Carnegie Mellon University
Software Engineering Institute

Application Programming Interface (API) Vulnerabilities and Risks

McKinley Sconiers-Hasan

April 2024

SPECIAL REPORT
CMU/SEI-2024-SR-DM24-0295
DOI: 10.1184/R1/25282342

CERT Division

https://www.sei.cmu.edu

Design: REV-03.18.2016.0 | Template: 02.22.2023

DM24-0295

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Table of Contents

List of Figures

Abstract

Application programming interfaces (APIs) are increasingly common, and they are often designed and implemented in a way that creates security risks. This report describes 11 common vulnerabilities and 3 risks related to APIs, providing suggestions about how to fix or reduce their impact. Recommendations include using a standard API documentation process, using automated testing, and ensuring the security of the identity and access management system.

1 Introduction to APIs

Application programming interfaces (APIs) have unique risks, especially when these risks are compared to other types of network security risks. One of the main goals of cybersecurity is to reduce the attack surface that malicious actors can use to penetrate a network, but APIs are intentionally designed to be exposed to the public or third parties for use. This means that APIs, by nature, increase the attack surface of a network, and if APIs are not designed correctly, their availability can lead to significant security issues [Hussain 2022].

From the attacker's perspective, one of the most time-consuming parts of planning a network attack is the reconnaissance required to find potential network attack vectors. Since APIs must be exposed to the public, the amount of time the attacker spends finding attack vectors into the API's network is greatly reduced, making them an easier target for potential breaches.

The rise in implementing microservice architectures in place of monolithic software architectures now makes APIs more common than ever. To maintain a secure, zero trust network, this pervasiveness requires a focused effort to identify API-specific vulnerabilities.

1.1 API Endpoints

An API provides a programmed entry point into an application. If the application is designed to interact with other applications, then an API is put into place to enable that interaction. Within each API, there are multiple application connection points called *endpoints*.

Each endpoint serves a different purpose, but they are all connected to the same application. The following are a few examples of how an API endpoint might be used:

- logins (The necessary inputs for that endpoint are a username and password.)
- password resets (The required input might only be an email, so a password reset link can be sent.)
- administrators changing application settings (The required input for access might be a username and password.)
- enabling interaction between microservices (The required inputs might be user IDs or other user information, product or order information, or whole data structure objects.)

1.2 Microservice Architectures

In the past decade, there has been a huge increase in the use of microservice architectures.[1]

Microservice architectures focus on creating many small, loosely coupled micro applications to produce a larger application. For example, a company might have a Product Descriptions

[1] In previous years, monolithic architectures were dominant.

microservice, a Payment Processing microservice, and a Login microservice. Each of these microservices can be executed and deployed independently of the others. Each microservice has its own API and endpoints for client-to-microservice or microservice-to-microservice interactions.

The distributed, loosely coupled design of these microservice architectures allow them to have greater scalability, flexibility, and continuous deployments when compared to monolithic architectures. According to National Institute of Standards and Technology (NIST) special publication (SP) 800-204, "Microservices generally communicate with each other using Application Programming Interfaces (APIs), which require several core features to support complex interactions between a substantial number of components" [Chandramouli 2019]. There are often support structures that also help API ecosystems by offering solutions, such as API gateways and service meshes. This report only discusses APIs, but the use and configuration of supporting structures (e.g., identity access management [IAM] servers, API gateways, service meshes) are also important to overall API security.

1.3 Common API Risks and Vulnerabilities

APIs can have vulnerabilities due to design flaws, incorrect deployments, or business logic that an attacker can then exploit. Each part of the API transaction process—from entity authentication, client request processing, and server response—must be secured. This report describes some of the most common risks and vulnerabilities of modern APIs. Many of these vulnerabilities and risks are identified and detailed in the release of the Open Worldwide Application Security Project's (OWASP's) document *Top 10 API Security Risks—2023* [OWASP 2023].

Furthermore, API architectures come in many variations, each with its own design and security strengths and weaknesses. Examples include the following:

- The commonly used representational state transfer (REST) API is usually used with JavaScript Object Notation (JSON); it is lightweight and flexible with few security features [Red Hat 2020].
- The Simple Object Access Protocol (SOAP) uses Extensible Markup Language (XML) and is much more complex than REST; it has its own web services security (WS-Security) extension [Singhal 2007].
- The Graph Query Language (GraphQL) is another type of API that was designed to optimize API queries; it can easily be exploited with simple misconfigurations [Red Hat 2019].

The specific API architecture and configuration a designer uses to create their API will have a major impact on its security. However, a deep analysis of API architectural design options and their associated security configurations is beyond the scope of this report.

Risks and vulnerabilities associated with APIs are similar to security risks associated with any other network. These vulnerabilities include authorization problems, network connectivity issues, lack of awareness and training for development staff, hardware malfunctions, and Internet-based attacks associated with poor authentication. While this report could include extensive discussions about each of these topics, it instead focuses on the risks that are exclusive to or more common in APIs as compared to non-API connections.

2 Top API Security Vulnerabilities

This section describes the top API vulnerabilities, including examples and recommendations.

2.1 Broken Object Level Authorization

Description. Broken Object Level Authorization (BOLA) occurs when an authenticated (or unauthenticated) user can access sensitive objects and data that they are not authorized to access. This vulnerability occurs when users type different Object IDs in the endpoint URL, either in the API, query string, or as part of the request payload.

OWASP's Security Risk Ranking. BOLA is number 1 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume an authenticated user named John Doe enters the following:

```
localbank.com/access/users?id=110
```

It returns the response shown below with his account information. The API returns his own record correctly.

```
{
    "userid": "110",
    "firstname": "John",
    "lastname": "Doe",
    "accountbalance": "5000.00"
}
```

Assume that the same authenticated John Doe enters the following:

```
localbank.com/access/users?id=111
```

It returns account information for the user associated with id 111, Catherine Ming. See below. In this case, the API has a BOLA issue because John Doe should not be able to access Catherine Ming's information.

```
{
    "userid": "111",
    "firstname": "Catherine",
    "lastname": "Ming",
    "accountbalance": "11325.16"
}
```

See Figure 1 for an illustration of this vulnerability.

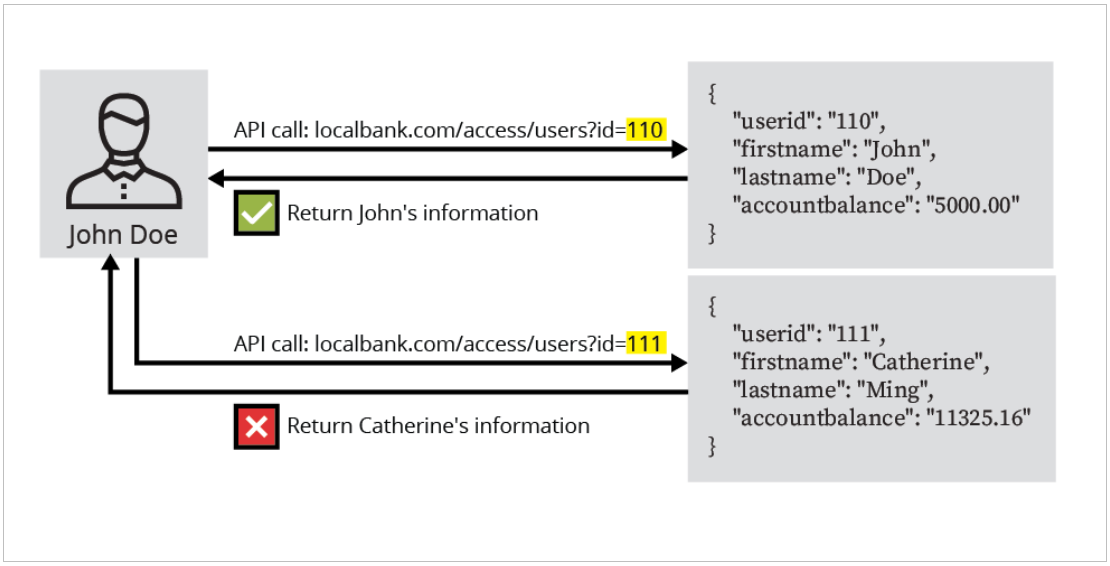

Figure 1: BOLA Example

Recommendations. The following recommendations apply to this vulnerability.

1. Use role-based access controls to limit access to resources; access to resources should follow the principle of least privilege.
2. When possible, use random object IDs for access to objects instead of sequential IDs or any other pattern that could be easily guessed or derived.
3. Write tests that attempt to access, modify, update, and delete objects with and without proper authorization. The tests without proper authorizations should return an error unless a BOLA vulnerability is present. These tests can be conducted throughout the software development lifecycle (SDLC).
4. Recognize that BOLAs are difficult to detect through automated testing because BOLA exploits are often design specific, and vulnerabilities vary greatly between different API designs and architectures. However, penetration testing would help identify BOLA vulnerabilities in a way that automated testing cannot.
5. Carefully design application and API logic to help prevent BOLA vulnerabilities. BOLA exploits often happen because of design issues that were incorrectly programmed into the API, so having clearly defined application flow and logic helps prevent BOLA issues.

2.2 Broken Authentication

Description. *Authentication* is the process of verifying that an entity or user is who they claim to be. If there are vulnerabilities in this process—through incorrect authentication implementation or weak or insufficient authentication policies—then the API is at risk for broken authentication.

If an attacker gains authentication on an account that is not theirs, then they can read or alter the victim's sensitive data or change the victim's account settings. Once the attacker gets control of the account, that access is difficult to detect.

Broken Authentication is often caused by weak password policies, insufficient authentication mechanisms, misconfigurations, etc. Fortunately, many of these policies and mechanisms are easy to detect using automated testing software.

OWASP's Security Risk Ranking. Broken Authentication is number 2 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. One common type of broken authentication is called credential stuffing. This attack works by using stolen credentials, often a username and password, and submitting those credentials on another site to break into the rightful user's account. This type of attack can be successful because some users reuse the same username and password for multiple sites. Once an attacker finds a username/password set that grants them access to the API, they not only compromise the victim's account, but they now have greater access to the internal network where the API is running.

Recommendations. The following recommendations apply to this vulnerability.

General

1. Implement rate limiting and lockout protection for all authentication endpoints.
2. Clearly understand the different mechanisms that are intended for authentication and those that are intended for authorization.
3. Create clear documentation for all authentication flows.
4. Use multifactor authentication (MFA) to prevent broken authentication attacks, such as credential stuffing.
5. Use step-up authentication when anomalous or higher risk requests are received. This type of authentication can include requiring MFA when an authentication request is submitted at an uncommon time of day or when someone requests a money transfer.
6. Use automated testing to verify the strength of the authentication mechanisms and policies, including strong passwords, rate limiting, anti-brute-force mechanisms, and lockout protection.

Password Policies

1. Require complex, hard-to-guess passwords that contain numbers, symbols, and letters with a mix of capitalization,.
2. Limit the number of authentication attempts in a short time frame to prevent brute-force attacks and credential stuffing.
3. Require email confirmation for password changes.
4. Ensure that URLs do not disclose password information.
5. Ensure that all passwords are stored in hashed text, not plaintext.

Token Policies

1. Ensure that values have enough randomness that they are difficult to guess.
2. Accept tokens for authentication only if they are signed.
3. Set a short expiration time for tokens to prevent their reuse.

Note: There are many types of authentication tokens, each with its own unique security configuration. Providing details about each type of token and subsequent security considerations will is beyond the scope of this report.

2.3 Broken Object Property Level Authorization

Description. Broken Object Property Level Authorization (BOPLA) allows a user to view or modify sensitive data that they should not have permission to view or alter. This action can be performed through a specially crafted API request or simply by misconfiguring the API response mechanism.

OWASP's Security Risk Ranking. BOPLA is number 3 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume that the `www.university.com/students HTTP/1.1` API GET request returns the following list of names of 2023 graduates from a university.

```
"students":
{
 {
    "firstname": "Jake",
    "lastname": "Care",
    "major": "political science",
    "email": "jakecare@gmail.com",
    "address": "123 Fourth St",
    "birthday": "04/07/92"

 },
 {
    "firstname": "Haley",
    "lastname": "Miles",
    "major": "civil engineering",
    "email": "haleymiles@gmail.com",
    "address": "934 Eighth St",
    "birthday": "01/17/92"

 }
}
```

The information provided to a user sending a GET request for the graduates should show only the graduates' names and possibly their degrees. However, the API request also returns the students' email addresses, postal addresses, and birthdays. This is sensitive information that should not be disclosed without authorization.

See Figure 2 for an illustration of this vulnerability.

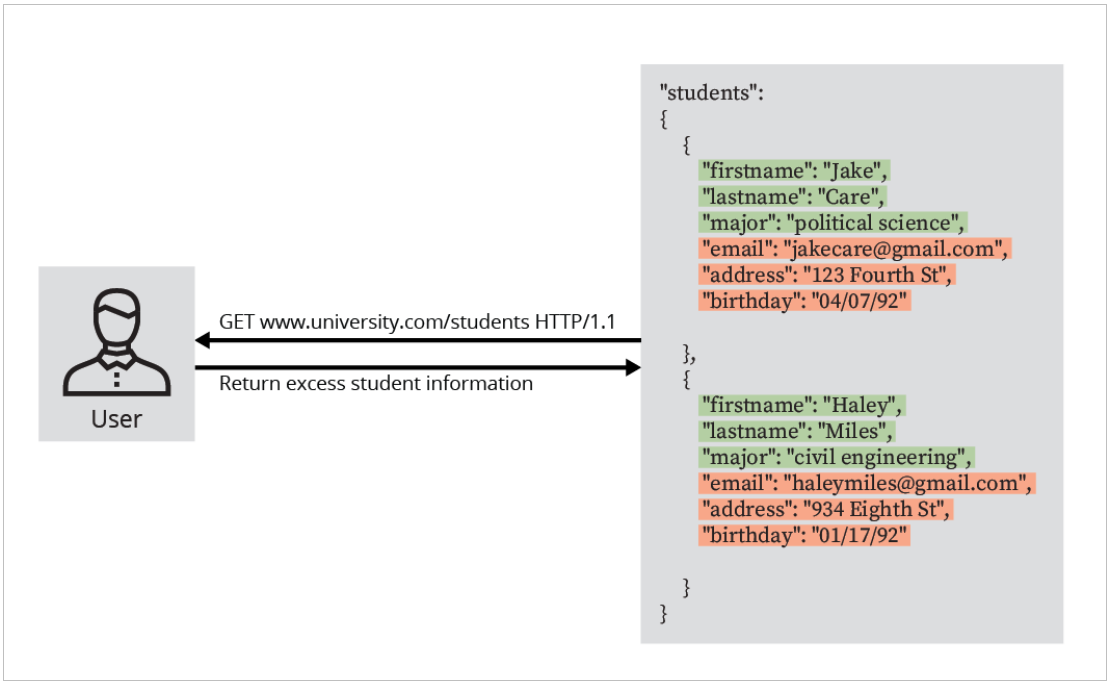

Figure 2: Broken Object Property Level Authorization Example

Recommendations. The following recommendations apply to this vulnerability.

1. Clearly define responses from each API method in the API documentation.
2. Test the API by reviewing the API responses to ensure there is no excessive data exposure.
3. Use automated testing and fuzzing techniques to test for API BOPLA vulnerabilities.
4. Validate and sanitize all user input according to the clearly defined API input specifications to avoid the unauthorized alteration of data.
5. Avoid functions that automatically bind user input to objects.

2.4 Unrestricted Resource Consumption

Description. Unrestricted Resource Consumption occurs when various API resources are excessively consumed, negatively effecting the technical or monetary purposes of the API. If a user or entity is allowed to constantly make manual or automated API requests that the API must service, it could lead to a slower API service or even a complete shutdown. Even if the API service is not disrupted because of excessive API calls, the business may incur higher infrastructure costs due to high resource consumption. This problem is even more costly when using an infrastructure scheme like function-as-a-service (FaaS), where resources are allocated and paid for on a per-request basis.

Failure to implement the following or configuring them incorrectly could lead to an unrestricted resource consumption vulnerability:

- execution timeouts
- maximum allocatable memory

- maximum number of file descriptors
- maximum number of processes
- maximum upload file size
- number of operations allowed in a single request
- number of records allowed in a single request

OWASP's Security Risk Ranking. Unrestricted Resource Consumption is number 4 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume that the following GraphQL query returns the titles of the two most recently published *New York Times* articles:

```
query getCompany(name: "New York Times") {
       getArticles(last: 2) {
              title
}
}
```

Assume that the following GraphQL query returns the 10,000 most recently published *New York Times* articles with the title, author, text, and date of each. (This query is obviously large and would require a huge number of resources for the API to service, which might slow down servicing for other clients or shut down the API altogether.)

```
query getCompany(name: "New York Times") {
       getArticles(last: 10000) {
              title
              author
              text
              date
       }
}
```

Recommendations. The following recommendations apply to this vulnerability.

1. Recognize that some APIs allow API request parameters that specify the number of resources to be returned. If this type of functionality is implemented in an API, ensure it is capped at a reasonable size and monitored to prevent excessive use.
2. Use memory and other resource-limiting functions if the platform being used provides them.
3. Limit the number of API calls a user is allowed to send within a single time frame and notify the user when the limit is exceeded.
4. Implement server-side validation of user inputs to check for parameters that specify the number of resources requested.
5. Define and implement mechanisms for controlling maximum input size. This input size includes maximum string sizes, maximum number of items in arrays, etc.

2.5 Broken Function Level Authorization

Description. Broken Function Level Authorization (BFLA) allows an attacker to alter or delete data. The attacker is able to perform their attack by sending API calls to endpoints they should not be able to access. This vulnerability results from incorrect user privileges and authorizations. It can result in not only unauthorized tampering of data on the attacker's own privilege level but also privilege escalation if the attacker can target API functions that can elevate their privileges. For this reason, admin accounts are common targets of BFLA attacks, and they should be securely configured to prevent exploitation. Hypertext Transfer Protocol (HTTP) verb tampering could be an indication of a BFLA vulnerability.

OWASP's Security Risk Ranking. BFLA is number 5 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume a school with an online system stores students' grades in a way that both students and teachers can access. Students should be allowed to view their own grades, but only teachers should be able to modify them. The student API endpoint for accessing their grades is `/viewgrades/`, and the endpoint for teachers to alter student grades is `/altergrades/`. A BFLA vulnerability is present if, for example, a student named Jimmy logs into their account, types the following URL, and is able to access the teacher page where student grades can be changed: www.school.com/altergrades?student=jimmy.

The authorization to use a function that can alter or delete data has been given to a user who should not have that authorization.

See Figure 3 for an illustration of this vulnerability.

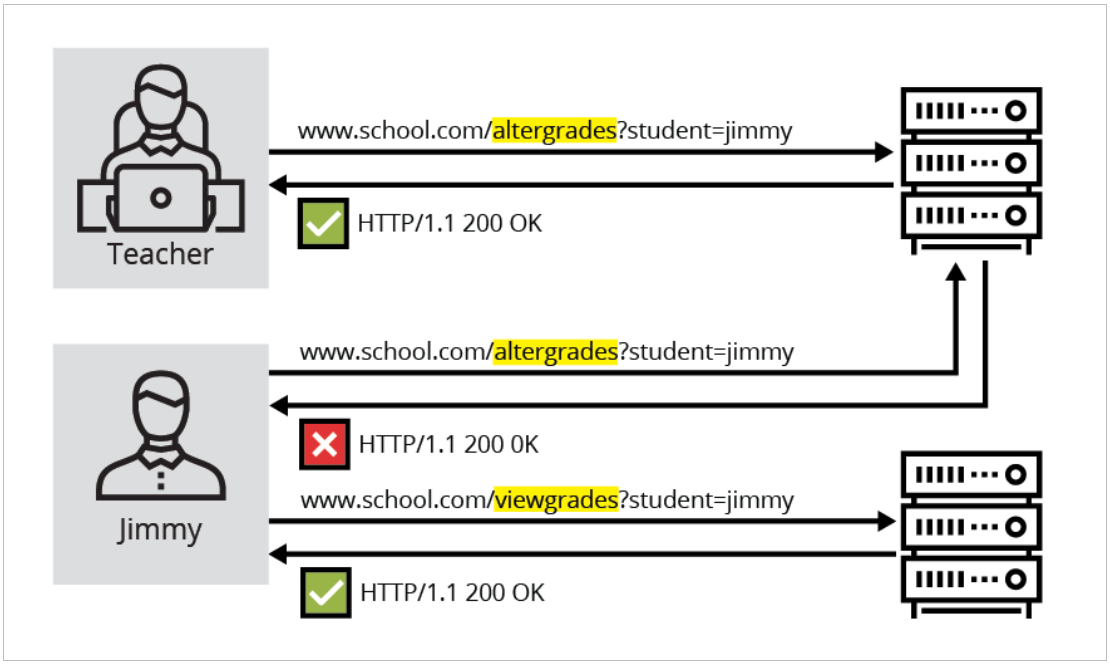

Figure 3: BFLA Example

Recommendation. The following recommendations apply to this vulnerability.

1. Create and enforce clearly documented rules about which users and groups are allowed to create, alter, or delete resources (including user and admin accounts). This is especially important for admin accounts and sensitive resources.
2. Restrict access to sensitive endpoints, such as admin accounts.
3. Restrict the use of HTTP verbs to only the roles that have the need and authorization to use those verbs. Sensitive verbs tend to be PUT, DELETE, and POST.
4. For resource access, adopt a deny-by-default policy that denies access to all functions unless explicitly granted.
5. Use both automated testing and manual testing to check the API for potential BFLA vulnerabilities.

2.6 Unrestricted Access to Sensitive Data Flows

Description. Unrestricted Access to Sensitive Data Flows is not a technical flaw, but it is a vulnerability in business flow logic that can cause harm if exploited through automation. When an attacker exploits unrestricted sensitive data flows in APIs, their goal is to disrupt or damage the business. To exploit this vulnerability, the attacker does not necessarily need a lot of technical skill or knowledge about the API. Instead, the attacker needs to know more about the business model behind the API to determine what type of data flows are sensitive. Attackers can use this information either for personal gain or to damage the business, monetarily or otherwise. Common examples include attackers writing automated scripts to spam a public forum with negative comments about a company to damage its reputation or reserving all the slots at a company's free event to prevent people from attending.

OWASP's Security Risk Ranking. Unrestricted Access to Sensitive Data Flows is number 6 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume that Company A recently acquired a shipment of 1,000 highly sought gaming personal computers (PCs). A scalper then sets up a script to automatically buy all 1,000 computers just as they became available to buy online. The scalper then takes all 1,000 computers and sells them at a higher price. This attack disrupts the regular business flow by preventing regular customers from buying the PCs at the market price.

Recommendations. The following recommendations apply to this vulnerability.

1. Identify sensitive business flows, and use that information to design APIs in a way that mitigates the risk of these sensitive flows.
2. Use human detection mechanisms (e.g., CAPTCHA)[2] to prevent automated access to sensitive flows.
3. Implement mechanisms that detect non-human API usage patterns, such as too many requests coming into the API in a short time frame.

[2] CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart.

2.7 Server-Side Request Forgery

Description. Server-Side Request Forgery (SSRF) occurs when a client URL is not validated and results in an attacker accessing or modifying resources that should not be accessed or modified. This vulnerability occurs when an API allows data to be uploaded from URLs or when proper safety measures are not put into place on fetching data through URLs. SSRF is a common vulnerability that is not specific to APIs; however, APIs are more susceptible to it than other network structures because they are, by design, exposed to the public (or to whatever user or entity they are designed for).

OWASP's Security Risk Ranking. SSRF is number 7 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Consider the following example:

1. Assume an attacker sends a GET request that includes an internal network URL (e.g., to a database or test server). In the example shown in Figure 4, the target URL is the admin page for a database.
2. The API that sent the request forwards it to the target machine.
3. The Public API receives the response URL from the target machine with the requested admin page.
4. The Public API responds to the attacker's original GET request with the admin page from the target machine.

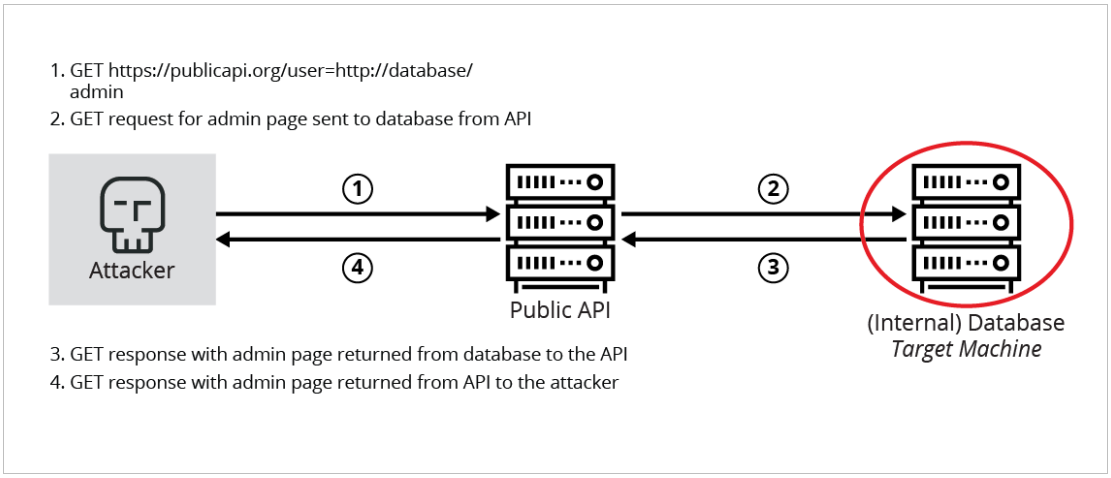

Figure 4: Server Side Request Forgery Example

Recommendations. The following recommendations apply to this vulnerability.

1. Disable unused URL schemas. APIs will usually communicate over HTTP and HTTP Secure (HTTPS) exclusively, so other URL schemas such as file:/// and dict:/// should not be executable.
2. Create allowlists of acceptable media types that are allowed to be sent and received by the API.
3. Disable or limit access to remote origins that should not be accessible through the API.

4. Do not send unprocessed API responses directly to users.
5. Use allowlists on URLs that clients can access. By design, APIs should allow access only to data the client is authorized to use within the API ecosystem, so allowlisting only available API endpoints helps prevent access to other internal Internet Protocol (IP) addresses and Domain Name System (DNS) names that clients should not access.
6. Implement authentication measures on all sensitive internal resources. These measures mitigate the risk of exploitation if an SSRF occurs within an API. (Often databases and internal file systems have no authentication mechanisms by default—a practice that makes them vulnerable if a poorly configured API is used in an attack vector to carry out an SSRF.)
7. Enable logging for all internal resources to help detect any tampering or malicious activity in the network.
8. Disable HTTP redirections.
9. Validate and sanitize all incoming user data.
10. Implement least privilege when granting access to internal resources to help prevent data leakage or tampering in the case of an SSRF exploitation.

2.8 Security Misconfiguration

Description. Security Misconfigurations can happen at any level of network communication and at any point in the API client-server interaction process. These misconfigurations include authentication, authorization, data transit, and data manipulation/retrieval. Some misconfigurations are API specific, such as implementing verbose error messaging that gives API users insight into the API backend. Other misconfigurations are more general, such as using default credentials for admin accounts. These API-specific or general misconfigurations can individually cause security vulnerabilities, but they can also compound into more severe vulnerabilities when there are multiple misconfigurations. These security misconfigurations can lead to data leaks, data exfiltration, data manipulation, overall knowledge of the internal network system, or even full server takeovers. Issues such as unpatched systems and unprotected endpoints, files, directories, and legacy systems can result in security misconfigurations.

There is a wide range of configurations that can be mishandled in during the API design and implementation process from the network level to the application level. These include misconfigured headers, misconfigured data encryption for data transfers, allowance of HTTP verb tampering, insufficient data validation and sanitation, or overly descriptive error messages.

OWASP's Security Risk Ranking. Security Misconfigurations is number 8 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Consider the following example:

```
Allow-Access-Control-Origins: *
Allow-Access-Control-Credentials: true
```

This kind of default configuration in cross-origin resource sharing (CORS) HTTP headers can enable an attacker to fetch and send information to an external location.

Recommendations. The following recommendations apply to this vulnerability.

1. Encrypt all data that is being transported.
2. Always change default credentials.
3. Disable or limit HTTP verbs that are unnecessary for API functionality.
4. Use an automated API vulnerability scanner to check for commonly known security misconfigurations.
5. Ensure that the secure API configuration process is well documented and automated where possible. This helps ensure quick and secure deployments.
6. Regularly check API security configurations, including supporting components and documentation such as the API gateway, firewall, and service mesh.
7. If the API is using CORS, ensure it is configured securely and has well-configured security headers.
8. Implement proper error handling to avoid revealing excessive information to the client.
9. If the API is in the cloud, ensure there are staff members who are knowledgeable about secure cloud configurations. (Poorly configured deployments in the cloud can be as detrimental as any security misconfiguration.)

2.9 Improper Inventory Management

Description. Improper Inventory Management occurs when old, unsupported versions of APIs or pre-production APIs are exposed for client use. These non-production APIs are often outdated, have unpatched vulnerabilities, or simply have weak security configurations. Poor communication and documentation standardization practices within API development teams can lead to improper inventory management or a lack of awareness of API changes by other developers or API consumers [Lercher 2023].

Since deployment is easy with microservice architectures, it is now easy to deploy and subsequently forget about certain microservices and their associated APIs. Unless the inventory of all APIs is properly managed in a microservice system, these forgotten APIs will eventually become outdated and insecure but will remain discoverable by DNS enumeration or other Internet scanning. Often, developers fail to add proper security to microservices and APIs that are in development, which also makes pre-production APIs vulnerable. Exploitation of these insecure APIs can result in sensitive data leaks, server takeovers, and lateral or vertical movement within network systems.

OWASP's Security Risk Ranking. Improper Inventory Management is number 9 on OWASP's *Top 10 API Security Risks—2023* [OWASP 2023].

Example. Assume there is an API currently on version 3 (e.g., https://company.com/api/**v3**/), and the older version is accessible at a similar URL (e.g., https://company.com/api/**v1**/). If that older API version is still discoverable, has been forgotten, and now has an unpatched vulnerability, that constitutes an improper inventory management vulnerability.

Recommendations. The following recommendations apply to this vulnerability.

1. Review old API documentation and repository version histories to discover lost APIs.
2. If there is a versioning scheme to a specific API (e.g., /v1/, /v2/, /v3/, /api?v=1, /api?v=2, api.company.org), check older versions to ensure they are properly patched or disabled if they are no longer in use.
3. Document and implement a secure development environment with secure configurations to prevent pre-production APIs from being exploited.
4. Use automated API discovery scanners or fuzz testing to discover unsupported API paths. Looking through old API documentation to identify the common API path naming conventions that were used can help in the guess/fuzzing process.
5. Maintain an inventory of all APIs and their associated versions with auto-generated documentation. This documentation should include, but should not be limited to, all endpoints, security configurations, third-party software or integrations being used; authentication and authorization information; and error handling.
6. Ensure that all client-facing APIs communicate through the network API gateway or firewall to ensure proper security, visibility, and logging.
7. Conduct a risk assessment during the API design process and when new versions of an API are being deployed. These assessments help determine if older versions should be disabled, phased out, or maintained for compatibility.
8. Implement detailed digital asset management. Often, APIs are used to retrieve digital assets, and if those digital assets are not inventoried correctly, then it is more likely that the APIs used to retrieve these unaccounted assets will also be lost.

2.10 Unsafe Consumption of APIs

Description. Unsafe Consumption of APIs focuses on the consumption of data from third-party API providers. Security measures on the communication between APIs are often more relaxed than communication between the API and a client. However, trusting the communication between an API and another third-party API more than an API to a client can lead to API exploitation if the third-party API is compromised.

Exploitation of this vulnerability requires the user to have information about the third-party integrations for the specific API. If a third-party API integrated with the API is compromised, it could lead to sensitive data leaks, denial of service (DoS), data injections, or other attacks.

OWASP's Security Risk Ranking. Unsafe Consumption of APIs is number 10 on OWASP's *Top 10 API Security Risks—023* [OWASP 2023].

Example. As illustrated in Figure 5, assume an API gateway accepts all incoming requests for an API. When it gets a request from a user, the request goes through a sanitization process to ensure it does not have any malicious attributes like Structured Query Language (SQL) or cross-site scripting (XSS) injections. However, when the API gateway gets a request from a third-party API, it does not do the same sanitation checks and instead simply forwards the request to the backend

API. If the third-party API was compromised by an attacker, then that attacker could send an SQL or XSS injection to the API through the third-party API and successfully attack it.

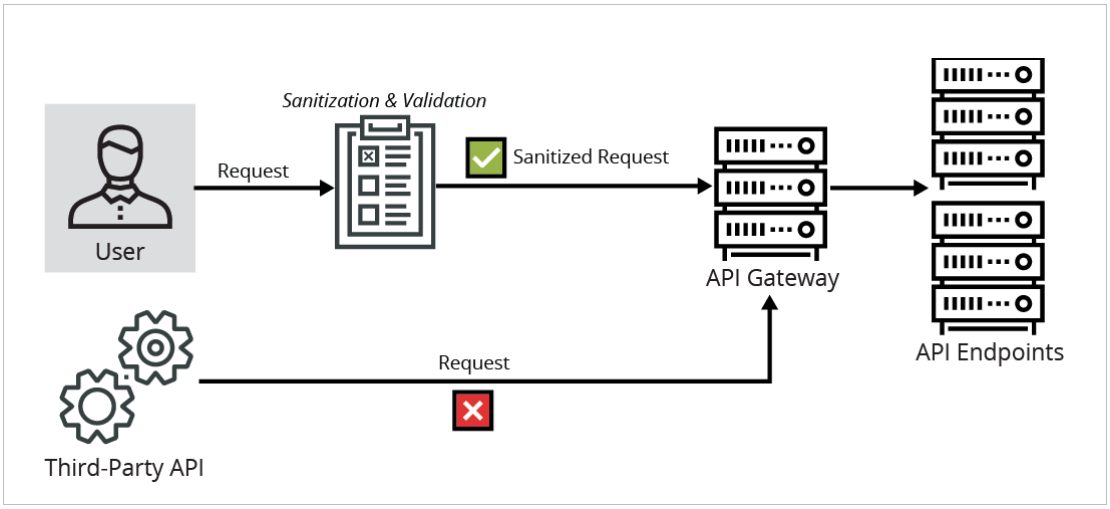

Figure 5: Unsafe Consumption of APIs Example

Recommendations. The following recommendations apply to this vulnerability.

1. Limit the availability of API documentation that contains information about third-party integrations to only those who need to see it.
2. Treat all input, whether from a client or third-party API, uniformly when it comes to authentication, authorization, validation, and sanitation.
3. Maintain an allowlist of the sites to which third-party API integrations are allowed to redirect the API.
4. Monitor API error and access logs for anomalous activity that could indicate a compromised API integration.
5. Conduct risk assessments on all potential third-party API integrations before deployment. Stay abreast of new vulnerabilities discovered within existing third-party API software integrations and conduct new risk assessments when they are discovered.
6. Consider having a team conduct penetration testing on API integrations before the API is deployed.
7. Ensure that all API communication is transmitted through encrypted channels.

2.11 Injections and Other Internet-Based Attacks

Description. There are many types of injections. Some of the most common are SQL, Lightweight Directory Access Protocol (LDAP), and XSS injections. These injections can be done through whatever attack vector an attacker can find, including from client input sources or third-party API integrations. The impact of an injection depends on the specific injection, but some of the common impacts include sensitive data leaks, network information exposure, DoS, or

a complete takeover of the host machine. Injections are difficult to detect or remediate without thoroughly logging and monitoring the API system, including conducting asset integrity checks.

Example. Consider an LDAP query. Assume an attacker wants to get all the usernames and passwords in an LDAP backend database; they could go to an application's login page and type the following:

```
Username: *)(username=*
Password: *
```

The result is an LDAP query as follows:

```
`(&(username=*)(password=*))`
```

This query returns all the *non-null* usernames and passwords in the database, which constitutes a data leak of sensitive information.

Recommendations. The following recommendations apply to this vulnerability.

1. Implement a comprehensive logging and alert system for the API using a security information and event management (SIEM) solution. This solution alerts on failed authentication attempts and input validation errors and implements other failsafes to prevent and detect the unsafe use of the API [Green 2016]. The logging system should also provide pattern and anomaly detection to alert on potential injection attacks.
2. Protect the logging system and log files from potential tampering.
3. Employ user input validation and sanitization to filter out commonly known injections. This can be done at the API gateway, firewall, or router.
4. Ensure that error handling does not reveal excessive information about the API or the network. This includes HTTP codes, HTTP header information, and database response messages.
5. Consider adopting a zero trust approach in which no entity or user is inherently trusted. This can help increase visibility and create logging and monitoring systems that detect and prevent breaches [Almeida 2022].

3 Top API Security Risks

This section describes the top API risks, including recommendations for mitigating them.

3.1 Third-Party Software Integrations

Description. Often when creating new software, including APIs, existing software packages and libraries are used to make the design and implementation process faster and easier. However, integrating third-party software means introducing all the possible vulnerabilities of that software into the new API that is being built. Even the highest quality code can have up to 600 defects per million lines of code while average quality code has around 6,000 defects per million lines of code. Many of these defects can result in vulnerabilities [Wallen 2023].

Example. Log4J is a critical vulnerability in the Java-based logging library that was discovered in 2022. It allows remote code execution and the complete takeover of the victim machine. Any system that has Log4j integrated into its software is vulnerable to attack until the vulnerability is patched.

Recommendations. The following recommendations apply to this risk.

1. Perform risk assessments on all third-party software that might be used in a new API to prevent introducing additional API vulnerabilities.
2. Stay informed about the latest vulnerabilities in all third-party software that is integrated into currently deployed APIs.
3. Use automated vulnerability scanners on third-party software if possible.
4. Consider implementing a software bill of materials framework to help secure the supply chain and mitigate the risks of using third-party software [Wallen 2023].

3.2 Cascading Failures

Description. As previously mentioned, APIs are closely tied to microservices, which are small applications designed to perform a specific function. These microservices should be loosely coupled with high cohesion, which means that each of them works independently or mostly independently from other microservices. Designing the microservice architecture with independent applications prevents the entire API ecosystem from shutting down if a single microservice becomes broken or nonfunctional [Chandramouli 2019]. When one microservice's inoperability causes issues with other microservices, it is called *cascading failures*, and it should be avoided to reduce the risk of DoS attacks on the API.

Example. As illustrated in Figure 6, assume a Sales Information API must call a Product Description API to complete a client request it receives from the API gateway, but also assume that the Product Description API goes down. In this case, the Sales Information microservice can

no longer complete its request. As a result, the Sales Information application cannot be used because another application it relies on is down.

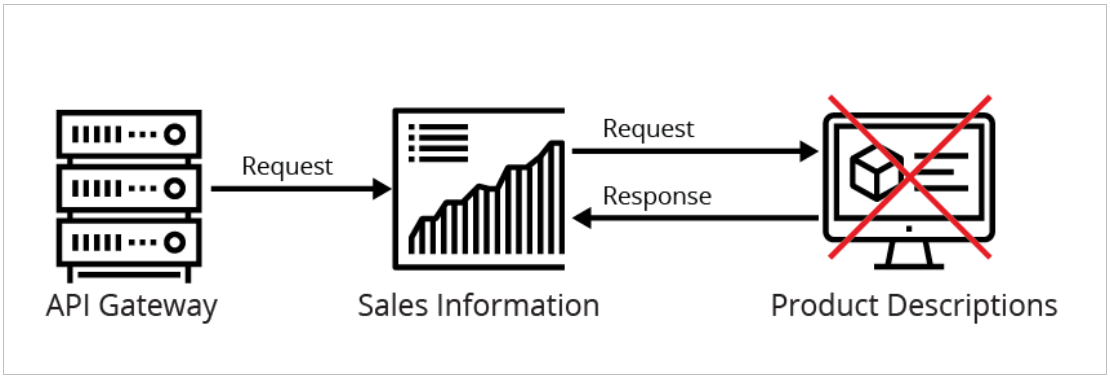

Figure 6: Cascading Failures Example

Recommendations. The following recommendations apply to this risk.

1. Design APIs with the goal of loose coupling in mind, including deployment coupling, temporal coupling, and implementational coupling.
2. Create API documentation that clearly shows which APIs are coupled and the communication flow that happens between coupled microservices. This documentation will be useful for testing and troubleshooting if there are issues with the API after deployment.
3. Conduct unit testing, integration testing, and end-to-end testing to confirm the correct functionality of the microservice architecture according to the documented design.
4. Strive to prevent single points of failure in the API network by adding redundancy measures in authentication and authorization points and other network structures [de Almeida 2022].

3.3 Increased Network Attack Surface

Description. Microservice architectures that heavily use APIs have many structural differences when compared to monolithic architectures. Those differences can often increase the overall network attack surface. Many non-microservice architectures have a web server in a demilitarized zone (DMZ) with a backend service one layer behind it that contains the database the API is accessing one layer behind that. This structure provides more defense in depth compared to microservices that can collapse this layered approach, which leads to more exposure [Chandramouli 2019].

Monolithic API architectures expose fewer IP-addressable remote procedure calls compared to microservice architectures [Chandramouli 2019]. The result is that these microservice architectures often have a larger attack surface. There are also many support structures, like the service discovery mechanism, needed to create a secure microservice system that creates more avenues for potential compromise if they are tampered with. Microservice architectures are generally more complex than monolithic architectures, which increases the likelihood of missed checks and misconfigurations that lead to vulnerabilities [Chandramouli 2019].

Example. Assume a microservice architecture currently has 10 API endpoints supported by one API gateway with a transactions per second (TPS) rate limit of 100 /second. If 10 more API endpoints are added with the same TPS rate limit, that means the same number of clients as before can now produce twice as many transactions in the same amount of time among the APIs. This could lead to delays in the API gateway processing the increased number of transactions, resulting in decreased customer service.

Recommendations. The following recommendations apply to this risk.

1. Carefully plan API versioning and deployments. Be strategic about adding new APIs so the maintenance and inventory of the existing APIs do not become overwhelming.
2. Add redundancy measures to prevent overload from API use in support structures, such as the API gateway and authentication servers [de Almeida 2022].
3. Maintain a thorough inventory of all APIs, preferably through auto-generated documentation for uniformity and searchability.
4. Configure thorough logging and monitoring efforts for all API activity, including logging all authentication and input failures.

4 Summary and Future Work

APIs are increasingly common, and they are often designed and implemented in a way that creates security risks. This report summarized 11 common vulnerabilities and 3 risks associated with APIs, and provided recommendations about fixing or reducing their impact. Some of the most common recommendations include the following:

- having a standard API documentation process
- using automated testing throughout the development process
- ensuring the identity and access management system is secure

Future work could extend the findings in this report to (1) explore integrating zero trust measures into the design of APIs and the development process and (2) create a set of rules and recommendations for API best practices.

References/Bibliography

URLs are valid as of the publication date of this report.

[Chandramouli 2019]
Chandramouli, Ramaswamy. *Security Strategies for Microservices-based Application Systems*. NIST SP 800-204. August 2019. https://csrc.nist.gov/pubs/sp/800/204/final

[de Almeida 2022]
de Almeida, Murilo Góes & Canedo, Edna Dias. Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. *Journal of Applied Science*. Volume 12. Issue 3023. https://doi.org/10.3390/app12063023

[Green 2016]
Green, Matthew & Smith, Matthew. Developers Are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy*. Volume 14. Issue 5. 2016. Pages 40–46. https://ieeexplore.ieee.org/abstract/document/7676144

[Hussain 2022]
Hussain, Al Alaa Abdul Al Muhsen; & Ipate, Pro Florentin. Application Programming Interface (API) Security: Cybersecurity Vulnerabilities Due to the Growing Use of APIs in Digital Communications. *International Research Journal of Innovations in Engineering and Technology*. *Volume* 6. Number 6. 2022. Page 108. https://www.proquest.com/openview/34b6b229f88e1cfb944a42b8dfebe934/1?pq-origsite=gscholar&cbl=5314840

[Lercher 2023]
Lercher, Alexander; Glock, Johann; Macho, Christian; & Pinzger, Martin. *Microservice API Evolution in Practice: A Study on Strategies and Challenges*. Cornell University. August 24, 2023. https://arxiv.org/pdf/2311.08175.pdf

[OWASP 2023]
Open Worldwide Application Security Project (OWASP). Top 10 API Security Risks—2023. *OWASP Website*. 2023. https://owasp.org/API-Security/editions/2023/en/0x00-header/

[Red Hat 2019]
Red Hat, Inc. What Is a GraphQL? *Red Hat Website*. January 8, 2019. https://www.redhat.com/en/topics/api/what-is-graphql

[Red Hat 2020]
Red Hat, Inc. What Is a REST API? *Red Hat Website*. May 8, 2020. https://www.redhat.com/en/topics/api/what-is-a-rest-api

[Singhal 2007]
Singhal, Anoop; Winograd, Theodore; & Carfone, Karen. *Guide to Secure Web Services*. NIST SP 800-95. National Institute of Standards and Technology (NIST). August 2007. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-95.pdf

[Traceable 2023]
Traceable Inc. API Security Reference Architecture for a Zero Trust World. *Traceable Website*. 2023. https://www.traceable.ai/resources/lp/whitepaper-api-security-reference-architecture

[Wallen 2023]
Wallen, Charles; Alberts, Christopher; Bandor, Michael; & Woody, Carol. Software Bill of Materials Framework: Leveraging SBOMs for Risk Reduction. Software Engineering Institute, Carnegie Mellon University. June 2023. https://insights.sei.cmu.edu/library/software-bill-of-materials-framework-leveraging-sboms-for-risk-reduction/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	April 2024	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Application Programming Interface (API) Vulnerabilities and Risks	FA8702-15-D-0002

6. AUTHOR(S)

McKinley Sconiers-Hasan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2024-SR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Application programming interfaces (APIs) are increasingly common, and they are often designed and implemented in a way that creates security risks. This report describes 11 common vulnerabilities and 3 risks related to APIs, providing suggestions about how to fix or reduce their impact. Recommendations include using a standard API documentation process, using automated testing, and ensuring the security of the identity and access management system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
application programming interfaces, APIs, common vulnerabilities, risks	29

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102