute of Standards and Technology, 1 April 1981.

- **[FIPS81]** Federal Information Processing Standard (FIPS 81), *DES Modes of Operation*, U.S. Department Of Commerce/National Institute of Standards and Technology, 2 December 1980.
- **[FIPS197]** Federal Information Processing Standards Publication FIPS PUB 197 *Advanced Encryption Standard (AES)*, U.S. Department Of Commerce/National Institute of Standards and Technology, 26 November 2001.
- **[MENE]** Menezes, van Oorschot and Vanstone, *Handbook of Applied Cryptography*, CRC Press LLC, 1997.
- **[NZEDI]** New Zealand Customs Service, *EDI Message Implementation Guidelines for Customs Declarations*, 15 November 1999.
- **[PKCS5]** PKCS #5, *Password-Based Encryption Standard*, RSA Laboratories, Version 2.0, March 1999.
- **[PKCS7]** PKCS #5, *Cryptographic Message Syntax Standard*, RSA Laboratories, Version 1.5, November 1993.
- **[RANK]** Rankl, W and W. Effing, *Smart Card Handbook*, John Wiley, 1997.
- **[SCHN]** Bruce Schneier, *Applied Cryptography - Protocols, Algorithms and Source Code in C*, second edition, John Wiley, 1996.
- **[STAL]** William Stallings, *Cryptography and Network Security: Principles and Practice*, 2nd edition, Prentice Hall 1998, ISBN 0138690170 (3rd edition 2002, ISBN 0130914290).
- **[TMOVS]** NIST Special Publication 800-20 *Modes of Operation Validation System for the Triple Data Encryption Algorithm (TMOVS): Requirements and Procedures*, April 2000.

Other Information

- An introduction to using keys in cryptography.
- Ciphertext is not text! - Storing and representing ciphertext.
- Cross-platform encryption.
- Encryption with international character sets.
- Encrypting variable-length strings with a password.

Contact

To comment on this page or ask a question, please send us a message.

This page last updated 28 January 2013

Comments

10 comments so far

[Comments are now closed]

> Great explanation. Especially about advises about; "So when do I use padding and when don't I?"
>
> ***PublicDinamicPi*** | *Spain_Codebreakers - Sun, Jan 30 2011 13:53 GMT*
>
> Liked a lot the text, very clear!
>
> ***LuKe*** | *Braszil - Wed, Apr 27 2011 16:29 GMT*

Really great article. The reason I found this page was one quesiton I did not see answered. What should the convention be if your unpadded data does fit an exact blocksize. Since you agree to use padding, depending on those last few bytes, it could be misread as actual padding bytes instead of real data. I imagine the solution is to add a full block of straight padding following the real data. Is that a reasonable approach?

Encrpyt_Newb | *- Sat, Jul 30 2011 01:35 GMT*

Encrpyt_Newb: A very good point. Methods 1 to 4 use the convention that a padding block is *always* added, thus avoiding the problem you correctly foresee. With method 5 the convention is usually that no padding block is added, because this is usually used only for simple plain text where extra spaces at the end do not affect the main message.

See the section

http://www.di-mgt.com.au/cryptopad.html#whenandwhennot

Dave | *Moderator - Sun, Jul 31 2011 18:06 GMT*

Very helpful documentation, thanks on that. There is one thing I'm not sure about. The text says "Don't forget to check first that the number of characters to be stripped is between one and eight" for method 1. In my opinion the check should be between one and seven? Eight chars to strip would mean that the whole block had only padding data. So the user data was a multiple of the blocklength and no padding is needed at all?

ltc_user | *- Thu, Mar 1 2012 15:10 GMT*

ltc_user: See the reply to Encrypt_Newb above. With method 1 padding is *always* added. If the user data is an exact multiple of the blocklength then we add a whole extra padding block of length blocklength. This then needs to be stripped afterwards.

If you think about it, this provides an unambiguous method of padding, which is simple to program to remove when decrypting. Providing, of course, that both parties agree to use it. This is the method used in almost all approved encryption protocols.

Dave | *Moderator - Tue, Mar 6 2012 11:42 GMT*

Can you explain CSSM_PADDING_PKCS1 padding which is essentially PKCS1 v1.5 signature padding?

webie | *- Fri, Apr 20 2012 16:01 GMT*

webie: See our RSA page "Signing using PKCS#1v1.5":

http://www.di-mgt.com.au/rsa_alg.html#signpkcs1

Dave | *Moderator - Sat, Apr 21 2012 04:46 GMT*

Hi, nice posting,. So when we use CTR mode in AES, is there any things which we need to take care ourselves for this padding.

Suresh P | *INDIA - Fri, Jun 22 2012 08:06 GMT*

Suresh P: With CTR mode you *must* make sure the IV is unique for *every* message you send with a given key. If you ever send two messages with the same IV and key, it is a trivial matter for someone to break the ciphertext.

You can safely use a 16-byte random IV each time because the probability of two 16-byte random values clashing is infinitesimal, providing the generator is OK. Another way is to use a counter that *always* increments even if the system crashes (this is actually very hard to do in practice). Or you can combine say a 32-bit counter with some random bits.

Dave | *Moderator - Fri, Jun 22 2012 09:06 GMT*
