OWASP TOP 10 API

Kavitha Venkataswamy
https://www.linkedin.com/in/kavitha-venkataswamy-90b23ab/

ABOUT ME

- *Software engineer background*
- *Application Security for 8+ years*
- *Anything in Appsec, Threat modeling, SSDLC, SAST, DAST, Software composition analysis, vulnerability management, shift left security, Security Champions program…*

Traditional Web Application

1) Traditional Web Application

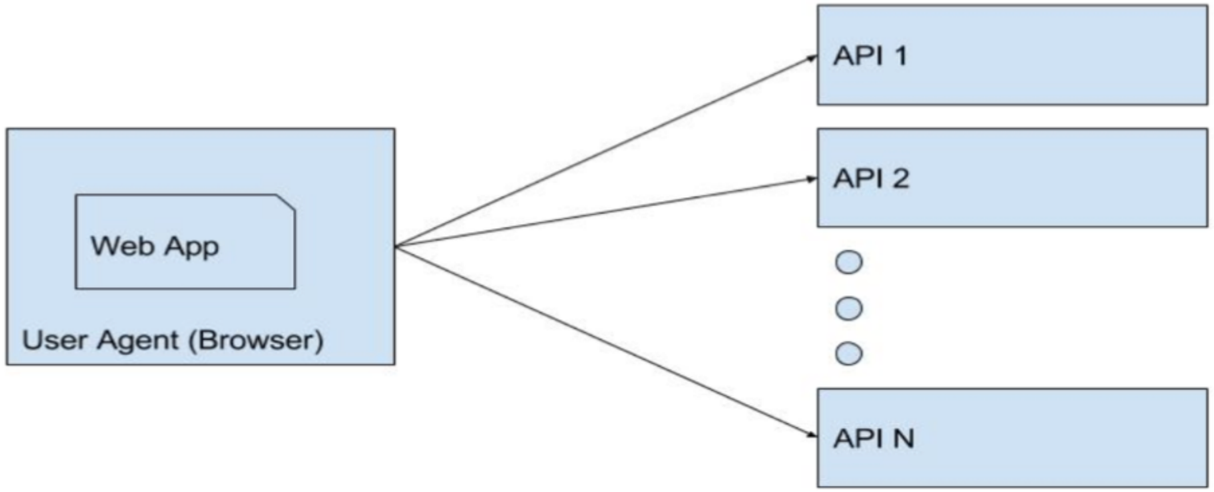
- Full HTML page returned on each HTTP call
- Client and Server code are tightly coupled
- Regardless of authentication method, security session is tracked with a cookie (JSESSIONID or similar) that contains an opaque token or reference token.

Single Page Application – Early Adoption

2) Single Page Application (SPA) + Single API Endpoint

- Static content and javascript downloaded from server.
- Application authenticates user, obtains some type of token or cookie for security session tracking, and makes a series of API calls.
- client and API are likely tightly coupled]

Single Page Application + Multiple API Endpoints



3)	Single Page Application (SPA) + Multiple API Endpoints

- Static content and APIs served from Web Server.
- Application authenticates user, obtains some type of token or cookie for security session tracking, and makes a series of API calls.
- client and API are likely tightly coupled

Many Consumers / Providers

4) Diagram 4: N API Consumers, Multiple types of consumers; N API Providers

- Many types of APIs
- Organization has a common API Gateway
- APIs are reusable across different consumers and types of consumers.
- Many types of consumers.
- API Providers and API Consumers are loosely coupled.

WEB SECURITY

- *Web applications, data processing is done on the server side, resulting webpage sent to to client browsers*
- *WAF plays a major role blocking malicious traffic*

API SECURITY

- *Evolution of Mobile apps, Microservices, richer interactive user experience*
- *Bots and malicious actors shifting from attacking web sites to attacking public APIs*
- *Apps can store more data and preserve more state between sessions, new approaches to secure like securing API key etc*
- *API fetches data and clients render and maintain the state*
- *API expose application logic and sensitive data like PII*
- *Exposes access keys, tokens etc*

TOP 10 API SECURITY RISK

API1:2019 - Broken Object Level Authorization	APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.
API2:2019 - Broken User Authentication	Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising system's ability to identify the client/user, compromises API security overall.
API3:2019 - Excessive Data Exposure	Looking forward to generic implementations, developers tend to expose all object properties without considering their individual sensitivity, relying on clients to perform the data filtering before displaying it to the user.
API4:2019 - Lack of Resources & Rate Limiting	Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force.
API5:2019 - Broken Function Level Authorization	Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers gain access to other users' resources and/or administrative functions.

TOP 10 API SECURITY RISK

API6:2019 - Mass Assignment	Binding client provided data (e.g., JSON) to data models, without proper properties filtering based on a whitelist, usually lead to Mass Assignment. Either guessing objects properties, exploring other API endpoints, reading the documentation, or providing additional object properties in request payloads, allows attackers to modify object properties they are not supposed to.
API7:2019 - Security Misconfiguration	Security misconfiguration is commonly a result of unsecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, permissive Cross-Origin resource sharing (CORS), and verbose error messages containing sensitive information.
API8:2019 - Injection	Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
API9:2019 - Improper Assets Management	APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. Proper hosts and deployed API versions inventory also play an important role to mitigate issues such as deprecated API versions and exposed debug endpoints.
API10:2019 - Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems to tamper with, extract, or destroy data. Most breach studies demonstrate the time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

1. BROKEN OBJECT LEVEL AUTHORIZATION

- *Exploiting API endpoints by manipulating ID of an object sent in the request*
- *IDs are sent from the client to API*

  GET : getUserDetails/241
- *Leads to unauthorized access to sensitive data*

1. BROKEN OBJECT LEVEL AUTHORIZATION

Scenario :

POST : product/updateReview. { user:140 review:1 point}

What is Missing in here

- *Access Control Checks*
- *Validate that logged-in user does have access to perform the requested action on the requested object.*

1. BROKEN OBJECT LEVEL AUTHORIZATION

Prevention

- *Implement a proper authorization mechanism that relies on the user policies and hierarchy.*
- *Authorization check in every function*
- *User GUID instead of UID*
- *Write testcases*

1. BROKEN OBJECT LEVEL AUTHORIZATION

Account Take Over :

- *Step #1 Getting user UUID of any Uber user with phonenumber*

Request : {“nationalPhoneNumber”:”99999xxxxx”,”countryCode”:”1"}

Response : {“status”:”failure”,”data”:{“code”:1009,”message”:”Driver ‘47d063f8–0xx5e-xxxxx-b01a-xxxx’ not found”}}

Outcome : Leaked UUID of user

1. BROKEN OBJECT LEVEL AUTHORIZATION

Step #2 : Use victim UUID , get PII including Mobile Access Token here

```
POST /marketplace/_rpc?rpc=getConsentScreenDetails

Host:"bonjour.uber.com
Accept:"application/json
Origin":"https"::"xxxx
Content-Type:"application/json
Accept-Encoding":"gzip","deflate
Cookie:"xxxxx"{"language":""en"",""
userUuid"":"xxxx-776-4xxxx1bd-861a-837xxx604ce"}
```


Vuln Disclosure : AppSecure

2. BROKEN USER AUTHENTICATION

- *Authentication endpoints and flows are assets that need to be protected*
- *Forgot Password / Reset Password protected same as login*

2. BROKEN USER AUTHENTICATION

VULNERABLE API

- *Credential Stuffing*
- *Unsigned/Weak JWT tokes ("alg":"none")*
- *Auth tokens, password in URL*
- *Weak encryption keys*
- *Weak hashes*
- *No Account Lockout*

2. BROKEN USER AUTHENTICATION

- *POST /api/health/verificationcodes*
- *Forgot Password / Reset Password protected same as login*
- *Sample Reset Password flow below.*

POST api/healthsystem/VerifyCodes

Verify Code = (6 digit code)

Without RateLimit API, brute forcing attempts to set verify code “x times” eventually to reset password .

2. BROKEN USER AUTHENTICATION

Prevention

- *Implement MFA when possible*
- *Anti brute force, Rate limit API*
- *Implement Account Lockout*
- *Captcha*
- *Use standards for authentication, token generation and password storage*

3 . EXCESSIVE DATA EXPOSURE

- *Sniffing API traffic for sensitive data*
- *API responses – more data than required*
- *Client filters data for display*

3 . EXCESSIVE DATA EXPOSURE

Request :
GET /api/user/reviews

Response:
Returns comments and user data also.
The endpoint implementation uses a generic **toJSON()** method on the User model, which is PII data.

3 . EXCESSIVE DATA EXPOSURE

BE CONSERVTATIVE IN EXPOSING DATA

- *Principle of Least Privilege - Expose only required data for API call*
- *Minimizes accidental exposure but also prevents disclosures of personal information that can be inferred by correlating data from different datasets*

3 . EXCESSIVE DATA EXPOSURE

Disclosure:Ibrahim Balic

3 . EXCESSIVE DATA EXPOSURE

- *Dec 2019 - Design flaw in Twitter Android API – Disclosed*

- *Twitter's API was allowing massive generated lists of phone numbers to be uploaded to their account matching API. Via this method, over 17 million phone numbers were matched to Twitter accounts*

- *Upload entire lists of generated phone numbers through Twitter's contacts upload feature and fetches user data in return.*

4 . EXCESSIVE DATA EXPOSURE

- *Never rely on client to filter data*
- *Review API responses*
- *Developers ask this question “ Who is the consumer of data”*
- *Avoid using to_json(), to_string()*
- *Return only specific properties*
- *Data classification*

4 . LACK OF RESOURCES AND RATE LIMITING

- *Multiple concurrent resources request*
- *Requests from multiple API clients compete for resources, causing Denial or service attacks*

4 . LACK OF RESOURCES AND RATE LIMITING

Request :

GET /api/users?page=1&size=100

Attacker changes size from 100 to 800000

Response:

API unresponsive

Buffer overflow errors

Denial of service

4 . LACK OF RESOURCES AND RATE LIMITING

- *Limit on how often a client can call the API within a defined timeframe*
- *Server side validation on control parameters*
- *Define maximum size on input data*

5 . BROKEN FUNCTION LEVEL AUTHORIZATION

- *Attackers sending legitimate API calls to the API endpoint they should not have access to*
- *API s are structured. Easy for attacker to figure out this flaw.*

Example : Changing HTTP method from GET to PUT

Changing url string from “users” to “admin”

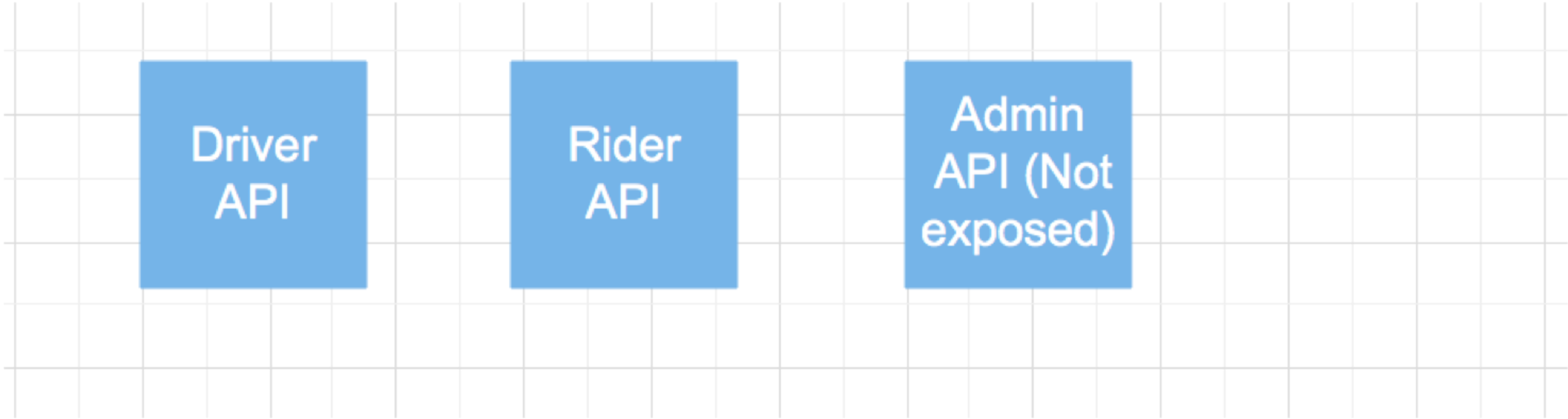
GET api/users/100

DELETE api/users/100 (EASY to guess)

5 . BROKEN FUNCTION LEVEL AUTHORIZATION

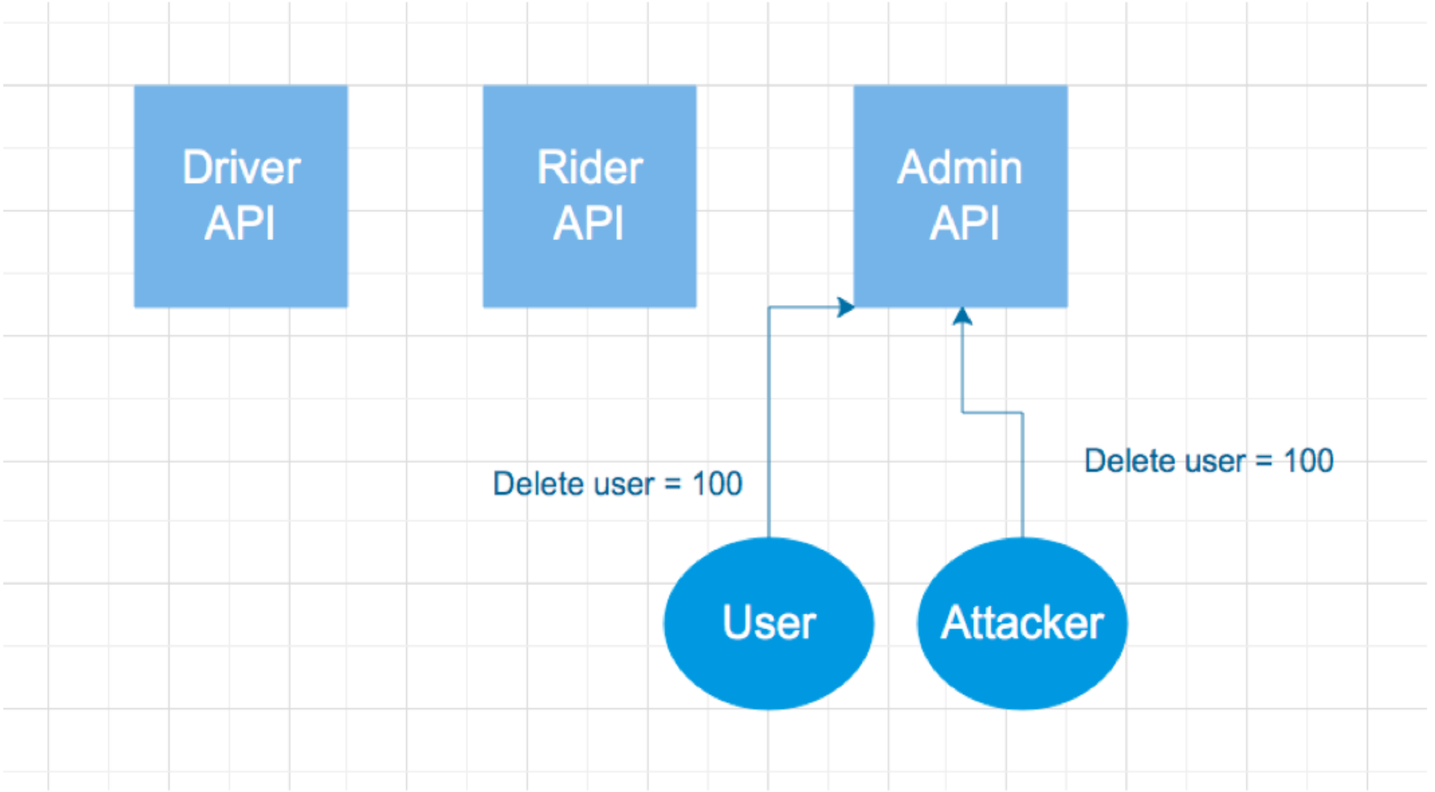
- *Can regular user access admin end point?*
- *Can user can perform sensitive actions where supposed not to, say from GET to DELETE?*

5 . BROKEN FUNCTION LEVEL AUTHORIZATION



5 . BROKEN FUNCTION LEVEL AUTHORIZATION

Function level authorization checks missing



5 . BROKEN FUNCTION LEVEL AUTHORIZATION

Prevention

- *Deny all access by default, requiring explicit grants to specific roles for access to every function*
- *Implement in Code, configuration and API Gateway*
- *Disable unused HTTP methods*
- *Have authorization checks*

6. MASS ASSIGNMENT

- *Binding of client-provided data (e.g., JSON) to data models, without properties filtering based on a whitelist,*
- *Blindly relying on an ORM.*
- *Automatically binding objects and their properties to incoming data.*
- *Either guessing objects properties, exploring other API endpoints, reading the documentation*
- *Not preventing the client from changing values that should be read-only*
- *Not enforcing schemas, types & patterns server-side.*

6. MASS ASSIGNMENT

```
<form>
    <input name=username type=text>
    <input name=password type=text>
    <input name=email text=text>
    <input type=submit>
</form>
```

The object that the form is binding to is as follows:

```
public class User {
   private String username;
   private String password;
   private String email;
   private boolean isAdmin;
   //Getters & Setters
 }
```

6. MASS ASSIGNMENT

```
@RequestMapping(value = "/editUser", method = RequestMethod.
  public String submit(User user) {
     userService.edit(user);
     return "successPage";
  }
```

POST /editUser

..

..

username=malicious&password=somepass&email=malicious@mal.com

&isAdmin=true

6. MASS ASSIGNMENT

- *Harbor – 2019 (cloud registry)*
- *Popular open-source system for registering container base images*
- *Used an ORM for writing objects to the database.*
- *Researchers found that the user objects have an admin role flag on them that directly maps to a json object field.*
- *They added the json flag to their create user post request and were able to create an admin account.*
- *Attacker tampered with the base images used at a company.*

6. MASS ASSIGNMENT

Prevention

- *Do not bind automatically bind client input to objects*
- *Whitelist only properties updated by client*

7. SECURITY MISCONFIGURATION

- *Unpatched flaws, common endpoints, or unprotected files and directories to gain unauthorized access*
- *Can expose not only sensitive data but server compromise also*

7. SECURITY MISCONFIGURATION

- *Lack of Secure Headers*
- *Unnecessary exposure of HTTP Methods*
- *Missing CORS policy*
- *Improper Cloud configurations*

7. SECURITY MISCONFIGURATION

Equifax - 2018

- *Lack of Secure Headers Unpatched Apache Struts*
- *Lack of control over Content-Type HTTP header content*
- *Hackers exploited crafting a header with a malicious payload.*
- *Their Alerting system failed due to a security certificate that had been expired for 19 months!*
- *148+ million customers impact*

7. SECURITY MISCONFIGURATION

Prevention

- *Hardening, Lock down*
- *Review configurations across API stack. Orch files, API, s3 buckets ..*
- *Automate process to find configuration flaws*
- *Disable unused HTTP methods*
- *API's accessed by browser based clients - implement CORS*
- *Make sure certs are up to date*

8. INJECTION

- *Unpatched flaws, common endpoints, or unprotected files and directories to gain unauthorized access*
- *Client-supplied data is not validated, filtered, or sanitized by the API*
- *Client-supplied data is directly used or concatenated to queries*

8. INJECTION

DELETE /api/bookings?bookingId=XX

```
router.delete('/bookings', async function (req, res, next) {
  try {
      const deletedBooking = await Bookings.findOneAndRemove({'_id' : req.query.bookingId});
      res.status(200);
  } catch (err) {
     res.status(400).json({error: 'Unexpected error occured while processing a request'});
  }
});
```

The attacker intercepted the request and changed `bookingId` query string parameter as shown below. In this case, the attacker managed to delete another user's booking:

```
DELETE /api/bookings?bookingId[$ne]=678
```

8. INJECTION

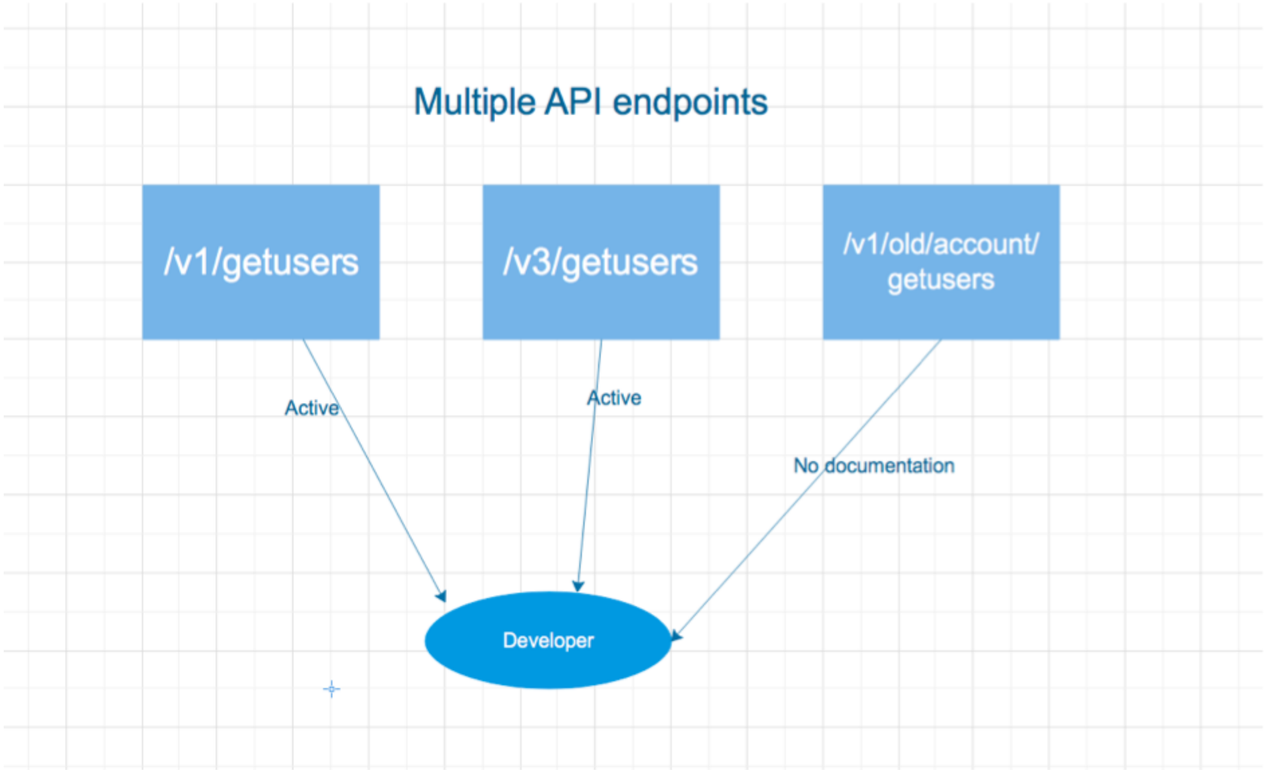
Prevention

- *Validate, Filter and sanitize all client data*
- *Special characters should be escaped using the specific syntax for the target interpreter*
- *Prefer a safe API that provides a parameterized interface.*

8. IMPROPER ASSET MANAGEMENT

- *Mismanagement of API Assets*
- *Having undocumented API endpoints.*
- *Having old APIs that are dormant or abandoned.*
- *Not having the same security standards between your Prod and non-prod environments.*

9. IMPROPER ASSET MANAGEMENT



9. IMPROPER ASSET MANAGEMENT

9. IMPROPER ASSET MANAGEMENT

- *JustDial – 2019 Breach*
- *Old unprotected API leaked data on all their users*

9. IMPROPER ASSET MANAGEMENT

Prevention

- *Inventory API hosts (Prod, staging ..)*
- *Generate API documentation using open standards, include in CI/CD pipeline*
- *Protection measures for all API , not only PROD*
- *Move to NEW API secure version*

10. INSUFFICIENT LOG & MONITORING

- *Video sharing Platform Dailymotion hit by “Credential stuffing attack”*
- *No alerts were triggered due to lack of monitoring*
- *After customer complaints, logs analyzed and found attack – password reset sent later*

10. INSUFFICIENT LOG & MONITORING

Prevention

- *Login failed attempts*
- *Use SIEM like tool to collect, aggregate logs*
- *Monitor, Create Alerts, Dashboard …*

BEFORE WE LEAVE

Usecase :

GET api/v4/orders/600

```
{
 “user id” : 341
  "name": "Flor",
  " email ": " ace@x.com ",
"order": {
   "id": "600",
   "item1": "XXX"
   "item1": "YYY"
  }
}
  "payment": {
   "Amount": "800",
   "Card": "7889990000099999"
  }
}
```

Some of Questions to ask :

- Can we change order id as GUID instead of guessable id ?
- Who is consuming this API?
- Do we really need payment method in response? Is client filtering response which is not recommended?
- Oh! I see card data not encrypted. ...
- Email is PII data, is this really needed by client? If not, remove this property from API response.
- I see v4 version, are older versions still active?
- What if GET api/v4/orders/**700** . Is API validation in place, not serving for user:700 since logged in user is #600. Does this error out?
- Are we logging sensitive info? check the logs
- RateLimit?
- ..

REFERENCES

- https://github.com/OWASP/API-Security/blob/master/2019/en/src/
- https://unit42.paloaltonetworks.com/critical-vulnerability-in-harbor-enables-privilege-escalation-from-zero-to-admin-cve-2019-16097/
- https://medium.com/appsecure/how-i-could-have-hacked-your-uber-account-e98e64ab51bb
- https://www.levvel.io/resource-library/api-security-vs-web-application-security

Thank You