Extracted from:

Enterprise Recipes with Ruby and Rails

Store Passwords Securely

Problem

Believe me, even if you think you already know how to store passwords securely, you probably don't. There's a lot of folklore code wandering around the Internet, and most of it is wrong. In this recipe, you'll learn what the biggest threats to your passwords are and how to store them the right way.

Ingredients

- Install the *bcrypt-ruby*[18] gem (at the time of this writing, it is not available for the Windows platform):

  ```
  $ gem install bcrypt-ruby
  ```

Solution

Let's say we have a User model that is represented in the database as follows:

Download security/bcrypt_demo/db/migrate/20080803070736_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :name
      t.string :hashed_password

      t.timestamps
    end
  end

  def self.down
    drop_table :users
  end
end
```

Admittedly, it's rather simplistic, but it's sufficient for demonstration purposes: our users have a name and a password. At least most people know that they should never store passwords as plaintext, so usually

18. http://bcrypt-ruby.rubyforge.org/

passwords are run through a mathematical one-way function such as MD5 or SHA1. These algorithms produce a hash value (also called a *fingerprint*). In other words, the same input value always results in the same output value, and you should not be able to deduce the input value from the output value. Instead of storing the password itself, you store only its hash value.

If a user tries to log in now, she sends her username and password to the application as plaintext (over a secure network connection such as HTTPS, of course). Then the server calculates the password's hash value and compares it to the hash value that has been stored in the database. If they are equal, the password is correct. Otherwise, it's not.

The biggest security threat is that someone gets a copy of all usernames and their according password hashes, because in the worst case (that is, if you did not store your passwords really securely) the attacker could derive the original passwords from the hash values. If, for example, you have hashed your passwords using MD5, this is easier than you think, because of *rainbow tables*. Simply put, these tables contain the MD5 hashes for all possible character sequences up to a certain length. Breaking a password is basically reduced to a table lookup.

To protect yourself from rainbow table attacks, you can add a little bit of random information, called *salt*, to every password before you turn it into a hash value. This way, an attacker would need a new rainbow table for every single password. But that's still insufficient, because with today's computing power, it's actually possible to perform this kind of attack. Typical hash algorithms can be computed very quickly on a modern computer, and they can be calculated even faster on special devices that have become pretty cheap in the past few years.

Most of today's password-cracking tools aren't based on tables anymore; instead, they use sophisticated algorithms based on cryptanalysis and statistics. That is, if you want to make an attacker's life more difficult, you have to drastically increase the time needed to crack your passwords. This can be achieved by hashing your passwords not only once but several times and by adding a new random bit of salt for every iteration. Several algorithms are available for doing this. One of the most popular is *bcrypt*, which is used by OpenBSD for encrypting passwords, for example.[19]

19. You can find an excellent article explaining all this in detail at http://www.matasano.com/log/958/.

We use a *bcrypt* library for Ruby to add a secure password scheme to our User model:

Download security/bcrypt_demo/app/models/user.rb

```
Line 1  require 'bcrypt'
     -
     -  class User < ActiveRecord::Base
     -    def password
     5      @password ||= BCrypt::Password.new(self.hashed_password)
     -    end
     -
     -    def password=(new_password)
     -      @password = BCrypt::Password.create(new_password, :cost => 10)
    10      self.hashed_password = @password
     -    end
     -
     -    def self.authenticate(name, password)
     -      if user = self.find_by_name(name)
    15        user = nil if user.password != password
     -      end
     -      user
     -    end
     -  end
```

We define a virtual password attribute. That is, we can read and write it, but it is not stored in the database. Only the hashed password gets stored. In line 5, we implement the reader. If the password has been created already, we simply return it. Otherwise, we create a new BCrypt::Password object from the hashed password and return this. The Password class hides all the cryptographic details and provides some convenience methods that we will use later.

Our writer's implementation starts in line 9. Here we create a new Password object from a plaintext password that has been input by a user. The cost attribute allows us to control the security level of the password. The higher the cost value, the longer it takes to break the password. We store the hashed password in @password and in self.hashed_password, so it gets stored in the database, too. Note that we do not have to store a salt value separately.

Finally, we need an authenticate() method that actually checks whether a certain combination of username and password is valid. First we check whether the user exists in the database, and if the user does, we compare the password entered to the password that has been stored in the database in line 15. Because the Password class overrides the ==() operator, the code looks very elegant, doesn't it? Be assured: behind the scenes a lot of cryptography is performed.

Let's use our new User class on the Rails console:

```
mschmidt> ruby script/console
Loading development environment (Rails 2.1.0)
>> user = User.create(:name => 'Maik', :password => 't0p$ecret')
=> #<User id: 2, name: "Maik",
   hashed_password:
     "$2a$10$fveY1Zte2p37XsQ0tTtsYeUGLWRgJtWPx8zXYcuFle0Z...",
   created_at: "2008-06-30 13:22:14",
   updated_at: "2008-06-30 13:22:14">
>> User.authenticate('Maik', 'wrong password')
=> nil
>> User.authenticate('Maik', 't0p$ecret')
=> #<User id: 2, name: "Maik",
   hashed_password:
     "$2a$10$fveY1Zte2p37XsQ0tTtsYeUGLWRgJtWPx8zXYcuFle0Z...",
   created_at: "2008-06-30 13:22:14",
   updated_at: "2008-06-30 13:22:14">
>>
```

We created a new user named *Maik* who has the password *t0p$ecret*. As you can see, only a hashed version of the password has been stored. Then, we tried to authenticate ourselves using a wrong password. As expected, we've got nil as a result. Finally, we used the right password and got a User object back.

Although it's easy to use the *bcrypt* library directly, there is even a Rails plug-in named *acts_as_authentable*[20] for it.

Discussion

Whenever you are writing code related to security, you should be extremely cautious and skeptical. Always try to get the latest information available about security holes in all the tools and algorithms you're going to use. That's true for *bcrypt*, too.

At the moment, *bcrypt* is sufficient for most purposes, but it uses the *Blowfish* encryption algorithm[21] internally, which has been succeeded already by *Twofish*.[22] It's a good idea to look for alternative solutions as early as possible, and stronger hashing algorithms such as SHA-256 are interesting candidates.[23]

Your software can never be totally secure, but it should be as secure as possible.

20. http://code.google.com/p/acts-as-authentable/
21. http://en.wikipedia.org/wiki/Blowfish_(cipher)
22. http://en.wikipedia.org/wiki/Twofish
23. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html