wallarm

WALLARM.COM

A Technical Guide for Practitioners

Securing Enterprise APIs

+1 (415) 940-7077

188 KING ST. UNIT 508

SAN FRANCISCO, CA 94107

Contents

Executive Summary

APIs are the digital backbone of modern enterprises, enabling seamless integration, innovation, and scalability. However, their growing use has made them a primary target for malicious actors, putting sensitive data and systems at significant risk.

This white paper explores the importance of API security for enterprises, highlights common vulnerabilities, and presents best practices for securing APIs using a defense-in-depth approach supported by dedicated security tools.

We'll discuss why enterprises must prioritize API security, the pitfalls to watch out for, and how implementing layered security measures can mitigate risks. Designed for enterprise architects, CISOs, and security architects, this white paper provides actionable insights to safeguard APIs and ensure resilience against evolving security threats.

How are APIs used today?

Interconnectivity

APIs allow different systems, platforms, and devices to work together seamlessly. For example, APIs enable your fitness app to sync with your smartwatch.

Efficiency

Developers can use APIs to integrate pre-built functionalities (like Google Maps or payment systems) instead of building them from scratch, saving time and resources.

Scalability

APIs make it easier to scale applications by connecting to external services or databases without overloading the core system.

Innovation

APIs empower businesses to create new products and services by leveraging existing technologies. For instance, Uber uses APIs for maps, payments, and notifications to build its platform.

Data Sharing

APIs facilitate secure and structured data sharing between organizations, enabling better decision-making and collaboration.

Automation

APIs enable automation by allowing systems to interact without human intervention. For example, APIs can automate workflows in marketing, sales, or customer support.

User Experience

APIs enhance user experiences by integrating multiple services into a single platform. For example, social media platforms use APIs to let users log in to third-party apps with their accounts.

In short, APIs are the backbone of modern digital ecosystems, enabling innovation, efficiency, and seamless user experiences across industries.

In short, APIs are the backbone of modern digital ecosystems, enabling innovation, efficiency, and seamless user experiences across industries.

The Primary Challenges in API Security

APIs come in many flavors, from REST and SOAP to GraphQL and gRPC. Despite their differences, these technologies share common vulnerabilities that attackers exploit. Some of the key challenges include improper management, misconfiguration, and insufficient monitoring, all of which can significantly compromise security.

Let's review the primary challenges in detail:

wallarm

Common Pitfalls in API Security

Lack of Proper Authentication and Authorization

Lack of proper authentication and authorization in API security is a critical weakness that can open the doors to numerous threats, including unauthorized access, data breaches, and exploitation of sensitive information.

Authentication ensures only verified users or systems can access an API, while authorization dictates what actions or resources they can use. When these mechanisms are poorly implemented or absent, attackers can easily impersonate legitimate users or escalate their privileges to access restricted data. For instance, weak API keys or failure to enforce token validation can allow cybercriminals to bypass basic security measures, exposing private user information or manipulating malicious data.

A failure in proper authentication and authorization often results in serious real-world consequences. Take, for example, the case of a poorly designed e-commerce API that lacks fine-grained access controls. Attackers could gain unauthorized access to customer records, payment details, or even modify orders. A more alarming scenario may occur in industries like healthcare, where unprotected APIs expose confidential patient records to anyone who can exploit such vulnerabilities. This results in massive data privacy violations and can lead to regulatory fines and damage to organizational reputation. Prioritizing strong authentication and authorization protocols, such as OAuth 2.0 or multi-factor authentication, can mitigate these risks and safeguard APIs from becoming an easy target for exploitation.

PRO TIP

Implement an authorization layer that validates the user's permissions for every action.

Use randomly generated Globally Unique Identifiers (GUIDs) as object identifiers, which reduce guessability.

Ensure these mechanisms apply consistently across all resources.

Broken Object Level Authorization (BOLA) Vulnerabilities

BOLA (also known as Insecure Direct Object Reference, or IDOR) occurs when an API fails to properly validate whether a user is authorized to access a specific object, allowing attackers to manipulate object identifiers to access unauthorized data or perform unauthorized actions. It's the #1 threat in the OWASP API Security Top 10 (2023) due to its ease of exploitation and severe impact.

Example

Scenario

An e-commerce platform provides an API endpoint to display revenue charts for shop owners: `GET /shops/{shopName}/revenue_data.json`. The API uses the shopName parameter to fetch revenue data but does not verify if the authenticated user owns the specified shop.

Exploitation

An attacker, who is a legitimate user with their own shop, inspects the API request and replaces their `shopName` (e.g., my-shop) with another shop's name (e.g., competitor-shop). The API returns the competitor's revenue data without checking the user's ownership.

Business Impact

Unauthorized access to sensitive financial data, potentially leading to competitive advantage, blackmail, or data breaches.

Exposure of Sensitive Data

Exposure of sensitive data is a critical pitfall in API security that can lead to significant risk. APIs often handle sensitive information, such as user credentials, payment data, and personal identifiers, making them an attractive target for attackers.

For instance, an API might inadvertently return excessive data in its responses or fail to properly sanitize input and output, exposing it to threats like data breaches or unauthorized access. The consequences of such exposure are severe, ranging from financial losses and legal penalties due to non-compliance with regulations like GDPR or HIPAA, to long-term reputational damage for organizations. Mitigating this risk requires implementing robust encryption protocols, employing strict access controls, and conducting regular security audits to identify and address gaps before they are exploited.

Business Impact

APIs that expose sensitive data—like credentials, payment details, or personal identifiers—can lead to financial loss, legal penalties, reputational damage, and loss of customer trust. Such breaches enable identity theft, fraud, and competitive exploitation, making strict access control, encryption, and data filtering essential.

PRO TIPS

Explicitly define what data your API should return rather than exposing entire objects.

Validate API responses against a schema to ensure only necessary properties are accessible.

Detect Sensitive Data in API requests and responses to identify data exposed where it shouldn't be.

Poor Inventory and Visibility

Poor inventory and visibility of APIs pose a significant challenge to API security, as organizations often rely on many APIs, some of which may be undocumented, outdated, or improperly managed. Without a clear inventory, tracking which APIs are active, identifying potential vulnerabilities, or monitoring unauthorized access becomes nearly impossible.

These blind spots in visibility can lead to data breaches, exploitation of exposed endpoints, and compliance failures. Addressing this issue involves maintaining an up-to-date inventory of all APIs, ensuring proper documentation, and implementing monitoring tools to gain real-time insights into their usage. We at Wallarm have hundreds of conversations with customers that, due to the lack of proper security practices and modern tools, lack visibility into their API structure and services. This proactive approach is critical for mitigating security risks and safeguarding sensitive data. Failing to maintain an accurate inventory of API endpoints can create hidden vulnerabilities.

Example

Scenario

A healthcare company provides a cloud-based platform for managing patient health records. Over the years, it has developed multiple APIs to support integrations with hospitals, insurance providers, and third-party health apps. However, due to poor API lifecycle management, some APIs have been forgotten or left undocumented. The company had an API for a now-defunct mobile app that allowed patients to view their health records. This API (a zombie API) was never decommissioned, even though the app was retired.

Exploitation

The zombie API uses outdated authentication methods (e.g., basic authentication instead of OAuth 2.0) and lacks modern security measures like rate limiting or encryption for sensitive data.

Business Impact

Undocumented or outdated APIs (“zombie” APIs) increase the risk of breaches, compliance failures, and exploitation. Without full API visibility, organizations can’t monitor usage or secure endpoints, leading to regulatory fines, operational disruption, and exposure of sensitive data.

PRO TIP

Create an inventory of all API infrastructure, tracking who can access each asset and its data.

Implement continuous API discovery to identify changes, shadow APIs, or rogue endpoints.

Thoroughly document APIs, including authorization policies, error reporting, and security measures. Share these with relevant teams to ensure consistent review and testing.

Avoid using production data for testing unless strictly necessary, and implement safeguards to prevent data leaks.

Insufficient Rate Limiting and Throttling

Insufficient rate limiting and throttling in APIs can expose systems to abuse, leading to security and performance issues. Malicious actors can flood an API with excessive requests without proper controls, resulting in Distributed Denial-of-Service (DDoS) attacks or resource exhaustion.

The result is service disruption for legitimate users and data scraping, brute-force attacks, and other exploits. Additionally, uncontrolled access can increase operational costs due to overused resources. Addressing this issue with robust rate limiting and throttling mechanisms ensures fair usage, maintains performance, and safeguards the API from potential misuse.

Example

Scenario

A company provides a telehealth platform that allows patients to book appointments, view medical records, and communicate with doctors. The platform offers an API for third-party developers to build integrations with their platform.The company's API includes an endpoint `/api/userinfo` that allows developers to retrieve user details (e.g., name, email, and phone number) by providing a unique user ID. The endpoint is intended for internal use but is not adequately secured. The API does not enforce rate limiting, meaning an attacker can send unlimited requests without being blocked or flagged.

Exploitation

The attacker writes a script to iterate through user IDs (e.g., 1001 to 9999) and sends requests to the `/api/userinfo` endpoint. For valid user IDs, the API returns detailed user information, including names, email addresses, and phone numbers. For invalid or inactive user IDs, the error messages reveal additional information, such as whether the user account exists or is inactive.

Business Impact

The attacker compiles a database of thousands of user records, including PII. The attacker uses the leaked data to launch targeted phishing campaigns or sell the data to other malicious actors. The company is fined under data protection laws like HIPAA or GDPR for failing to secure patient data.

PRO TIP

Set rate limits for every API or endpoint to restrict the number of requests within a specified timeframe.

Apply request size limits to prevent excessive data submissions, mitigating the risk of overloading your system.

Deploy bot mitigation tools to block illegitimate, automated traffic.

Apply Geo and Traffic Restrictions to scrub undesirable traffic

Security Misconfigurations

Security misconfiguration in API security is a critical pitfall that can expose sensitive data and lead to devastating breaches. This issue often arises from poorly implemented security settings, such as unused features left enabled, weak or default credentials, mismanaged permissions, or unencrypted data in transit.

These oversights can give attackers easy entry points to exploit vulnerabilities, compromise systems, and access confidential information. The consequences include data leaks, financial loss, and reputational damage. Effective security practices, such as enforcing strong access controls, frequent configuration reviews, and proper encryption, are essential to mitigate these risks.

PRO TIP

Regularly review deployment and configuration processes, ensuring that all dependencies and environments adhere to secure practices.

Limit API interactions to secure, authorized channels with appropriate HTTP verbs.

Set proper Cross-Origin Resource Sharing (CORS) policies for public APIs.

Example

Scenario

A company operates a popular e-commerce platform. To support its mobile app and third-party integrations, the company provides an API that allows developers to access product catalogs, user profiles, and order details.

Exploitation

Misconfigured API Endpoint includes an endpoint `/api/orders` that allows users to retrieve their order history. However, the API is misconfigured to accept requests without proper authentication or authorization checks. The API does not restrict access to specific user data. Instead, it allows any request to retrieve order details for any user, as long as the order ID is provided.

A malicious actor discovers the misconfiguration by testing the API. They notice that:

- The `/api/orders` endpoint does not require an API key or token.
- Providing a valid order ID (e.g., 12345) returns the full details of the order, including: customer name; shipping address; email address; pone number; payment method (e.g., “Visa ending in 1234”)

The attacker writes a script to brute-force order IDs (e.g., 10001, 10002, 10003) and retrieves sensitive data for thousands of customers.

Business Impact

The attacker compiles a database of customer information, including: name; addresses; contact details; partial payment information. The attacker sells the data on the dark web, leading to identity theft and fraudulent activities targeting the company's customers. News of the breach spreads, causing customers to lose trust in the company's platform. The company faces penalties under data protection laws like GDPR or CCPA for failing to secure customer data.

Server-side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) is a critical vulnerability in API security that occurs when an attacker manipulates a server to make unauthorized requests to internal or external resources. This often happens when APIs accept user input to fetch data from a URL without proper validation or sanitization.

The pitfall lies in the server's trust in its network, allowing attackers to exploit SSRF to access sensitive internal systems, retrieve confidential data, or execute malicious commands. To mitigate this risk, developers should implement strict input validation, enforce allowlists for permissible URLs, and restrict network access for APIs to prevent unauthorized internal or external requests.

PRO TIP

Restrict Access to Internal Resources and DMZs (Demilitarized Zones) to prevent direct access from untrusted networks or systems.

Use Safe Libraries and Frameworks that provide built-in protection against SSRF vulnerabilities.

Example

Scenario

A company provides a cloud-based document management platform. It offers an API that allows users to upload documents, retrieve metadata, and fetch external resources (e.g., previewing a document from a URL). The API endpoint /api/fetch-url allows users to provide a URL, and the server fetches the URL's content to display a preview. For example:

```
POST /api/fetch-url
```

```
{
  "url": "https://example.com/document.pdf"
}
```

The server fetches the URL's content and returns it to the user. However, the API does not validate or sanitize the user-provided URL, making it vulnerable to SSRF.

Exploitation

A malicious actor can use this endpoint /api/fetch-url and they can provide any URL, including internal URLs within the company's infrastructure. The attacker can send the request

```
{
  "url": "http://localhost:8080/admin"
}
```

The server fetches the internal resource (e.g., an admin panel or internal API) and returns the content to the attacker.

Business Impact

This will allow the attacker to:

- Access sensitive internal services.
- Extract configuration files, such as /`etc/passwd` or cloud metadata endpoints (e.g., `http://169.254.169.254/latest/meta-data/` in AWS environments).
- Enumerate internal network services and endpoints.

Over-Reliance on Perimeter-Based Security

Traditional defenses rely heavily on securing the network perimeter using firewalls, intrusion detection systems (IDS), and intrusion prevention systems (IPS). However, APIs often operate beyond the traditional perimeter, especially in cloud environments, making them vulnerable to attacks that bypass these defenses.

Traditional tools are not designed to provide deep visibility into API traffic. They may fail to detect malicious API calls, parameter tampering, or unauthorized access attempts, as they lack the context to understand API-specific behaviors. Traditional defenses often rely on signature-based detection, which is reactive and ineffective against zero-day API vulnerabilities or sophisticated attacks that exploit business logic.

Scenario

Example

A company provides a payment processing platform for e-commerce businesses. Their architecture includes a public-facing API for merchants to process transactions and an internal API for managing sensitive operations like user authentication, account management, and payment reconciliation. The company relies heavily on a perimeter-based security model, using a firewall and API gateway to protect its APIs.The internal API is assumed to be secure because it is only accessible from within the company's network. As a result, the internal API lacks robust authentication and authorization mechanisms.

Exploitation

Let's assume an attacker gains access to a trusted system within the company's network, such as a developer's laptop or a misconfigured server, through phishing or exploiting a vulnerability.Once inside the network, the attacker can bypass the perimeter defenses (firewall and API gateway) and directly access the internal API.

PRO TIP

Adopt a layered security approach by combining perimeter security with other measures like advanced API security, network segmentation, and application-level security.

Adopt Zero Trust and Strong Authentication: Treat all API requests as untrusted, enforce strict identity verification, and use OAuth 2.0, MFA, and least privilege access.

Monitor, Test, and Educate: Regularly log and monitor API activity, conduct vulnerability assessments, and train teams on secure API practices.

The attacker discovers that the internal API lacks proper authentication and authorization checks. For example:

- The endpoint `/api/internal/refund` allows refunds to be processed by simply providing a transaction ID and amount.
- The endpoint `/api/internal/update-account` allows account details to be modified without verifying the user's identity.

Business Impact

The attacker sends requests to these endpoints to Issue fraudulent refunds to their own accounts, modify account details to redirect payments to their bank accounts, and extract sensitive customer data, such as payment card details and personal information.

Broken Object Property Level Authorization

Broken Object Property Level Authorization (BOPLA) occurs when an API fails to enforce proper authorization checks at the property level of an object. This means that while a user may have permission to access or modify an object (e.g., their user profile), they can also manipulate sensitive or restricted properties within that object (e.g., setting themselves as an admin) without proper validation.

BOPLA is important because it can lead to privilege escalation, unauthorized access to sensitive data, or manipulation of critical system settings. Ensuring property-level authorization is crucial to prevent exploitation and maintain the security and integrity of APIs.

Example

Scenario

A company operates a social media platform where users can create profiles, post updates, and manage account settings. It provides an API for its mobile app and third-party integrations. This API design is with Insufficient Property-Level Authorization.

The API includes an endpoint for updating user profiles:

```
PUT /api/user/{userId}/profile
```

The API accepts a JSON payload with various properties, such as:

```
{
  "username": "new_username",
  "email": "new_email@example.com",
  "isAdmin": true
}
```

PRO TIP

Implement Fine-Grained Authorization: Ensure access control checks are applied at the object property level, not just at the endpoint level.

Use Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC): Define clear roles and permissions for accessing specific properties.

Conduct Security Testing: Regularly test APIs for authorization flaws using tools like penetration testing and automated scanners.

Follow the Principle of Least Privilege: Limit access to only the data and functionality necessary for a user's role.

Audit and Monitor API Activity: Log and monitor API requests to detect and respond to unauthorized access attempts.

While the API enforces user-level authorization (e.g., ensuring that users can only update their profiles), it does not enforce property-level authorization. This means:

Any authenticated user can modify all properties in their profile, including sensitive or restricted fields like isAdmin.

Exploitation

An attacker finds that isAdmin property is included in the JSON payload and is not restricted to authorized users. They can exploit this to escalate their privileges. The attacker sends a request to the API to update a profile:

```
PUT /api/user/12345/profile
```

With the following payload:

```
{
  "username": "Test",
  "email": "test@example.com",
  "isAdmin": true
}
```

The API processes the request and updates the profile, granting the attacker admin privileges because it does not validate whether they are authorized to modify the isAdmin property.

Business Impact

The attacker gains access to sensitive admin-only features, such as:

- Viewing all user data, including private messages and email addresses.
- Deleting or modifying other users' accounts.
- Accessing analytics and financial data for the platform.

Unrestricted Resource Consumption

Unrestricted resource consumption in APIs happens when an API allows users to consume excessive server resources, such as CPU, memory, or bandwidth, without proper limits. This can lead to performance issues, server crashes, or even denial-of-service (DoS) attacks, disrupting services for legitimate users.

Addressing this is important because it ensures fair usage, protects system stability, and prevents malicious actors from exploiting the API to overload infrastructure. Implementing rate limits, quotas, and resource caps is essential to safeguard APIs and maintain a reliable user experience.

Example

Scenario

A company operates a popular video streaming platform. It provides an API for its mobile app and third-party integrations, allowing users to search for videos, stream content, and manage their accounts. Their API is designed with no resource limits.

API includes an endpoint for searching videos:

```
GET /api/videos/search
```

The endpoint accepts query parameters like `keyword`, `page`, and `limit`:

```
GET /api/videos/
search?keyword=action&page=1&limit=10
```

However, the API does not enforce any restrictions on the limit parameter, allowing users to request an unlimited number of results in a single query.

PRO TIP

Rate Limiting: Set limits on the number of requests a user or client can make within a specific time frame.

Throttling: Slow down the response rate for users exceeding their allowed limits.

Authentication and Authorization: Only authenticated and authorized users can access the API.

Resource Quotas: Define quotas for resource usage, such as data transfer or compute time.

Monitoring and Alerts: Continuously monitor API usage and set up alerts for unusual activity.

Caching: Use caching mechanisms to reduce the load on backend systems for frequently requested data.

Exploitation

A malicious actor discovers the flaw in the API that by setting a very high value for the limit parameter, they can force the server to process and return a massive amount of data in a single request:

```
GET /api/videos/
search?keyword=action&page=1&limit=1000000
```

The attacker writes a script to repeatedly send requests to the `/api/videos/search` endpoint with an extremely high limit value. Each request consumes significant server resources (e.g., CPU, memory, and bandwidth) as the server processes the query, retrieves the data, and formats the response.

Business Impact

The excessive resource consumption caused by the attacker's requests overwhelms the server, leading to a Denial of Service event for the service, resulting in:

- slower response times for legitimate users;
- the server crashes, making the API and platform unavailable to all users.

Unrestricted Access to Sensitive Business Flows

Unrestricted Access to Sensitive Business Flows in APIs occurs when APIs fail to enforce proper authorization for critical operations, such as processing refunds, modifying user roles, or accessing financial data. This allows unauthorized users to exploit these flows, leading to fraud, data breaches, or system abuse.

Securing sensitive business flows is crucial to prevent exploitation, protect financial and operational integrity, and maintain user trust. Proper authorization checks, role-based access control, and activity monitoring are essential to ensure only authorized users can perform these critical actions.

Example

Scenario

A company operates a financial platform that allows users to send and receive payments, manage accounts, and process refunds. It provides an API for its mobile app and third-party integrations.

The API includes an endpoint for processing refunds with unrestricted access:

```
POST /api/refunds
```

The endpoint accepts a JSON payload with the following parameters:

```
{
  “transactionId”: “123456”,
  “refundAmount”: 100.00
}
```

However, the API does not enforce proper authorization checks to ensure only authorized users (e.g., merchants or admins) can initiate refunds. Any authenticated user can call the endpoint and process refunds for any transaction.

PRO TIP

Rate Limiting: Set limits on the number of requests a user or client can make within a specific time frame.

Throttling: Slow down the response rate for users exceeding their allowed limits.

Authentication and Authorization: Only authenticated and authorized users can access the API.

Resource Quotas: Define quotas for resource usage, such as data transfer or compute time.

Monitoring and Alerts: Continuously monitor API usage and set up alerts for unusual activity.

Caching: Use caching mechanisms to reduce the load on backend systems for frequently requested data.

Exploitation

An attacker discovers this vulnerability that the API `/api/refunds` endpoint does not verify whether they own the transaction attempting to refund.

They can refund transactions belonging to other users or merchants by simply providing a valid `transactionId`.

The attacker writes a script to automate refund requests. She uses a brute-force approach to guess valid `transactionId` values (e.g., sequential IDs like 123456, 123457, 123458).

For each valid transaction ID, they send a refund request:

```
{
  “transactionId”: “123456”,
  “refundAmount”: 100.00
}
```

The API processes the requests and issues refunds to the attacker’s account, even though they are not authorized to initiate refunds for those transactions.

Business Impact

The company can suffer significant financial losses as fraudulent refunds are processed and funds are transferred to unauthorized accounts. Merchants using the platform experience financial discrepancies and disruptions in their operations and could result in potential damages or loss of revenue.

Conclusion

APIs are the backbone of our digital ecosystem, enabling critical data exchanges and unlocking business innovation. But with every opportunity comes risk. Unauthorized access, data breaches, and compliance failures are just a few of the threats lurking when API security is overlooked.

Enterprise architects and CISOs are on the frontlines of protecting sensitive data. Have you recently evaluated your API security posture? Are you confident your current measures align with industry best practices? If not, now is the time to act.

Neglecting API security puts your organization at risk of costly breaches, reputational damage, and regulatory penalties. On the other hand, implementing strong API security measures mitigates these risks, fosters customer trust, and strengthens your overall cyber resilience.

Here's your next step

Don't leave your APIs vulnerable! Schedule a consultation with our experts to assess your current posture. Your data integrity and compliance depend on focused, proactive action. Act today, and secure your APIs before it's too late.

Schedule a consultation

wallarm

Book a Demo

+1 (415) 940-7077

188 KING ST. UNIT 508

SAN FRANCISCO, CA 94107