

Module 14 - Pentesting APIs & Cloud Applications

▼ Introduction to APIs

API ⇒ Application Programming Interface

- non-GUI collection of endpoints in a standardized form to be used by human user as well as a machine.
- It often has documentation that can be both machine and human-readable form.

-
- Lots of APIs
 - Windows API
 - Remote APIs like RPC (remote procedure call)

Our focus will be on Web APIs

- Web Services (SOAP/XML)
- REST APIs (JSON)

From a technical standpoint, API differs from a website because:

- It has a standardized input/output form so that it can be scripted.
- It is language independent (it should work on each platform in the same way).
- It aims to be secure (e.g., it allows only some predefined methods).

▼ SOAP API utilizes Simple Object Access Protocol

SOAP Messages (HTTP Requests) are an XML type and must contain some special elements.

- Content type text/xml is also allowed.
- SOAPAction is sometimes used just for the standard and sometimes needs to hold the called method name.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:m="http://www.example.com">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:MethodName>
      <m:ParamName>PARAMETER VALUE</m:ParamName>
    </m:MethodName>
  </soap:Body>
</soap:Envelope>
```

Here is the sample response which follows the SOAP standard and is in XML format.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <MethodResult xmlns="http://tempuri.org/">
      <ResultValue>TheValue</ResultValue>
    </MethodResult>
  </soap:Body>
</soap:Envelope>
```

- For SOAP APIs, documentation is stored in WSDL files.

are stored under the „?wsdl” path, for example,
<https://api.example.com/api/?wsdl>.

You can take a look at an exemplary calculator service online at address:

<http://www.dneonline.com/calculator.asmx>

At the following address,
<http://www.dneonline.com/calculator.asmx?op=Add>, you
can see an exemplary SOAP request that was issued in
order to speak to the calculator service.

You can also see the full WSDL file at:
<http://www.dneonline.com/calculator.asmx?wsdl>

▼ REST - Representational State Transfer

Another type of API is REST (Representational State Transfer) APIs. Usually, the method client is about to call is in the resource path:

GET /api/v2/**methodName**

In REST APIs, HTTP methods have some special meaning:

- GET – Read resource
- POST – Create resource
- PUT – Update resource
- DELETE – Delete resource
- PATCH – Update resource partially

An exemplary REST API request can be seen to the right:

- Path often contains the API version
- Content-Type application/json header is required
- Parameters are passed as JSON array

```
POST /api/2.2/auth/signin HTTP/1.1
HOST: my-server
Content-Type: application/json
Accept: application/json

{
  "credentials": {
    "name": "administrator",
    "password": "password",
    "site": {
      "contentUrl": ""
    }
  }
}
```

It is also often possible to pass the REST API parameters as XML, so the equivalent of the request from the previous slide would look like the listing to the right.

```
POST /api/2.2/auth/signin HTTP/1.1
HOST: my-server
Content-Type: text/xml

<tsRequest>
  <credentials name="administrator"
password="passw0rd">
    <site contentUrl="" />
  </credentials>
</tsRequest>
```

REST API also has a documentation standard called the WADL file. A sample WADL can be viewed here:

<https://www.w3.org/Submission/wadl/>

Similar to WSDL, we will shortly present tools that help to parse the lengthy file in order not to rewrite all the methods manually.

In order to make developer's (and penetration testers') lives easier, some APIs include a more human-friendly API representation. For example, a very popular API engine named Swagger is often found with its demo page, which contains forms with description and possibility to issue a request to each method.

You can see sample Swagger API here: <https://swagger.io/tools/swagger-ui/>. Click on „Live Demo” to try it yourself.

▼ API testing & Attacking

It is possible than when you request /api/anything then every character past „anything” is parsed by the API engine, but you can still find interesting files on the server under, for example /version.txt.

Regardless of the fact that APIs make use of predefined methods, you should be aware that there can still be vulnerabilities related to:

- Parameters to these predefined functions
- The API parsing itself
- Access to sensitive methods

When Facing an API during a pentest, first thing to do it *Recon*

- What's API name and version?
- is it a custom implementation or an open-source product?
- is there any online documentation available?
- are there any interesting methods?
- Does the documentation exist on the target server (?wsdl, ?wadl, or similar)?
- Does the API require authentication, or is publicly available?
- If there is both local and public documentation for an API, do they match?
- Maybe some methods were hidden from local users (typically ones that allow insecure operations).

Tools:

- Postman
- SOAPUI
- Burp pro WSLDer extention

▼ API Access Control

In larger APIs, not every method is designed to be used by each user. For example, the most common split is between read-only users and read+write users. The latter has the possibility to modify the contents of the API backend.

In APIs, you will rarely see cookies being used. More often, the authentication mechanism will be basic authorization or a kind of token – it can be a pre-generated token that will be equivalent of a cookie, for example in the form of a header, like **X-API-Token: adk32Kds38au39aU0s**.

Broken access control are often found in APIs

- Authorization Bypasses are very common

In order to test an API in a complex way for Access control flaws, one needs to:

- Prepare a working request to each API endpoint
- Generate a token (or authorization header) for each of the API users
- Combine each API request with each token to see which will work and which do not
- Remember to test each request, also without any token

Again, such test cases might be generated using SoapUI, which allows us to issue a request to each API endpoint. Also, as a reminder, double-check if the API implementation uses all the methods provided by the original version.

For example, with Rundeck API there is a default possibility of running OS commands, which might be hidden from the documentation on a local API implementation.

- <https://docs.rundeck.com/docs/api/rundeck-api.html#adhoc>

API Tokens are susceptible to vulnerabilities commonly diagnosed in session cookies, for example:

- Low entropy or predictable value
- Lack of invalidation
- Possible token leaks from the application infrastructure, or possibility to generate tokens in advance.

Tokens that might grant you access to an API interface are **JWT** and **Bearer Authentication**.

▼ Resource Sharing

SOP loses some constraints by CORS

Interesting Headers for us:

- Access-Control-Allow-Origin: VALUE
 - specify the allowed domains? allowed to access website's response.
 - VALUE can be domain, wildcard or null.
- Access-Control-Allow-Credentials: true/false

▼ Attacking Cloud Based Applications

▼ Microservices

Monolithic design:

- One server hosts the web application and required services like databases.
- While easy to set up and maintain at a low cost, it's difficult to scale.
- Updates may cause downtime, and a single point of failure can be catastrophic without a backup plan.

Tiered Monolithic:

- Services are separated, with the web server hosting the application and another server managing the database.

- This architecture allows updates without downtime, and clustering and load balancing can improve performance.
- However, scalability remains a challenge, and the cluster can be a single point of failure, requiring recovery from backups in case of disaster.

Cloud solutions:

- Built-in elastic servers enable horizontal scaling and full automation, improving performance by creating new instances as needed.
- Updates can be performed without downtime, and disasters typically don't require backups.
- Despite advantages over previous designs, issues remain at the application layer due to the monolithic codebase, and costs can be unpredictable based on service requirements.

▼ Serverless Applications

FaaS - Function as a Service:

- Serverless applications running in a cloud environment, managed by the cloud operator.
- Advantages: Avoids complexity of building and maintaining infrastructure.
- Limitations: Execution time, threads, disk space, RAM, code package size, dependencies.
- Requirements: Trigger/event to run the application, routing method or API gateway.
- Suitability: Not ideal for tasks needing more than 10 minutes of execution.

▼ Details of Serverless Architecture

Serverless architecture differs from normal web applications.

Key concepts:

- API Routing: Routing layer based on URL association, rules, and parameters, enabling functions to be accessed from the internet. In

AWS, it's called API Gateway.

- State: Function lifespan is brief, leading to no local cache. Vulnerabilities like file command injections or uploads are exploited differently due to this.

Cold Start:

- Limited lifespan requires code to be downloaded, containerized, booted, and primed for initial execution.
- Third-party plugins like Serverless-plugin-warmup can mitigate this.

Debugging:

- Lack of infrastructure management limits debugging access to logs.
- Alternative approaches like printing variables or using local lambda are typically used for debugging.

No Ops:

- Minimal sysadmin tasks as environment is managed by cloud operator.
- However, backups, security monitoring, and logging are still necessary.

▼ **Serverless Application Example**

DVSA - Damn Vulnerable Serverless Application

- by OWASP ^_^

▼ **S3 Buckets**

S3 - Simple Storage Service

Tool:

- <https://github.com/mxm0z/awesome-sec-s3>
- S3Recon

Wordlists:

- <https://github.com/koaj/aws-s3-bucket-wordlist/blob/master/common-s3-bucket-names-list.txt>

Running S3 recon with the wordlist file can be done with the following command:

```
s3recon "word-list.txt" -o "results.json" --public
```

```
root@kali:~# s3recon "word-list.txt" -o "results.json" --public
- PRIVATE https://s3.ap-south-1.amazonaws.com/apple-test
- PRIVATE https://s3.us-west-2.amazonaws.com/test-uber
- PRIVATE https://s3.ca-central-1.amazonaws.com/google-dev
- PRIVATE https://s3.ap-southeast-1.amazonaws.com/apple-test
- PRIVATE https://s3.ap-southeast-2.amazonaws.com/google-testing
- PRIVATE https://s3.ap-south-1.amazonaws.com/google-dev
- PRIVATE https://s3.ap-northeast-2.amazonaws.com/lyft
- PRIVATE https://s3.us-west-2.amazonaws.com/google
- PRIVATE https://s3.us-west-1.amazonaws.com/dev-microsoft
- PUBLIC https://s3.us-east-1.amazonaws.com/amazon-dev
- PRIVATE https://s3.ca-central-1.amazonaws.com/test-google
- PRIVATE https://s3.us-east-1.amazonaws.com/google
```

Buckets marked as "public" could give access to restricted content. Objects could be accessed via aws-cli.

▼ S3 AWS Signed URLs

▼ Serverless Event Injection

Serverless functions listen for events or triggers in order to be executed. These events can be injected from other trusted sources in cloud environments leading to a Serverless Event Injection vulnerability. These trusted sources can be:

- Actions on S3 Objects
- Alerting Systems (Cloudwatch)
- API Gateway Calls
- Changes in the code repository
- Database events
- HTTP APIs

▼ GraphQL APIs

- only **Query** and **Mutate** instead of (GET, POST, PUT, DELETE, PATCH ⇒ REST)
- There's only one endpoint to an API (instead of too many endpoints ⇒ REST)
- usually example.com/graphql or something similar (Google Dorks ^^)

- REST usually has one endpoint for each type of object (users, groups, items, books, orders, shipments...etc) with 3 or more operations on each endpoint
- In graphql, the same endpoint serves all predefined objects under both Query and Mutation methods.

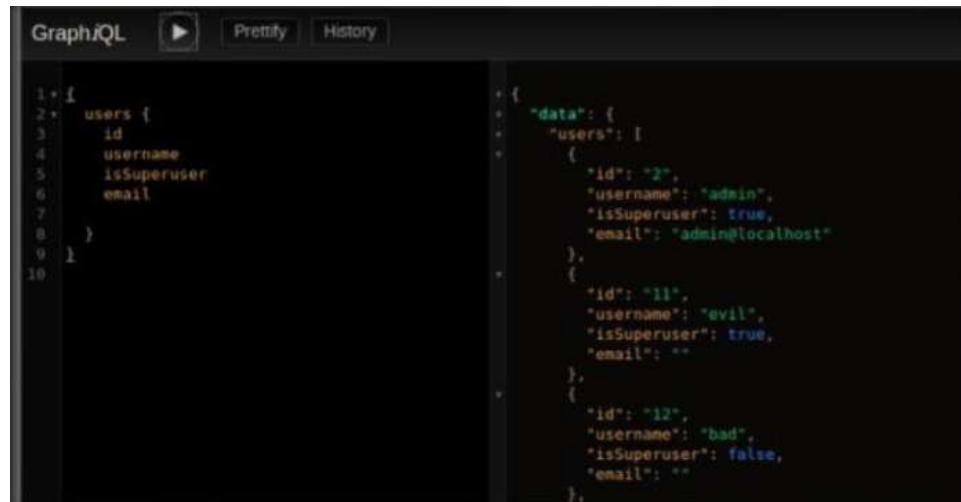
GraphQL Terms:

- **Query:**
 - a query operation on an object or type.
- **Mutate:**
 - an update operation on an object like
 - creating a new one
 - updating it fully
 - updating it partially
 - or deleting it.
- **Type (objecttype):**
 - A type of object, like a class or table, eg: Users, Orders, Books

More GraphQL Terms:

- **Schema:**
 - Describes the types, fields and actions available.
- **Introspection:**
 - A method to learn more about the schema details like types and fields.
- **Resolver:**
 - A function that connects schema definitions to actual backend data sources like SQL tables.
- **Scalar Type:**
 - Type of data for a field, like string, int or custom types.

Simple GraphQL query:



The screenshot shows the GraphQL Playground interface. On the left, a query is entered:

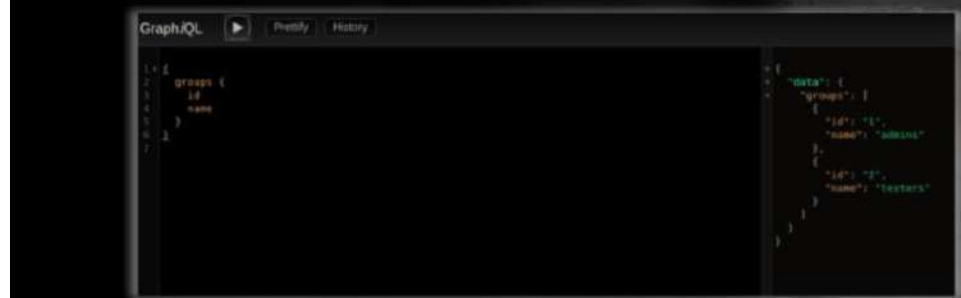
```
1 query {
2   users {
3     id
4     username
5     isSuperuser
6     email
7   }
8 }
9
10
```

 On the right, the JSON response is displayed:

```
{
  "data": {
    "users": [
      {
        "id": "2",
        "username": "admin",
        "isSuperuser": true,
        "email": "admin@localhost"
      },
      {
        "id": "11",
        "username": "evil",
        "isSuperuser": true,
        "email": ""
      },
      {
        "id": "12",
        "username": "bad",
        "isSuperuser": false,
        "email": ""
      }
    ]
  }
}
```

GraphQL can also be called from the command line using curl.

- Using POST
- Content-type is JSON
- Output is sent to jq for pretty JSON



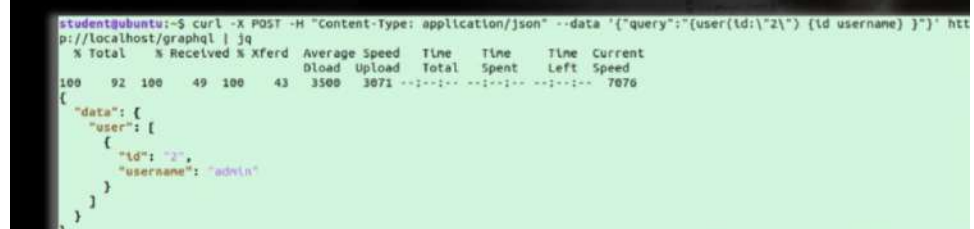
The screenshot shows the GraphQL Playground interface. On the left, a query is entered:

```
1 query {
2   groups {
3     id
4     name
5   }
6 }
7
```

 On the right, the JSON response is displayed:

```
{
  "data": {
    "groups": [
      {
        "id": "1",
        "name": "admins"
      },
      {
        "id": "2",
        "name": "testers"
      }
    ]
  }
}
```

Calling a particular object in GraphQL:



The screenshot shows a terminal window with the following command and output:

```
student@ubuntu:~$ curl -X POST -H "Content-Type: application/json" --data '{"query":"{user(id:\"2\") {id username}}}" http://localhost/graphql | jq
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 92 100 49 100 43 3500 3071 --:--:-- --:--:-- --:--:-- 7076
{
  "data": {
    "user": {
      "id": "2",
      "username": "admin"
    }
  }
}
```

Graphql nesting queries:

- Display each user with his group subscriptions using graphql, showing the id and name of the group
- Hint: groups {id name}
- Try both the GraphiQL and Curl

Security in GraphQL:

- GraphQL has no built-in understanding of security. It will return the object as it was requested.
- Without explicit filtering, sensitive data could be exposed and extracted.
- Can we read user sensitive info such as passwords?

Making updates in graphql:

- In GraphQL, updates (Addition, Creation, Deletion) are called mutations.
- Let's check the source code
- We have 3 mutations

```
class Mutation(graphene.ObjectType):
    create_user = CreateUser.Field()
    update_user = UpdateUser.Field()
    delete_user = DeleteUser.Field()
```

Deleteuser mutation

- The deleteUser mutation can be called by:
 - Defining the query type to be a mutation
 - Selecting the named deleteUser mutation
 - Supplying the id to be deleted, and a sub selection for response (ok field here)

```
mutation deleteUser {
  deleteUser(id:24) {
    ok
  }
}
```

```
student@ubuntu:~$ curl -s -X POST -H "Content-Type: application/json" --data '{"query":"mutation {deleteUser(id:22){ok}}"}' http://localhost/graphql | jq
{
  "data": {
    "deleteUser": {
      "ok": true
    }
  }
}
```