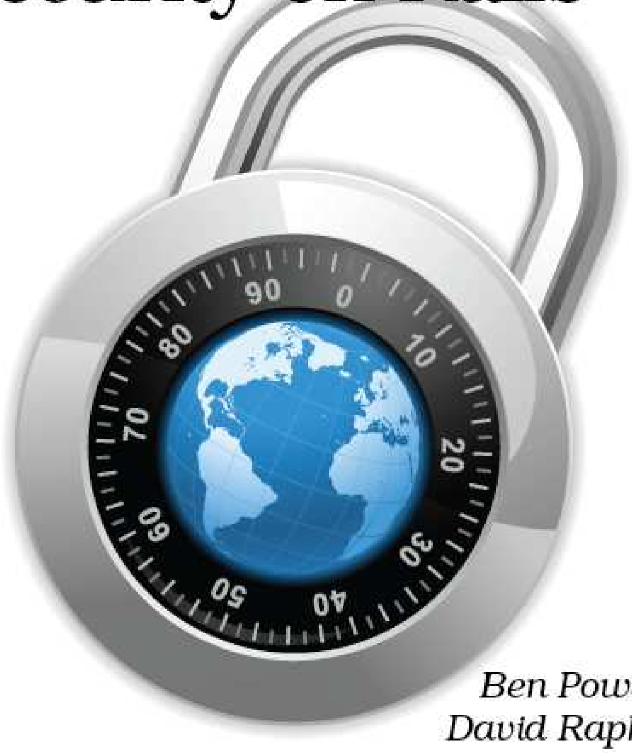
Extracted from:

Security on Rails

The Pragmatic Programmers

Security on Rails



Ben Poweski
David Raphael

Edited by Colleen Toporek

ISBN-10: 1-934356-48-4

ISBN-13: 978-1-934356-48-7

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-16

We can also employ a *salt*, which is a small amount of random data added to the original message before the digest function processes it. The salt is not a secret, and you need it to verify that a message matches the digest. We cover salts in depth in Section 6.2, *Adding Salts to the Passwords*, on page 104.

When Hashes Attack

Earlier we said the hash values produced for different inputs should themselves be different. This determines the quality of a hash function. When two inputs are found that return the same message digest, a *collision* is said to have occurred. We know collisions exist because of how hash functions are calculated; in fact, all hash functions have an infinite number of collisions! How can this be? Hash functions yield a fixed-length output with a variable-length input, so an infinitely large input divided by a finite number is infinity. Don't be alarmed just yet. The goal is to create a hash function where collisions can't occur. Collision attacks do exist for many hash functions in use today, but the context of their use plays a larger role in selection.

Choosing a Hash Function

We have only a handful of options when it comes to choosing a cryptographic hash function. If our application must interoperate with an existing application, the choice is typically made for us. For new development, the SHA family of hash digests is preferred. While recent developments signal a possible collision attack against the SHA1 algorithm, it remains the most widely accepted algorithm. It is also the current U.S. standard. That could (and probably will) change in the future, but the likely replacements are SHA-2 variants (SHA-224, SHA-256, SHA-384, or SHA-512).

We, the authors, believe you should avoid the MD5 algorithm. Many well-known collision attacks exist for MD5, and the cost of using the SHA family is minimal.

8.2 Encrypting Data

It's not uncommon to hear someone say, "We must encrypt this data." In most cases, the person who says it expects the person handling the encryption to use *block cipher* encryption. Block ciphers are black boxes that house the algorithms. We use block ciphers to encrypt information called the *plaintext*. The plaintext is whatever we want to protect. Block ciphers operate on a fixed length of data, called a *block*.

Many variable length block ciphers exist, but they are not the norm.

Types of Cryptography

For the purposes of the book, we can divide cryptography into two main groups: symmetric and asymmetric. Each group has its own strengths and weaknesses, and neither group is always better than the other.

Symmetric cryptography

Symmetric cryptography gets its name from the way the same key is used to encrypt and decrypt the information protected. We refer to encrypted information as the *ciphertext*.

The following example illustrates basic usage of the OpenSSL module in the Ruby standard library:

Download data_protection/aes_test.rb

```
#!/bin/env ruby
require 'openssl'

plain_text = "To the limit!"
algorithm = "AES-128-CBC"
aes = OpenSSL::Cipher::Cipher.new(algorithm)
key = aes.random_key

puts %(clear plain_text:    "#{plain_text}")
puts %(symmetric key: "#{key.inspect}")
puts %(cipher alg:    "#{algorithm}")

aes.encrypt(key)
cipher_text = aes.update(plain_text)
cipher_text << aes.final
puts %(encrypted plain_text: #{cipher_text.inspect})

# clear the aes cipher
aes.reset

# decrypt and display output
aes.decrypt(key)
out = aes.update(cipher_text)
out << aes.final
puts %(decrypted plaintext: "#{out}")
```

```
data_protection> ruby aes_test.rb
clear plain_text:    "To the limit!"
symmetric key: ""\204\016<#p\216\310;\256\227\366\2064\017#\355""
cipher alg:    "AES-128-CBC"
encrypted plain_text: "\376KS\001I\020\317\307i\202)\271+\0225K"
decrypted plaintext: "To the limit!"
```

Joe Asks...

Why Should I Use an OpenSSL random_key?

The question that really needs to be answered is this: why should we use the OpenSSL::Cipher.random_key() class instead of the Kernel::srand() class? The OpenSSL library uses a pseudorandom number algorithm designed specifically for cryptography. Producing a cryptographically random number generator is a nontrivial, computationally intensive task. Ruby's Kernel::srand() class is better than many others due to its underlying Mersenne Twister algorithm, but our goal was never to replace a true cryptographically random number generator. In short, we should use OpenSSL because it provides an easy way to defend against attacks that rely on sequence predictability.

This code begins by instantiating the OpenSSL::Cipher::Cipher class, passing the algorithm, key length, and block mode to the initialization() method. In our example, we use AES, 128-bit key length, and block mode *cipher block chaining*.

We can generate a random key from an initialized Cipher object by invoking the random_key() method. We can adjust the length of the key based upon the key length value passed to the initialize() method. Encryption and decryption occur by triggering the cipher mode with the encrypt(key) and decrypt(key) methods. Once we put the cipher into a mode, we call the update() method, passing it the plaintext or ciphertext.

The next example illustrates how to use an *initialization vector*, or IV. An initialization vector is an array of bits used to introduce additional randomness to the ciphertext stream:

Download data_protection/aes_iv_test.rb

```ruby
#!/bin/env ruby
require 'openssl'

plain_text = "To the limit!"
algorithm = "AES-128-CBC"
aes = OpenSSL::Cipher::Cipher.new(algorithm)
key = aes.random_key
iv = aes.random_iv

```

```
10  puts %(clear plain_text:    "#{plain_text}")
 -  puts %(symmetric key: #{key.inspect})
 -  puts %(initialization vector: "#{iv.inspect}")
 -  puts %(cipher alg:    "#{algorithm}")
 -
15  aes.encrypt(key, iv)
 -  cipher_text = aes.update(plain_text)
 -  cipher_text << aes.final
 -  puts %(encrypted plain_text: #{cipher_text.inspect})
 -
20  # clear the aes cipher
 -  aes.reset
 -
 -  # decrypt and display output
 -  aes.decrypt(key, iv)
25  out = aes.update(cipher_text)
 -  out << aes.final
 -  puts %(decrypted plaintext: "#{out}")
```

```
data_protection> ruby aes_iv_test.rb
clear plain_text:    "To the limit!"
symmetric key: "]\364\323T\317\377\216\264\221\216b\314;\237\021\267"
initialization vector: ""\004\332\235\320\315\250K\244\227}\232`\326_\237H""
cipher alg:    "AES-128-CBC"
encrypted plain_text: "\3162jqS\224\305B\347:\267\350><N\203"
decrypted plaintext: "To the limit!"
```

You can see the only significant change between these two examples on line 15. To use an initialization vector, we must pass it as the second parameter to the encrypt() and decrypt() methods of the Cipher class.

When using block ciphers that support initialization vectors, we must take care to use those vectors! Initialization vectors add additional strength to the ciphertext, and they can help reduce the amount of information that can be gleaned from the encrypted information. For example, if we did not use an initialization vector and two plaintexts were the same, the ciphertext of each message would be the same. This might not pose an immediate problem, but it could become a serious issue if someone learned one of the plaintext values.

Keep in mind the initialization vector is not a secret. It's an array of bytes required to decrypt the message. If we were to lose this information, we would lose any means of decrypting the message. We should store our initialization vector with the ciphertext. Some developers choose to prepend the IV to the ciphertext, while others prefer to create a separate field to hold the value. Each approach works, but prepending the value to the ciphertext confers a slight advantage because an attacker would need to determine its location.

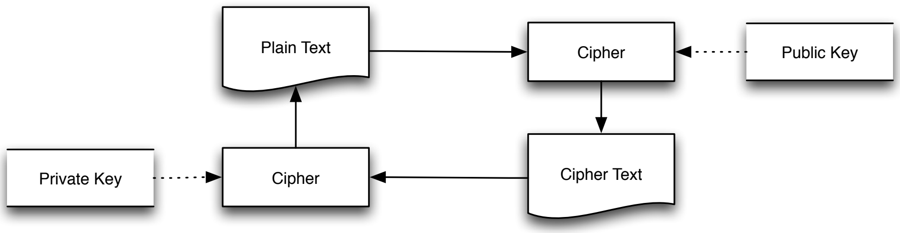


Figure 8.3: Asymmetric encryption

We should use symmetric encryption when we face each of the following situations:

- We need to encrypt and decrypt large amounts of data in a timely manner. Symmetric cryptography is an order of magnitude faster than asymmetric cryptography.
- We do not need key distribution because the same application encrypts and decrypts the information.
- Key exchange can occur out of band, in an secure manner.

Asymmetric cryptography

Asymmetric cryptography uses two keys. Each key can be used to decrypt ciphertexts, as long as the other key was used to encrypt it (see Figure 8.3). This approach enables bidirectional message communication without divulging the private key. This is a remarkable concept. Key distribution is a serious pain, and asymmetric cryptography can help us alleviate it.

Let's look at an example that uses the RSA algorithm to encrypt and decrypt a message:

Download data_protection/rsa_test.rb

```ruby
#!/bin/env ruby
require 'openssl'
include OpenSSL

rsa = PKey::RSA.generate(512)
txt = "Chunky Bacon"
puts "Plain Text: #{txt}"

```

```
 -  cipher_text_with_priv = rsa.private_encrypt(txt)
10  cipher_text_with_pub = rsa.public_encrypt(txt)
 -
 -  puts "== Plain Text Decrypted With Public Key =="
 -  puts pt_with_pub = rsa.public_decrypt(cipher_text_with_priv)
 -
15  puts "== Plain Text Decrypted With Private Key =="
 -  puts pt_with_priv = rsa.private_decrypt(cipher_text_with_pub)
```

```
data_protection> ruby rsa_test.rb
Plain Text: Chunky Bacon
== Plain Text Decrypted With Public Key ==
Chunky Bacon
== Plain Text Decrypted With Private Key ==
Chunky Bacon
```

On line 5, we instantiate an RSA public key. Next, on lines 9 and 10, we generate two ciphertexts using the private and public keys. Finally, on lines 13 and 16, we decrypt each ciphertext using the opposite key.

Choosing the right cryptography

So far we've learned about symmetric block ciphers like AES and asymmetric algorithms like RSA. Next, we need to determine when to use a specific algorithm and the consequences of choosing one over the other.

We can choose from several well-respected algorithms for symmetric block ciphers. The current number-one choice is the *advanced encryption standard*, or *AES*. AES was established using a contest where various groups submitted proposals, and the top submission was chosen. This approach contrasts distinctly with a design-by-committee approach, where any specific vision is normalized to the lowest common denominator. An algorithm known as *Rijndael* was selected as the winner, and it now bears the name AES. AES is the current U.S. government standard for symmetric encryption, and it is ideal for cases where third parties do not need to send or receive encrypted data.

If choosing the same algorithm the U.S. government uses makes you uncomfortable, don't fret; you have many other options. *Serpent* was another AES finalist, as was the *Twofish* algorithm. These algorithms currently represent a sound approach, but new attacks are being developed constantly, so what's sound for now might not hold true for the future. The *DES* algorithm is a good example of this; attacks targeting the DES algorithm have rendered it undesirable, except in the case of backwards compatibility.

Joe Asks...

Why Is ECB Undesirable?

From the previous examples, you might have noticed some extra information passed into the initialize() method of the Cipher classes. These values instruct the cipher how to handle messages that grow beyond a single block. Block sizes are typically in the eight-byte range, so this situation occurs frequently:

```
$ ruby ecb_versus_cbc.rb
aes with mode cbc
"\267X\254lI\216M\021\255\300`\"\251\220\272\243"
"\036\237\234\224p\324D{i\036\266\342MTiV"
"\203r\246\232\220\301=\205\267cN\314\376n?."
"\331\v\225\024j\301\310\037l(\222>\246\274\230T"
"s\213n\212\270\232\022\230v\327\035\215\237\eZ\005"

aes with mode ecb
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"
"d\026\325\252d\305D\304\031H C\310\310\215{"
```

In our example we can see that the plaintext is identical, but the corresponding ciphertext is not. The output of each line corresponds to one AES block. The first group of lines displays a pseudo-random distribution of block contents; it used *cipher block chaining* to achieve that effect. The second example shows four equal consequence blocks within the cipher text; it appears this way because the cipher wasn't instructed to extend its algorithm across blocks. This is bad because it allows for known plaintext attacks or attacks where the part of the message is known. An example of this would be attacking protocols that contain standard headers included in the plaintext. By limiting the protection to a single block and knowing the order of certain parts of the plaintext, we can narrow drastically how much of the plaintext must be deciphered.

To sum up, *don't use ECB!*

If an application requires that you exchange messages with encrypted information, then managing symmetric keys will become a nightmare once any party involved is no longer trusted. This is where asymmetric, or public-key cryptography, shines. Unfortunately, using public-key cryptography is more difficult because it has significantly more going on within it. Given this fact, we must concede proper encryption is beyond the scope of this book. That said, both RSA and DSA serve as excellent starting points for encryption.

8.3 Using Cryptography with ActiveRecord

Now that we're acquainted with the basics of using cryptography with Ruby, it's time to improve our data-protection implementation. One way to improve it: we can use extension techniques common to the construction of Rails.

ActiveRecord Encryptor

The book *Agile Web Development with Rails* [RTH08],includes an example of leveraging ActiveRecord hooks to provide transparent encryption and decryption of a model's properties. That example focuses on teaching developers how to use ActiveRecord callbacks, but it leaves out the finer details of using cryptography. Our next task is to provide an implementation for that book's example.

Add a security question

When authenticating users, it is important to provide additional factors. One example of this involves incorporating a security question to further challenge the user. Let's add a security question to LunchedIn that helps us authenticate a user who has lost her password.

We can run the following command to create a Migration class that builds the additional fields required for our security question:

```
$ ruby script/generate migration add_secret_user_question
```

Next add the following code to 011_add_secret_user_question.rb:

Download data_protection/lunchedin/db/migrate/011_add_secret_user_question.rb

```
class AddSecretUserQuestion < ActiveRecord::Migration
  def self.up
    add_column "users", "secret_question", :string
    add_column "users", "secret_answer", :string
  end
```