**OWASP API Security Top 10**

[**API1:2019 – Broken object level authorization** 2](#_Toc79550886)

[**Example Attack Scenarios** 2](#_Toc79550887)

[**How To Prevent** 3](#_Toc79550888)

[**API2:2019 – Broken Authentication** 4](#_Toc79550889)

[**Example Attack Scenarios** 4](#_Toc79550890)

[**Scenario 1** 4](#_Toc79550891)

[**Scenario 2** 4](#_Toc79550892)

[**How To Prevent** 5](#_Toc79550893)

[**API3:2019 – Excessive data exposure** 7](#_Toc79550894)

[**Example Attack Scenarios** 7](#_Toc79550895)

[**How To Prevent** 7](#_Toc79550896)

[**API4:2019 – Lack of resources and rate limiting** 8](#_Toc79550897)

[**Example Attack Scenarios** 8](#_Toc79550898)

[**Scenario 1** 8](#_Toc79550899)

[**Scenario 2** 9](#_Toc79550900)

[**How To Prevent** 10](#_Toc79550901)

[**API5:2019 – Broken function level authorization** 10](#_Toc79550902)

[**Example Attack Scenarios** 10](#_Toc79550903)

[**How To Prevent** 11](#_Toc79550904)

[**API6:2019 – Mass assignment** 12](#_Toc79550905)

[**Example Attack Scenarios** 12](#_Toc79550906)

[**How To Prevent** 13](#_Toc79550907)

[**API7:2019 – Security Misconfiguration** 14](#_Toc79550908)

[**Example Attack Scenarios** 14](#_Toc79550909)

[**Scenario 1** 14](#_Toc79550910)

[**Scenario 2** 14](#_Toc79550911)

[**Scenario 3** 14](#_Toc79550912)

[**How To Prevent** 15](#_Toc79550913)

[**API8:2019 – Injection** 16](#_Toc79550914)

[**Example Attack Scenarios** 16](#_Toc79550915)

[**How To Prevent** 16](#_Toc79550916)

[**API9:2019 – Improper Assets Management** 17](#_Toc79550917)

[**Example Attack Scenarios** 17](#_Toc79550918)

[**How To Prevent** 18](#_Toc79550919)

[**API10:2019 – Insufficient Logging & Monitoring** 19](#_Toc79550920)

[**Example Attack Scenarios** 19](#_Toc79550921)

[**Scenario 1** 19](#_Toc79550922)

[**Scenario 2** 20](#_Toc79550923)

[**How To Prevent** 21](#_Toc79550924)

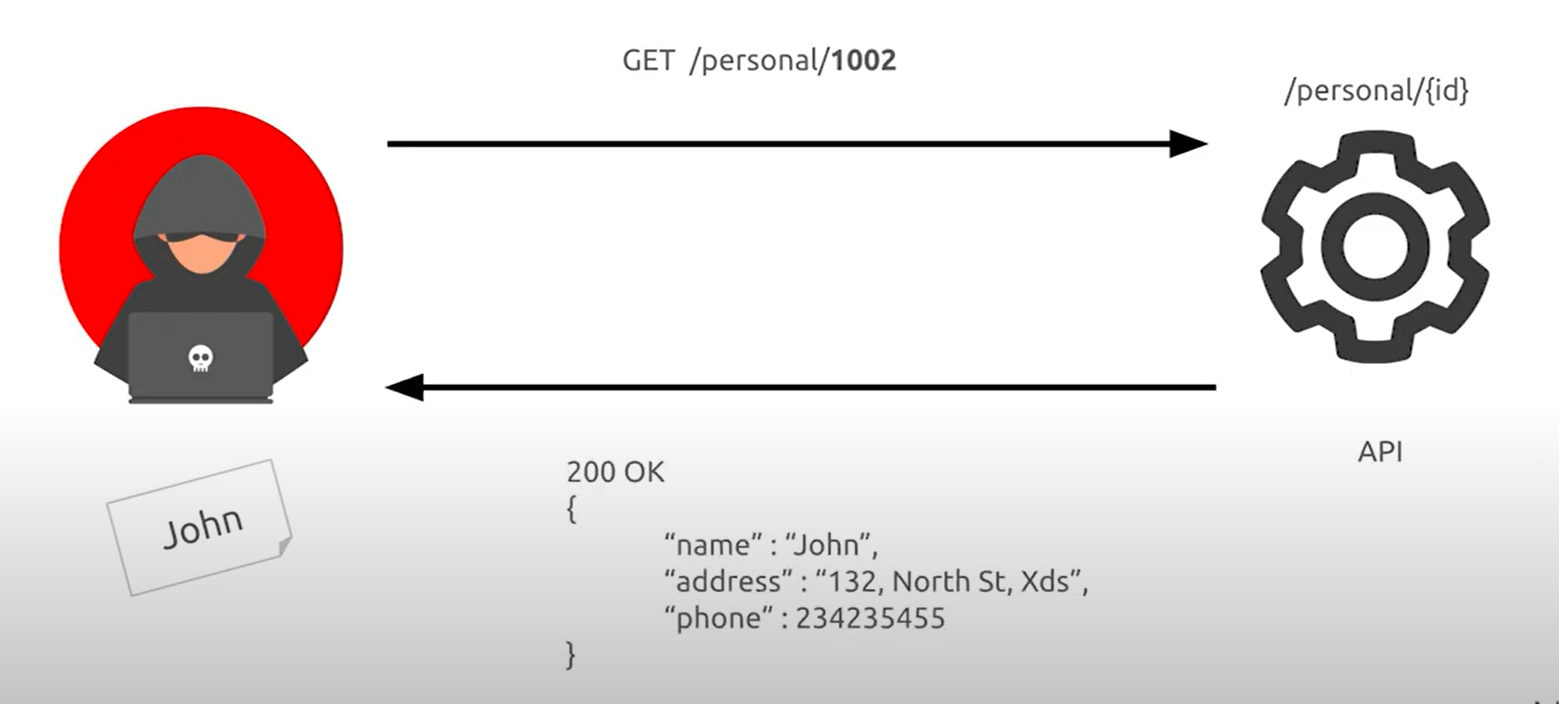
# **API1:2019 – Broken object level authorization**

APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.

## **Example Attack Scenarios**

User John has id is 1002.

User John call api GET /personal/1002. API response user’s information.

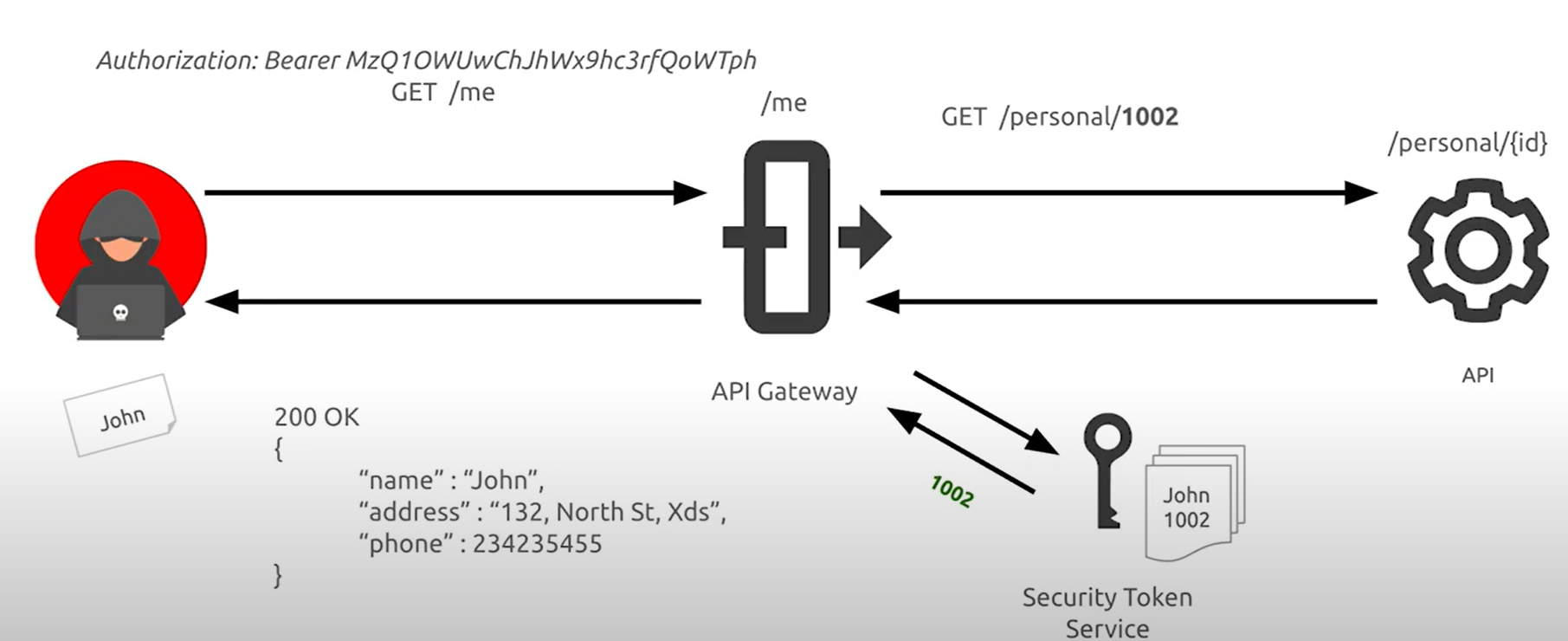


User John change id 1002 to 1001 and call api GET /personal/1001 . API response Mary’s information because Mary has id 1001.

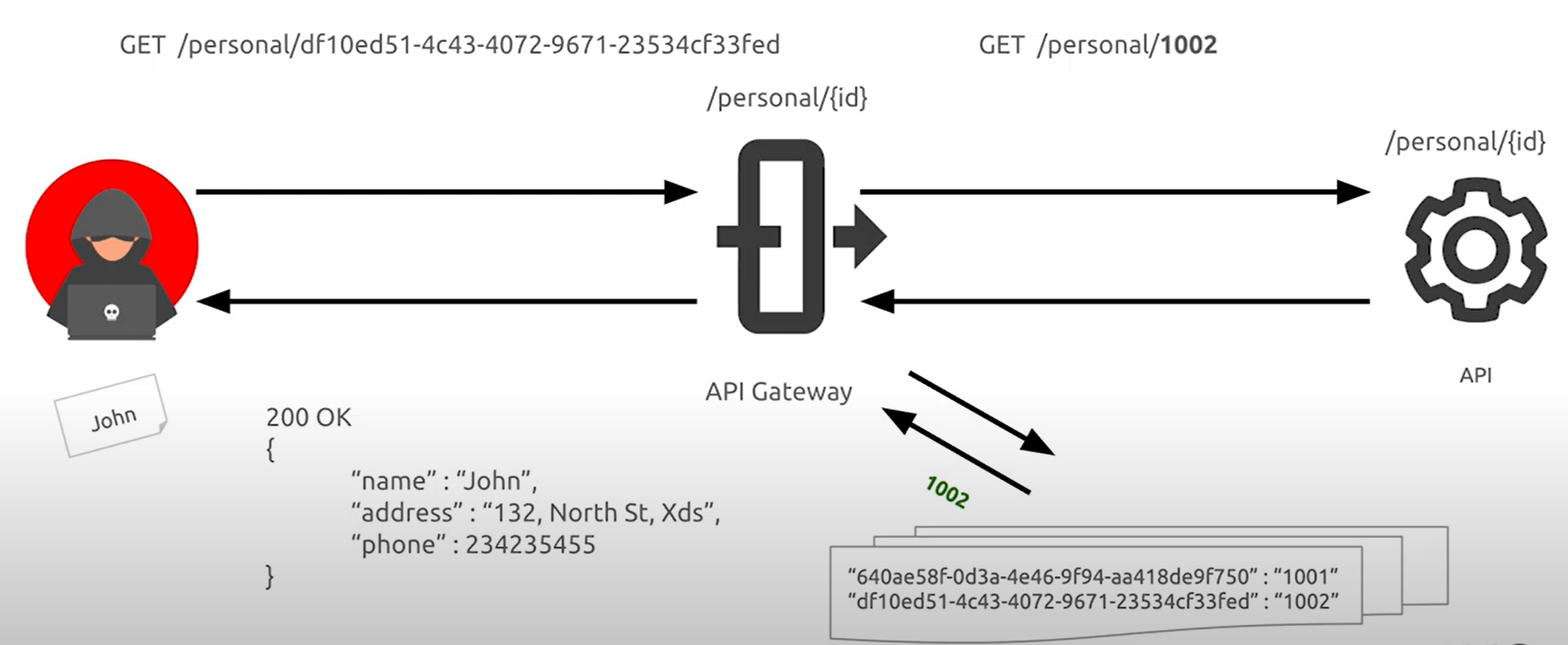


## **How To Prevent**

* Implement a proper authorization mechanism that relies on the user policies and hierarchy.
* Use an authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database.
* Prefer to use random and unpredictable values as GUIDs for records’ IDs.
* Write tests to evaluate the authorization mechanism. Do not deploy vulnerable changes that break the tests.



* Prefer to use random and unpredictable values as GUIDs for records’ IDs.



# **API2:2019 – Broken Authentication**

Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user’s identities temporarily or permanently. Compromising a system’s ability to identify the client/user, compromises API security overall.

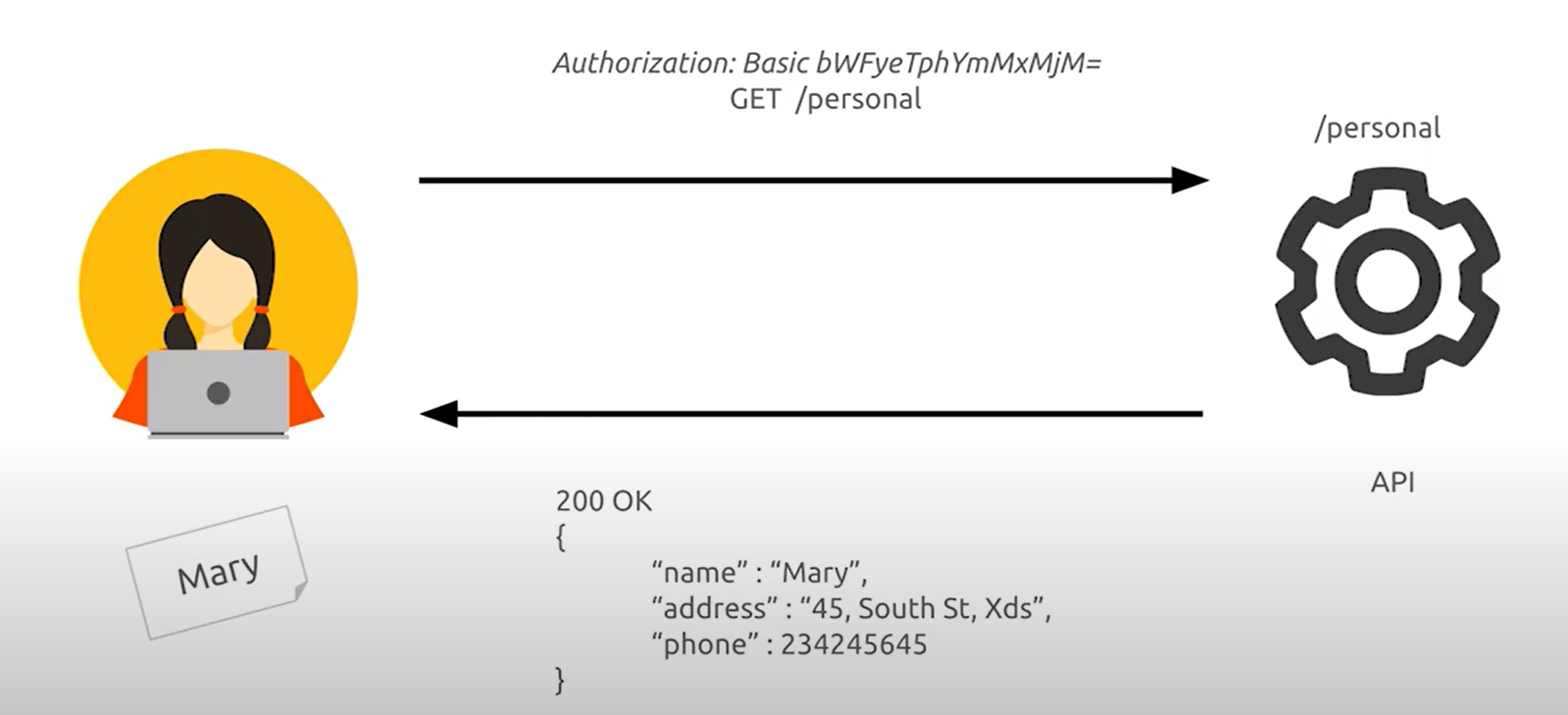
## **Example Attack Scenarios**

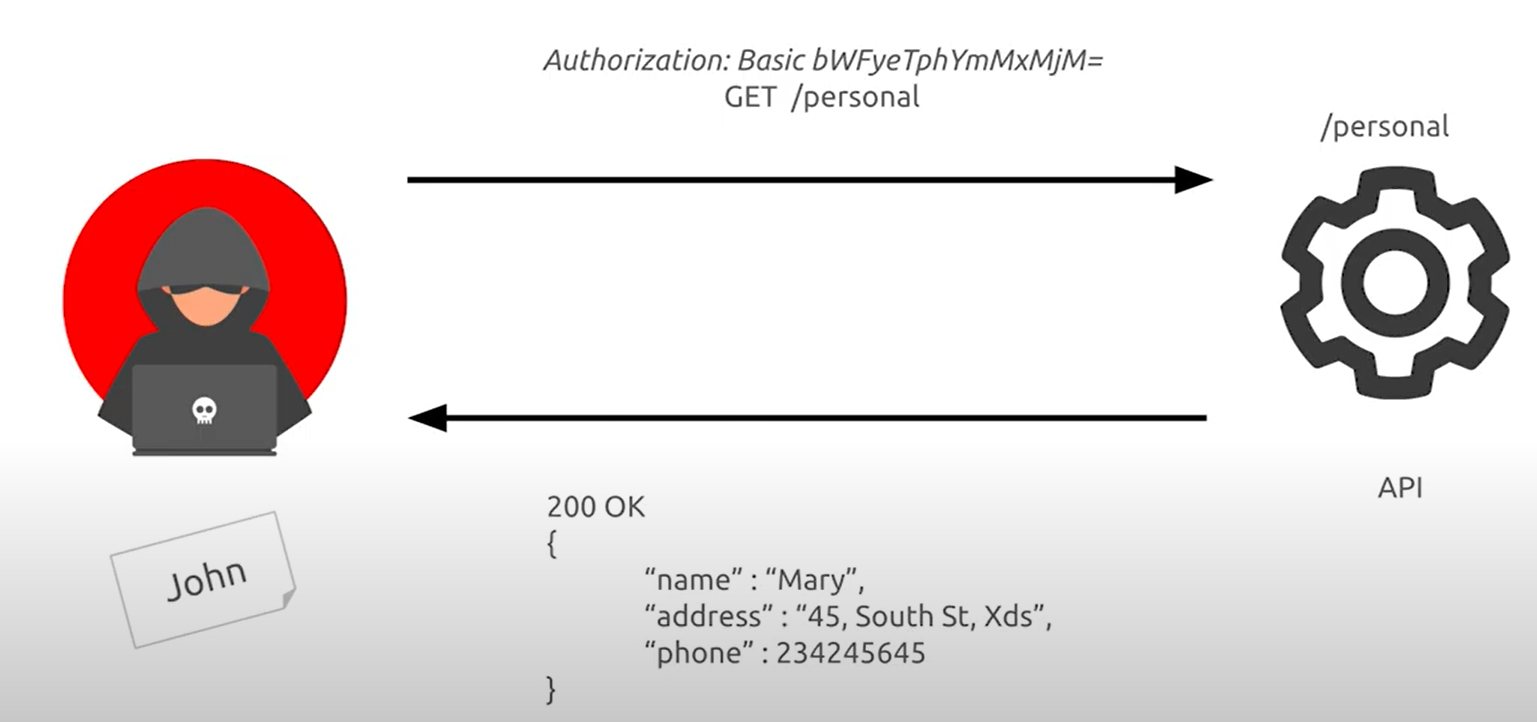
## **Scenario 1**

Credential stuffing (using lists of known usernames/passwords), is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle (tester) to determine if the credentials are valid.

## **Scenario 2**

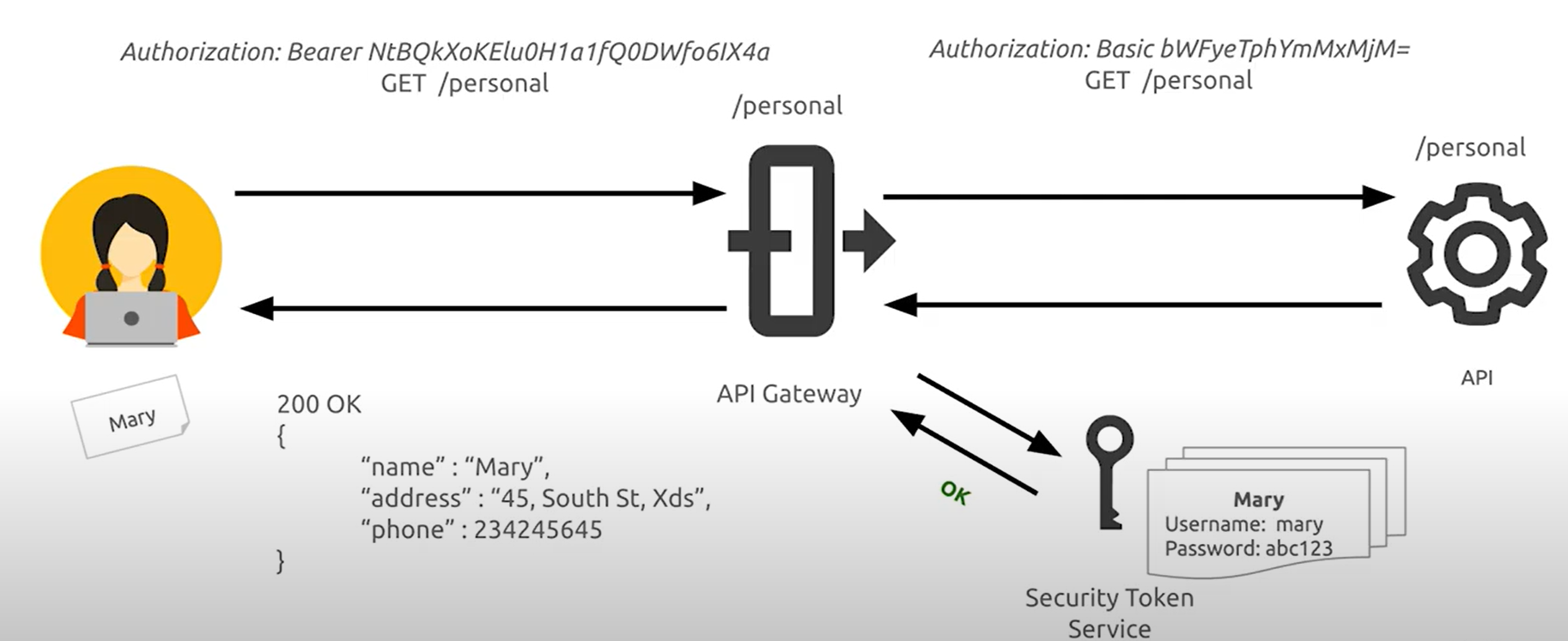
An attacker starts the password recovery workflow by issuing a POST request to /api/system/verification-codes and by providing the username in the request body. Next an SMS token with 6 digits is sent to the victim’s phone. Because the API does not implement a rate limiting policy, the attacker can test all possible combinations using a multi-threaded script, against the /api/system/verification-codes/{smsToken} endpoint to discover the right token within a few minutes.

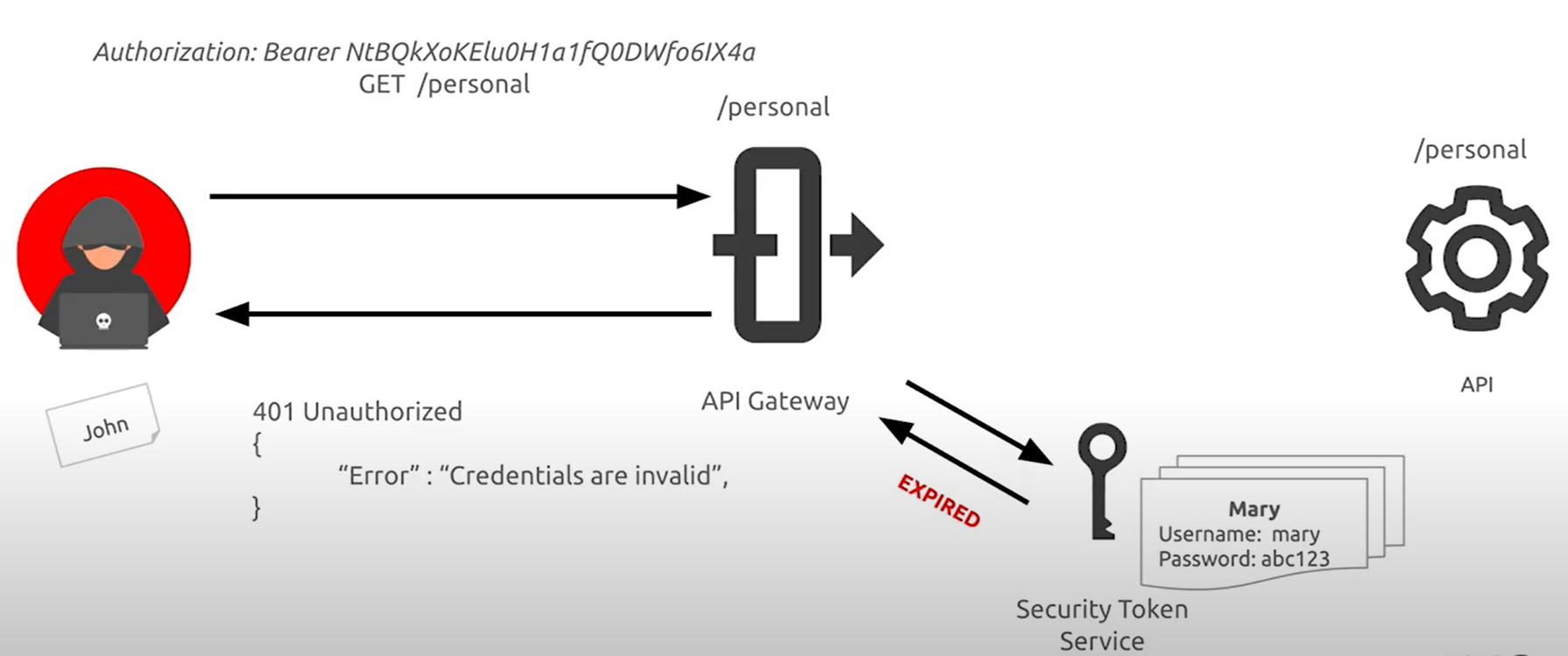




## **How To Prevent**

* Make sure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.)
* Ask your engineers what flows you missed.
* Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither is API keys.
* Don't reinvent the wheel in authentication, token generation, password storage. Use the standards.
* Credential recovery/forget password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections.
* Use the [OWASP Authentication Cheatsheet](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html).
* Where possible, implement multi-factor authentication.
* Implement anti brute force mechanisms to mitigate credential stuffing, dictionary attack, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanism on your API.
* Implement [account lockout](https://www.owasp.org/index.php/Testing_for_Weak_lock_out_mechanism_(OTG-AUTHN-003)) / captcha mechanism to prevent brute force against specific users. Implement weak-password checks.
* API keys should not be used for user authentication, but for [client app/ project authentication](https://cloud.google.com/endpoints/docs/openapi/when-why-api-key).





# **API3:2019 – Excessive data exposure**

Looking forward to generic implementations, developers tend to expose all object properties without considering their individual sensitivity, relying on clients to perform the data filtering before displaying it to the user.

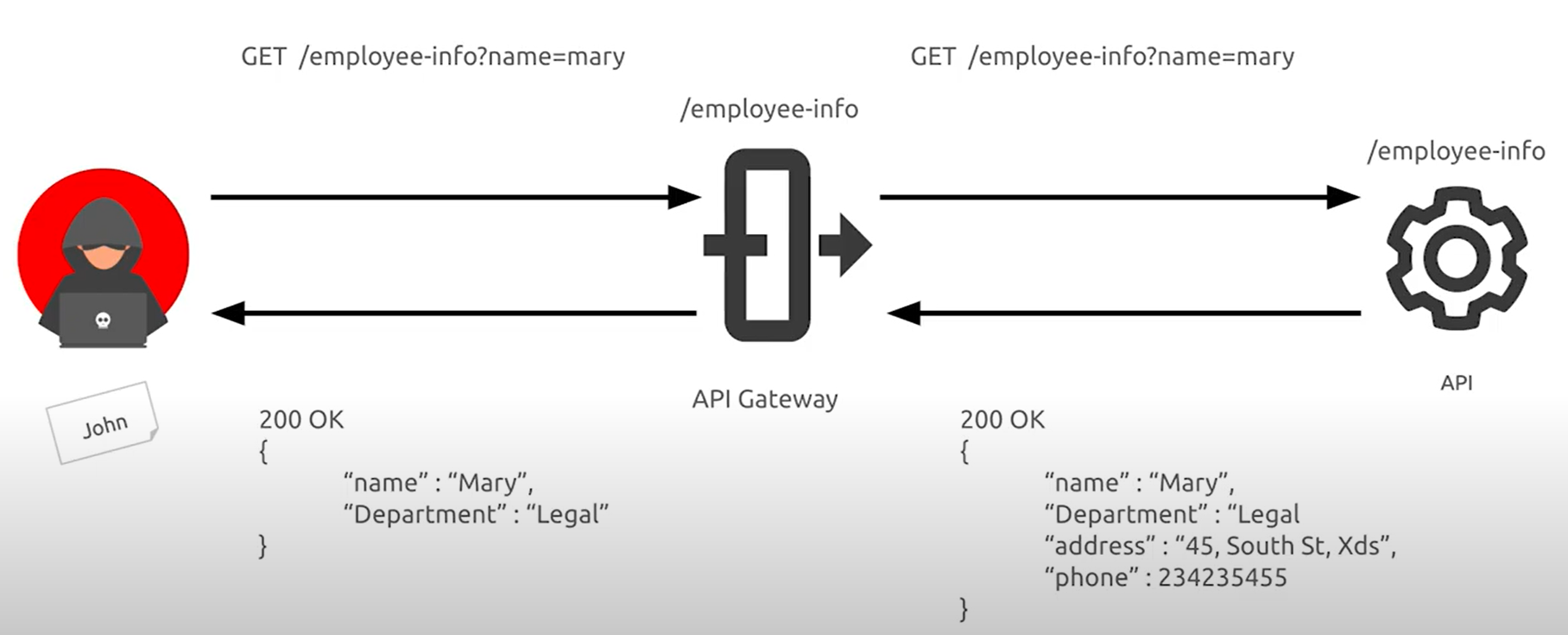
## **Example Attack Scenarios**

User call API to see information employee GET /employee-info?name=mary and API response has other information importan as phone number, address,etc…



## **How To Prevent**

* Never rely on the client side to filter sensitive data.
* Review the responses from the API to make sure they contain only legitimate data.
* Backend engineers should always ask themselves "who is the consumer of the data?" before exposing a new API endpoint.
* Avoid using generic methods such as to\_json() and to\_string(). Instead, cherry-pick specific properties you really want to return
* Classify sensitive and personally identifiable information (PII) that your application stores and works with, reviewing all API calls returning such information to see if these responses pose a security issue.
* Implement a schema-based response validation mechanism as an extra layer of security. As part of this mechanism define and enforce data returned by all API methods, including errors.



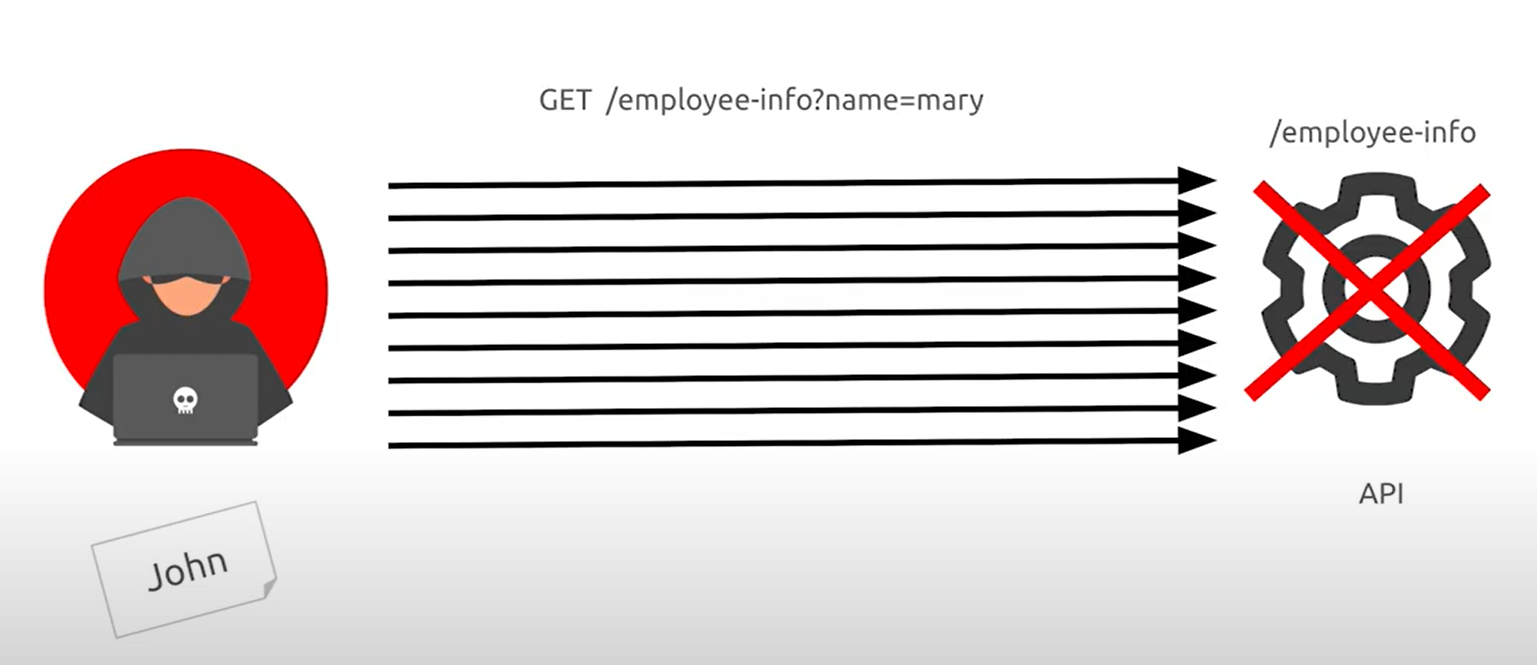
# **API4:2019 – Lack of resources and rate limiting**

Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force.

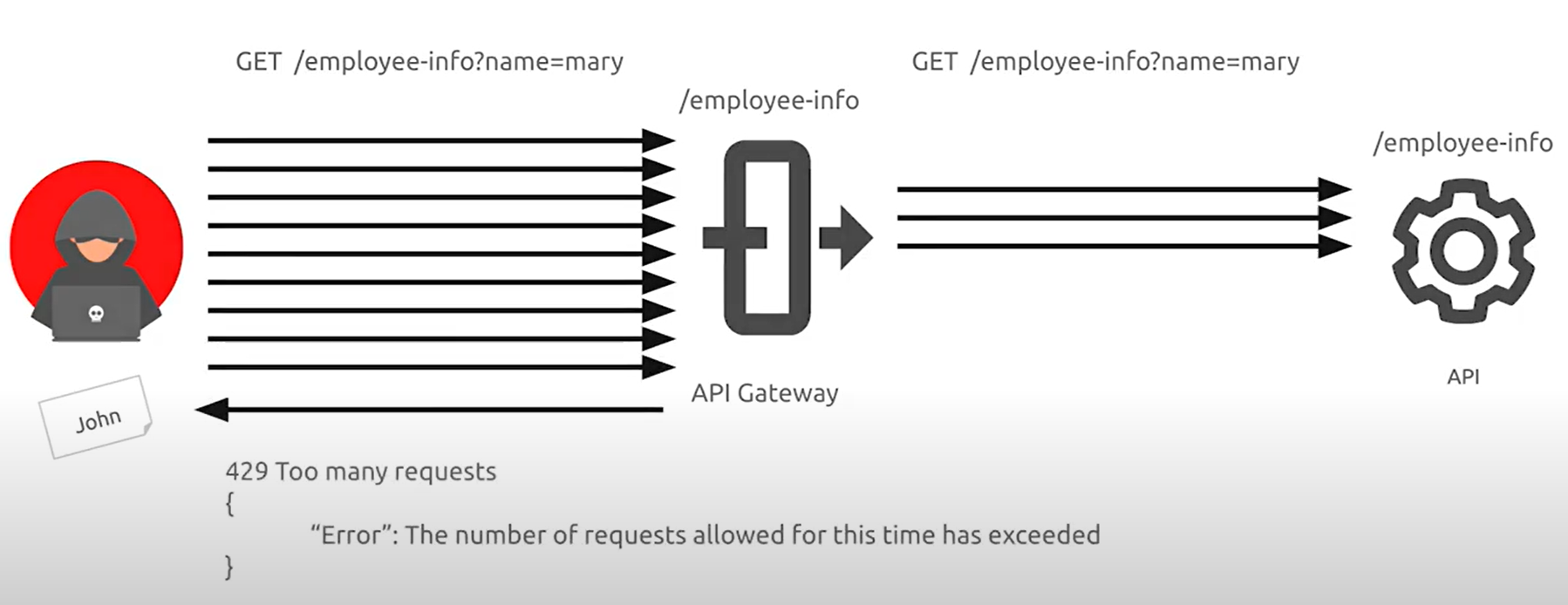
## **Example Attack Scenarios**

### **Scenario 1**

Attacker send multi threads request API and API system not response (DoS).

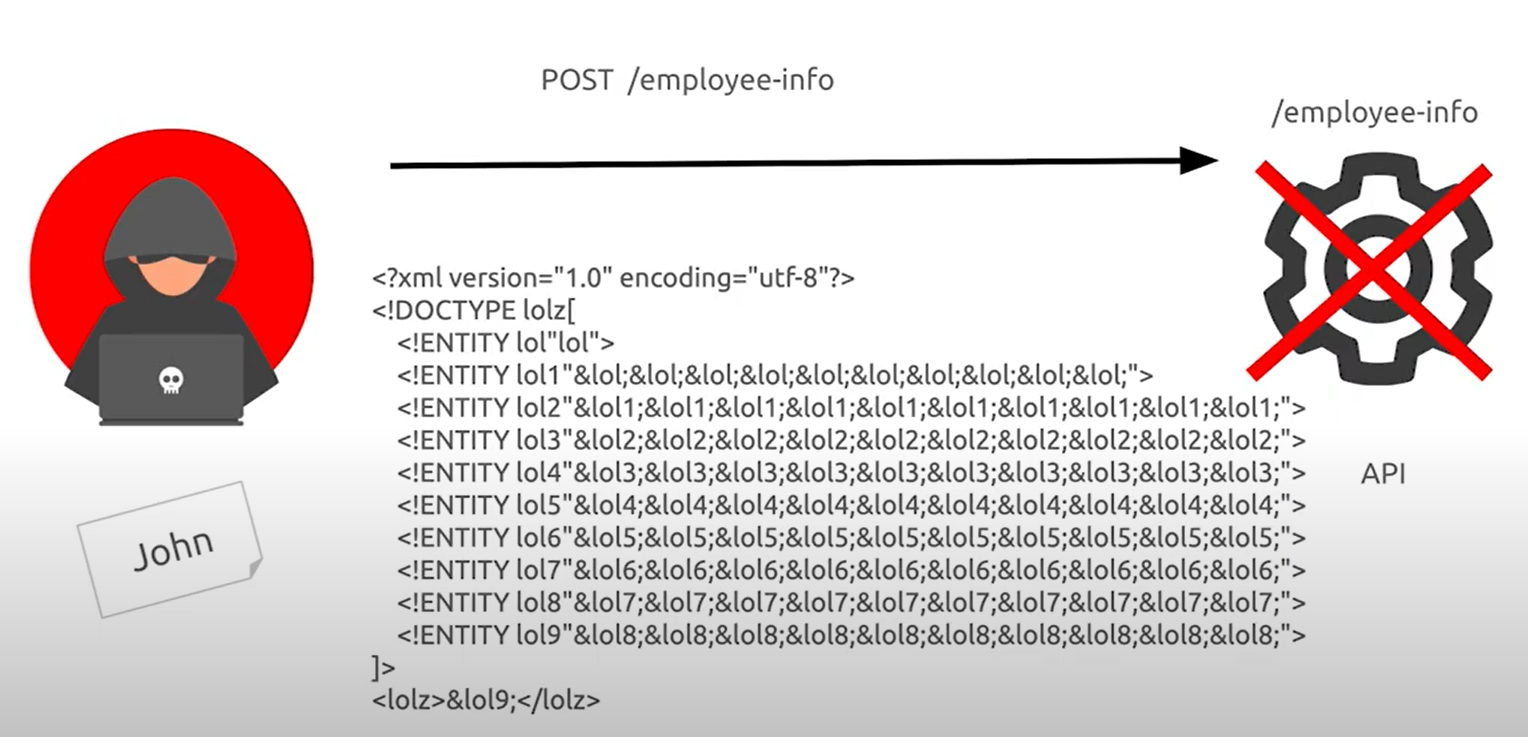


**Solution:**

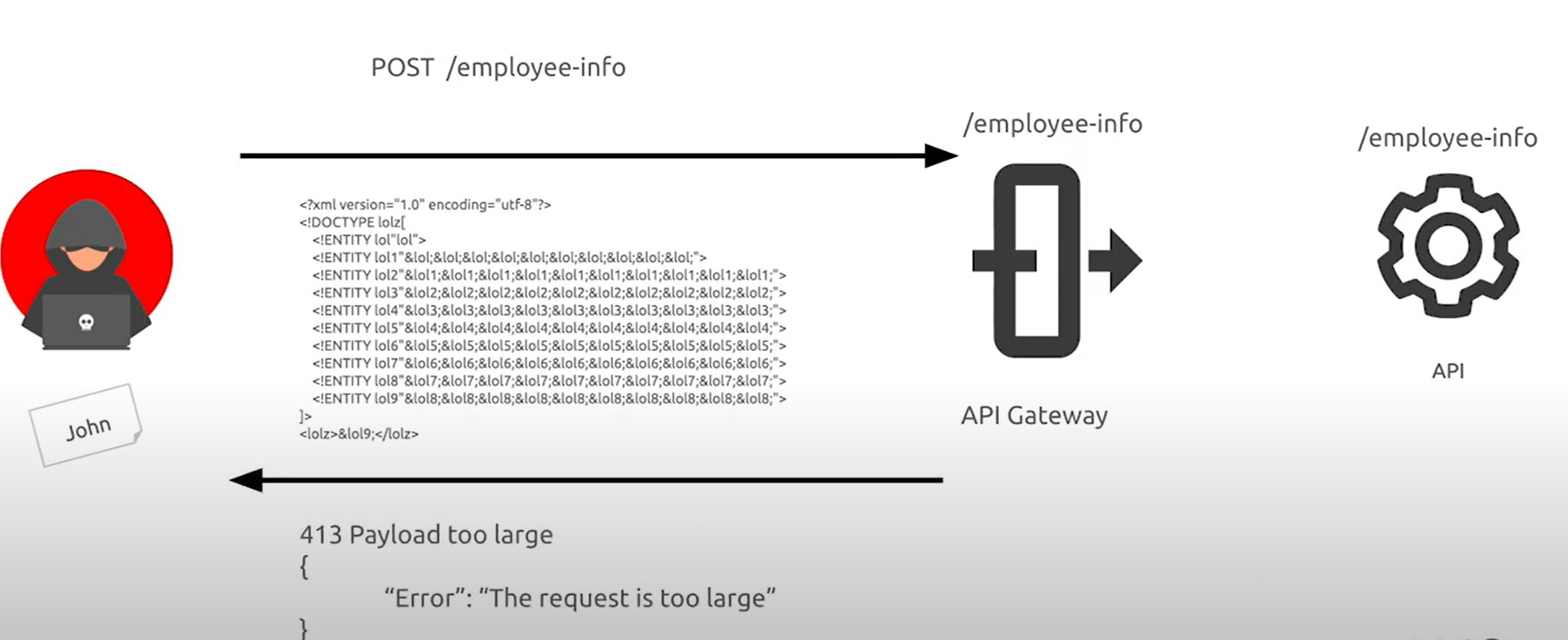


### **Scenario 2**

Attacker send request with big data and API is not response (DoS).



**Solution:**



## **How To Prevent**

* Docker makes it easy to limit [memory](https://docs.docker.com/config/containers/resource_constraints/#memory), [CPU](https://docs.docker.com/config/containers/resource_constraints/#cpu), [number of restarts](https://docs.docker.com/engine/reference/commandline/run/#restart-policies---restart), [file descriptors, and processes](https://docs.docker.com/engine/reference/commandline/run/#set-ulimits-in-container---ulimit).
* Implement a limit on how often a client can call the API within a defined timeframe.
* Notify the client when the limit is exceeded by providing the limit number and the time at which the limit will be reset.
* Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response.
* Define and enforce maximum size of data on all incoming parameters and payloads such as maximum length for strings and maximum number of elements in arrays.

# **API5:2019 – Broken function level authorization**

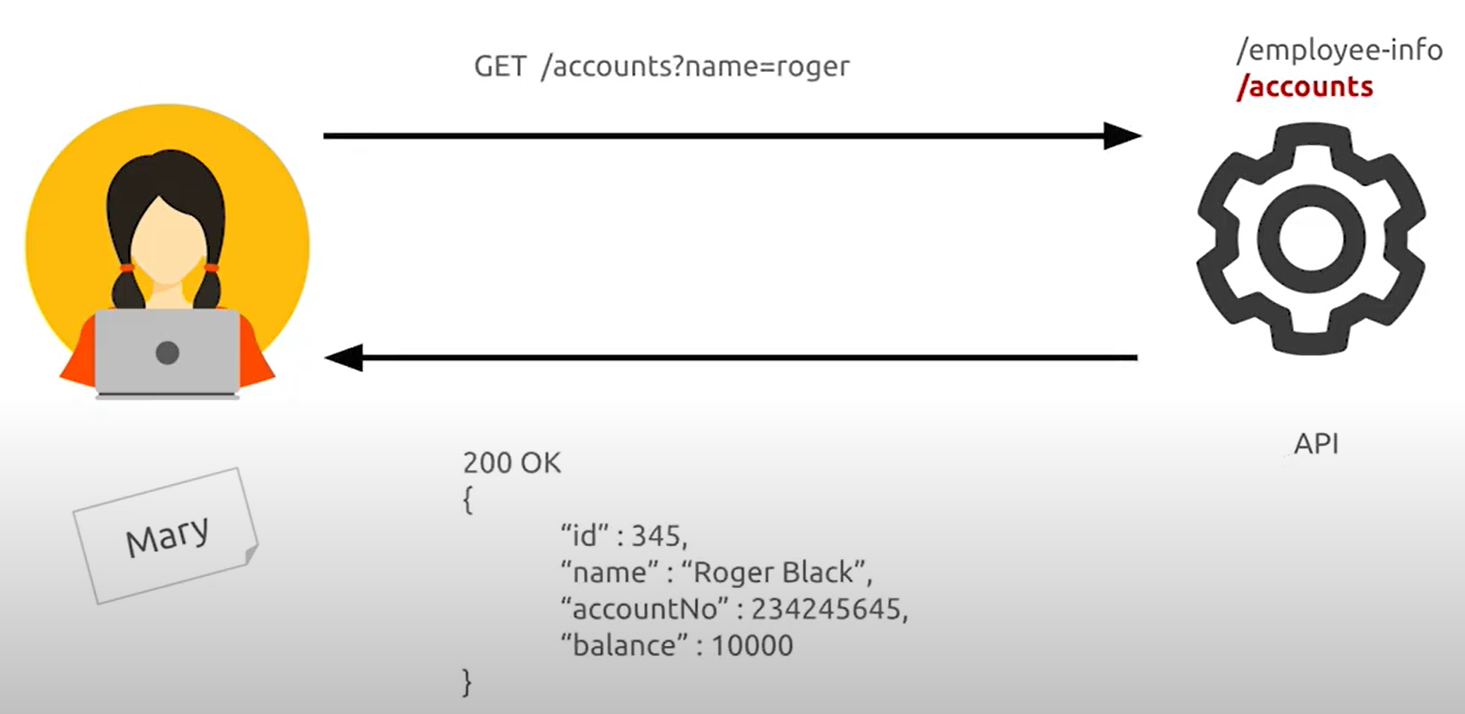
Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers gain access to other users’ resources and/or administrative functions.

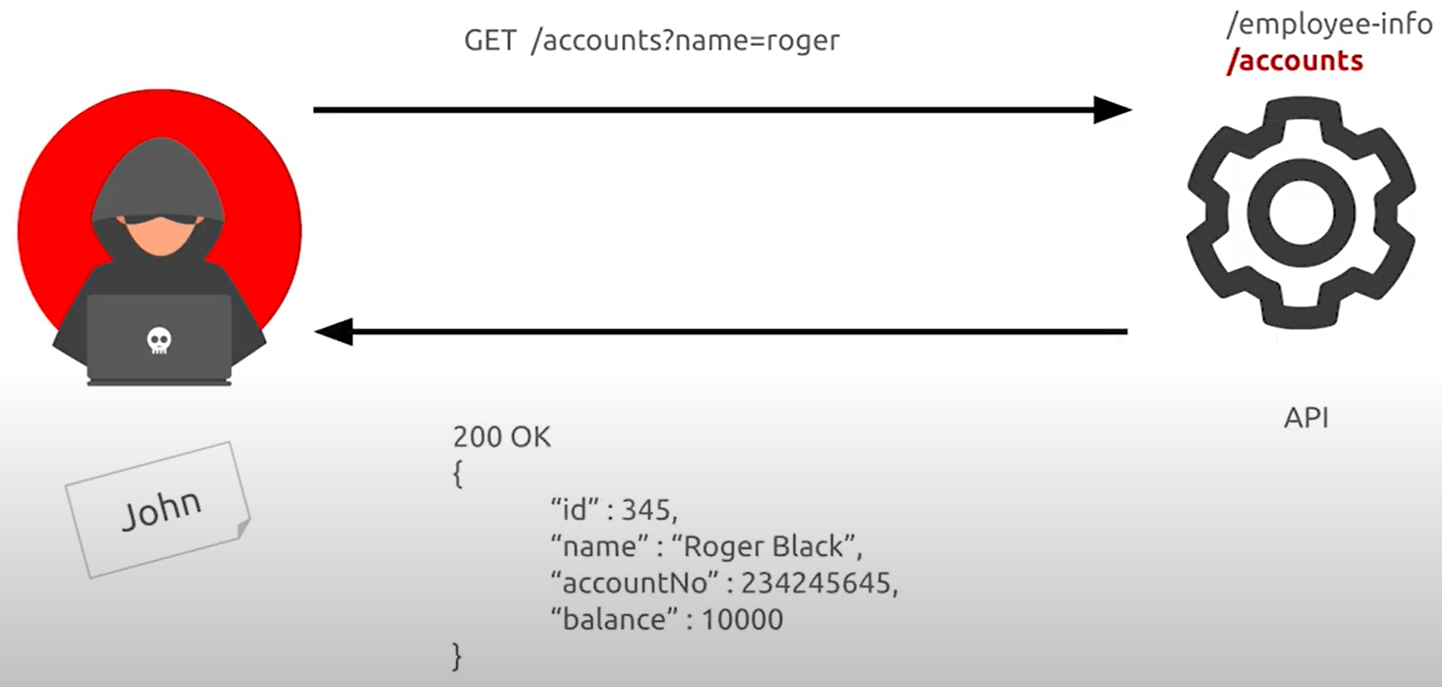
## **Example Attack Scenarios**

API see information account only user in group Finance request.

Employee Mary in group Finance call api GET /accounts?name=roger to see information name’s employee is Roger Black.

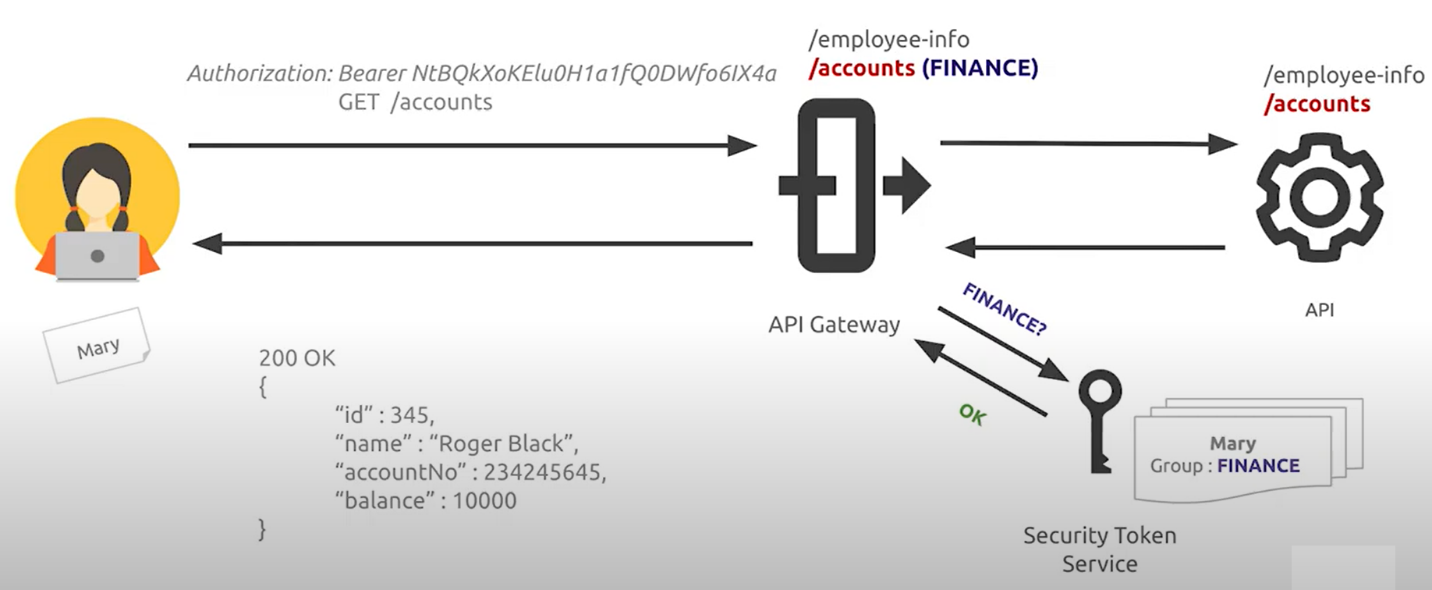
User John in group IT call api GET /accounts?name=roger to see information name’s employee is Roger Black and API ressponse send back employee’s information.

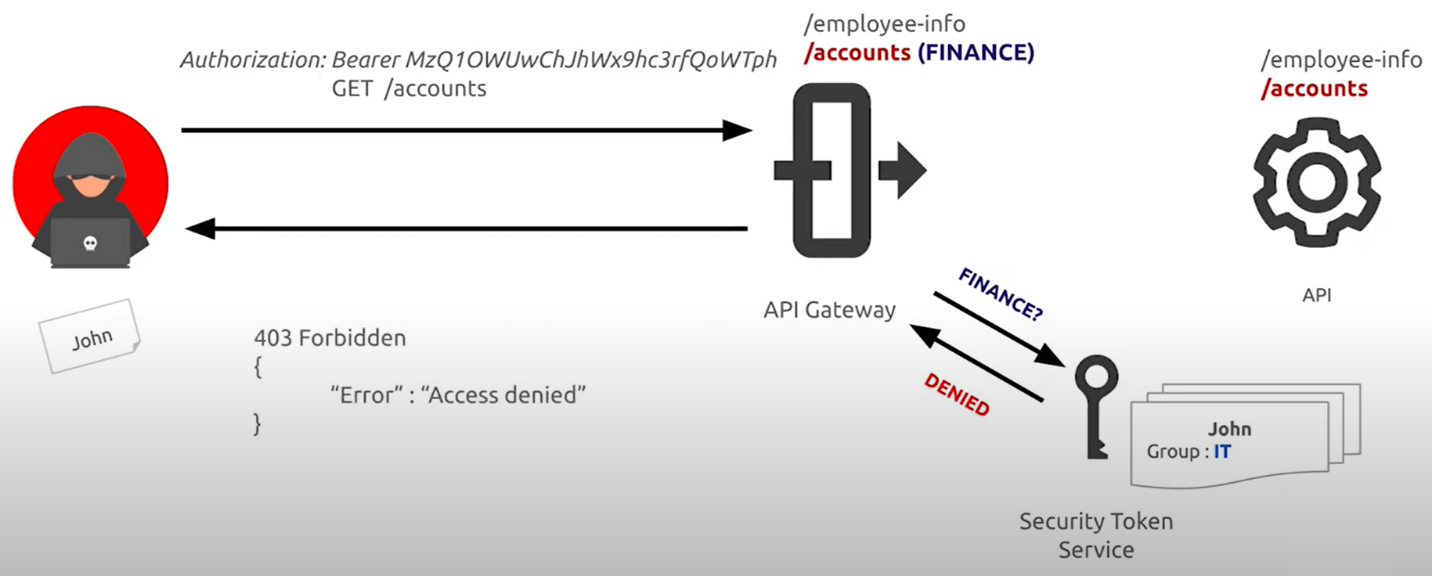




## **How To Prevent**

* The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
* Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy.
* Make sure that all of your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user’s group/role.
* Make sure that administrative functions inside a regular controller implements authorization checks based on the user’s group and role.





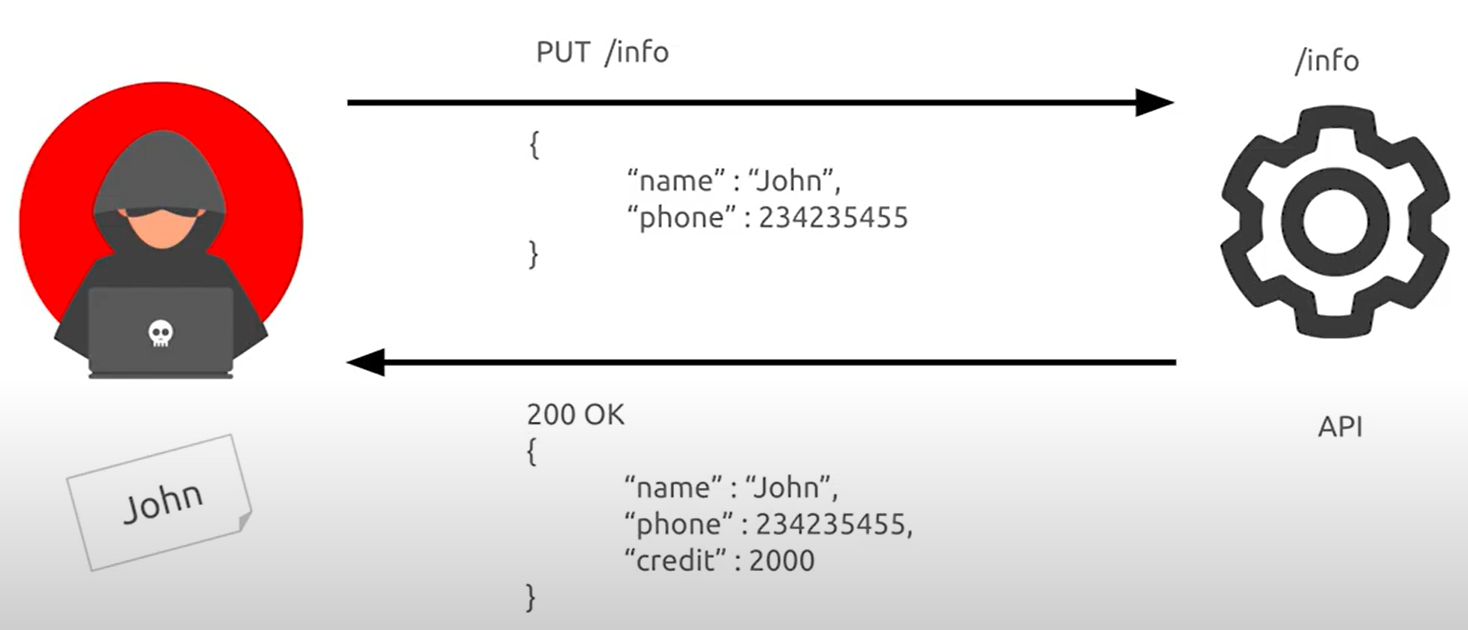
# **API6:2019 – Mass assignment**

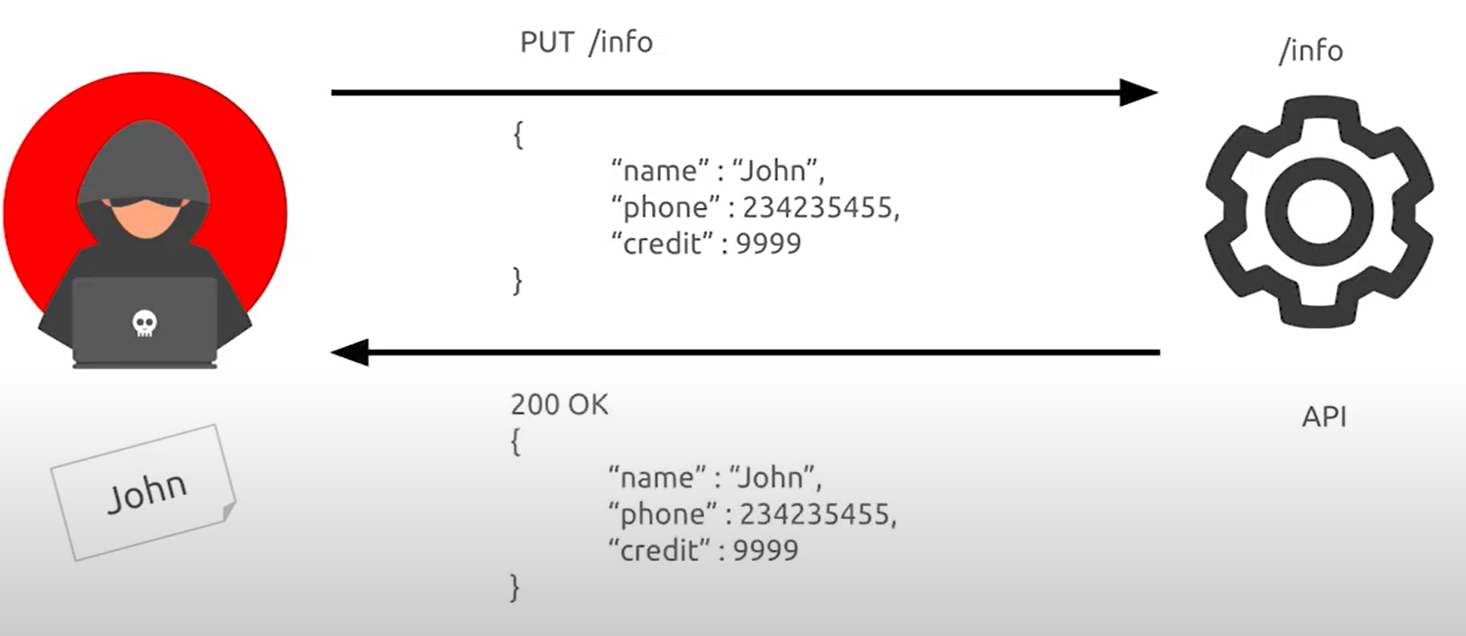
Binding client provided data (e.g., JSON) to data models, without proper properties filtering based on an allowlist, usually leads to Mass Assignment. Either guessing objects properties, exploring other API endpoints, reading the documentation, or providing additional object properties in request payloads, allows attackers to modify object properties they are not supposed to.

## **Example Attack Scenarios**

User normal only update name and phone, api response send have information credit.

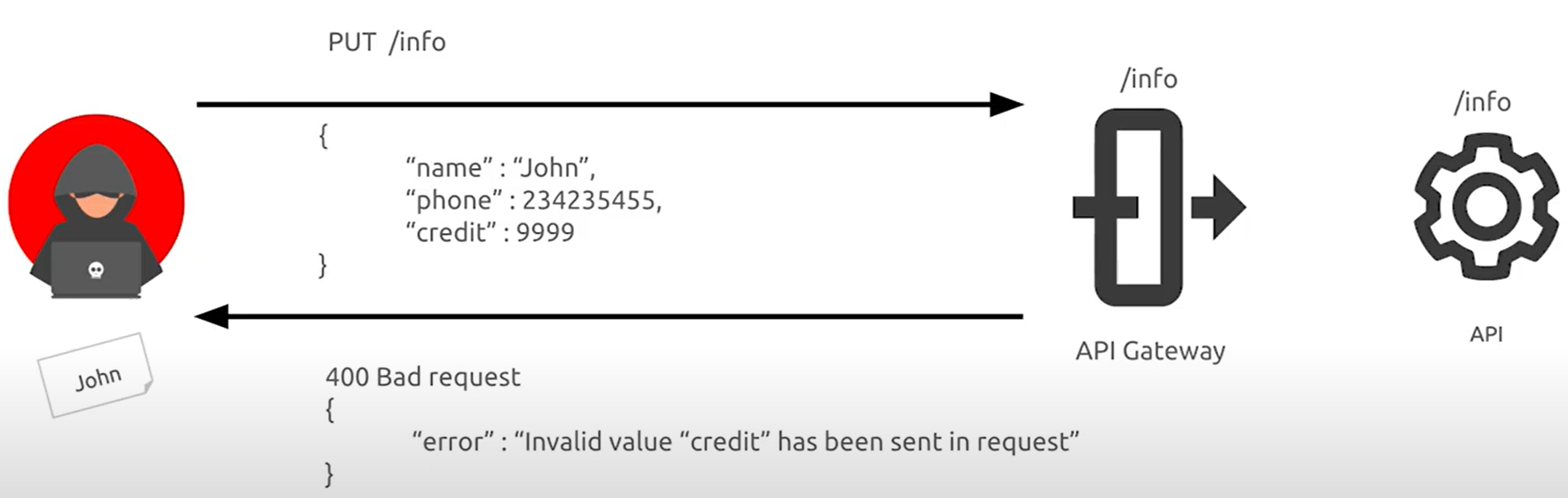
User normal update endpoint credit, because api doesn’t has filter to validate data so data update successfully.





## **How To Prevent**

* If possible, avoid using functions that automatically bind a client’s input into code variables or internal objects.
* Whitelist only the properties that should be updated by the client.
* Use built-in features to blacklist properties that should not be accessed by clients.
* If applicable, explicitly define and enforce schemas for the input data payloads.



# **API7:2019 – Security Misconfiguration**

Security misconfiguration is commonly a result of unsecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, permissive Cross-Origin resource sharing (CORS), and verbose error messages containing sensitive information.

## **Example Attack Scenarios**

### **Scenario 1**

An attacker finds the .bash\_history file under the root directory of the server, which contains commands used by the DevOps team to access the API:

|  |
| --- |
| $ curl -X GET 'https://api.server/endpoint/' -H 'authorization: Basic Zm9vOmJhcg==' |

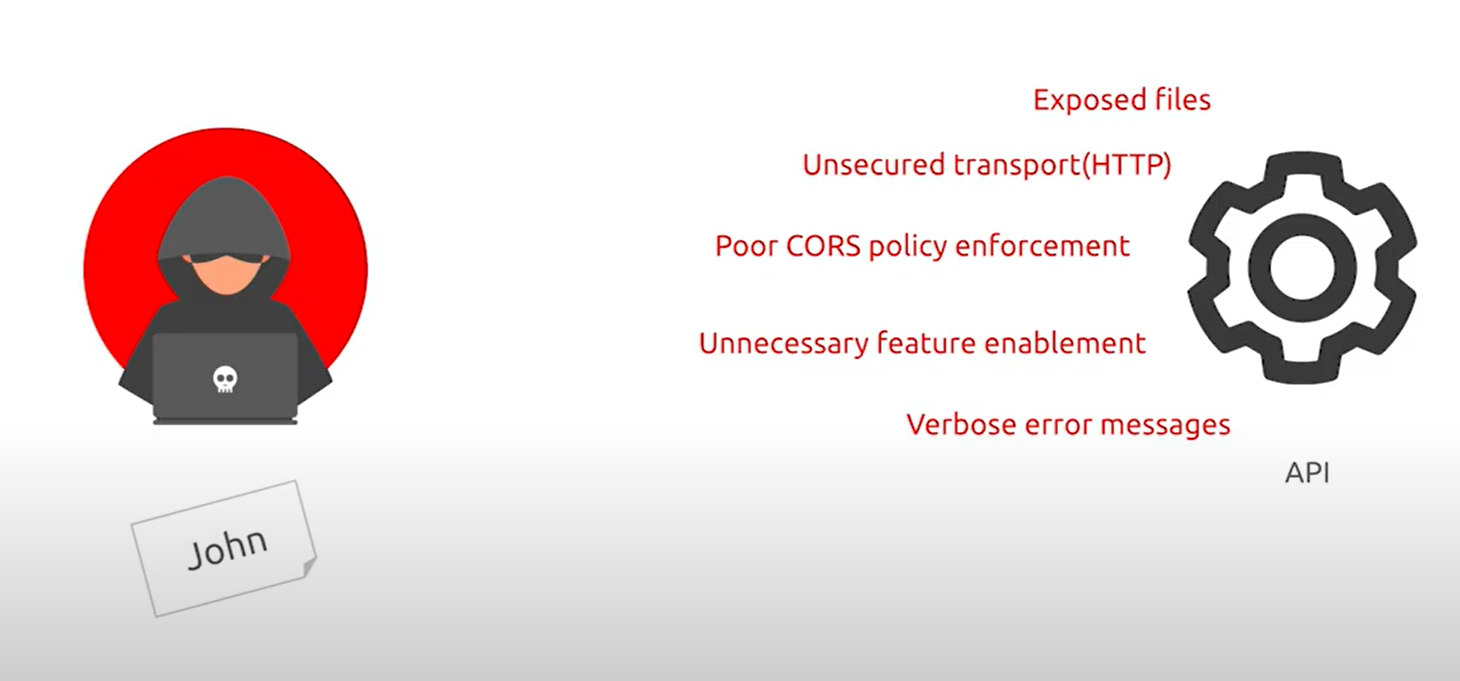
An attacker could also find new endpoints on the API that are used only by the DevOps team and are not documented.

### **Scenario 2**

To target a specific service, an attacker uses a popular search engine to search for computers directly accessible from the Internet. The attacker found a host running a popular database management system, listening on the default port. The host was using the default configuration, which has authentication disabled by default, and the attacker gained access to millions of records with PII, personal preferences, and authentication data.

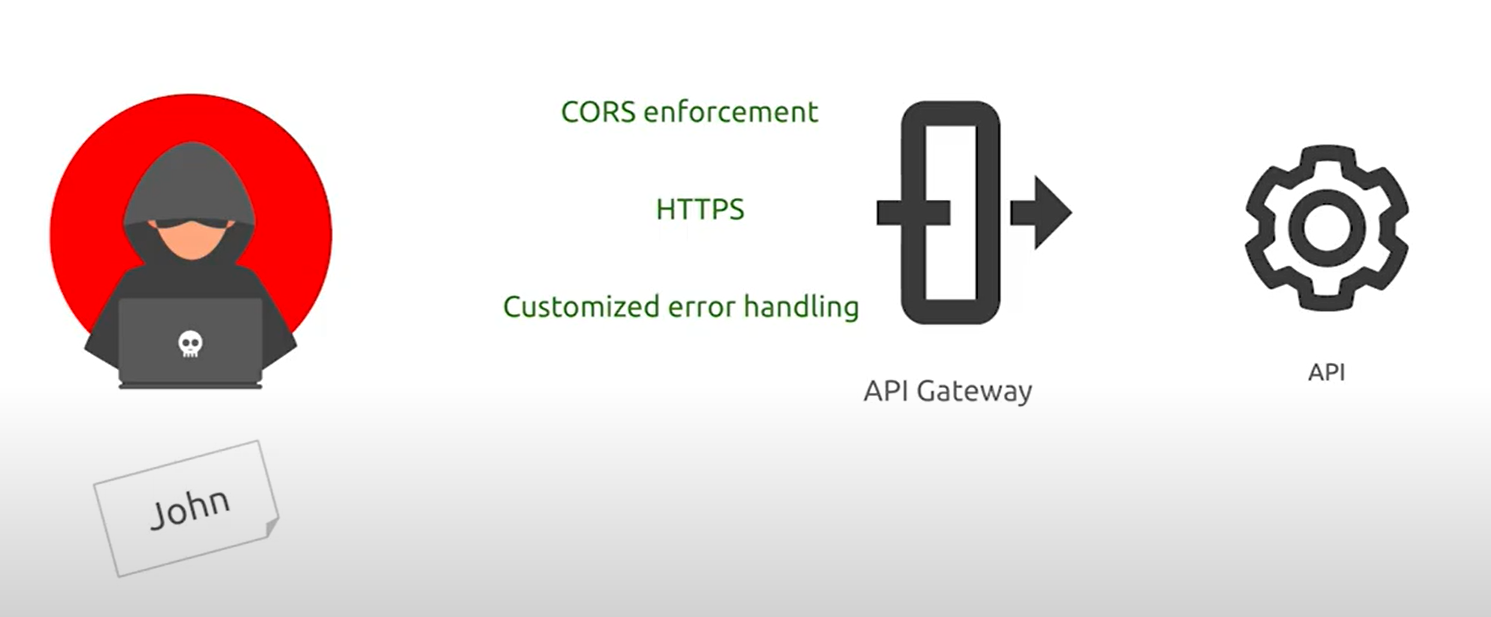
### **Scenario 3**

Inspecting traffic of a mobile application an attacker finds out that not all HTTP traffic is performed on a secure protocol (e.g., TLS). The attacker finds this to be true, specifically for the download of profile images. As user interaction is binary, despite the fact that API traffic is performed on a secure protocol, the attacker finds a pattern on API responses size, which he uses to track user preferences over the rendered content (e.g., profile images).



## **How To Prevent**

* To prevent exception traces and other valuable information from being sent back to attackers, if applicable, define and enforce all API response payload schemas including error responses.
* Ensure API can only be accessed by the specified HTTP verbs. All other HTTP verbs should be disabled (e.g. HEAD).
* APIs expecting to be accessed from browser-based clients (e.g., WebApp front-end) should implement a proper Cross-Origin Resource Sharing (CORS) policy.

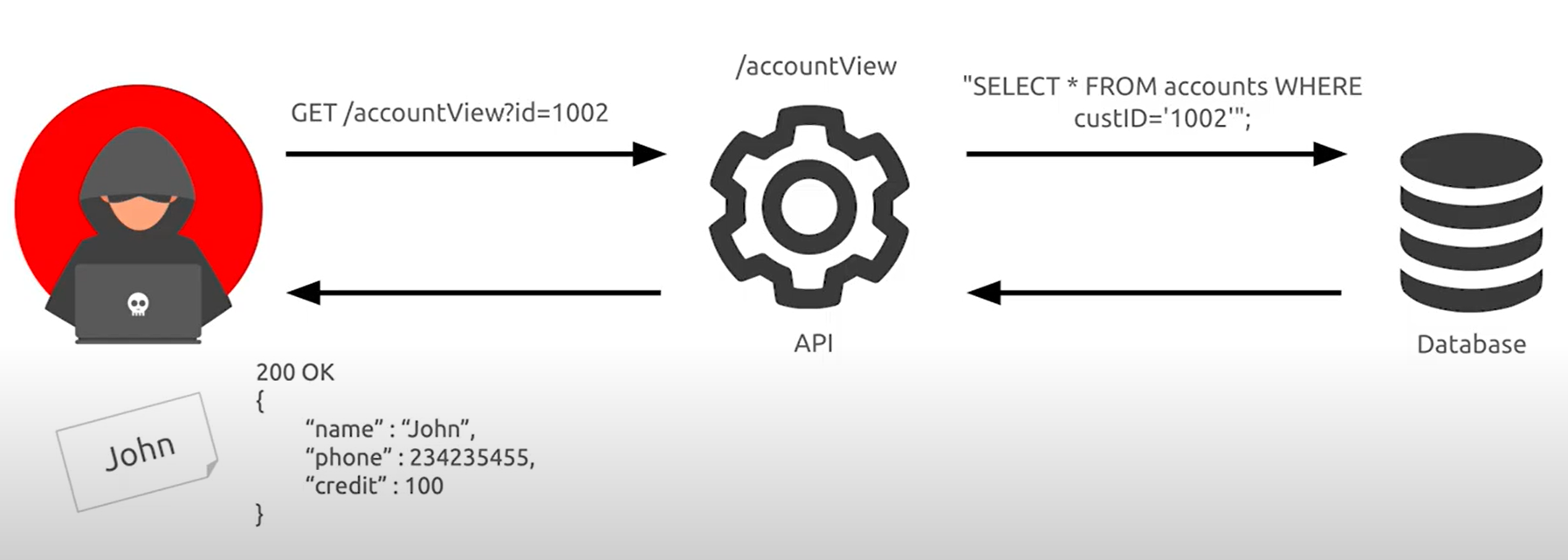


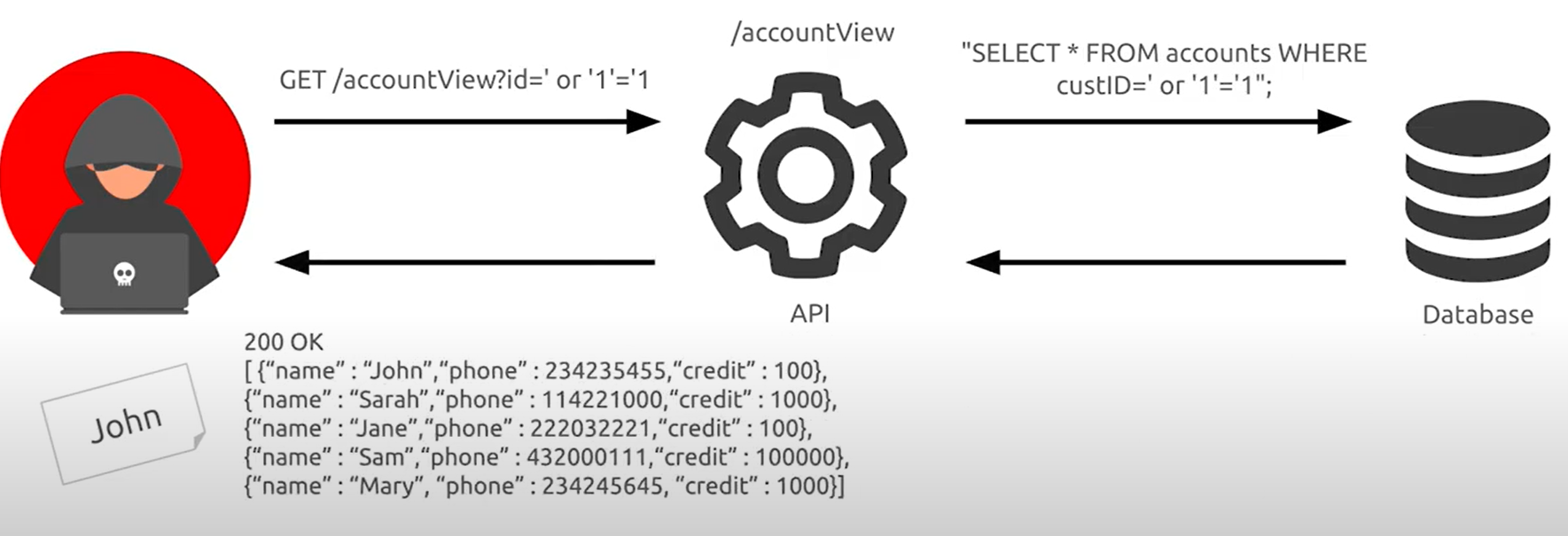
# **API8:2019 – Injection**

Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

## **Example Attack Scenarios**

Attacker attack SQL injection entry point id API GET /accountView?id

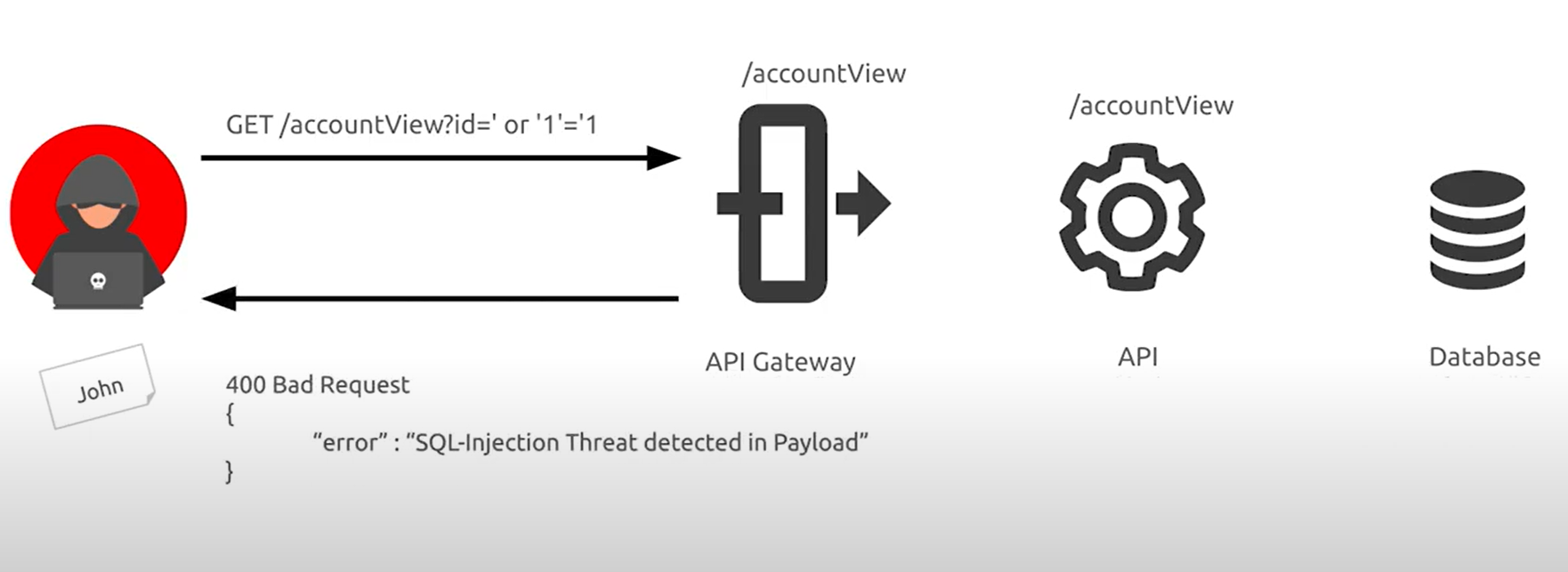




## **How To Prevent**

Preventing injection requires keeping data separate from commands and queries.

* Perform data validation using a single, trustworthy, and actively maintained library.
* Validate, filter, and sanitize all client-provided data, or other data coming from integrated systems.
* Special characters should be escaped using the specific syntax for the target interpreter.
* Prefer a safe API that provides a parameterized interface.
* Always limit the number of returned records to prevent mass disclosure in case of injection.
* Validate incoming data using sufficient filters to only allow valid values for each input parameter.
* Define data types and strict patterns for all string parameters.

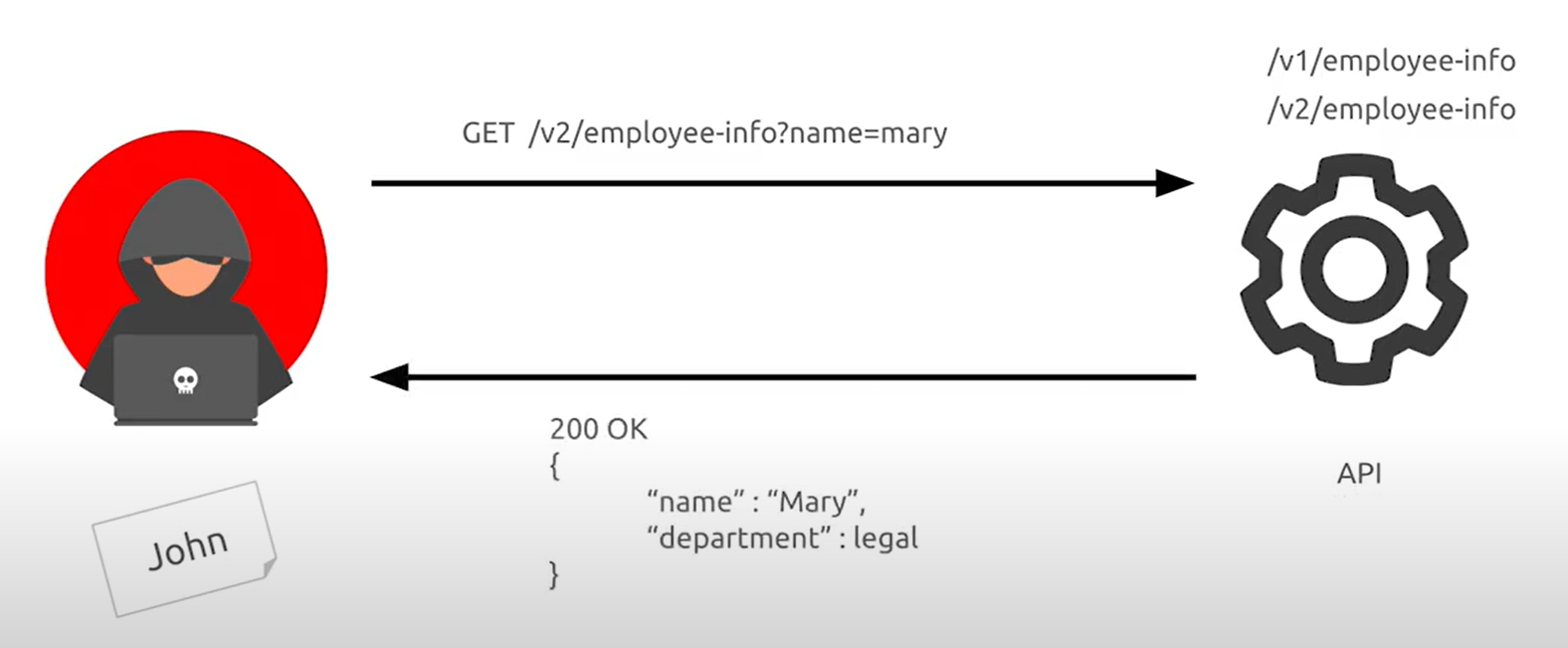


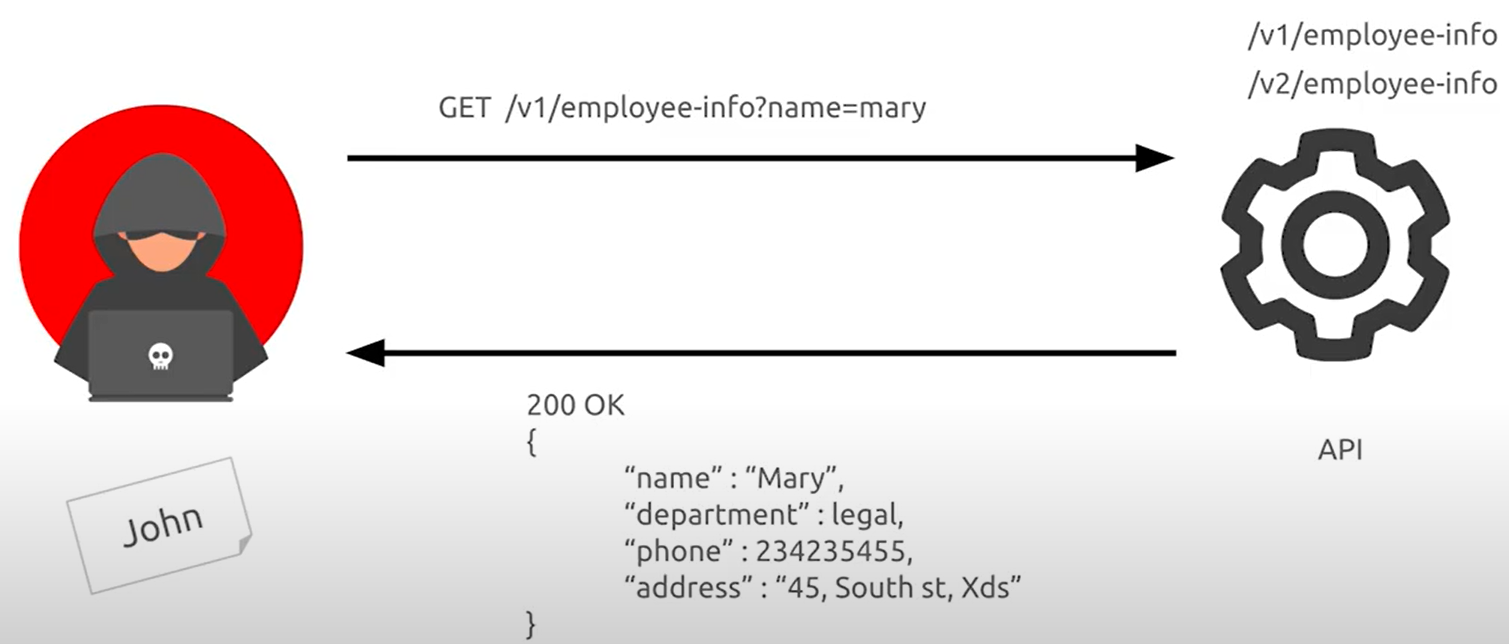
# **API9:2019 – Improper Assets Management**

APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. Proper hosts and deployed API versions inventory also play an important role to mitigate issues such as deprecated API versions and exposed debug endpoints.

## **Example Attack Scenarios**

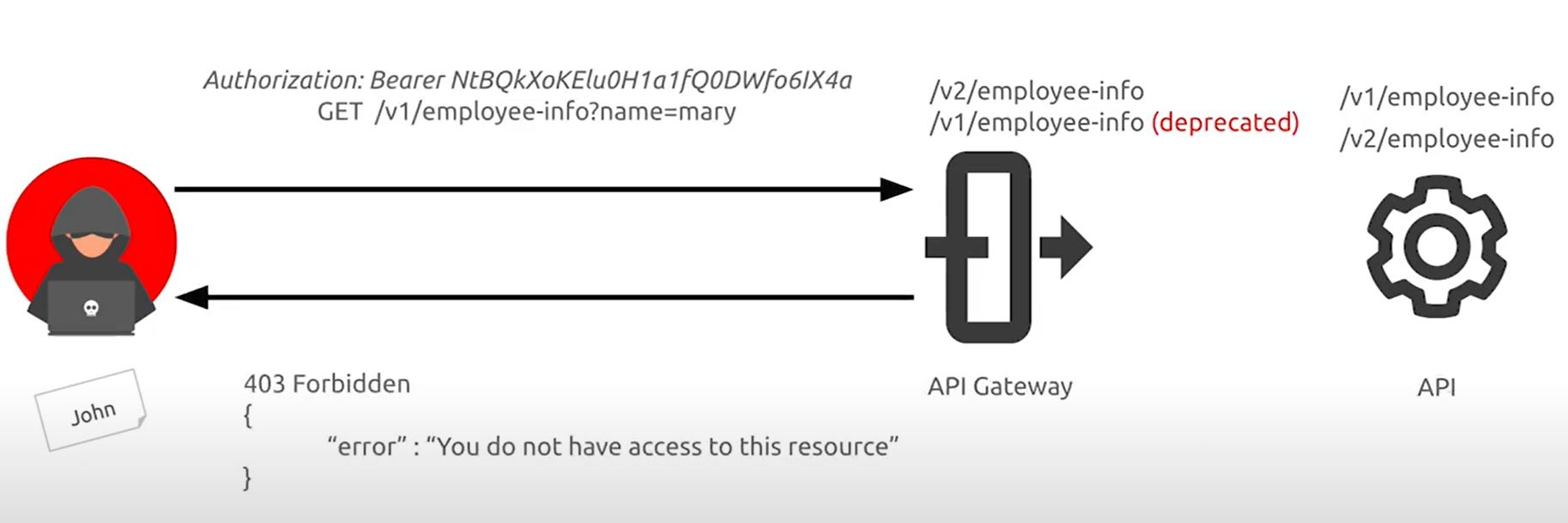
After redesigning their applications, a local search service left an old API version (api.someservice.com/v1) running, unprotected, and with access to the user database. While targeting one of the latest released applications, an attacker found the API address (api.someservice.com/v2). Replacing v2 with v1 in the URL gave the attacker access to the old, unprotected API, exposing the personal identifiable information (PII) of over 100 Million users.

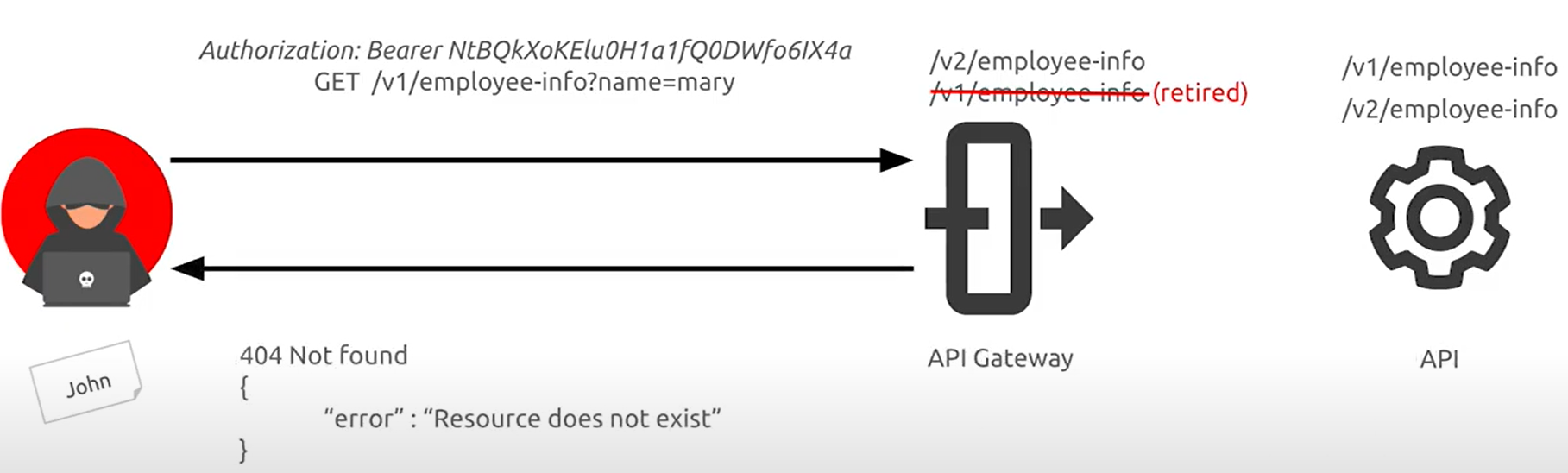




## **How To Prevent**

* Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g., production, staging, test, development), who should have network access to the host (e.g., public, internal, partners) and the API version.
* Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and its sensitivity.
* Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy and endpoints, including their parameters, requests, and responses.
* Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline.
* Make API documentation available to those authorized to use the API.
* Use external protection measures such as API security firewalls for all exposed versions of your APIs, not just for the current production version.
* Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones.
* When newer versions of APIs include security improvements, perform risk analysis to make the decision of the mitigation actions required for the older version: for example, whether it is possible to backport the improvements without breaking API compatibility or you need to take the older version out quickly and force all clients to move to the latest version.





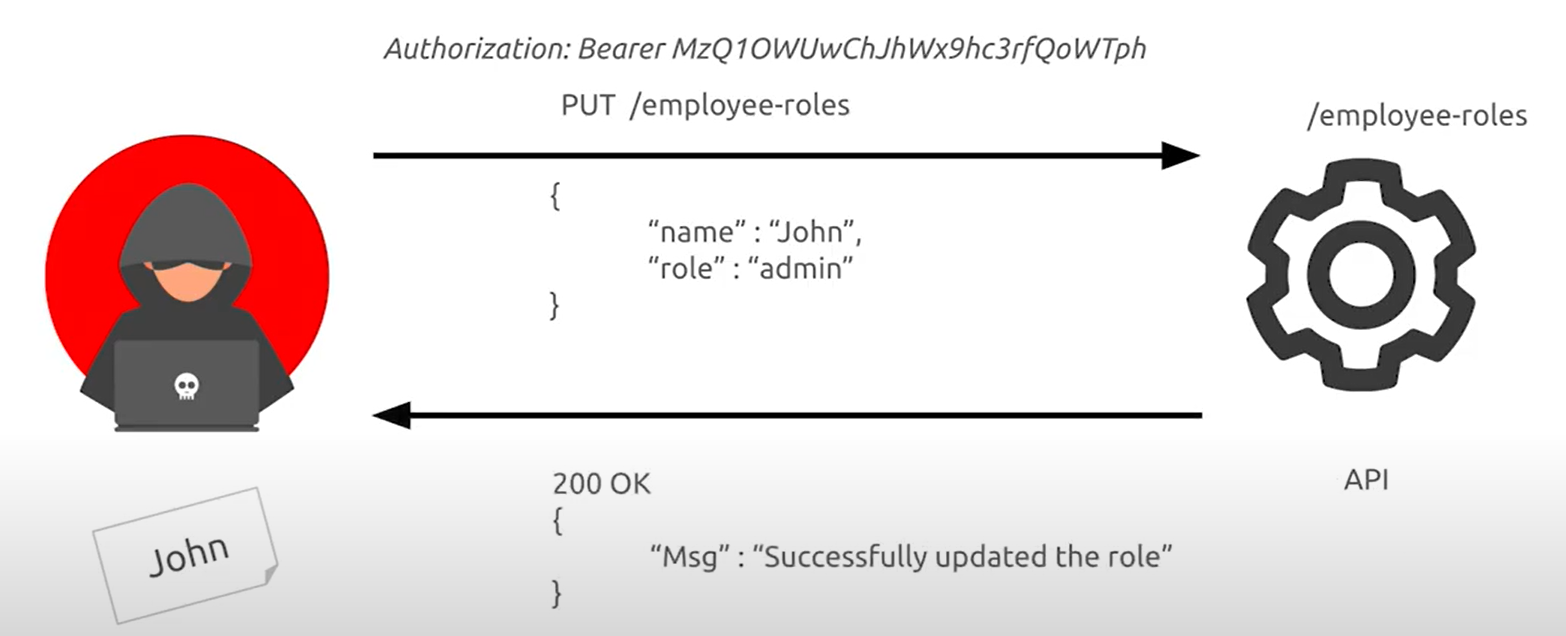
# **API10:2019 – Insufficient Logging & Monitoring**

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems to tamper with, extract, or destroy data. Most breach studies demonstrate the time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

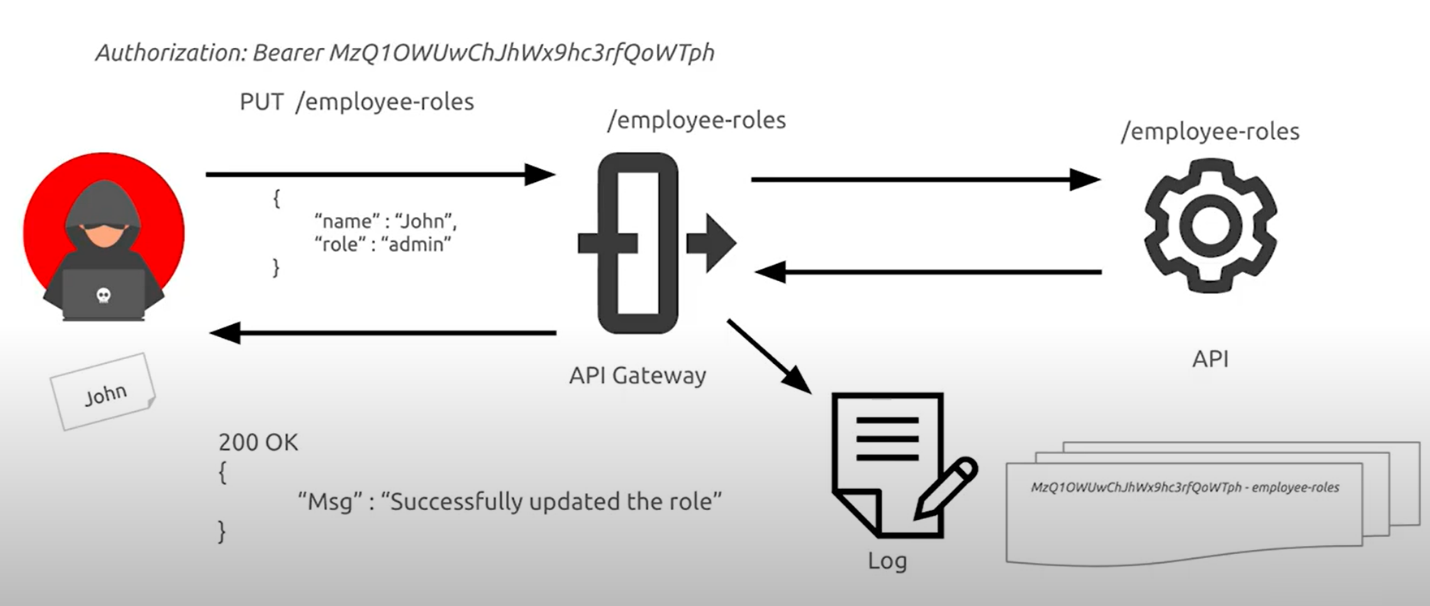
## **Example Attack Scenarios**

### **Scenario 1**

Access keys of an administrative API were leaked on a public repository. The repository owner was notified by email about the potential leak, but took more than 48 hours to act upon the incident, and access keys exposure may have allowed access to sensitive data. Due to insufficient logging, the company is not able to assess what data was accessed by malicious actors.

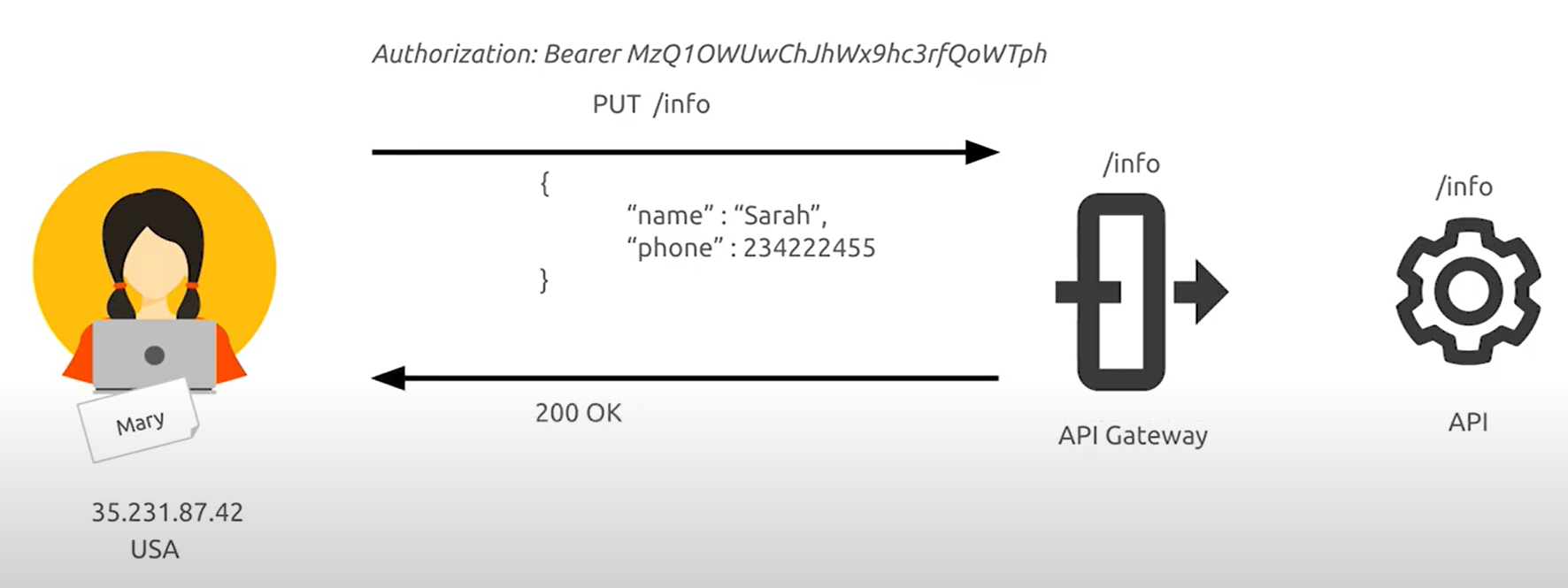


**Solution:**

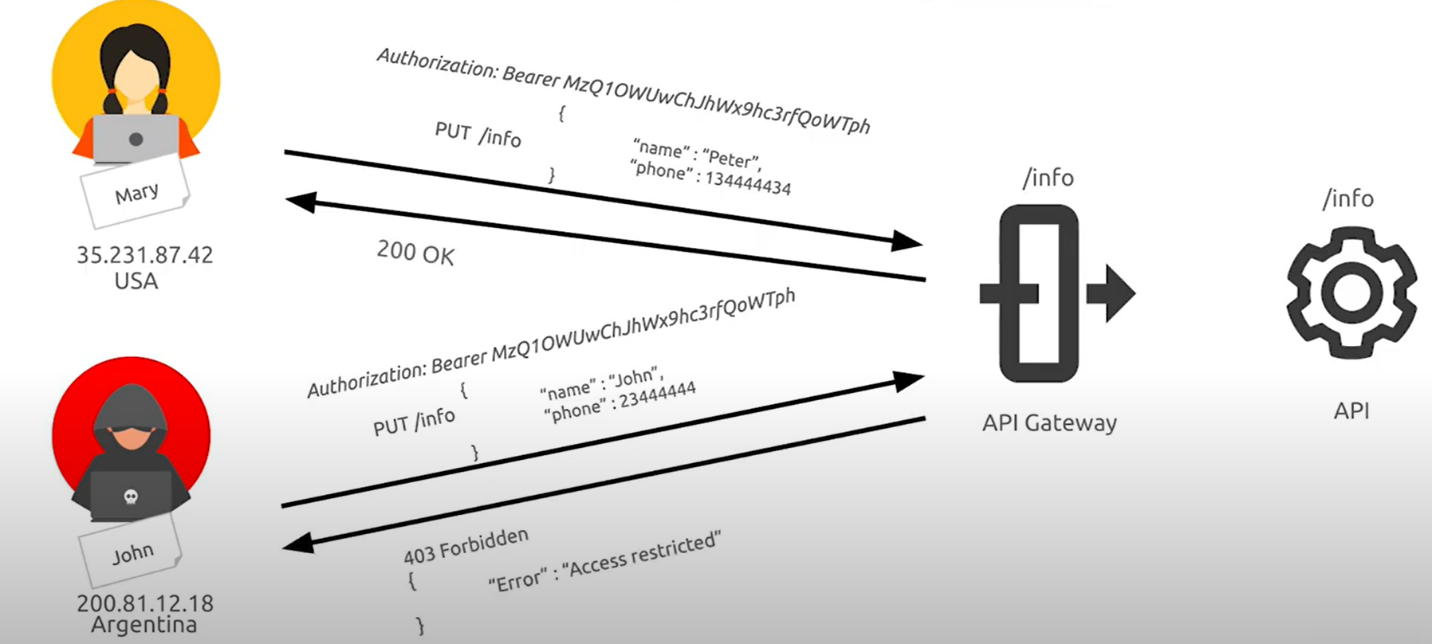


### **Scenario 2**

A video-sharing platform was hit by a “large-scale” credential stuffing attack. Despite failed logins being logged, no alerts were triggered during the timespan of the attack. As a reaction to user complaints, API logs were analyzed and the attack was detected. The company had to make a public announcement asking users to reset their passwords, and report the incident to regulatory authorities.



**Solution:**



## **How To Prevent**

* Log all failed authentication attempts, denied access, and input validation errors.
* Logs should be written using a format suited to be consumed by a log management solution, and should include enough detail to identify the malicious actor.
* Logs should be handled as sensitive data, and their integrity should be guaranteed at rest and transit.
* Configure a monitoring system to continuously monitor the infrastructure, network, and the API functioning.
* Use a Security Information and Event Management (SIEM) system to aggregate and manage logs from all components of the API stack and hosts.
* Configure custom dashboards and alerts, enabling suspicious activities to be detected and responded to earlier.