Fedora Security Team

Secure Ruby Development Guide

Guide to secure software development in Ruby

Ján Rusnačko

Fedora Security Team Secure Ruby Development Guide
Guide to secure software development in Ruby
Edition 1

Author Ján Rusnačko jrusnack@redhat.com

This guide covers security aspects of software development in Ruby.

Environment

Development environment can significantly affect quality and security of code and investing certain effort into proper setup can result in saved development time, better code coverage, more readable and secure code etc. In general, automated checks provide a good baseline and are less prone to unintentional mistakes than developers.

1.1. Code quality metrics

Security is just one aspect of code quality along with reliability, correctness and others. These metrics overlap a lot, for example denial of service can be seen as both security and reliability issue. Therefore improvement in any of these areas is likely to affect others.

Increasing code quality by reducing complexity, duplication of code and mainaining good readability is a good first step towards security. All other things being equal, more complex code will have more weaknesses than simpler one.

Several gems can help with improving code quality:

- *Rails Best Practices*[1] is a popular gem among rails developers and new checks are implemented based on voting of community.
- *rubocop*[2] is a style checker and implements vast amount of checks based on *Ruby Style Guide*[3]
- *metric_fu*[4] combines several popular code metric tools like *Reek*[5], *Flog*[6], *Flay*[7], *Cane*[8] etc.

These are just few examples and actual setup may vary from project to project. However, they help developers keep code complexity low in an automated fashion and can be easily integrated into workflow.

1.2. Dependency management

Dependencies in form of gems can be another source of vulnerabilities in Ruby applications.

1.2.1. Outdated Dependencies

Bundler[9] is the de facto standard for managing Ruby application dependencies. Developer can specify required dependencies and their versions in Gemfile and bundler automatically resolves dependencies and prepares environment for application to run in. Bundler freezes exact versions of dependencies in Gemfile.lock and everytime this file is present, depencency resolution step is skipped and exact versions of gems from Gemfile.lock are installed.

Freezing versions of dependencies has a security impact. If a dependency is vulnerable and new version contains the fix, Gemfile.lock has to be updated. Detection of outdated versions of

[1] https://github.com/railsbp/rails_best_practices
[2] https://github.com/bbatsov/rubocop
[3] https://github.com/bbatsov/ruby-style-guide
[4] https://github.com/metricfu/metric_fu
[5] https://github.com/troessner/reek
[6] https://github.com/seattlerb/flog
[7] https://github.com/seattlerb/flay
[8] https://github.com/square/cane
[9] http://bundler.io/

dependencies is something that can be automated and several gems help with this using information provided by *rubysec-db*[10].

Rubysec[11] project maintains rubysec-db database of all security advisories related to Ruby libraries. This database covers most of the popular gems and provides data to identify vulnerable and patched versions of dependencies.

bundler-audit[12] is a gem maintainted by rubysec project that automatically scans Gemfile.lock and reports any unpatched dependencies or insecure sources.

gemsurance[13] also works on top of rubysec-db. Unlike bundler-audit it outputs html report and lists outdated gems as well. Another useful feature is possibility to integrate the check with RSpec and make your tests fail whenever vulnerable dependency is detected.

Other gems or services that provide similar functionality include *HolePicker*[14] and *gemcanary*[15].

Important

It is highly recommended to set up automated checks for outdated dependencies.

1.2.2. Vendoring dependencies

Another way of freezing dependencies is checking their source code into vendor folder in application. With bundler this practice becomes obsolete. Another, still valid, usecase is when dependency needs to be slightly modified to suit needs of application.

By checking the dependency into the application`s repository, developer takes responsibility of tracking bugs and vulnerabilities and updating vendored gems. However, backporting commits that fix security issues from upstream version will render automatic tools for checking dependencies useless, as they will rely on gem versions, which will not correspond with the vendored code.

1.2.3. Gem signing

Gem signing is already implemented in rubygems and is based on x509 certificates, even though discussion about future implementation is *ongoing*[16]. There is no PKI, so user who wants to verify gem`s integrity must explicitly download and trust certificate that was used to sign the gem. Establishing trust in certificate of party user has no prior relationship with over internet can be diffucult and unscalable.

[10] https://github.com/rubysec/ruby-advisory-db/
[11] http://www.rubysec.com
[12] https://github.com/rubysec/bundler-audit
[13] https://github.com/appfolio/gemsurance
[14] https://github.com/jsuder/holepicker
[15] https://gemcanary.com/
[16] https://github.com/rubygems-trust/rubygems.org/wiki

Important

Assuming user verified the certificate belongs to the developer it says, signature protects integrity of gem as it is distributed and gives user a mechanism to detect modifications of gem after it was signed.

However, signatures do not guarantee trustworthiness of gem author.

Developer can generate his private key and self signed certificate with:

```
$ gem cert --build <email address>
...
$ chmod 600 gem-private_key.pem gem-public_cert.pem
```

This command will generate self-signed 2048 bit RSA with SHA1 certificate (this configuration is currently hardcoded) stored in PEM format.

Important

Generated private key will not be passphrase protected, and it has to be encrypted manually:

```
$ openssl rsa -des3 -in <private key> -out <encrypted private key>
```

To sign the gem, following needs to be added to gemspec:

```
s.cert_chain = <path to public certificate>
s.signing_key = <path to private key> if $0 =~ /gem\z/
```

After building the gem, one can verify it has been signed with:

```
$ gem spec testgem-1.0.0.gem cert_chain
...
$ tar tf testgem-1.0.0.gem
data.tar.gz
metadata.gz
data.tar.gz.sig
metadata.gz.sig
```

1.2.3.1. Installation and policies

To make use of signatures in gems, user has to specify security policy during gem installation (it is turned off by default):

```
$ gem install -P HighSecurity testgem
```

There are 4 available security policies:

No policy
: Signed packages are treated as unsigned.

LowSecurity
: Still pretty much no security. Rubygems will make sure signature matches certificate and certificate hasn`t expired.

MediumSecurity
: For signed gems, signature is verified against certificate, certificate validity is checked and certificate chain is checked too. Packages from untrusted sources won`t be installed (user has to explicitly trust the cerficate, see below). Unsigned gems are installed normally.

HighSecurity
: Same as medium, but unsigned gems are not installed.

Warning

Since signatures protect integrity of gem as it`s being distributed from developer to users, the only policy with security impact is `HighSecurity`. With `MediumSecurity`, attacker can always intercept gem, strip signatures, modify it and serve users that accept unsigned gems.

To install signed gem under medium or high security policy, user has to download certificate from external source, verify it`s authenticity and explicitly add it to his local database of trusted certificates:

```
$ gem cert --add <certificate>
```

This command will store public certificate to **`~/.gem/trust`** directory. Name of the certificate will contain hexdigest of the subject of certificate, so if users adds another certificate with the same subject as one of the already trusted ones, original one will be overwritten without notice.

To avoid overwriting existing certificate, make sure subject of certificate being added is different from certificates that are already trusted:

```
$ openssl x509 -text -in <certificate> | grep Subject:
  Subject: CN=test, DC=example, DC=com
$ gem cert --list
...
```

Bundler supports gem signing and trust policies since version 1.3 and user can specify security policy during installation:

```
$ bundle install --trust-policy=HighSecurity
```

Warning

Gems that are installed by bundler from repository like

```
gem 'jquery-datatables-rails', git: 'git://github.com/rweng/jquery-datatables-rails.git'
```

bypass security policy, as they are not installed using **`gem`** command, but cloned into bundler folder.

A small gem **`bundler_signature_check`** can be used to check **`Gemfile`** and determine which gems are signed, with suggestion which security policy can be currently safely used (note that **`bundler_signature_check`** is signed and it`s dependencies **`bundler`** and **`rake`** are likely already installed, so `HighSecurity` can be used):

```
$ gem install -P HighSecurity bundler_signature_check
$ bundler_signature_check
...
```

1.2.3.2. References:

- *Rubygems Security page*[17]
- *Documentation of **`Gem::Security`** module*[18]
- Ben Smith`s Hacking with gems presentation *on youtube*[19]

1.3. Static code analysis with Brakeman

Brakeman[20] is a static code scanner for Ruby on Rails applications. It does not require any configuration and can be run out-of-the-box on source of rails application. It performs static code analysis, so it does not require rails application to be set up, but rather parses the source code and looks for common vulnerable patterns.

Brakeman gem is signed, but some of its dependencies are not, so to install run:

```
$ gem install -P MediumSecurity brakeman
```

To execute scan on application, run brakeman from rails application repository:

```
$ brakeman -o report.html --path <path to rails app>
```

[17] http://guides.rubygems.org/security/
[18] http://rubygems.rubyforge.org/rubygems-update/Gem/Security.html
[19] http://www.youtube.com/watch?v=z-5bO0Q1J9s
[20] http://brakemanscanner.org

The format of the output is determined by file extension or by **`-f`** flag. Currently supported formats are `html`,`json`,`tabs`, `csv` and `text`.

Brakeman output contains warnings in format

```
+------------+-------+--------+------------------+---------------------------------------+
| Confidence | Class | Method | Warning Type     | Message                               |
+------------+-------+--------+------------------+---------------------------------------+
| High       | Foo   | bar    | Denial of Service | Symbol conversion from unsafe String .. |
```

As static code scanner Brakeman does not analyze the behaviour of code when run and lacks execution context (e.g. it does not know about dead code that`s never executed). Therefore Brakeman output usually contains also false warnings. There are 3 confidence levels to help developers determine possible false warnings and prioritize when reviewing the output: `High`, `Medium` and `Weak`.

1.3.1. Continuous integration

Good way to use Brakeman is to integrate it into workflow of a project and fix the reported problems before they are committed into repository.

Creating a rake task is easy with

```
$ brakeman --rake
```

which creates file **`lib/tasks/brakeman.rake`**

Another useful options is to create a configuration file from a command line options:

```
$ brakeman -C <config file> <options>
```

which can be later used:

```
$ brakeman -c <config file>
```

Very useful feature is comparison with older scan result and outputting only difference between reports - developers can then easily identify warnings that were just added or fixed:

```
$ brakeman --compare <old result in json> -o <output in json>
```

The output is always in json (**`-f`** is ignored).

1.3.2. Reducing number of false warnings

There are several ways to reduce number of false warnings, most of which can be dangerous. Reducing number of false warnings might be meaningful when Brakeman is adopted by an existing project - in such cases initial report can be overwhelming and ignoring warnings that are likely to be false can be crucial. However, this shall be considered only temporary solution.

Important

Reduction of false warnings by skipping certain checks or ignoring certain files is dangerous. Even if all currently reported warnings are false, future commits might introduce flaws that would otherwise be reported. This greatly reduces effectiveness of Brakeman and its value for project.

One way to reduce number of warnings is to set minimum confidence level:

```
$ brakeman -w <level>
```

where level 1 indicates Weak confidence, level 2 Medium and 3 High confidence.

Another option is to specify list of safe methods:

```
$ brakeman -s <comma separated list of methods>
```

This will add methods to the set of known safe methods and certain checks will skip them without producing a warning. For example, Cross site scripting checker maintains a set of methods which produce safe output (it contains methods like **`escapeHTML`**) and safe methods specified as command line argument are added to the list.

You can skip processing **`lib`** directory and/or specify files to be skipped:

```
$ brakeman --skip-libs
$ brakeman --skip-files <comma separated list of files>
```

Language features

2.1. Tainting and restricted code execution

Ruby language includes a security mechanism to handle untrusted objects and restrict arbitrary code execution. This mechanism consists of two parts: first is an automated way of marking objects in Ruby as coming from untrusted source, called tainting. The second part is mechanism for restricting code execution and prevents certain potentially dangerous functions being executed on tainted data. Ruby interpreter can run in several safe levels, each of which defines different restrictions.

This mechanism (especially restricted code execution) is implementation specific and is not part of Ruby specification. Other Ruby implementations such as Rubinius and JRuby do not implement safe levels. However, taint flag is part of the rubyspec.

2.1.1. Object.tainted?

Each object in Ruby carries a taint flag which marks it as originating from unsafe source. Additionally, any object derived from tainted object is also tainted. Objects that come from external environment are automatically marked as tainted, which includes command line arguments (`ARGV`), environment variables (`ENV`), data read from files, sockets or other streams. Environment variable `PATH` is exception: it is tainted only if it contains a world-writable directory.

To check whether object is tainted and change taintedness of object, use methods **`Object.tainted?`**, **`Object.taint`** and **`Object.untaint`**:

```
>> input = gets
exploitable
=> "exploitable\n"
>> input.tainted?
=> true
>> input.untaint
=> "exploitable\n"
>> input.tainted?
=> false
```

Note

Literals (such as numbers or symbols) are exception: they do not carry taint flag and are always untainted.

2.1.2. Object.untrusted?

At higher safe levels (see safe level 4 below) any code is automatically untrusted and interpreter prevents execution of untrusted code on trusted objects. In Ruby 1.8, taint flag is also used to mark objects as untrusted, so untrusted code is not allowed to modify untainted objects. In addition, any object created by untrusted code is tainted. This effectively allows to sandbox an untrusted code, which will not be allowed to modify "trusted" objects.

Mixing taint and trust of object has serious drawback - untrusted code is allowed to modify all tainted objects (even if they come from trusted code).

Ruby 1.9 adds another flag to each object to mark it as untrusted. Untrusted code is now allowed only to modify untrusted objects (ignoring taint flag), and objects created by untrusted code are automatically marked as untrusted and tainted. To check and modify trust flag use methods **`Object.untrusted?`**, **`Object.untrust`** and **`Object.trust`**.

However, Ruby 2.1 deprecates trust flag and the behaviour of above methods is the same as **`Object.tainted?`**, **`Object.taint`** and **`Object.untaint`**. This change comes together with removal of safe level 4, which makes trust flag useless (see *issue on ruby-lang*[1] or read below).

2.1.3. $SAFE

Ruby interpreter can run in restricted execution mode with several levels of checking, controlled by global variable $SAFE. There are 5 possible levels: 0,1,2,3,4 with 0 being default safe level. $SAFE is thread-local and its value can only be increased (at least in theory - in practice there are well known ways how to work around restricted code execution or decrease a safe level. See *Section 2.1.3.1, "Security considerations of $SAFE"*). Safe level can be changed by assigning to $SAFE or with **`-T<level>`** argument.

Safe levels have following restrictions:

level 0
: strings from streams/environment/ARGV are tainted (default)

level 1
: dangerous operations on tainted values are forbidden (such as **`eval`**, **`require`** etc.)

level 2
: adds to the level 1 also restrictions on directory, file and process operations

level 3
: in addition all created objects are tainted and untrusted

level 4
: code running in this level cannot change trusted objects, direct output is also restricted. This safe level *is deprecated*[2] since Ruby 2.1

There is a lack of documentation of what is restricted in each safe level. For more exhausting description refer to *Programming Ruby: Pragmatic programmer`s guide*[3].

2.1.3.1. Security considerations of $SAFE

Design of restricted code execution based on $SAFE is inherently flawed. Blacklist approach is used to restrict operation on each level, which means any missed function creates a vulnerability. In past several security updates were related to restricted code execution and taint flag (see *CVE-2005-2337*[4], CVE-2006-3694, *CVE-2008-3655*[5], *CVE-2008-3657*[6], *CVE-2011-1005*[7], *CVE-2012-4464*[8],*CVE-2012-4466*[9] and *CVE-2013-2065*[10]).

[1] https://bugs.ruby-lang.org/issues/8468
[2] https://bugs.ruby-lang.org/issues/8468
[3] http://ruby-doc.com/docs/ProgrammingRuby/
[4] https://www.ruby-lang.org/en/news/2005/10/03/ruby-vulnerability-in-the-safe-level-settings/
[5] https://www.ruby-lang.org/en/news/2008/08/08/multiple-vulnerabilities-in-ruby/
[6] https://www.ruby-lang.org/en/news/2008/08/08/multiple-vulnerabilities-in-ruby/
[7] https://www.ruby-lang.org/en/news/2011/02/18/exception-methods-can-bypass-safe/
[8] https://www.ruby-lang.org/en/news/2012/10/12/cve-2012-4464-cve-2012-4466/
[9] https://www.ruby-lang.org/en/news/2012/10/12/cve-2012-4464-cve-2012-4466/

Warning

Design of restricted code execution based on $SAFE is inherently flawed and cannot be used to run untrusted code even at the highest safe level. It must not be used as mechanism to create a secure sandbox, as attacker will be able to work around the restrictions or decrease safe level.

One example of how exploitable the design is comes from *CVE-2013-2065*[11]:

```
require 'fiddle'

$SAFE = 1
input = "uname -rs".taint
handle = DL.dlopen(nil)
sys = Fiddle::Function.new(handle['system'], [Fiddle::TYPE_VOIDP], Fiddle::TYPE_INT)
sys.call DL::CPtr[input].to_i
```

Even though safe level 1 should restrict execution of system commands, this can be bypassed using Fiddle library, which is an extension to translate a foreign function interface with Ruby. Exploit above bypasses safe level by passing input to system call as numeric memory offset. Since numbers as literals cannot be tainted, code cannot check taintedness of input.

Note

However, running application with higher safe level is still useful for catching unintended programming errors, such as executing **`eval`** on tainted string.

2.2. Dangerous methods

Ruby contains number of methods and modules that should be used with caution, since calling them with input potentially controlled by attacker might be abused into arbitrary code execution. These include:

- **`Kernel#exec`**, **`Kernel#system`**, backticks and **`%x{...}`**
- **`Kernel#fork`**, **`Kernel#spawn`**
- **`Kernel#load`**, **`Kernel#autoload`**
- **`Kernel#require`**, **`Kernel#require_relative`**
- **`DL`** and **`Fiddle`** module
- **`Object#send`**, **`Object#__send__`** and **`Object#public_send`**
- **`BasicObject#instance_eval`**, **`BasicObject#instance_exec`**

[10] https://www.ruby-lang.org/en/news/2013/05/14/taint-bypass-dl-fiddle-cve-2013-2065/
[11] https://www.ruby-lang.org/en/news/2013/05/14/taint-bypass-dl-fiddle-cve-2013-2065/

- **`Module#class_eval`**, **`Module#class_exec`**, **`Module#module_eval`**, **`Module#module_exec`**
- **`Module#alias_method`**

2.3. Symbols

Symbols in MRI Ruby are used for method, variable and constant lookup. They are implemented as integers so that they are faster to look up in hastables. Once symbol is created, memory allocated for it is never freed. This creates opportunity for attacker: if he is able to create arbitrary symbols, he could flood the application with unique symbols that will never be garbage collected. Memory consumption of Ruby process would only grow until it runs out of memory, resulting in Denial of Service attack.

Application developers should be careful when calling **`to_sym`** or **`intern`** on user-supplied strings. Additionally, other methods may convert supplied arguments to symbols internally, for example **`Object.send`**, **`Object.instance_variable_set`**, **`Object.instance_variable_get`**, **`Module.const_get`** or **`Module.const_set`**:

```
>> Symbol.all_symbols.size
=> 2956
>> Module.const_get('MY_SYMBOL')
NameError: uninitialized constant Module::MY_SYMBOL
>> Symbol.all_symbols.size
=> 2957
```

Array of all currently defined symbols is available through **`Symbol.all_symbols`** class method.

Starting from Ruby 2.0, method **`rb_check_id`** is available to Ruby C extensions, which returns 0 when String passed as argument is not already defined as Symbol. This makes overriding default intern methods possible.

SafeIntern[12] gem makes use of **`rb_check_id`** and provides a patch for **`to_sym`** or **`intern`** methods of String. When the conversion from String to Symbol would define a new Symbol, either nil is returned or exception raised. Such approach prohibits creating any new Symbols other than those that are already defined by the application. In case the string is trusted, new symbol can be created by calling **`intern(:allow_unsafe)`**.

Ruby 2.2 *released 25th of Dec 2014*[13] includes substantial improvements to it`s garbage collector, on of which is ability to garbage collect Symbols. This solves denial of service problems with Symbols present in previous versions of Ruby.

2.4. Serialization in Ruby

Deserialization of untrusted data has been on the top of critical vulnerabilities in 2013 (prominent examples are deserialization issues found in Ruby on Rails, see *CVE-2013-0156*[14], *CVE-2013-0277*[15] or *CVE-2013-0333*[16]). There are several ways how to serialize objects in Ruby:

[12] https://github.com/jrusnack/safe_intern
[13] https://www.ruby-lang.org/en/news/2014/12/25/ruby-2-2-0-released/
[14] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0156
[15] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0277
[16] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0333

2.4.1. Marshal.load

`Marshal.dump` and **`Marshal.load`** can serialize and deserialize most of the classes in Ruby. If application deserializes data from untrusted source, attacker can abuse this to execute arbitrary code. Therefore, this method is not suitable most of the time and should never be be used on data from unstrusted source.

2.4.2. YAML.load

YAML is a popular serialization format among Ruby developers. Just like **`Marshal.load`** it can be used to deserialize most of the Ruby classes and also should never be used on untrusted data.

2.4.2.1. SafeYAML

Alternative approach is taken by *SafeYAML*[17] gem - by default it allows deserialization of only few types of objects that can be considered safe, such as `Hash`, `Array`, `String` etc. When application requires serialization of certain types, developer can explicitly whitelist trusted types of objects:

```
SafeYAML.whitelist!(FrobDispenser, GobbleFactory)
```

This approach is more versatile, since it disables serialization of unsafe classes, yet allows developer to serialize know benign object. Requiring `safe_yaml` will patch method **`YAML.load`**.

2.4.3. JSON.parse and JSON.load

JSON format supports only several primitive data types such as strings, arrays, hashes, numbers etc. This certainly limits the attack surface, but it should not give developer false sense of security - one example is *CVE-2013-0333*[18] vulnerability in Ruby on Rails, when parser used for deserialization of JSON data actually converted data to a subset of YAML and used **`YAML.load`** to deserialize.

However, it is possible to extend Ruby classes to be JSON-dumpable:

```
class Range
def to_json(*a)
  {
    'json_class'   => self.class.name,
    'data'         => [ first, last, exclude_end? ]
  }.to_json(*a)
end

def self.json_create(o)
  new(*o['data'])
end
end
```

This will allow instances of Range class to be serialized with JSON:

```
>> (1..10).to_json
=> "{\"json_class\":\"Range\",\"data\":[1,10,false]}"
```

[17] http://danieltao.com/safe_yaml/
[18] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0333

During deserialization, JSON gem will try to look up class referenced by "json_class", which might create new Symbol if the class does not exist, possibly allowing Denial of Service (see *Section 2.3, "Symbols"*):

```
>> Symbol.all_symbols.size
=> 3179
>> JSON.parse('{"json_class":"NonexistentClass"}')
ArgumentError: can't get const NonexistentClass: uninitialized constant NonexistentClass
>> Symbol.all_symbols.size
=> 3180
```

To disable this, `:create_additions => false` option can be passed as second argument:

```
>> JSON.parse('{"json_class":"NonexistentClass"}',:create_additions => false)
=> {"json_class"=>"NonexistentClass"}
```

This behaviour has changed in response to *CVE-2013-0269*[19] and **`JSON.parse`** now defaults to `:create_additions => false`. However, default behaviour has not changed for **`JSON.load`**, which is dangerous to call on untrusted input.

2.4.4. Exploiting deserialization vulnerabilities

To exploit deserialization vulnerability, there must already be a dangerous class loaded in the current namespace. In particular, it contains unsafe **`init_with()`** or **`[]=`** methods, that get called during deserialization. This might seem like an unlikely event, however, its very likely in case of big projects like Ruby on Rails.

CVE-2013-0156[20] vulnerability in Ruby on Rails can be used as an example. A vulnerable class in this case was `ActionDispatch::Routing::RouteSet::NamedRouteCollection`, which contained code like this:

```
class NamedRouteCollection
alias []=   add

def add(name, route)
  routes[name.to_sym] = route
  define_named_route_methods(name, route)
end

def define_named_route_methods(name, route)
  define_url_helper route, :"#{name}_path",
    route.defaults.merge(:use_route => name, :only_path => true)
  define_url_helper route, :"#{name}_url",
    route.defaults.merge(:use_route => name, :only_path => false)
end

def define_url_helper(route, name, options)@module.module_eval <<-END_EVAL
    def #{name}(*args)
      # ... code
    end
  END_EVAL
end
```

[19] https://www.ruby-lang.org/en/news/2013/02/22/json-dos-cve-2013-0269/
[20] https://groups.google.com/forum/?fromgroups=#!topic/rubyonrails-security/61bkgvnSGTQ

```
...
```

Even though **`module_eval`** is hidden under several layers of method calls, calling **`[]=`** effectively passes first argument to the **`define_url_helper`**, where it gets evaluated.

To exploit vulnerable class, it is enough to deserialize YAML payload below:

```
--- !ruby/hash:NamedRouteCollection
foo; end; system 'rm /etc/passwd'; def bar: baz
```

Before deserialization, Ruby's YAML parser Psych first looks at the declared type, which says this object is an instance of `NamedRouteCollection` and subclass of Ruby's `Kernel::Hash` class.

Deserialization of hashes from YAML to Ruby makes use of **`[]=`** method. Given YAML like

```
--- !ruby/hash:MyHash
key1: value1
key2: value2
```

deserialization process is equivalent to calling

```
newobj = MyHash.new
newobj['key1'] = 'value1'
newobj['key2'] = 'value2'
newobj
```

In the case of YAML payload, key and value pair is

```
['foo; end; system 'rm /etc/passwd'; def bar','baz']
```

so deserialization process will call **`[]=`** method on `NamedRouteCollection` with key **`'foo; end; system 'rm /etc/passwd'; def bar'`**.

This gets passed to **`define_url_helper`** as an argument and following code gets evaluated:

```
def foo; end; system 'rm /etc/passwd'; def bar(*args)
# ... code
end
```

Reordering the code above to be more readable, this is equivalent to

```
def foo
end

system 'rm /etc/passwd'

def bar(*args)
# ... code
end
```

2.4.4.1. References

- Aaron Patterson's *blog*[21]
- Charlie Sommerville's *blog*[22]
- Metasploit's *blog*[23]
- *Extending Hash*[24]

2.5. Regular expressions

A common gotcha in Ruby regular expressions relates to anchors marking the begninning and the end of a string. Specifically, ^ and $ refer to the beginning and the end of a line, rather then a string. If regular expression like **`/^[a-z]+$`** is used to whitelist user input, attacker can bypass it by including newline. To match the beginning and the end of a string use anchors **`\A`** and **`\z`**.

```
>> puts 'Exploited!' if /^benign$/ =~ "benign\n with exploit"
Exploited!
=> nil
>> puts 'Exploited!' if /\Abenign\z/ =~ "benign\n with exploit"
=> nil
```

2.6. Object.send

`Object.send` is a method with serious security impact, since it invokes any method on object, including private methods. Some methods in Ruby like **`eval`** or **`exit!`** are private methods of `Object` and can be invoked using **`send`**:

```
>> Object.private_methods.include?(:eval)
=> true
>> Object.private_methods.include?(:exit)
=> true
>> Object.send('eval', "system 'uname'")
Linux
=> true
```

Alternative is **`Object.public_send`**, which by definition only invokes public methods on object. However, this does not prevent attacker from executing only private methods, since **`Object.send`** itself is (and has to be) public:

```
>> Object.public_send("send","eval","system 'uname'")
Linux
=> true
>> Object.public_send("send","exit!")     # exits
```

Developers should be careful when invoking **`send`** and **`public_send`** with user controlled arguments.

[21] http://tenderlovemaking.com/2013/02/06/yaml-f7u12.html
[22] https://charlie.bz/blog/rails-3.2.10-remote-code-execution
[23] https://community.rapid7.com/community/metasploit/blog/2013/01/09/serialization-mischief-in-ruby-land-cve-2013-0156
[24] http://www.yaml.org/YAML_for_ruby.html#extending_kernel::hash

2.7. SSL in Ruby

Ruby uses OpenSSL implementation of common cryptographic primitives, which are accessible through `OpenSSL` module included in standard library. This module is then used by other parts of standard library to manage SSL, including `Net::HTTP`, `Net::POP`, `Net::IMAP`, `Net::SMTP` and others.

There are four valid verification modes `VERIFY_NONE`, `VERIFY_PEER`, `VERIFY_FAIL_IF_NO_PEER_CERT` and `VERIFY_CLIENT_ONCE`. These correspond to underlying *OpenSSL modes*[25].

SSL connection can be created using OpenSSL module directly:

```
>> require 'openssl'
=> true
>> require 'socket'
=> true
>> tcp_client = TCPSocket.new 'redhat.com', 443
=> #<TCPSocket:fd 5>
>> ssl_context = OpenSSL::SSL::SSLContext.new
=> #<OpenSSL::SSL::SSLContext:0x00000000fcf918>
>> ssl_context.set_params
=> {:ssl_version=>"SSLv23", :verify_mode=>1, :ciphers=>"ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:
+HIGH:+MEDIUM:+LOW", :options=>-2147480585}
>> ssl_client = OpenSSL::SSL::SSLSocket.new tcp_client, ssl_context
=> #<OpenSSL::SSL::SSLSocket:0x0000000106a418>
>> ssl_client.connect
=> #<OpenSSL::SSL::SSLSocket:0x0000000106a418>
```

Note the call to **`ssl_context.set_params`**: by default, when context is created, all its instance variables are nil. Before using the context, **`set_params`** should be called to initialize them (when called without argument, default parameters are chosen). In case this call is omitted and variables are left uninitialized, certificate verification is not performed (effectively the same as `VERIFY_NONE` mode). Default parameters are stored in the constant:

```
>> OpenSSL::SSL::SSLContext::DEFAULT_PARAMS
=> {:ssl_version=>"SSLv23", :verify_mode=>1, :ciphers=>"ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:
+HIGH:+MEDIUM:+LOW", :options=>-2147480585}
```

One of the side effects of **`set_params`** is that it also sets up certificate store with certificates from default certificate area (see *Section 2.7.1, "Certificate store"* below):

```
>> ssl_context.cert_store
=> nil
>> ssl_context.set_params
=> {:ssl_version=>"SSLv23", :verify_mode=>1, :ciphers=>"ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:
+HIGH:+MEDIUM:+LOW", :options=>-2147480585}
>> ssl_context.cert_store
=> #<OpenSSL::X509::Store:0x00000000fea740>
```

[25] https://www.openssl.org/docs/ssl/SSL_CTX_set_verify.html#NOTES

2.7.1. Certificate store

Class **`OpenSSL::X509::Store`** implements certificate store in Ruby. Certificate store is similar to store in web browsers - it contains trusted certificates that can be used to verify certificate chain. When new certificate store is created, it contains no trusted certificates by default.

To populate certificate store with certificates, use one of methods:

- **`Store#add_file`** takes a path to DER/PEM encoded certificate
- **`Store#add_cert`** takes instance of `X509::Certificate`
- **`Store#add_path`** takes a path to a directory with trusted certificates
- **`Store#set_default_path`** adds certificates stored in default certificate area

OpenSSL installation usually creates a directory, which stores several trusted certificates (approach similar to web browsers, that also come with predefined certificate store). To populate certificate store with certificates that come with OpenSSL use **`Store#set_default_path`**. The path to default certificate area is defined as:

```
>> OpenSSL::X509::DEFAULT_CERT_AREA
=> "/etc/pki/tls"
```

2.7.2. Ruby libraries using OpenSSL

There are several libraries that build on top of OpenSSL. Depending on how a library uses `SSLContext`, users may encounter exception from OpenSSL code saying the certificate verification failed:

```
>> ssl_client.connect
OpenSSL::SSL::SSLError: SSL_connect returned=1 errno=0 state=SSLv3 read server certificate B:
 certificate verify failed
from (irb):7:in `connect'
from (irb):7
```

This usually happens when `verify_mode` is set to check the certificate, but the certificate store used does not contain trusted certificate required to verify the SSL sent by the server.

Note

The worst advice that can be found on internet on how to fix SSL is to set

```
OpenSSL::SSL::VERIFY_PEER = OpenSSL::SSL::VERIFY_NONE
```

This redefines constant `OpenSSL::SSL::VERIFY_PEER` to have the same effect as `OpenSSL::SSL::VERIFY_PEER`, effectively globally disabling certificate checking.

Take `Net::IMAP` as example (the code below refers to Ruby 1.9.3): initialize method for creating a new IMAP connection has takes the following arguments:

```
def initialize(host, port_or_options = {},
               usessl = false, certs = nil, verify = true)
        ...
```

When SSL connection is used but `certs` and `verify` arguments are left to be assigned defaults values, SSLError may be thrown when certificate sent by server cannot be verified.

Important

The correct solution is to always make sure certificate store used by `SSLContext` contains a trusted certificate that can be used to verify the certificate sent by the server.

2.7.2.1. Behaviour in different Ruby versions

Default behaviour differs across Ruby versions: in Ruby 1.8, SSL enabled libraries usually falled back to `VERIFY_NONE` mode. The above mentioned `Net::IMAP#initialize` looks like this:

```
def initialize(host, port = PORT, usessl = false, certs = nil, verify = false)
        ...
```

Starting from Ruby 1.9, standard library defaults to `VERIFY_PEER` mode.

Web Application Security

Web application development is one of the most popular usages of Ruby language thanks to the popularity of Ruby on Rails. Following chapter is dedicated to security of web applications with most of the content being framework-independent, while examples and implmentation specific problems are targeted to Ruby on Rails.

Ruby on Rails as a popular web framework already helps with a web application security by providing secure defaults, useful helper methods, automatic html escaping etc.

3.1. Common attacks and mitigations

Creating a secure web application is impossible without knowing most common attack vectors and taking proactive actions to prevent them, such as code reviews, coding standards etc.

This section lists some of the most common attacks with in depth explanations, examples of real world vulnerabilities and specifics concerning Rails applications.

3.1.1. Command injection

One of the most widespread types of attack is command injection attack, where data from untrusted source are being used by application to construct a command. The command is executed in the context of application and when the untrusted data is not sanitized properly, attacker might use this weakness to execute arbitrary command, potentially with elevated privileges.

3.1.1.1. SQL injection

SQL injection is the most common type of command injection, where application constructs a SQL query from user supplied data. If not properly escaped, malicious attacker might be able to execute any SQL command on application's database, that can lead to information disclosure, unauthorized modification of data, execution of administrative operations or destruction of data.

Ruby on Rails provides a good protection against SQL injection attacks by escaping several special SQL characters by default. However, this is far from making Rails applications safe against SQL injection. Consider a query against database:

```
User.where("name = '#{params[:name]}'")
```

This would be translated to following SQL query:

```
SELECT "users".* FROM "users" WHERE (name = 'username')
```

Such statement is vulnerable to SQL injection, since part of the SQL statement is passed as string in argument and Rails does not perform any escaping. Malicious string can match apostrophe and bracket in the statement, the follow with semicolon as statement separator and arbitrary SQL query. At the end double hyphens are necessary to comment out superfluous apostrophe:

```
>> params[:name] = "'); <arbitrary statement> --"
```

Using Rails console we can see this how such input is translated to a SQL query:

```
>> params[:name] = "noname'); SELECT name, password_digest FROM users where userid = 'admin'
 --"
=> "noname'); SELECT name, password_digest FROM users where userid = 'admin' --"

>> User.where("name = '#{params[:name]}'")
  User Load (2.4ms)  SELECT "users".* FROM "users" WHERE (name = 'noname'); SELECT name,
 password_digest FROM users where userid = 'admin' --')
=> [#<User name: "Administrator", password_digest: "$2a
$10$m7XI628GGkdTH1JmkdMfluJyA360V1.QBtSbFMrc5Jwm...">]
```

3.1.1.1.1. (Un)safe Active Record queries

Safer approach is to pass either array or hash as an argument and use Rails escaping mechanism to protect against SQL, as in

```
User.where("name = ?", params[:name])
```

or

```
User.where(name: params[:name])
```

Alternatively, ActiveRecord also provides **`ActiveRecord::sanitize`** method which can be used to sanitize a string explicitly.

However, other ActiveRecord methods may be vulnerable to surprising SQL injection attacks, too. Consider **`exists?`** - when given string as an argument, it tries to convert it to integer, returning 0 when the conversion is impossible:

```
>> User.exists?("1")
  User Exists (0.9ms)  SELECT 1 AS one FROM "users" WHERE "users"."id" = 1 LIMIT 1
=> true

>> User.exists?("abc")
  User Exists (0.8ms)  SELECT 1 AS one FROM "users" WHERE "users"."id" = 0 LIMIT 1
=> false
```

This might look like a safe behaviour and imply the following query is safe from SQL injection attack:

```
User.exists?(params[:id])
```

The **`exists?`** method also accepts array as an argument - in which case first element of array is used directly in SQL query without escaping:

```
>> params[:id] = ["id = '1'"]
=> ["id = '1'"]

>> User.exists?(params[:id])
  User Exists (0.8ms)  SELECT 1 AS one FROM "users" WHERE (id = '1') LIMIT 1
=> true
```

This makes SQL injection attack possible:

```
>> params[:id] = ["1=1);UPDATE users SET password_digest='my_digest' WHERE userid='admin'
 --"]
=> ["1=1);UPDATE users SET password_digest='my_digest' WHERE userid='admin' --"]

>> User.exists?(params[:id])
  User Exists (67.6ms)  SELECT 1 AS one FROM "users" WHERE (1=1);UPDATE users SET
 password_digest='my_digest' WHERE userid='admin' --) LIMIT 1
=> false

>> User.where(userid: 'admin').first.password_digest
  User Load (1.0ms)  SELECT "users".* FROM "users" WHERE "users"."userid" = 'admin' LIMIT 1
  User Inst (0.4ms - 1rows)
=> "my_digest"
```

The last obstacle is passing the user supplied parameter as an Array. Usually, all values of parameters are passed by Rack as strings, but it is also possible to explicitly specify that value of parameter is supposed to be Array in the HTTP request. If the parameter looks like

```
key[]=value
```

Rack assumes it should be an Array and performs conversion before the parameter is passed to Rails application. HTTP request that exploits **`exists?`** method called on `params[:id]` then looks like this:

```
GET /controller/action?id[]=1 = 1);UPDATE users SET password_digest='my_digest' WHERE
 userid='admin' --
```

3.1.1.2. OS command injection

Another common vulnerability is invoking underlying OS commands with user supplied input without proper sanitization. Ruby provides several commands that can be used and if user's input is used as parameter to a system command without sanitization, he might be able to misuse it to execute arbitrary command.

For example, when application contains call like

```
system "echo Hello #{params[:name]}!"
```

user can use semicolon to terminate **`echo`** command and invoke command of his choice:

```
>> params[:name] = 'Joe;rm -rf /'
=> "Joe;touch /tmp/abc"
>> system "echo Hello #{params[:name]}"
Hello Joe
=> true        # and rm gets executed
```

`system` command can be used to explicitly separate OS command to invoke from the arguments passed to it:

```
system(command, *parameters)
```

Important

Whenever system command is executed with arguments from untrusted source, extra care must be taken to prevent arbitrary code execution.

Also see *Section 2.2, "Dangerous methods"*.

3.1.1.3. References

- Ruby on Rails Security Guide: *SQL injection*[1]
- *Rails SQL Injection*[2]
- OWASP: *SQL Injection*[3]
- CWE-89: *Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*[4]
- CWE-77: *Improper Neutralization of Special Elements used in a Command ('Command Injection')*[5]

3.1.2. Cross site scripting (XSS)

Cross site scripting (usually abbreviated as XSS) is a special type of command injection, where attacker supplied malicious payload is interpreted within the context of victim page. The weakness that is a cause for this vulnerability is embedding untrusted data in the web page without sanitization. In the most trivial example, imagine social networking website which displays names of users without sanitization to others. If the name is pasted into resulting page as-is, one can change his name to

```
<script>alert('Hello');</script>
```

This code embedded in the html sent by the server will be parsed in client's browsers and interpreted as javascript code, which runs and displays a popup with a greeting.

3.1.2.1. Types of XSS

Over the time XSS attack evolved and based on the way payload is delivered we recognize three types.

Reflected XSS is the most common type, when data sent by the user is directly embedded into a web page by server and returned back to user. For example think of query parameters which are displayed in a web page:

```
https://bugzilla.redhat.com/show_bug.cgi?id=<script>alert('Hello')</script>
```

[1] http://guides.rubyonrails.org/security.html#sql-injection
[2] http://rails-sqli.org/
[3] https://owasp.org/index.php/SQL_Injection
[4] http://cwe.mitre.org/data/definitions/89.html
[5] http://cwe.mitre.org/data/definitions/77.html

The attack vector in this example is attacker tricking a user into clicking on this malicious URL. If the server returns back user supplied bugzilla ID without escaping, it would get interpreted in the context of bugzilla.redhat.com. Looking at the source of actual response, we find that value provided is escaped:

```
'&lt;script&gt;alert(10)&lt;/script&gt;' is not a valid bug number nor an alias to a bug.
```

In **Stored XSS** the payload is saved on the server and later displayed to the users. Examples include unescaped user profile data on social networking site, unescaped posts on internet forums, unescaped filenames on file sharing sites etc.

DOM based XSS is the newest and so far least common type of XSS. With more and more webpage processing moving over to client side in form of javascript code and frameworks, users can browse sites and change them without sending data to be processed by the server. For example hypothetical vulnerable webpage would contain a javascript to display current URL of the page:

```
<script>
document.write("URL is: " + document.location.href);
<script>
```

If attacker appends fragment with script to the URL, script running in client's browser would inject URL as-is along with the payload. The URL would look like this:

```
http://vulnerable.com/#<script>alert('Hello')</script>
```

Note that fragment part of URL is not usually sent to the server in the HTTP request.

3.1.2.2. XSS protection in Rails

In past Ruby on Rails required developers to explicity escape any potentially unsafe strings with **`html_escape()`** method (or alias **`h()`**). This is methodically bad approach to security, since any omission of the method on critical place lead to potentiall exploitable weakness.

Since Rails 3 instead or relying on developer to provide html escaped output Rails automatically escapes everything. This is implemented in **`ActiveSupport::SafeBuffer`** class, which extends String and represents escaped (i.e. html safe) version of a string. The most important aspect of SafeBuffer is that it stays html safe even when standard String methods are called with unsafe inputs:

```
> sb = ActiveSupport::SafeBuffer.new
 => ""
> sb << "<string>"
 => "&lt;string&gt;"
> sb + "</string>"
 => "&lt;string&gt;&lt;/string&gt;"
```

This way a response to the client can be built part by part without allowing unsafe characters. If a string is safe and can be rendered without escaping, developer has to mark it html safe explicitly using **`html_safe`** method. This method actually returns an instance of SafeBuffer:

```
> safe =  "  <b>HTML safe</b> string".html_safe
 => "  <b>HTML safe</b> string"
```

```
> safe.class
 => ActiveSupport::SafeBuffer
> sb << safe
 => "&lt;string&gt;&lt;/string&gt;  <b>HTML safe</b> string"
```

3.1.2.3. Real world examples

With automatic escaping of output XSS vulnerabilities in Rails are much less common, but still occur. One common cause is misuse of **`html_safe()`** on untrusted strings. Example of this is CVE-2014-3531: XSS flaw in operating system name/description in Foreman. Weakness in this case was present in helper method, which returned name of the OS:

```
def os_name record, opts = {}
  "#{icon(record, opts)} #{record.to_label}".html_safe
end
```

The first part of string, **`icon(record, opts)`** returns html safe string, but second part is untrusted and may contain html unsafe characters. Calling **`html_safe`** on the resulting String caused vulnerability and the fix was straightforward:

```
def os_name record, opts = {}
  icon(record, opts).html_safe << record.to_label
end
```

Another way to create XSS vulnerability in Rails is to bypass the automatic ERB escaping by rendering response directly. Example of this is CVE-2014-3492: XSS from stored YAML in Foreman. Vulnerable code serialized untrusted data into YAML and directly rendered it on page as preview:

```
begin
  respond_to do |format|
    format.html { render :text => "<pre>#{@host.info.to_yaml}</pre>" }
    format.yml { render :text => @host.info.to_yaml }
  end
rescue
...
```

In this case output has to be explicitly escaped before returning to the client:

```
begin
  respond_to do |format|
    format.html { render :text => "<pre>#{ERB::Util.html_escape(@host.info.to_yaml)}</pre>" }
    format.yml { render :text => @host.info.to_yaml }
  end
rescue
...
```

3.1.2.4. References

- Ruby on Rails Security Guide: *Cross-site Scripting*[6]

[6] http://guides.rubyonrails.org/security.html#cross-site-scripting-xss

- OWASP: *Cross-site Scripting*[7]
- OWASP: *XSS (Cross Site Scripting) Prevention Cheat Sheet*[8]
- OWASP: *Reviewing Code for Cross-site scripting*[9]
- OWASP: *DOM Based XSS*[10]
- OWASP: *DOM Based XSS Prevention Cheat Sheet*[11]
- OWASP: *Testing for Reflected Cross site scripting*[12]
- OWASP: *Testing for Stored Cross site scripting*[13]
- OWASP: *Testing for DOM-based Cross site scripting*[14]

3.1.3. Cross site request forgery (CSRF)

By default, browsers include user's authentication tokens (such as cookies, HTTP basic authentication credentials etc.) with every request to the web application. This allows client to authenticate once and each following request to the web application will be authenticated without prompting the user for credentials. However, this gives client's browser ability to make authenticated requests on behalf of the user without user's explicit consent.

This behaviour can be misused by the attacker to confuse client's browser into issuing an authenticated request. For example, if attacker's website contains this simple script tag

```
<script src="http://victimbank.com/transfermoney?to=attacker&amount=1000"/>
```

browser will issue a HTTP GET request to victimbank.com with parameters supplied by the attacker. The browser does not know anything about the resource that is being requested by the attacker's site - whether it is malicious of harmless - and it requests the script from the specified URL. If the user is authenticated at that moment, browser will also include his credentials, so the request would look like this:

```
GET /transfermoney?to=attacker&amount=1000 HTTP/1.1
Host: victimbank.com
Cookie: ...
```

Even though browser believes it is asking for a resource, web application will perform action specified in the request from the client - in this case, send money to the attacker. Such web application is vulnerable to Cross Site Request Forgery.

[7] https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29
[8] https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet
[9] https://www.owasp.org/index.php/Reviewing_Code_for_Cross-site_scripting
[10] https://www.owasp.org/index.php/DOM_Based_XSS
[11] https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
[12] https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_%28OWASP-DV-001%29
[13] https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_%28OWASP-DV-002%29
[14] https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_%28OTG-CLIENT-001%29

Important

Web application should not change state or perform security sensitive actions upon receiving HTTP GET requests. Such behaviour is not compliant with HTTP and may create problems with caches, browser prefetching etc.

It is not enough to make sure that web application does not use HTTP GET requests to perform security sensitive actions - it is important that such requests are forbidden by the application. For example, Rails application's action can be invoked only with non-GET requests throughout the application, but still be routable through GET requests.

Restricting security-sensitive operations to non-GET requests does not protect from CSRF attack itself. Even though common HTTP tags like `<img>`, `<script>` and others can be used to issue HTTP GET requests, there are other means to issue arbitrary requests against vulnerable application.

As example consider the code below:

```
<body onload="document.getElementById('f').submit()">
  <form id="f" action="http://victimbank.com/transfermoney" method="post" name="form1">
    <input name="to" value="attacker">
    <input name="amount" value="1000">
  </form>
</body>
```

If user visits page containing a code similar to this one, upon loading the page browser will send a HTTP POST request with the parameters supplied by the attacker.

There are several mechanisms available, that allow web application to identify requests issued by a third-party web page from the client's browser.

3.1.3.1. Synchronizer token pattern

OWASP recommended method of CSRF protection is to include a challenge token in each sensitive request. The token must be unpredictable to the attacker, otherwise the attacker could guess it and include with his forged request. The token must also be tied to user's session - if the token is shared by users, they would be able to forge requests on behalf of others. It goes without saying that it cannot be part of the authentication tokens, since they are sent with each request automatically, which defeats the purpose of CSRF protection. However, this token needs to be generated only once per each session.

The CSRF challenge token should be included in all non-GET requests, including Ajax requests. On the server side, application has to verify the token is included in request and is valid, and reset session otherwise.

Synchronizer token pattern is also default CSRF protection mechanism for Rails applications. To enable CSRF protection, one has to enable it in application controller with

```
protect_from_forgery
```

which will automatically include CSRF token in all non-get and XHR requests. The token itself is sent by the server in meta tag of the web page like this:

```
<meta content="authenticity_token" name="csrf-param" />
<meta content="VBlgpnibfsxm1QykEmlOCbxqLRxx7kDGr57tjE+LLZk=" name="csrf-token" />
```

If the request is not verified to be CSRF-free, Rails resets the session by default:

```
def handle_unverified_request
  reset_session
end
```

If this does not effectively log out user due to application-specific behaviour, developers should redefine `handle_unverified_token`.

The disadvantage of synchronizer token pattern is the need to rememnber the challenge token for each session on the server side.

3.1.3.2. Double submit cookie pattern

This method mitigates the problem of keeping state on the server side. Each sensitive request shall include a random value twice: in cookie, and as a request parameter. After receiving request, server verified that both values are equal, so this mechanism is stateless.

Assuming the random value meets the requirements on CSRF token, attacker cannot forge the CSRF requests. To do that, he would need an access to random value stored in a cookie of another site, which is prevented by Same Origin Policy.

This mechanism is arguably less secure than synchronizer token pattern. While it is hard for the attacker to read the random value from cookie, it is easier to write a value, for example by writing an attacker-specified value from a subdomain.

3.1.3.3. Encrypted token pattern

Another stateless approach leverages encryption. The token sent by the server is triple User ID, Timestamp and Nonce, encrypted with server-side secret key. The token sent to the client in a hidden field, and returned by the client in a custom header field for Ajax requests or as a parameter for form-based requests.

Validation of token does not require any state on the server side aside from secret key. Upon receiving request, server decrypts the token and verifies User ID against session's User ID (if there is one) and Timestamp to prevent replay attacks. If decryption of the token yields malformed data or any of the checks fails, server blocks the potential attack.

3.1.3.4. Checking Referer header

Checking the Referer header to make sure that request does not originate from the third party site is a common stateless CSRF protection mechanism. Even though it is possible for the user to spoof referer header, it is not possible for the attacker in case of CSRF, since the Referer header is included by the client's browser and outside of attackers control.

Even though it may seem to be the easiest mechanism to implement, it carries a lot of cornercases, depends on configuration outside of applications control and is prone to compatibility issues.

One of the problems of Referer header is potential disclosure of private information, due to which some users may configure their browsers to not include Referer header at all. Referer header is also omitted when browsing from HTTPS secured site to HTTP. Since attacker can mount attack from

HTTPS protected page, web application has to deny requests without Referer header. This affects compatibility - for example, when user directly types the URL (or bookmarks it), Referer header will be empty and the application will the refuse request due to CSRF protection, creating usability problems.

From implementation standpoint, CSRF check needs to make sure that request originated from a page from trusted domain, however path with parameters do not matter. It is therefore tempting to implement the check by verifying that Referer start with the domain, ignoring the rest of the path. For example, if the Referer is "http://application.domain.com/some/page", the check would verify that it starts with "http://application.domain.com" and allow the request. This can be bypassed if the attacker mounts CSRF attack from "http://application.domain.com.evil.io".

Important

Checking the Referer header as CSRF protection mechanism is highly discouraged.

3.1.3.5. References

- OWASP: *Cross Site Request Forgery*[15]
- OWASP: *CSRF Prevention cheat sheet*[16]
- CWE-352: *Cross-Site Request Forgery (CSRF)*[17]
- *Encrypted Token pattern*[18]

3.1.4. Guidelines and principles

Following are general recommendations based on previous sections:

Always make sure output sent to client is escaped correctly
: Automatic ERB escaping in Rails works in most cases, however, developers should still be careful about rendering untrusted data directly to user or misusing **`html_safe`**.

Always make sure command arguments send to components (shell, database) are escaped or trusted
: Command injection is one of the most understood and best studied attack vectors. Ruby on Rails provides good defense against SQL injection, however developers should be always careful when executing OS command with potentiall untrusted arguments.

Verify routing exposes actions through expected HTTP verbs
: An important part of protecting agains CSRF attacks is to make sure actions reachable through HTTP GET do not have side effects. This is something to think about from the very beginning, since cleaning up routing later into development cycle tends to be intrusive and complex.

3.2. Client-side security

Web applications are based on interaction between client and a server and both ends can be attacked. In order to defend some of the attacks mentioned in previous chapter, reduce attack surface of client`s

[15] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29
[16] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet
[17] https://cwe.mitre.org/data/definitions/352.html
[18] http://insidethecpu.wordpress.com/2013/09/23/encrypted-token-pattern/

browsers and patch holes that were introduced without a second thought during browser wars in past, browsers include several security features.

3.2.1. Same origin policy

One of the most important concepts in web applications is same origin policy. It is a protection mechanism implemented by modern web browsers that isolates web applications from each other on the client side. This isolation is performed on domain names under the assumption that content from different domains comes from different entities. In theory, this means every domain has its own trust domain and interaction across domains is restricted. In practice, there are multiple ways of bypassing this mechanism, malicious ones often creating confused deputy problem where client`s browser is tricked into submitting attacker-specified request under his authority.

Same origin policy prevents Javascript and other scripting languages to access DOM across domains. In addition it also applies to XMLHttpRequest Javascript API provided by browsers and prohibits page of sending XMLHttpRequest requests against different domains. On the downside, actual implementation by different browsers may vary in important details. Since the actual behaviour depends on implementation in each browser, each vendor usually implements some exceptions intended to help web developers, which reduce the reliability of this mechanism.

Same origin policy

Two pages share the same origin if the protocol, hostname and port are the same for both.

Following is a table with outcome of same origin policy check against URL http://web.company.com/~user1

Table 3.1. Sample CALS Table

URL	Outcome	Reason
http://web.company.com/~user2	Success	
https://web.company.com/~user1	Fail	Different protocol
http://store.company.com/~user1	Fail	Different hostname
https://web.company.com:81/~user1	Fail	Different port

As the example above shows, if a company servers webpages of users from the same domain web.company.com, then pages of individual users are not restricted by same origin policy when accessing each other, as they are coming from the same domain.

Browsers treat hostname of server as string literal, which creates another exceptional case: even if IP address of company.com is 10.20.30.40, browser will enforce same origin policy between http://company.com and http://10.20.30.40.

3.2.1.1. Setting document.domain

A page can also define its origin by setting **`document.domain`** property to a fully-qualified suffix of the current hostname. When two pages have defined the same **`document.domain`**, same origin policy is not applied. However, **`document.domain`** has to be specified mutually - it is not enough for just one page to specify its **`document.domain`**. Also, when **`document.domain`** property is set, port is set to null, while still being checked. This means company.com:8080 cannot bypass same origin policy and access company.com by setting **`document.domain = "company.com"`**, as their ports (null vs 80) differ.

However, **`document.domain`** has several issues:

- When web.company.com and storage.company.com need to share resources and set **`document.domain = company.com`**, any subdomain can set its **`document.domain`** and access both of them, even though this access was not intended to be permitted.
- When this mechanism cannot be used, cross-domain requests are forbidden even for legitimate use, which creates problem for websites that use multiple (sub)domains.

3.2.1.2. Unrestricted operations

Same Origin Policy restricts Javascript access to DOM and XMLHttpRequest across domains. However, there are multiple operations that are not restricted:

- Javascript embedding with **`<script src=".."><script>`**
- CSS embedding with **`<link rel="stylesheet" href="...">`**
- Anything with **`<frame>`** and **`<iframe>`**
- .. and others

3.2.1.3. References:

- Google: *Browser Security Handbook*[19]
- Mozilla Developer Network: *Same Origin Policy*[20]

3.2.2. Bypassing same origin policy

Same Origin Policy as security mechanism leaves a lot to be desired: on one hand, it is not flexible enough to allow web developers use cross-domain resources in several legitimate usecases without exceptions to the rule and workarounds, on the other hand, such exceptions create opportunities for attacker.

There are several other mechanisms except **`document.domain`** that provide a way to relax Same Origin Policy.

3.2.2.1. Cross-origin resource sharing (CORS)

Cross-origin resource sharing is a mechanism that allows web application to inform browser, whether cross domain requests against the requested resource are expected.

Web browsers that conform to the CORS alter their behaviour of handling XMLHttpRequests: instead of denying the cross-domain request immediately, HTTP request is sent with **`Origin`** header. Let's assume http://example.com/testpage is making a XMLHttpRequest against http://content.com/wanted_image. Request would contain:

```
GET /wanted_image HTTP/1.1
Referrer: http://example.com/testpage
Origin: http://example.com
```

If the server allows sharing of the resource with domain that originated the request, the response would include:

[19] http://code.google.com/p/browsersec/wiki/Part2
[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://example.com
..
```

By sending `Access-Control-Allow-Origin` header, server explicitly tells browser that this cross domain request shall be allowed. Allowed values of `Access-Control-Allow-Origin` are: * (denoting any domain, effectively marking the resource public) or space separated list of allowed origins (in practice, this usually contains just a single domain - one that was specified in Origin header in request).

If the resource should not be accessible by the originating domain, server ought not include Access-Control-Allow-Origin header in the response. By default, upon receiving such response from server browser will not pass the response back to the page that originated the request.

Several additional considerations:

- If the browser is outdated and does not conform to CORS, cross domain request will be denied immediately without sending the request to the server. This means usability of web applications relying on CORS might be restricted on old browsers.
- If the web server does not conform to CORS, the Access-Control-Allow-Origin header will not be included in the response and the request will be denied on the client side.
- Cross-domain access to resources is enforced on the side of the client. However, since the request includes Origin header, server may also restrict access to resources from other domains (e.g. by returning nothing).
- If the origin of page is unknown (for example webpage is running from a file), browsers will send

  ```
  Origin: null
  ```

3.2.2.1.1. Using CORS in Rack-based applications

CORS support for Rack-based applications is provided by *rack-cors*[21] gem. After adding it to the applications Gemfile

```
gem 'rack-cors', :require => 'rack/cors'
```

and configure Rails by modifying config/application.rb:

```
module YourApp
class Application < Rails::Application

  # ...

  config.middleware.use Rack::Cors do
    allow do
      origins '*'
      resource '*', :headers => :any, :methods => [:get, :post, :options]
    end
  end
```

[21] https://github.com/cyu/rack-cors

```
end
end
```

This configuration permits all origins access to any resource on the server via GET, POST and OPTIONS methods. Customizing the configuration, developer of the application can restrict cross-domain acess to resources by origin, headers and methods.

3.2.2.2. JSON with padding (JSONP)

JSONP is a very common way of hacking around the Same Origin Policy. This mechanism makes use of **`<script>`** tag and the fact that embedding Javascript code from other domains is not resctricted by the same origin policy. Since the code references by src attribute of **`<script>`** tag is loaded, it can be used as a vehicle to carry data and return them after evaluation.

Lets assume webpage needs to access resource at http://example.com/resource/1, which returns JSON data like:

```
{"Key1": "Value1", "Key2": "Value2"}
```

When webpage requests the resource with

```
<source src="http://example.com/resource/1"></source>
```

after receiving the response, browser will try to evaluate received data. Since data are not executable, interpreter would end with error and data would not be accessible to the code that requested it.

To work around this, it would be enough if the returned data were enclosed with function, that would be able to parse them on the client side. Suppose function **`parseData`** can accept JSON data as argument, parse it and make it accessible to the rest of the page:

```
parseData({"Key1": "Value1", "Key2": "Value2"})
```

However, web server does not know the name of the function that will parse data. Final piece is to pass the name of data-parsing function to server as parameter in request:

```
<script src="http://example.com/resource/1?jsonp=parseData"></script>
```

This technique of sharing resources across domains carries bigger security risks than CORS. Since **`source`** tag does not fall under Same Origin Policy on the client side, browser sends normal HTTP GET request without `Origin` header. Server that receives request has no means to know that the request was generated on behalf of page from other domain. Since neither the browser nor the server checks this kind of cross-domain requests, last obstacle that prevents exploitation is the fact that returned response is evaluated as Javascript code.

Example of this type of vulnerability is *CVE-2013-6443*[22]. Cloud Forms Manage IQ application has been found vulnerable to cross-domain requests issued using JSONP. UI of application makes heavy use of Javascript and in this particular case changing the tab to "Authentication" would generate this HTTP request through XMLHttpRequest API:

[22] https://access.redhat.com/security/cve/CVE-2013-6443

```
GET /ops/change_tab/?tab_id=settings_authentication&callback=...
Referrer: ...
Cookie: ...
```

Response returned by the server would look like this:

```
HTTP/1.1 200 OK
....


miqButtons('hide');
Element.replace("ops_tabs", "<div id=\"ops_tabs\" ...");
```

where ops_tabs div contained html code of the Authentication tab including form with hidden CSRF token. To exploit this vulnerability, attacker would patch **`Element.replace`** function on his page and issue a JSONP request against CFME server.

```
<script src='http://code.jquery.com/jquery-1.10.2.min.js'></script>
<script>
function test() {
$.ajax({
    url: $( "input[name=url]" ).val() + '/ops/change_tab/?tab_id=settings_authentication',
    dataType: 'jsonp'
});
};

var Element = { replace: function (a,text) {
...
}
>/script>
```

This way attacker can run arbitrary code on returned response from the server: since the request also contains CSRF token, it is easy for attacker to steal it and issue successful CSRF request on behalf of currently logged-in user.

3.2.2.3. References:

- W3C Recommendation: *Cross-Origin Resouce Sharing/*[23]
- Mozilla: *cross-site xmlhttprequest with CORS*[24]
- Open Ajax Alliance: *Ajax and Mashup Security*[25]
- *CVE-2013-6443*[26]

3.2.3. Content Security Policy (CSP)

Content Security policy is a comprehensive web security mechanism that allows web applications to declaratively list all sources of trusted content. Originally developed at Mozilla and later adopted by Webkit, Content Security Policy is now a W3C Candidate Recommendation.

[23] http://www.w3.org/TR/access-control/
[24] http://hacks.mozilla.org/2009/07/cross-site-xmlhttprequest-with-cors/
[25] http://www.openajax.org/whitepapers/Ajax%20and%20Mashup%20Security.php
[26] https://access.redhat.com/security/cve/CVE-2013-6443

One of the persistent problems in web application security is the lack of distinction between content loaded from trusted sources and potentially malicious content injected or referenced in the web page. Content Security Policy takes a comprehensive approach: a new HTTP header is introduced to allow server to send a whitelist of trusted sources to the client. Conformant user agents follow the policy declared in the header and block content from untrusted sources.

Several headers are related to Content Security Policy:

- `X-Content-Security-Policy:` experimental header originally introduced by Mozilla
- `X-WebKit-CSP:` experimental header used in WebKit based browsers
- `Content-Security-Policy:` a standard header proposed by W3C, that shall be used as replacement for the two abovementioned experimenal headers. However, older versions of browsers may support only experimental versions of this header, so web application developers that seek the best coverage may want to use all three headers together.

Value of the header consists of several directives separated by semicolon, each of them followed by list of sources separated by spaces. Following simple policy declares http://example.com as a trusted sources of scripts and disables all other sources:

```
Content-Security-Policy: default-src 'none'; script-src http://example.com
```

Since CSP uses whitelist approach, loading scripts from any other domain would not be permitted. Suppose webpage contains following:

```
<script src="http://malicious.com"></script>
```

In Firefox this would generate following warning:

```
[13:16:03.713] CSP WARN:  Directive script-src http://example.com:80 violated by http://
malicious.com/
```

This approach works in case of content with known origin, but this does not solve problem with inlined scripts such as

```
<script>exploit()</script>
```

CSP addresses this problem by completely banning execution of any scripts or CSS inlined with `<script>` or JavaScript URI and similar restrictions apply on `eval()` like mechanisms. This is necessary from security standpoint, however, it also means that web application developers who want to adopt CSP need to make sure their application does not make use of banned functions. To mitigate this CSP includes reporting capability via `report-uri` directive, reporting only mode via `Content-Security-Policy-Report-Only` header and ability to disable protection with `'unsafe-inline'` and `'unsafe-eval'` sources (see below).

3.2.3.1. Directives and source lists

CSP defines several directives that define restricted content types:

- `script-src` restricts which scripts the protected resource can execute.
- `object-src` restricts from where the protected resource can load plugins.
- `style-src` restricts which styles the user applies to the protected resource.

- `img-src` restricts from where the protected resource can load images.
- `media-src` restricts from where the protected resource can load video and audio.
- `frame-src` restricts from where the protected resource can embed frames.
- `font-src` restricts from where the protected resource can load fonts.
- `connect-src` restricts which URIs the protected resource can load using script interfaces (like XMLHttpRequest).

and additional directives that control behaviour of CSP:

- `default-src` sets a default source list for all directives except sandbox. If not set, directives that are omitted permit all sources by default.
- `sandbox` is an optional directive that specifies an HTML sandbox policy that the user agent applies to the protected resource.
- `report-uri` specifies a URI to which the user agent sends reports about policy violation.

Source list syntax is fairly flexible: source can be specified from scheme only (https:) and hostname (example.com) to a fully qualified URI (https://example.com:443). Wildcards are also permitted instead of scheme, port or as prefix of domain name to denote arbitrary subdomain (*.example.com). Additionally, there are four keywords allowed in the source list:

- `'none'` matches nothing.
- `'self'` matches current origin.
- `'unsafe-inline'` allows inline JavaScript and CSS and can be used with script-src and style-src directives.
- `'unsafe-eval'` allows eval-list mechanisms that convert text to executable script and can be used with script-src directive.

```
Content-Security-Policy: default-src 'none'; script-src https://cdn.example.com 'self'
 'unsafe-inline'; connect-src https://api.example.com;
```

3.2.3.2. Reporting policy violations

Developers who are tuning CSP for their web applications or adopting CSP can use reporting capabilities of CSP. By including `report-uri` directive server can instruct client's user agent to send POST with JSON-formatted violation report to a specified URI.

```
Content-Security-Policy: ...; report-uri /csp_report_parser;
```

Reports sent back to server about CSP violation looks like this:

```
{
"csp-report": {
  "document-uri": "http://example.org/page.html",
  "referrer": "http://evil.example.com/haxor.html",
  "blocked-uri": "http://evil.example.com/image.png",
  "violated-directive": "default-src 'self'",
  "original-policy": "default-src 'self'; report-uri http://example.org/csp-report.cgi"
}
```

```
}
```

When deploying CSP it may be useful to test the policy in the wild before enforcing it. It is possible to achieve this by sending `Content-Security-Policy-Report-Only` header instead - this will indicate that the user agent must monitor any policy violations, but not enforce them. Combined with `report-uri` this gives developers tools to seamlessly deploy new CSP policy.

```
Content-Security-Policy-Report-Only: ...; report-uri /csp_report_parser;
```

3.2.3.3. References

- *W3C Content Security Policy 1.0*[27]
- *HTML5Rocks tutorial*[28]
- *GitHub's blog on CSP*[29]

3.2.4. HTTP Strict Transport Security

HTTP Strict Transport Security is a mechanism that allows server to inform client that any interactions with the server shall be carried over secure HTTPS connection.

HTTPS provides a secure tunnel between client and the server, yet there are still ways through which data can leak to the attacker. One of the most practical attacks on SSL is SSL stripping attack introduced by Moxie Marlinspike, in which active network attacker transparently converts HTTPS connection to insecure one. To the client it seems like web application does not support HTTPS and has no means to verify whether this is the case.

HTTP Strict Transport Security mechanism allows server to inform client's user agent that the web application shall be accessed only through secure HTTPS connection. When client`s UA conformant with HSTS receives such notice from server, it enforces following behaviour:

- all references to HSTS host are converted into secure ones before dereferencing
- connection is terminated upon any and all secure transport errors or warnings without interaction with user

User agents which receive response with HSTS header need to retain data about host enforcing strict transport security for the timespan declared by the host. User agent builds a list of known HSTS hosts and whenever request is sent to known HSTS host, HTTPS is used.

HSTS header sent by the server includes timespan during which UA should enforce strict transport security in seconds:

```
Strict-Transport-Security: max-age=631138519
```

Optionally, server can also specify that HSTS be enforced on all subdomains:

```
Strict-Transport-Security: max-age=631138519; includeSubDomains
```

[27] http://www.w3.org/TR/CSP/
[28] http://www.html5rocks.com/en/tutorials/security/content-security-policy/
[29] https://github.com/blog/1477-content-security-policy

Setting timespan to zero

```
Strict-Transport-Security: max-age=0
```

allows the server to indicate that UA should delete HSTS policy associated with the host.

This header protects client from visiting host he has visited before using unsecure connection, but when the client connects for the first time, he has no prior knowledge about HSTS policy for the host. This theoretically allows attacker to successfully perform attack against user that connect for the first time. To mitigate this, browsers include preloaded list of known HSTS hosts in the default installation.

3.2.4.1. Configuring HSTS in Rails

A single directive in Rail configuration

```
config.force_ssl = true
```

enables HSTS for the application.

3.2.4.2. References

- *RFC 6797*[30]
- Mozilla: *Preloading HSTS*[31]
- Chromium: *list of preloaded known HSTS hosts*[32]

3.2.5. X-XSS-Protection

Modern browsers usually come with built-in XSS filter, that is enabled by default. Originally IE 8 introduced new XSS filter and this header was created to give web application developers way to turn this feature off in case it breaks functionality of the web application for users. Later this concept was also adpoted by Webkit, which implements its own XSS filter.

XSS filter does not prevent XSS attacks by blocking malicious scripts, but rather tries to identify untrusted scripts and transform them into benign strings. Heuristics that identify untrusted scripts usually try to match scripts embedded within request to those included in response. If the script matches, browser assumes the script included in the content is not trusted, as it is most probably not part of the content of the application, but rather included as user-supplied parameter. This means XSS filters are effective only against reflective XSS, not other variants.

Setting value of the header to 1 should re-enable XSS filter, in case it was disabled by user.

```
X-XSS-Protection: 1
```

Sanitization of scripts by converting them to benign strings has been source of bugs and security vulnerabilities - sanitization in IE8 XSS filter has been found counterproductive as it actually introduced XSS vulnerabilities in websites that were previously not vulnerable to XSS (including bing.com,

[30] http://tools.ietf.org/html/rfc6797
[31] https://blog.mozilla.org/security/2012/11/01/preloading-hsts/
[32] https://src.chromium.org/viewvc/chrome/trunk/src/net/http/transport_security_state_static.json

google.com, wikipedia.com and others. For details, see whitepaper by Eduardo Vela Nava and David Lindsay *Abusing Internet Explorer 8's XSS Filters*[33]).

To remedy this, extension to the X-XSS-Protection header was introduced:

```
X-XSS-Protection: 1; mode=block
```

With mode set to block browser will outright block any script found untrusted instead of trying to sanitize and display it.

3.2.5.1. References

- IE Internals: *Controlling XSS Filter*[34]
- IE Blog: *The XSS Filter*[35]
- Chromium Blog: *Security in Depth: New Security Features*[36]

3.2.6. X-Frame-Options

X-Frame-Options header can be used by server to indicate that page returned shall not be rendered inside **`<frame>`** and **`<iframe>`** tags and sites can use this as a defense from clickjacking attacks.

DENY
: Content of the page shall not be displayed in a frame regardless of the origin of the page attempting to do so.

SAMEORIGIN
: Content of the page can be embedded only in a page with the same origin as the page itself.

ALLOW-FROM
: Content of the page can be embedded only in a page with top level origin specified by this option.

The header returned from server allowing content to be embedded within https://example.com/ looks like this

```
X-Frame-Options: ALLOW-FROM https://example.com/
```

3.2.6.1. References

- RFC 7034: *http://tools.ietf.org/html/rfc7034*

3.2.7. X-Content-Type-Options

To provide better compatibility modern browesers usually come with a content-type sniffing algorithm, which allows them to infer content type of file by inspecting its content. This is useful in cases when

[33] http://p42.us/ie8xss/
[34] http://blogs.msdn.com/b/ieinternals/archive/2011/01/31/controlling-the-internet-explorer-xss-filter-with-the-x-xss-protection-http-header.aspx
[35] http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx
[36] http://blog.chromium.org/2010/01/security-in-depth-new-security-features.html

HTTP reponse does not include Content-Type header or if its mismatched. By correctly rendering the content and ignoring mismatched MIME type browser gains competitive advantage over other browser who do not render such file correctly.

Even though such behaviour enhances user experience, it also has impact on security. Suppose web application allows users to upload and download content and to protect from malicious file types, it implements content type filters that ban possibly dangerous file types. Attacker can upload malicious file with benign `Content-Type` that will pass web applications filters and server will store the file along with declared MIME type. When users download such file, server will include stored type in `Content-Type` header. However, browser's content-type sniffing algorithm will determine the correct type and ignore received `Content-Type` header, making the client vulnerable.

To prevent browsers from using content-type sniffing, server can include

```
X-Content-Type-Options: nosniff
```

header to enforce the type sent in `Content-Type` header.

3.2.7.1. References

- Internet Explorer Dev Center: *Reducing MIME type security risks*[37]

3.2.8. Configuring Rails

Enabling security related headers in Rails application is simplified by *SecureHeaders*[38] gem. After installation, it automatically adds:

- Content Security Policy
- HTTP Strict Transport Security
- X-Frame-Options
- X-XSS-Protection
- X-Content-Type-Options

After adding the gem to project's Gemfile

```
gem 'secure_headers'
```

enable its functionality by adding `ensure_security_headers` directive to ApplicationController:

```
class ApplicationController < ActionController::Base
  ensure_security_headers
end
```

Configuration of the header values can be done by creating an initializer and overriding default gem configuration:

```
```

[37] http://msdn.microsoft.com/en-us/library/ie/gg622941
[38] https://github.com/twitter/secureheaders

```
::SecureHeaders::Configuration.configure do |config|
config.hsts = {:max_age => 20.years.to_i, :include_subdomains => true}
config.x_frame_options = 'DENY'
config.x_content_type_options = "nosniff"
config.x_xss_protection = {:value => 1, :mode => 'block'}
config.csp = {
  :enforce => true,
  :default_src => "https://* self",
  :frame_src => "https://* http://*.twimg.com http://itunes.apple.com",
  :img_src => "https://*",
  :report_uri => '//example.com/uri-directive'
}
end
```

It is important to set `:enforce` to `true` in CSP configuration, because SecureHeaders defaults to false, which indicates Content-Security-Policy-Report-Only header will be sent and the policy will not be enforced, only monitored (see *Section 3.2.3, "Content Security Policy (CSP)"*). SecureHeaders will also set value of `:default_src` to all empty directives explicitly and not rely on the user agent's behaviour.

3.2.9. Guidelines and recommendations

Following are general recommendations based on previous sections regarding client side security:

Avoid JSONP pattern for cross-origin resource sharing
: JSONP pattern emerged as a workaround of Same Origin Policy in case web application needs to share resources across domains. Such approach creates a big attack surface and JSONP hijacking is dangerous even for application that don't use JSONP pattern, but return JavaScript content on GET requests (see *Section 3.2.2.2, "JSON with padding (JSONP)"*).

Use SSL for all connections and use HSTS to enforce it
: Using non-SSL connection is a serious weakness of web application with regards to network attackers. Enforcing SSL connection by redirection is often insufficient too, and it is desirable to add HSTS header to SSL enabled web applications (see *Section 3.2.4, "HTTP Strict Transport Security"*).

Use Content Security Policy
: Content Security Policy is quickly becoming standardized and provides a robust solution against XSS attacks and untrusted content loaded in the context of web page in general. Adopting it requires a web application to be compliant and enforces already accepted good practices with regards to script inlining (see *Section 3.2.3, "Content Security Policy (CSP)"*).

Use experimental security related headers for additional hardening
: Several non-standard HTTP headers that control implementation-specific behaviour of some user agents can be used to provide additional hardening of web application. These include `X-Frame-Options`, `X-XSS-Protection` and `X-Content-Type-Options` (see *Section 3.2.7, "X-Content-Type-Options"*). In case of CSP `X-WebKit-CSP` and `X-Content-Security-Policy` can be used to provide better compatibility with older Mozilla and WebKit-based browsers (see *Section 3.2.3, "Content Security Policy (CSP)"*).

3.3. Application server configuration and hardening

Defense in depth approach to security reminds us to add layers of security further down the stack, rather than focus on the interface between client and the server.

This section lists common problems related to server configuration and hardening with examples of vulnerabilities together with best practices on how to avoid them.

3.3.1. Logging

Logging is another surprising attack vector for the attackers. Logs are used to collect information about the state of the system, used by administrators in decisions about the system configuration and, in case of machine compromise, post hoc analysis. In both cases it is important that logs contain correct and accurate information. To achieve this, application developers tend to log various inputs to the application, which helps with debugging the problems, but creates opportunity for the attacker to forge logs.

Prerequisite for log forging is pasting potentiall untrusted unescaped input directly into logs as part of the logged message. Ability to include special characters such as newline is often enough to create false log messages. Example of this kind of vulnerability is CVE-2014-0136 CFME: AgentController get/log application log forging. The vulnerable part of code was

```
$log.info "MIQ(agent-get): Request agent update for Proxy id [#{params[:id]}]"
```

The application logged the ID of proxy that was requested as-is, taken from the parameters supplied by the client. If the ID parameter looked like this ("%0A" is percent encoded line feed)

```
1%0AMIQ(agent-get): Agent shutdown
```

then logs would contain two messages, one logged by the agent, one specified by the attacker:

```
MIQ(agent-get): Request agent update for Proxy id 1
MIQ(agent-get): Agent shutdown
```

This example is simplified by omitting details like timestamps from logs, which are not secret and attacker would be able to spoof them, too.

Perhaps slightly surprising consequence of logging unescaped characters is the potential to "delete" or "modify" the existing log messages, using backspace control character. Of course, the existing log messages would be intact, but the backspace control characters would alter existing messages when viewed in text editor. If the ID parameter contained backspace characters:

```
%08%08%08%08%08%08%08%08%08Server id 7
```

so the log stored on disk would contain

```
MIQ(agent-get): Request agent update for Proxy id %08%08%08%08%08%08%08%08%08Server id 7
```

If this log was opened in text viewer, such as lsess, control sequences would be interpreted and the administrator would see

```
MIQ(agent-get): Request agent update for Server id 7
```

Important

Always make sure potentially untrusted input is escaped before being logged and make sure user supplied input is quickly recognizable and cannot be confused with data such as logged messages, timestamps etc.

Appendix A. Revision History

Revision 1-1 **Tue Feb 18 2014** **Ján Rusnačko** *jrusnack@redhat.com*

Initial creation of book

Revision 1-2 **Tue Jan 28 2015** **Ján Rusnačko** *jrusnack@redhat.com*

Completed section 3.3 Common Attacks and Mitigations
Added section on secure logging
Updated section 2.2 Symbols on Ruby 2.2 symbol garbage collection
Lots of minor fixes

Index