Developer's guide to OWASP

Top 10 API Security vulnerabilities & MITRE ATT&CK framework relation

Go Lang Edition

prancer

Build Attack Ready Cloud

Authors

Farshid Mahdavipour

Kumar Chandramoulie

Joe Vadakkan

Contributors

Prancer Threat Research Team

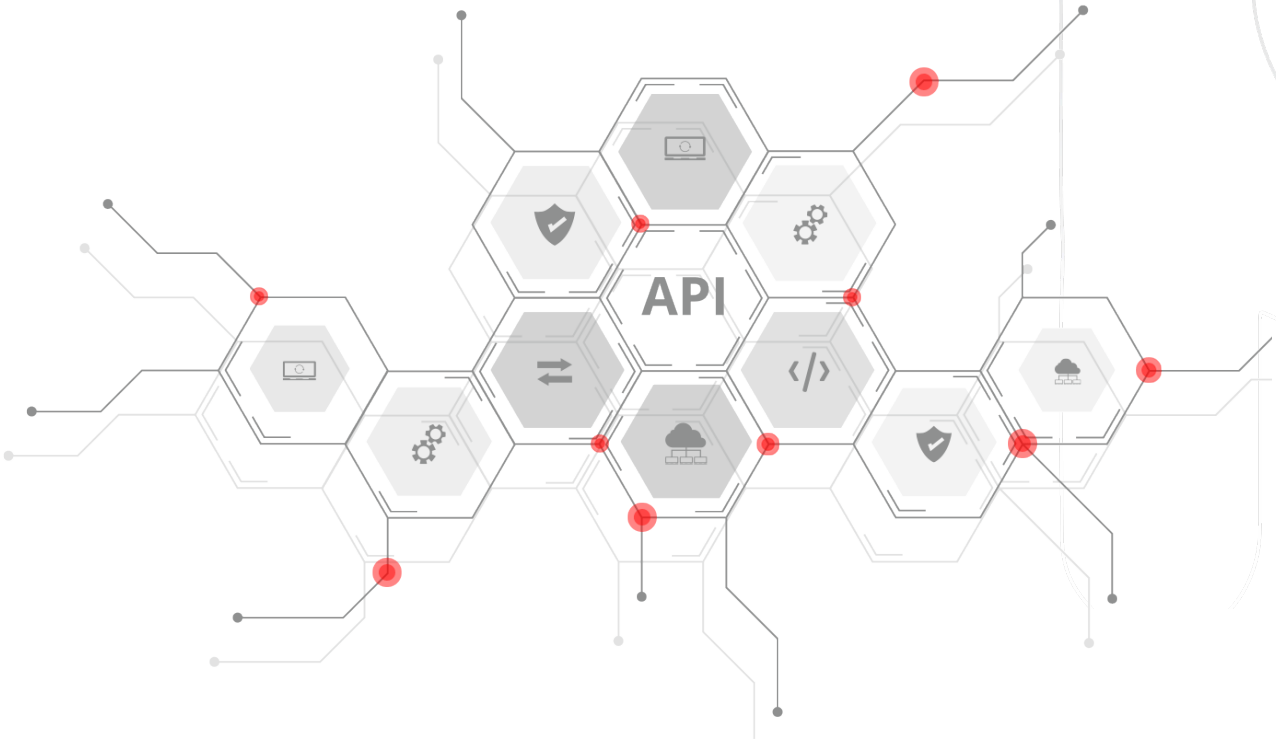
Aakhash Ganesh, Threat Developer

Akshat Verma, Threat Developer

z

Introduction

The OWASP API Security Top 10 is a list of the most common and critical risks that organizations face when developing and exposing APIs (Application Programming Interfaces). APIs allow different systems and applications to communicate with each other, and are often used to expose data and functionality to external parties. However, exposing APIs can also introduce a variety of security risks if not properly secured. The OWASP API Security Top 10 aims to provide guidance on the most important security risks to consider when developing and exposing APIs.



1. Broken Object Level Authorization:

This refers to the risk of improper authorization controls, where APIs may allow unauthorized access to sensitive data or functionality.

2. Broken User Authentication:

This refers to the risk of weak or inadequate authentication controls, which can allow attackers to gain unauthorized access to APIs.

3. Excessive Data Exposure:

This refers to the risk of exposing sensitive data through APIs, either intentionally or unintentionally.

4. Lack of Resources and Rate Limiting:

This refers to the risk of APIs being overwhelmed or exhausted by excessive requests, which can lead to denial of service attacks.

5. Broken Function Level Authorization:

This refers to the risk of improper authorization controls at the function level, where APIs may allow unauthorized access to sensitive functionality.

6. Mass Assignment:

This refers to the risk of allowing untrusted parties to set values for sensitive fields, which can lead to unauthorized access or manipulation of data.

7. Security Misconfiguration:

This refers to the risk of APIs being improperly configured, which can lead to vulnerabilities being exposed.

8. Injection:

This refers to the risk of injecting malicious code into APIs, which can lead to unauthorized access or manipulation of data.

9. Improper Asset Management:

This refers to the risk of failing to properly manage APIs and the data and functionality they expose, which can lead to vulnerabilities being introduced.

10. Insufficient Logging and Monitoring:

This refers to the risk of failing to properly log and monitor API activity, which can make it difficult to detect and respond to security incidents.

MITRE ATT&CK framework

the OWASP API Security Top 10 can be mapped to the following tactics and techniques:

1. Broken Object Level Authorization:

Tactic: Privilege Escalation;
Techniques: Exploitation of Uncontrolled Linkage to a Third-party Domain, Uncontrolled Search Path Element

2. Broken Authentication:

Tactic: Initial Access;
Techniques: Brute Force, Credential Dumping

3. Excessive Data Exposure:

Tactic: Discovery;
Techniques: Data from Information Repositories

4. Lack of Resources and Rate Limiting:

Tactic: Denial of Service;
Techniques: Flooding

5. Broken Function Level Authorization:

Tactic: Privilege Escalation;
Techniques: Exploitation of Uncontrolled Linkage to a Third-party Domain, Uncontrolled Search Path Element

6. Mass Assignment:

Tactic: Privilege Escalation;
Techniques: Exploitation of Uncontrolled Linkage to a Third-party Domain, Uncontrolled Search Path Element

7. Security Misconfiguration:

Tactic: Initial Access;
Techniques: Peripheral Device Discovery, System Information Discovery

8. Injection:

Tactic: Execution;
Techniques: Command Injection, SQL Injection

9. Improper Asset Management:

Tactic: Defense Evasion;
Techniques: Disabling Security Tools, Modify Registry

10. Insufficient Logging and Monitoring:

Tactic: Defense Evasion;
Techniques: Disabling Security Tools, Modify Registry

API security is of utmost importance as it ensures the protection of sensitive data and the integrity of systems when utilizing APIs. APIs are often used to connect different systems and applications, making them a common entry point for attackers. To ensure the security of APIs, it is essential to follow industry best practices and guidelines. One of the most widely recognized and respected sets of guidelines for API security is the OWASP Top 10 API Security Project. This project provides a list of the top 10 most critical security risks for APIs and recommendations for mitigating them. By following these recommendations, organizations can effectively protect against their APIs' most common and severe security threats. The OWASP Top 10 API Security Project is a valuable resource for any organization that utilizes APIs and wants to ensure the security of their systems and sensitive data.

01| Broken Object Level Authorization

Introduction

Broken Object Level Authorization refers to the risk of improper authorization controls in APIs, where API calls may allow unauthorized access to sensitive data or functionality. This can occur when API calls do not properly validate the permissions of the caller or when permissions are not correctly enforced on the server side.

Risks

Some common risks associated with Broken Object Level Authorization include:

- Sensitive data being accessed or modified by unauthorized parties
- Unauthorized access to sensitive functionality, such as the ability to delete or modify data
- Elevation of privileges by unauthorized parties

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts API calls and modifies the permissions of the caller to gain access to sensitive data or functionality
- An attacker uses a compromised account with higher permissions to access sensitive data or functionality
- An attacker exploits a vulnerability in the API to bypass authorization checks

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func deleteData(w http.ResponseWriter, r *http.Request) {
  // Get the user's ID from the request
  userID := r.Header.Get("X-User-ID")

  // Delete the data from the database
  err := database.DeleteData(userID)
  if err != nil {
    http.Error(w, "Error deleting data", http.StatusInternalServerError)
    return
  }

  // Return a success message to the user
  json.NewEncoder(w).Encode("Data deleted successfully")
}
```

In this example, the API call retrieves data from a database based on the user›s ID, which is passed in the request header. However, there is no validation or authorization check to ensure that the user is authorized to access the data. An attacker could exploit this vulnerability by intercepting the API call and modifying the user ID to access data that they should not have access to.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -H "X-User-ID: attacker_user_id" -X DELETE http://api.example.com/deletedata
```

In this example, the attacker is using curl to send a DELETE request to the API with a modified user ID in the request header. If the API is vulnerable to Broken Function Level Authorization, the attacker may be able to delete data that they should not have access to.

MITRE ATT&CK framework reference

Broken Function Level Authorization can be mapped to the Tactic: Privilege Escalation and the Techniques: Exploitation of Uncontrolled Linkage to a Third-party Domain, Uncontrolled Search Path Element in the MITRE ATT&CK framework. These techniques involve exploiting vulnerabilities in authorization controls to gain access to resources or functionality that the attacker should not have access to.

Mitigation

For mitigating Broken Object Level Authorization in cloud API applications, the following steps can be taken:

1. Use proper access controls: Implement role-based access controls and make sure that objects are only accessible by authorized users.
2. Validate user inputs: Ensure that all user inputs are validated and sanitized to prevent unauthorized access to objects.
3. Use encryption: Use encryption to protect sensitive data transmitted over the network and stored in the cloud.
4. Monitor API activity: Monitor API activity for unusual or suspicious behavior and implement proper logging and auditing mechanisms.
5. Implement rate limiting: Implement rate limiting to prevent brute-force attacks and limit the impact of DDoS attacks.
6. Use API keys: Use API keys to authenticate API requests and ensure that only authorized requests are processed.
7. Use API gateways: Use API gateways to control access to your API, enforce security policies, and monitor API usage.
8. Use OAuth2: Use OAuth2 for authentication and authorization to provide secure access to APIs.
9. Use API documentation: Use API documentation to provide clear instructions on how to use the API securely and prevent common security issues.
10. Stay up-to-date: Stay up-to-date with the latest security patches and updates for your cloud environment and API.

02| Object Level Authentication:

Introduction

Broken Authentication refers to the risk of weak or inadequate authentication controls in APIs, which can allow attackers to gain unauthorized access to the API. This can occur when the API uses weak or easily guessable passwords, fails to properly secure authentication tokens, or does not properly validate the authenticity of authentication credentials.

Risks

Some common risks associated with Broken Authentication include:

- Unauthorized access to sensitive data or functionality
- Compromise of user accounts
- Elevation of privileges by unauthorized parties

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts an API call and captures an authentication token, then uses the token to access the API as the authenticated user
- An attacker guesses or cracks a weak password to gain access to an API
- An attacker exploits a vulnerability in the API's authentication process to bypass authentication checks

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func login(w http.ResponseWriter, r *http.Request) {
  // Get the username and password from the request
  username := r.FormValue("username")
  password := r.FormValue("password")

  // Check if the username and password are correct
  if database.CheckCredentials(username, password) {
    // Generate an authentication token
    token, err := generateToken(username)
    if err != nil {
      http.Error(w, "Error generating token", http.StatusInternalServerError)
      return
    }

    // Return the token to the user
    json.NewEncoder(w).Encode(token)
  } else {
    http.Error(w, "Invalid username or password", http.StatusUnauthorized)
  }
}
```

In this example, the API call processes a login request by checking the provided username and password against a database. However, there are several vulnerabilities in this implementation: the password is transmitted in plaintext, there is no rate limiting to prevent brute force attacks, and there is no protection against session hijacking (e.g., by using secure cookies or rotating tokens). An attacker could exploit these vulnerabilities to gain unauthorized access to the API.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -d "username=attacker&password=attacker_password" http://api.example.com/login
```

In this example, the attacker is using curl to send a login request with a malicious username and password. If the API is vulnerable to Broken Authentication, the attacker may be able to gain access to the API and potentially compromise user accounts.

MITRE ATT&CK framework reference

Broken Authentication can be mapped to the Tactic: Initial Access and the Techniques: Brute Force, Credential Dumping in the **MITRE ATT&CK** framework. These techniques involve exploiting vulnerabilities in authentication controls to gain unauthorized access to a system or network.

Mitigation

The following are some mitigation techniques for Object Level Authentication in OWASP API Security 2019:

1. Access Tokens: Use access tokens to authenticate API requests and control access to resources.
2. Role-Based Access Control (RBAC): Implement role-based access control to restrict access to resources based on the user's role.
3. Ownership Verification: Verify ownership of objects before allowing access or modifications to ensure that the correct user has access to the correct data.
4. Input Validation: Validate all user inputs to ensure that they are authorized to access or modify the requested resources.
5. Encryption: Encrypt sensitive data to protect against unauthorized access in case of a breach.
6. Logging and Monitoring: Log API requests and responses and monitor for unusual activity.
7. Authorization Checks: Perform authorization checks before processing API requests to ensure that the user has the necessary permissions to access or modify the requested resources.
8. Least Privilege Principle: Adhere to the principle of least privilege, which states that a user should have the minimum level of access necessary to perform their job function.
9. Use Security Frameworks: Use security frameworks such as OWASP API Security Top 10 or SANS Top 25 to guide security efforts and reduce the risk of object level authentication vulnerabilities.

03| Excessive Data Exposure

Introduction

Excessive Data Exposure refers to the risk of exposing sensitive data through APIs, either intentionally or unintentionally. This can occur when APIs allow access to more data than is necessary, or when data is not properly protected or redacted when returned to the caller.

Risks

Some common risks associated with Excessive Data Exposure include:

- Sensitive data being accessed or compromised by unauthorized parties
- Loss of confidentiality or privacy for users whose data is exposed
- Reputational damage for the organization due to data breaches

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts an API call and modifies the request to access more data than they should have access to
- An attacker exploits a vulnerability in the API to access sensitive data without proper authorization
- An attacker uses an API to retrieve large amounts of data, potentially overwhelming the API and causing a denial of service

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func getUserData(w http.ResponseWriter, r *http.Request) {
  // Get the user's ID from the request
  userID := r.Header.Get("X-User-ID")

  // Retrieve the user's data from the database
  user, err := database.GetUser(userID)
  if err != nil {
    http.Error(w, "Error retrieving user data", http.StatusInternalServerError)
    return
  }

  // Return the user's data to the caller
  json.NewEncoder(w).Encode(user)
}
```

In this example, the API call retrieves a user's data from a database based on the user's ID, which is passed in the request header. However, there is no validation or authorization check to ensure that the caller is authorized to access the user's data, and the entire user record is returned to the caller without any redaction. An attacker could exploit this vulnerability by intercepting the API call and accessing sensitive data that they should not have access to.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -H "X-User-ID: attacker_user_id" http://api.example.com/getuserdata
```

In this example, the attacker is using curl to send an API request with a modified user ID in the request header. If the API is vulnerable to **Excessive Data Exposure**, the attacker may be able to access sensitive data belonging to the user with the specified ID.

MITRE ATT&CK framework reference

Excessive Data Exposure can be mapped to the Tactic: Discovery and the Technique: Data from Information Repositories in the **MITRE ATT&CK** framework. This technique involves accessing data from information storage and management systems, such as databases or APIs.

Mitigation

For Excessive Data Exposure in API applications, the following mitigation techniques are recommended:

1. Implement proper access controls based on the principle of least privilege, only returning the minimum data necessary for each request.
2. Use encryption for sensitive data in transit and at rest.
3. Validate and sanitize inputs to prevent injection attacks.
4. Properly implement error handling to avoid exposing sensitive information in error messages.
5. Use secure coding practices and input validation to prevent Cross-Site Scripting (XSS) attacks.
6. Monitor and log API activity for suspicious behavior.
7. Regularly perform security testing, including penetration testing.
8. Keep the API and its dependencies up-to-date with the latest security patches.
9. Consider using masking or redaction techniques for sensitive data.
10. Educate developers and stakeholders on the risks of excessive data exposure and the importance of following secure data handling practices.

04| Lack of Resources and Rate Limiting

Introduction

Lack of Resources and Rate Limiting refers to the risk of APIs being overwhelmed or exhausted by excessive requests, which can lead to denial of service attacks. This can occur when APIs do not properly handle high volumes of traffic, or do not implement sufficient rate limiting to prevent excessive requests from a single source.

Risks

Some common risks associated with **Lack of Resources and Rate Limiting** include:

- Denial of service for legitimate users of the API
- Loss of availability for the API and the systems and services it supports
- Reputational damage for the organization due to service disruptions

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker uses an API to send a large number of requests in a short period of time, overwhelming the API and causing it to become unavailable
- An attacker exploits a vulnerability in the API to send a high volume of requests, potentially causing a denial of service
- An attacker coordinates with other attackers to launch a distributed denial of service (DDoS) attack against an API

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func getData(w http.ResponseWriter, r *http.Request) {
  // Retrieve the data from the database
  data, err := database.GetData()
  if err != nil {
    http.Error(w, "Error retrieving data", http.StatusInternalServerError)
    return
  }

  // Return the data to the user
  json.NewEncoder(w).Encode(data)
```

In this example, the API call retrieves data from a database and returns it to the caller. However, there is no rate limiting in place to prevent excessive requests from a single source, and the API does not properly handle high volumes of traffic. An attacker could exploit this vulnerability by sending a large number of requests to the API in a short period of time, potentially causing a denial of service.

Sample Attack

A sample attack payload using the curl command might look like this:

```
while true; do curl http://api.example.com/getdata; done
```

In this example, the attacker is using a loop to send an endless stream of requests to the API using curl. If the API is vulnerable to Lack of Resources and Rate Limiting, this could potentially cause a denial of service.

MITRE ATT&CK framework reference

Lack of Resources and Rate Limiting can be mapped to the Tactic: Denial of Service and the Technique: Flooding in the MITRE ATT&CK framework. This technique involves overwhelming a system or network with excessive requests, potentially causing a denial of service.

Mitigation

To mitigate the risks of lack of resources and rate limiting in OWASP API Security 2019, organizations can use the following techniques:

1. Resource Pooling: Implement resource pooling to manage API resources and prevent over-utilization.
2. Cache Resources: Cache resources in memory to reduce the number of database queries and prevent resource exhaustion.
3. Load Balancing: Use load balancing to distribute API requests evenly across multiple servers to prevent resource exhaustion.
4. Rate Limiting: Implement rate limiting to limit the number of API requests that a user can make within a given time period.
5. Queuing: Use queuing to store API requests when resources are exhausted and process them when resources become available.
6. Circuit Breaking: Implement circuit breaking to prevent a single API request from consuming all available resources and causing a system-wide failure.
7. Monitoring and Logging: Monitor API performance and resource utilization, and log API requests and responses to identify potential resource exhaustion issues.
8. Use Security Frameworks: Use security frameworks such as OWASP API Security Top 10 or SANS Top 25 to guide security efforts and reduce the risk of resource exhaustion and rate limiting vulnerabilities.
9. Auto Scaling: Use auto scaling to dynamically add or remove resources based on the current load to prevent resource exhaustion.

05| Broken Function Level Authorization

Introduction

Broken Function Level Authorization refers to the risk of improper authorization controls in APIs, where API calls may allow unauthorized access to sensitive functionality. This can occur when API calls do not properly validate the permissions of the caller, or when permissions are not correctly enforced on the server side.

Risks

Some common risks associated with **Broken Function Level Authorization** include:

- Sensitive functionality being accessed or exploited by unauthorized parties
- Unauthorized modification or deletion of data
- Elevation of privileges by unauthorized parties

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts an API call and modifies the permissions of the caller to gain access to sensitive functionality
- An attacker uses a compromised account with higher permissions to access sensitive functionality
- An attacker exploits a vulnerability in the API to bypass authorization checks and access sensitive functionality

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func deleteData(w http.ResponseWriter, r *http.Request) {
  // Get the user's ID from the request
  userID := r.Header.Get("X-User-ID")

  // Delete the data from the database
  err := database.DeleteData(userID)
  if err != nil {
    http.Error(w, "Error deleting data", http.StatusInternalServerError)
    return
  }

  // Return a success message to the user
  json.NewEncoder(w).Encode("Data deleted successfully")
```

In this example, the API call allows a user to delete data from a database based on their ID, which is passed in the request header. However, there is no validation or authorization check to ensure that the user is authorized to delete the data, and any user with a valid ID could potentially delete data belonging to other users. An attacker could exploit this vulnerability by intercepting the API call and modifying the user ID to delete data that they should not have access to.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -H "X-User-ID: attacker_user_id" -X DELETE http://api.example.com/deletedata
```

In this example, the attacker is using curl to send a DELETE request to the API with a modified user ID in the request header. If the API is vulnerable to Broken Function Level Authorization, the attacker may be able to delete data that they should not have access to.

MITRE ATT&CK framework reference

Broken Function Level Authorization can be mapped to the Tactic: Privilege Escalation and the Techniques: Exploitation of Uncontrolled Linkage to a Third-party Domain, Uncontrolled Search Path Element in the MITRE ATT&CK framework. These techniques involve exploiting vulnerabilities in authorization controls to gain access to resources or functionality that the attacker should not have access to.

Mitigation

For Broken Function Level Authorization in cloud API applications, the following mitigation techniques are recommended:

1. Use a secure authentication mechanism, such as OAuth2 or JWT, to properly identify the user.
2. Verify the user's authorization for each requested action by checking their roles and permissions.
3. Implement access controls based on the principle of least privilege.
4. Use encryption and secure storage for sensitive data.
5. Use secure coding practices and validate user inputs to prevent injection attacks.
6. Implement rate limiting to prevent brute force attacks.
7. Monitor and log API activity for suspicious behavior.
8. Regularly perform security testing, including penetration testing.
9. Implement proper error handling and avoid exposing sensitive information in error messages.
10. Keep the API and its dependencies up-to-date with the latest security patches.

06| Mass Assignment

Introduction

Mass Assignment refers to the risk of insecurely handling user input in APIs, which can allow attackers to modify or manipulate data in unintended ways. This can occur when APIs do not properly validate or sanitize user input, or when APIs allow direct assignment of user input to object properties without proper authorization or validation.

Risks

Some common risks associated with **Mass Assignment** include:

- Unauthorized modification or manipulation of data
- Elevation of privileges by unauthorized parties
- Compromise of user accounts

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts an API call and modifies the request to modify or manipulate data in unintended ways
- An attacker exploits a vulnerability in the API to directly assign user input to object properties, bypassing authorization or validation checks
- An attacker uses an API to send malicious input in an attempt to exploit vulnerabilities or inject malicious code

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
type User struct {
  ID        int    `json:"id"`
  Email     string `json:"email"`
  Password  string `json:"password"`
  FirstName string `json:"first_name"`
  LastName  string `json:"last_name"`
  Role      string `json:"role"`
}

func updateUser(w http.ResponseWriter, r *http.Request) {
  // Get the user's ID from the request
  userID := r.Header.Get("X-User-ID")

  // Get the updated user data from the request body
  var user User
  err := json.NewDecoder(r.Body).Decode(&user)
  if err != nil {
    http.Error(w, "Error decoding request body", http.StatusBadRequest)
    return
  }

  // Update the user in the database
  err = database.UpdateUser(userID, user)
  if err != nil {
    http.Error(w, "Error updating user", http.StatusInternalServerError)
    return
  }

  // Return a success message to the user
  json.NewEncoder(w).Encode("User updated successfully")
}
```

In this example, the API call allows a user to update their own data in a database. However, the API directly assigns the user input from the request body to the properties of a User struct without any validation or authorization checks. An attacker could exploit this vulnerability by intercepting the API call and modifying the request body to update the user's data in unintended ways, such as changing the user's role or password.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -H "X-User-ID: attacker_user_id" -d '{"email":"attacker@example.com",
"password":"attacker_password","first_name":"Attacker","last_name":"Attacker",
"role":"admin"}' -X PUT http://api.example.com/updateuser
```

In this example, the attacker is using curl to send a PUT request to the API with a modified user ID in the request header and a modified request body that includes a new email, password, and role for the user. If the API is vulnerable to Mass Assignment, the attacker may be able to update the user's data in unintended ways.

MITRE ATT&CK framework reference

Mass Assignment can be mapped to the Tactic: Privilege Escalation and the Techniques: Exploitation of Uncontrolled Linkage to a Third-party.

MITIGATION

There are several mitigation techniques to prevent mass assignment vulnerability in APIs:

1. Input Validation: Ensure that only valid parameters are accepted by the API. Use libraries to validate user input.
2. Whitelisting: Only allow specific parameters to be updated and ignore all others.
3. Parameter Mapping: Map incoming parameters to a known set of parameters, ignoring any unknown parameters.
4. Role-Based Access Control (RBAC): Assign roles to users and only allow the appropriate parameters to be updated based on the user's role.
5. Encrypt Sensitive Data: Encrypt sensitive data on the client-side before sending it to the API.
6. Use Access Tokens: Use access tokens to authenticate API requests and limit the scope of what data can be accessed or modified.
7. Keep it Simple: Minimize the number of parameters that are accepted by the API to reduce the attack surface.
8. Logging and Monitoring: Log API requests and responses and monitor for unusual activity.

07| Security Misconfiguration

Introduction

Security Misconfiguration refers to the risk of APIs being improperly configured, which can lead to vulnerabilities or weaknesses in their security. This can occur when APIs are not properly secured during development or deployment, or when they are not properly maintained and kept up to date with security patches and updates.

Risks

Some common risks associated **Security Misconfiguration** include:

- Unauthorized access to sensitive data or functionality
- Compromise of user accounts
- Reputational damage for the organization due to data breaches or service disruptions

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker exploits a known vulnerability in an API due to a lack of proper patches or updates
- An attacker gains access to an API through default or easily guessable credentials
- An attacker discovers and exploits a misconfigured or poorly secured API endpoint

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func getData(w http.ResponseWriter, r *http.Request) {
  // Retrieve the data from the database
  data, err := database.GetData()
  if err != nil {
    http.Error(w, "Error retrieving data", http.StatusInternalServerError)
    return
  }

  // Return the data to the user
  json.NewEncoder(w).Encode(data)
}
```

In this example, the API call retrieves data from a database and returns it to the caller. However, there is no authentication or authorization in place to ensure that only authorized users can access the data. An attacker could exploit this vulnerability by simply making an API request to the endpoint and accessing the data without proper credentials.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl http://api.example.com/getdata
```

In this example, the attacker is using curl to send a request to the API without any authentication or authorization. If the API is vulnerable to Security Misconfiguration, the attacker may be able to access the data without proper credentials.

MITRE ATT&CK framework reference

Security Misconfiguration can be mapped to the Tactic: Initial Access and the Techniques: Obtain Credentials, Exploit Public-Facing Application in the MITRE ATT&CK framework. These techniques involve exploiting vulnerabilities or weaknesses in systems or applications to gain unauthorized access.

Mitigation

For Security Misconfiguration in API applications, the following mitigation techniques are recommended:

1. Follow secure configuration guidelines for the operating system, web server, and framework.
2. Remove unused features and configurations to reduce the attack surface.
3. Regularly update and patch all software, including dependencies.
4. Keep sensitive information, such as passwords and keys, secure and out of source control.
5. Use logging and monitoring to detect and respond to security incidents.
6. Use environment-specific configurations to prevent sensitive information from being leaked to unintended environments.
7. Implement proper access controls and authentication mechanisms.
8. Use file system permissions to prevent unauthorized access to sensitive files.
9. Use HTTPS to encrypt sensitive data in transit.
10. Regularly perform security testing, including penetration testing, to identify and remediate misconfigurations.

08| Injection

Introduction

Injection refers to the risk of attackers injecting malicious code or commands into APIs, which can allow them to exploit vulnerabilities or manipulate data in unintended ways. This can occur when APIs do not properly validate or sanitize user input, or when APIs do not properly handle external data sources or systems.

Risks

Some common risks associated Injection include:

- Compromise of user accounts or data
- Unauthorized access to sensitive data or functionality
- Reputational damage for the organization due to data breaches or service disruptions

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker intercepts an API call and injects malicious code or commands into the request
- An attacker exploits a vulnerability in the API to inject malicious code or commands into the response
- An attacker uses an API to send malicious input in an attempt to exploit vulnerabilities or inject malicious code

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func getData(w http.ResponseWriter, r *http.Request) {
  // Get the search term from the request
  searchTerm := r.URL.Query().Get("term")

  // Retrieve the data from the database
  data, err := database.SearchData(searchTerm)
  if err != nil {
    http.Error(w, "Error searching data", http.StatusInternalServerError)
    return
  }

  // Return the data to the user
  json.NewEncoder(w).Encode(data)
}
```

In this example, the API call allows users to search for data in a database based on a search term passed in the request. However, the API does not properly validate or sanitize the search term, allowing an attacker to inject malicious code or commands into the request. For example, an attacker could send a request with a search term such as **"; DROP TABLE users;"** which could potentially delete the entire users table in the database.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl http://api.example.com/getdata?term=%22%3B%20DROP%20TABLE%20users%3B%22
```

In this example, the attacker is using curl to send a request to the API with a malicious search term that includes a command to drop the users table in the database. If the API is vulnerable to Injection, the attacker may be able to execute the command and delete the table.

MITRE ATT&CK framework reference

Injection can be mapped to the Tactic: Execution and the Techniques: Command-Line Interface, Remote Command Execution in the MITRE ATT&CK framework. These techniques involve injecting malicious code or commands into systems or applications to execute them.

Mitigation

To mitigate the risk of injection attacks in OWASP API Security 2019, organizations can use the following techniques:

1. Input Validation: Validate all user inputs and reject any malicious inputs that contain special characters or unexpected data.
2. Use Parameterized Queries: Use parameterized queries instead of string concatenation to prevent SQL injection attacks.
3. Escaping: Escape all user inputs before using them in API queries to prevent injection attacks.
4. Use Prepared Statements: Use prepared statements instead of string concatenation to prevent SQL injection attacks.
5. Whitelisting: Use whitelisting to allow only approved characters in user inputs and reject all other characters.
6. Logging and Monitoring: Log API requests and responses and monitor for unusual activity to detect and respond to potential injection attacks.
7. Use Security Frameworks: Use security frameworks such as OWASP API Security Top 10 or SANS Top 25 to guide security efforts and reduce the risk of injection attacks.
8. Use Encryption: Encrypt sensitive data to protect against data exposure in case of an injection attack.
9. Use Access Tokens: Use access tokens to authenticate API requests and control access to resources.

09| Improper Asset Management

Introduction

Improper Asset Management refers to the risk of APIs not properly managing or securing their assets, which can lead to vulnerabilities or weaknesses in their security. This can occur when APIs do not properly track or secure their assets, such as secrets, keys, or credentials, or when they do not properly manage their dependencies or third-party libraries.

Risks

Some common risks associated with **Improper Asset Management** include:

- Unauthorized access to sensitive data or functionality
- Compromise of user accounts
- Reputational damage for the organization due to data breaches or service disruptions

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker gains access to sensitive assets, such as secrets or keys, through unsecured storage or poor access controls
- An attacker exploits a vulnerability in a third-party library or dependency used by the API
- An attacker discovers and exploits a misconfigured or poorly secured asset, such as an API endpoint

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
const apiKey = "abc123"

func getData(w http.ResponseWriter, r *http.Request) {
  // Get the API key from the request
  requestKey := r.Header.Get("X-API-Key")

  // Check the API key
  if requestKey != apiKey {
    http.Error(w, "Invalid API key", http.StatusUnauthorized)
    return
  }

  // Retrieve the data from the database
  data, err := database.GetData()
  if err != nil {
    http.Error(w, "Error retrieving data", http.StatusInternalServerError)
    return
  }

  // Return the data to the user
```

In this example, the API uses a hardcoded API key for authentication. However, this key is not properly secured and is easily accessible to anyone with access to the source code. An attacker could exploit this vulnerability by simply copying the key and using it to make unauthorized API requests.

Sample Attack

A sample attack payload using the curl command might look like this:

```
curl -H "X-API-Key: abc123" http://api.example.com/getdata
```

In this example, the attacker is using curl to send a request to the API with the hardcoded API key. If the API is vulnerable to Improper Asset Management, the attacker may be able to access the data without proper credentials.

MITRE ATT&CK framework reference

Improper Asset Management can be mapped to the Tactic: Initial Access and the Techniques: Obtain Credentials, Exploit Public-Facing Application in the MITRE ATT&CK framework. These techniques involve exploiting vulnerabilities or weaknesses in systems or applications to gain unauthorized access.

1. Use secure methods for storing and managing secrets, keys, and credentials, such as using a password manager or a secure storage service.
2. Implement proper access controls and permissions for assets, including rotating keys and credentials regularly and limiting access to sensitive assets to only authorized personnel.
3. Regularly review and update dependencies and third-party libraries to ensure they are secure and up to date.
4. Use security testing tools and techniques, such as static code analysis or penetration testing, to identify and fix vulnerabilities in assets.
5. Have proper logging and monitoring in place to detect and respond to potential asset management issues.
6. Educate and train employees on the importance of proper asset management and how to implement it effectively.

10| Insufficient Logging and Monitoring

Introduction

Insufficient Logging and Monitoring refers to the risk of APIs not having proper logging and monitoring in place to detect and respond to security threats or vulnerabilities. This can occur when APIs do not properly log or monitor events, such as authentication failures or unauthorized access attempts, or when they do not have proper alerts or notifications in place to alert security personnel of potential issues.

Risks

Some common risks associated with Insufficient Logging and Monitoring include:

- Difficulty in detecting and responding to security threats or vulnerabilities in a timely manner
- Difficulty in identifying the root cause of security incidents
- Increased risk of data breaches or service disruptions

Attack Scenarios

Attack scenarios for cloud applications may include:

- An attacker exploits a vulnerability in an API without being detected due to insufficient logging or monitoring
- An attacker gains unauthorized access to an API and is able to perform malicious actions without being detected
- An attacker is able to cover their tracks and evade detection by deleting or tampering with log files

Vulnerable Sample Code

A vulnerable sample of code in Go lang might look like this:

```
func login(w http.ResponseWriter, r *http.Request) {
  // Get the login credentials from the request
  var credentials struct {
    Email    string `json:"email"`
    Password string `json:"password"`
  }
  err := json.NewDecoder(r.Body).Decode(&credentials)
  if err != nil {
    http.Error(w, "Error decoding request body", http.StatusBadRequest)
    return
  }

  // Check the credentials against the database
  user, err := database.GetUser(credentials.Email)
  if err != nil {
    http.Error(w, "Error retrieving user", http.StatusInternalServerError)
    return
  }
  if user.Password != credentials.Password {
   http.Error(w, "Invalid email or password
```

Sample Attack

A sample attack payload using the curl command to exploit an API with **insufficient logging and monitoring** might look like this:

```
curl -X POST -H "Content-Type: application/json" -d '{"email":"attacker@example.com",
"password":"password123"}' http://api.example.com/login
```

In this example, the attacker is using curl to send a request to the API's login endpoint with a valid email and password. However, if the API is vulnerable to Insufficient Logging and Monitoring, the attacker may be able to perform malicious actions without being detected.

MITRE ATT&CK framework reference

Insufficient Logging and Monitoring can be mapped to the Tactic: Defense Evasion and the Techniques: Indicator Removal on Host, Indicator Removal from Tools in the MITRE ATT&CK framework. These techniques involve deleting or tampering with log files or other indicators of compromise in an attempt to evade detection.

Mitigation

To mitigate the risk of insufficient logging and monitoring in OWASP API Security 2019, organizations can use the following techniques:

1. Logging: Log API requests and responses, including the user's IP address, the API endpoint accessed, and the request and response data.
2. Monitoring: Monitor API activity for unusual behavior, such as a high number of requests from a single IP address or unexpected data in API requests.
3. Alerting: Set up alerts to notify administrators of potential security threats, such as failed login attempts or unexpected API activity.
4. Correlation and Analysis: Use correlation and analysis tools to identify patterns in API activity and detect potential security threats.
5. Use Security Information and Event Management (SIEM) Tools: Use SIEM tools to centralize and analyze API logs for security events and threats.
6. Access Control: Implement access controls to restrict access to API logs and monitor API activity.
7. Compliance Requirements: Meet regulatory and compliance requirements for logging and monitoring API activity.
8. Use Security Frameworks: Use security frameworks such as OWASP API Security Top 10 or SANS Top 25 to guide security efforts and reduce the risk of insufficient logging and monitoring vulnerabilities.
9. Regular Review: Regularly review API logs and monitor API activity to identify potential security threats and respond quickly to mitigate the risk.

How Prancer Cloud Security solution can mitigate the risk of vulnerabilities in OWASP API top 10

At Prancer, we are dedicated to providing our clients with the highest level of security for their APIs. Our automated testing process covers the OWASP API Security 10, a widely recognized list of the top 10 API security risks. Our testing methodology is designed to identify potential vulnerabilities and provide actionable recommendations for remediation.

One of the key aspects of our automated testing process is the mapping of common API vulnerabilities to the OWASP API Security 10. This approach ensures that our clients receive a complete and thorough security assessment of their APIs.

Some of the main API vulnerabilities we cover are:

1. Mass Injection (API8:2019 Injection)

2. Attribute Overwrite Vulnerabilities (API6:2019 Mass Assignment)

3. Exploitation of Bearer Tokens for Unauthorized Data Access and Modification (API2:2019 Broken User Authentication)

4. Accepting HTTP Methods That Should Not Be Accepted (API5:2019 Broken Function Level Authorization)

5. Improper Filtering and Validation of JSON Payload Inputs (API1:2019 Broken Object Level Authorization)

10. APIReDos (API4:2019 Lack of Resources & Rate Limiting)

6. Excessive Fingerprinting Headers (API3:2019 Excessive Data Exposure)

7. Missing Security Header (API9:2019 Improper Assets Management)

8. Exposed Debug Endpoints (API7:2019 Security Misconfiguration)

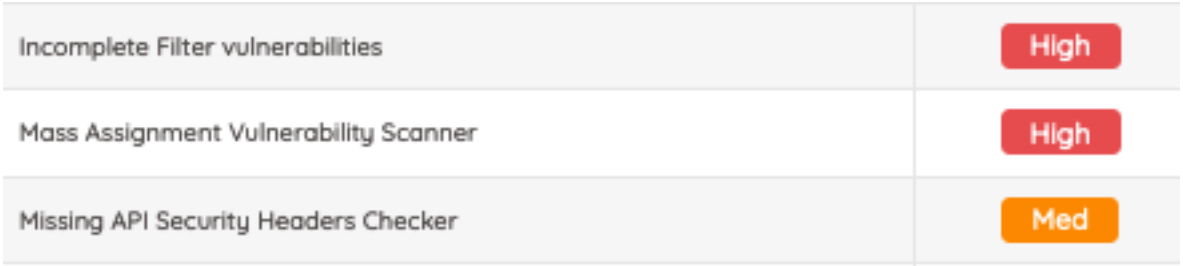
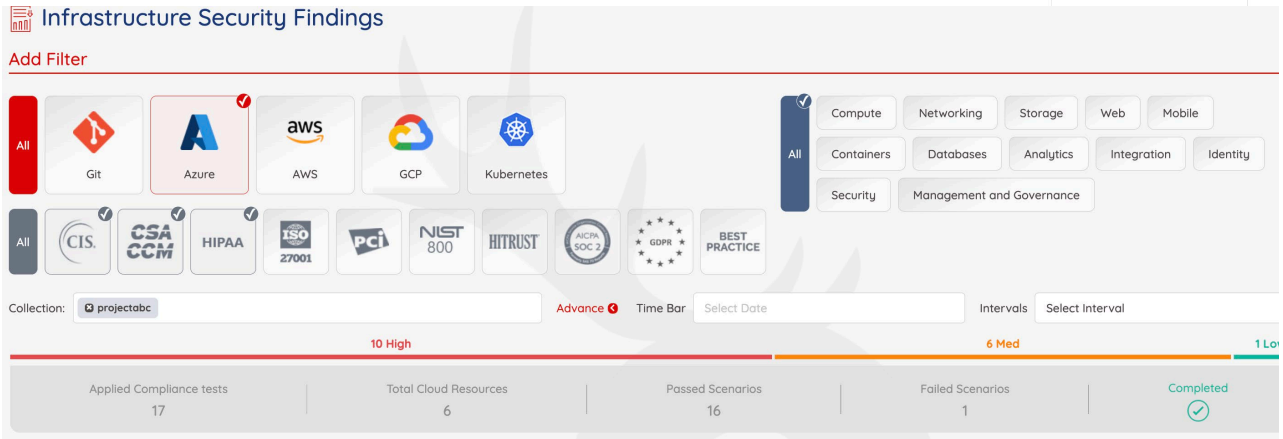


Figure 1 Example API Scan Results

In addition to the common API vulnerabilities mentioned previously. Prancer's testing process also covers a wide range of other security risks. Our automated testing process is designed to identify vulnerabilities that may go unnoticed by manual testing methods, and our expertise in API security allows us to provide clients with the most comprehensive security assessment available.

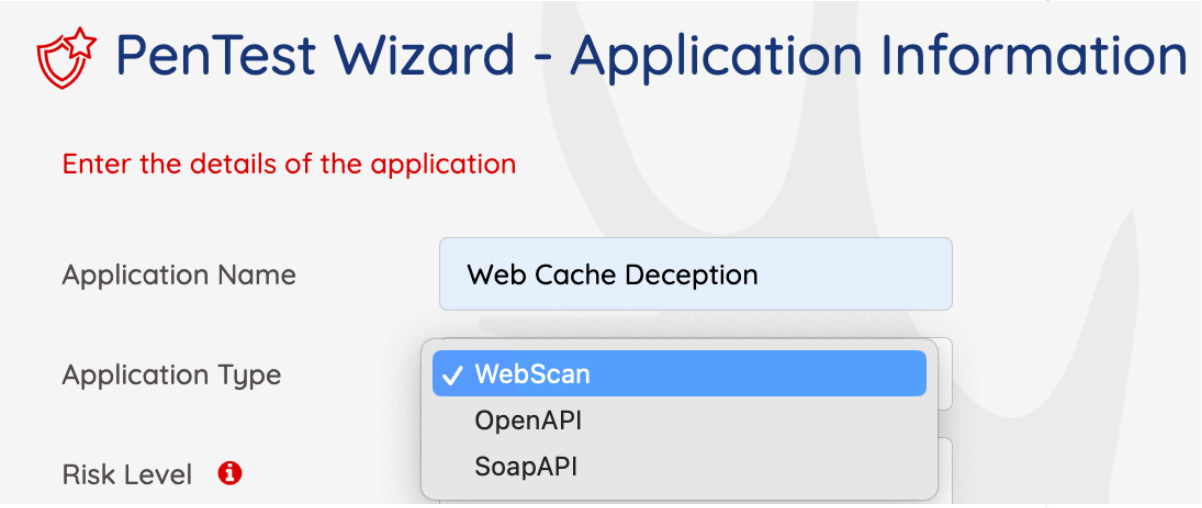
Prancer Security Solution Highlight Multi cloud and on prem support

Prancer's cloud security solution is designed to support multi-cloud and on-premise environments. It supports major cloud providers such as Azure, AWS, GCP, and private clouds based on Kubernetes clusters, Oracle cloud, and on-prem clouds. The solution extracts intelligence from the cloud and uses it to formulate attacks on APIs. This allows organizations to effectively validate the security of their cloud applications and infrastructure. The integration of multiple cloud providers and on-premise environments ensures that the solution is adaptable to the diverse needs of organizations, providing them with a comprehensive approach to cloud security.



Shift Left mindset

Prancer's cloud security solution features a shift-left security approach that allows for seamless integration into the software development life cycle (SDLC) process. The solution supports CI/CD integration, as well as integration with Postman, to provide a comprehensive security assessment of APIs during the development process. The solution supports multiple API formats, including OpenAPI, SOAP, and GraphQL, allowing for a flexible and versatile security validation process. With the ability to integrate early in the SDLC process, Prancer's cloud security solution helps to identify and remediate security issues before they become a problem, resulting in a more secure and reliable final product.



API Fuzzing

Prancer's cloud security solution features an advanced API fuzzer. The fuzzer is designed based on the OWASP API Security guidelines and incorporates error code and logic-based checkers. The solution provides a comprehensive assessment of the API by analyzing the entire specification of the API, be it OpenAPI or SoapAPI. Prancer then generates and intelligently executes vulnerability tests to identify any potential weaknesses in the API. This feature enables organizations to identify and remediate security issues before deployment, ensuring the security of their APIs.

Custom Attacks on API

Prancer's cloud security solution also offers the ability for clients to write custom codified attacks for API security. With this feature, organizations can tailor the security tests to meet their specific needs and address any potential vulnerabilities that may be unique to their API infrastructure. This allows for a more comprehensive and thorough evaluation of API security, and the ability to identify and address potential risks in a timely manner. The custom codified attack feature supports multiple API formats, including OpenAPI, SOAP, and GraphQL, ensuring compatibility with a wide range of API systems.

One example of how Prancer also integrates custom attacks on APIs is by scanning JWT vulnerabilities. **jwt_tool** makes the header and payload values nice and clear and has a "Playbook Scan" that can be used to target a web application and scan for common JWT vulnerabilities. This scan can be run by invoking this command:

```
$ jwt_tool -t http://target-site.com/ -rc "Header: JWT_Token" -M pb
```

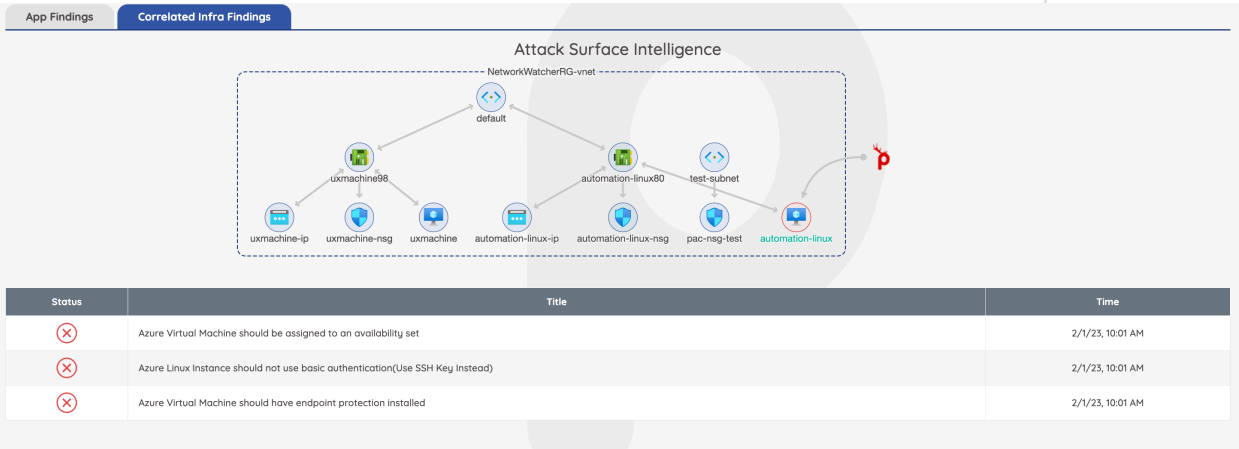
To use this command, we set the JWT header. With this information, replace "Header" with the name of the header we provide and "JWT_Token" with the actual token value that we generate.

The image below is a portion of the PAC Configuration file that allows you to load custom addons and attacks for you own business case with filters allowing you to fine tune what scans you want to run.

```
CVE:
- Path:
    Include:
    - ApiScanAddons
    Exclude: []
  Connector: cve_connector
```

API Applications and Infrastructure Correlation

Prancer's cloud security solution features a comprehensive approach to API security. In addition to providing pentesting findings for cloud API applications, the solution also includes a correlated infrastructure scan to provide a complete understanding of the attack surface. This approach contributes to attack surface intelligence and provides a deeper understanding of all potential attack vectors. With this feature, Prancer ensures that its clients have a thorough understanding of the security posture of their API applications and the infrastructure supporting them.



Authentication Mechanism

Prancer's cloud security solution features a modern authentication mechanism for APIs, supporting various methods including form-based authentication, JWT authentication, cookie authentication, OAuth authentication, and custom authentication. Additionally, the solution also enables unauthenticated scans to provide comprehensive security assessments. With support for a wide range of authentication methods, Prancer helps organizations secure their APIs and applications against potential threats and vulnerabilities.

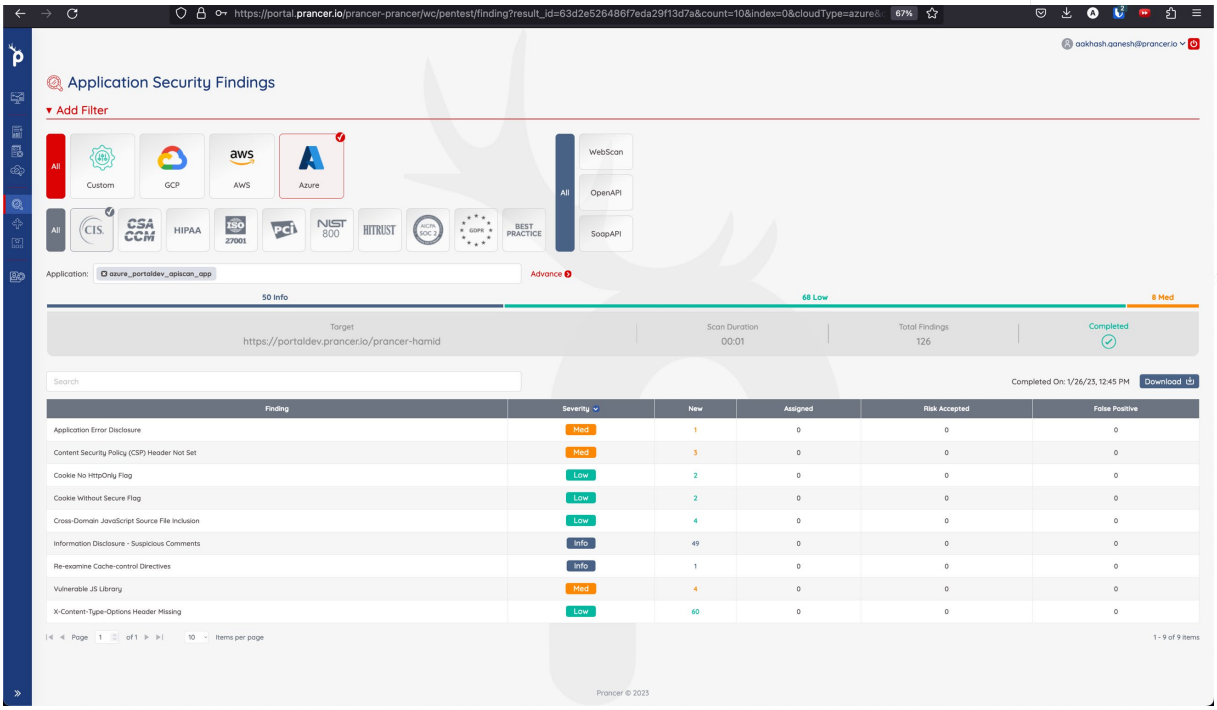
Sample Reports Examples for the API Security TOP 10

The following pages contain example screenshots from real results of vulnerabilities pertaining to the API Security TOP 10.

The screenshots were taken from the results page which contain information about the vulnerability such as its description, relevant tags, WASC & CWE IDs, remediation solutions, and links to relevant reference materials. The page also then presents specific information about the vulnerability based on the link selected.

Selecting a page from the selection brings up the request and response information about the vulnerability as it applies to the selected page. Information presented include header, and body information as well as any pertinent information available such as the evidence of the vulnerability, response parameters and the specific attack that was sent in order to uncover the vulnerability.

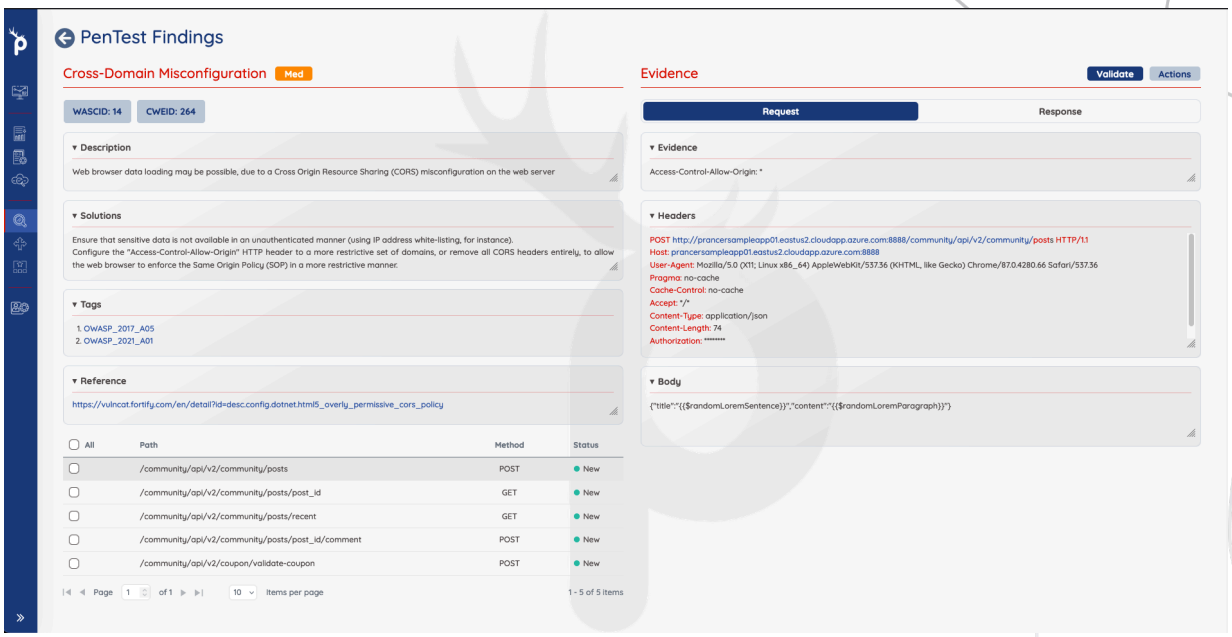
The following image is a “Security Finding Page” of a real API security scan done by prancer:

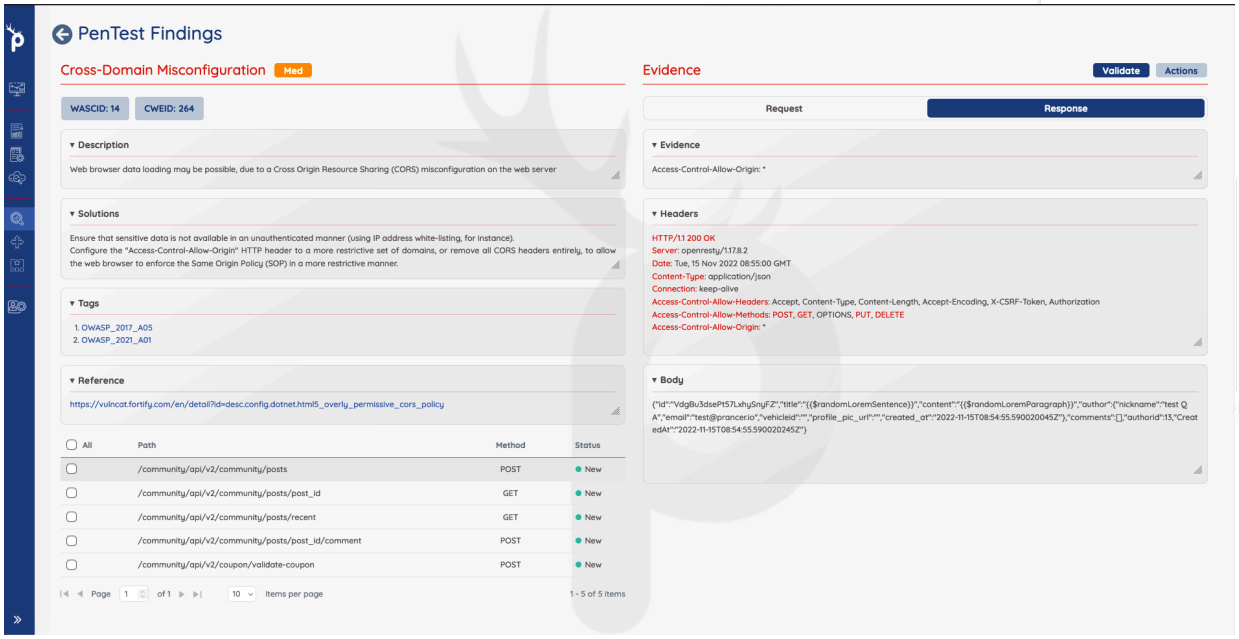


df

Security Misconfiguration

A sample screenshot of a security misconfiguration vulnerability alert is shown below:

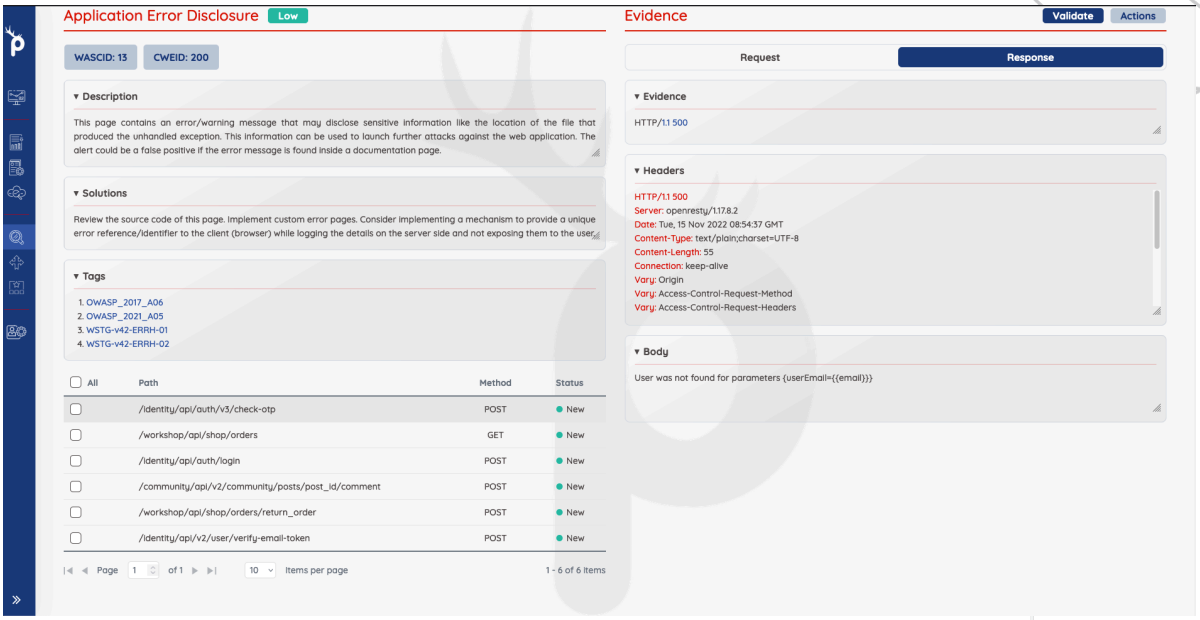


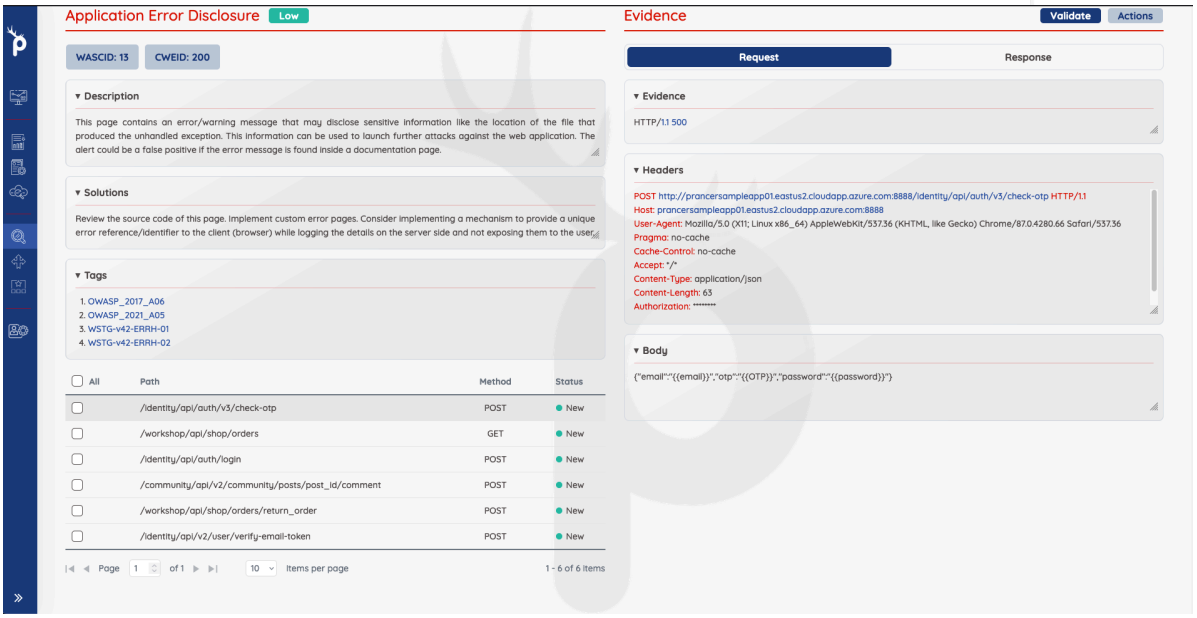


The result shown above is for a Cross-Domain misconfiguration vulnerability. It's classified as Security Misconfiguration Vulnerability because it can allow an attacker to bypass security restrictions and access sensitive data from an API, leading to information leakage and potential security breaches, which can occur when the API is not properly configured to restrict access to specific domains, allowing any domain to access the API's data, potentially exposing sensitive information to unauthorized parties.

Excessive Data Exposure

A sample screenshot of an Excessive Data Exposure vulnerability alert is shown below:

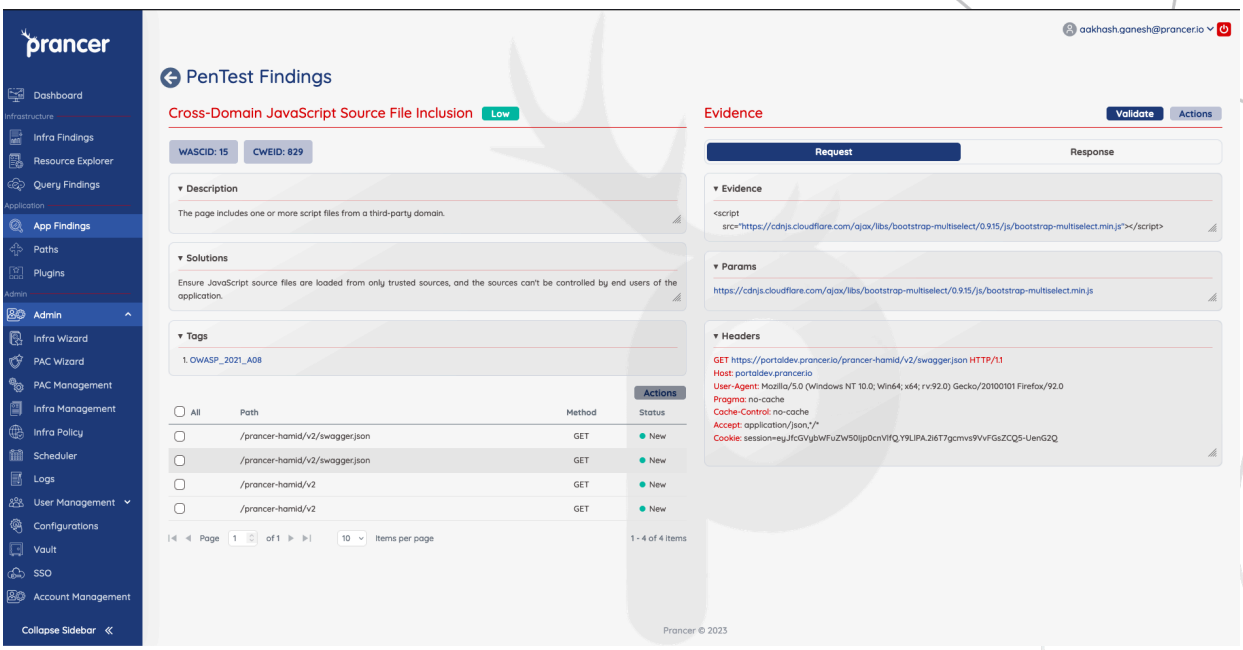


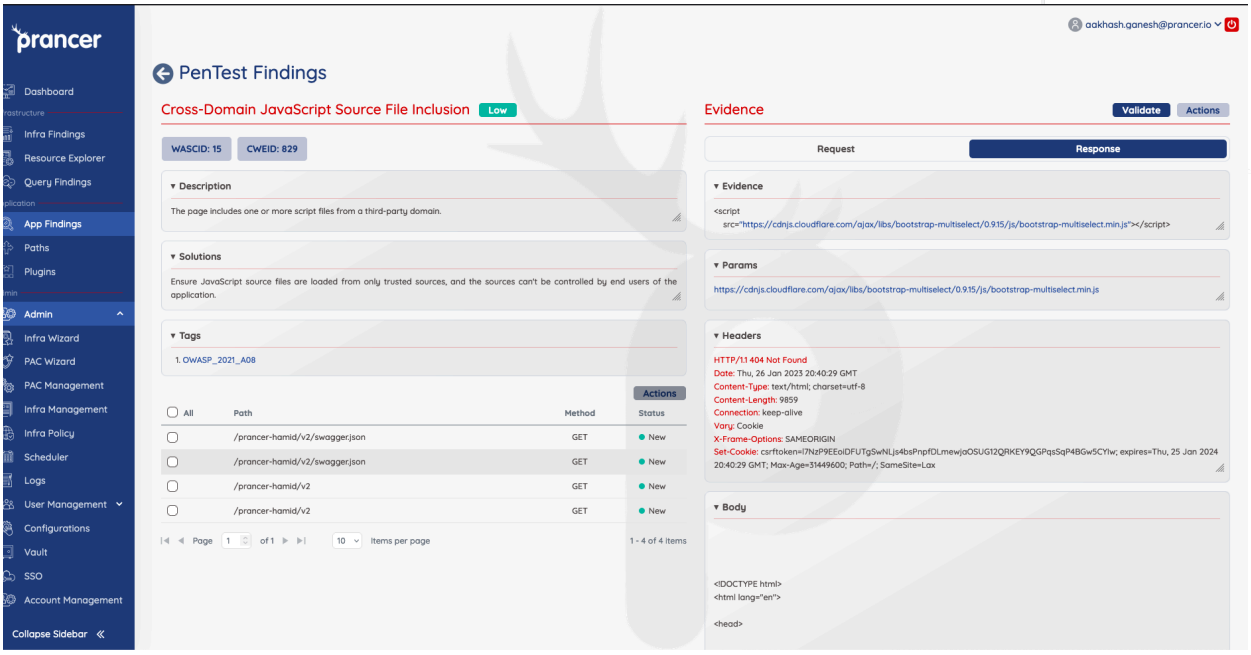


The result shown above is for an Application Error Disclosure vulnerability. It's classified as Excessive Data Exposure vulnerability because it can reveal sensitive information about the application's internal workings, systems, and data to attackers. This can occur when the application returns detailed error messages that include sensitive information, such as stack traces, database error messages, and other details that can help an attacker gain insight into the application's security weaknesses.

Broken Access Control

A sample screenshot of a Broken Access Control vulnerability alert is shown below:

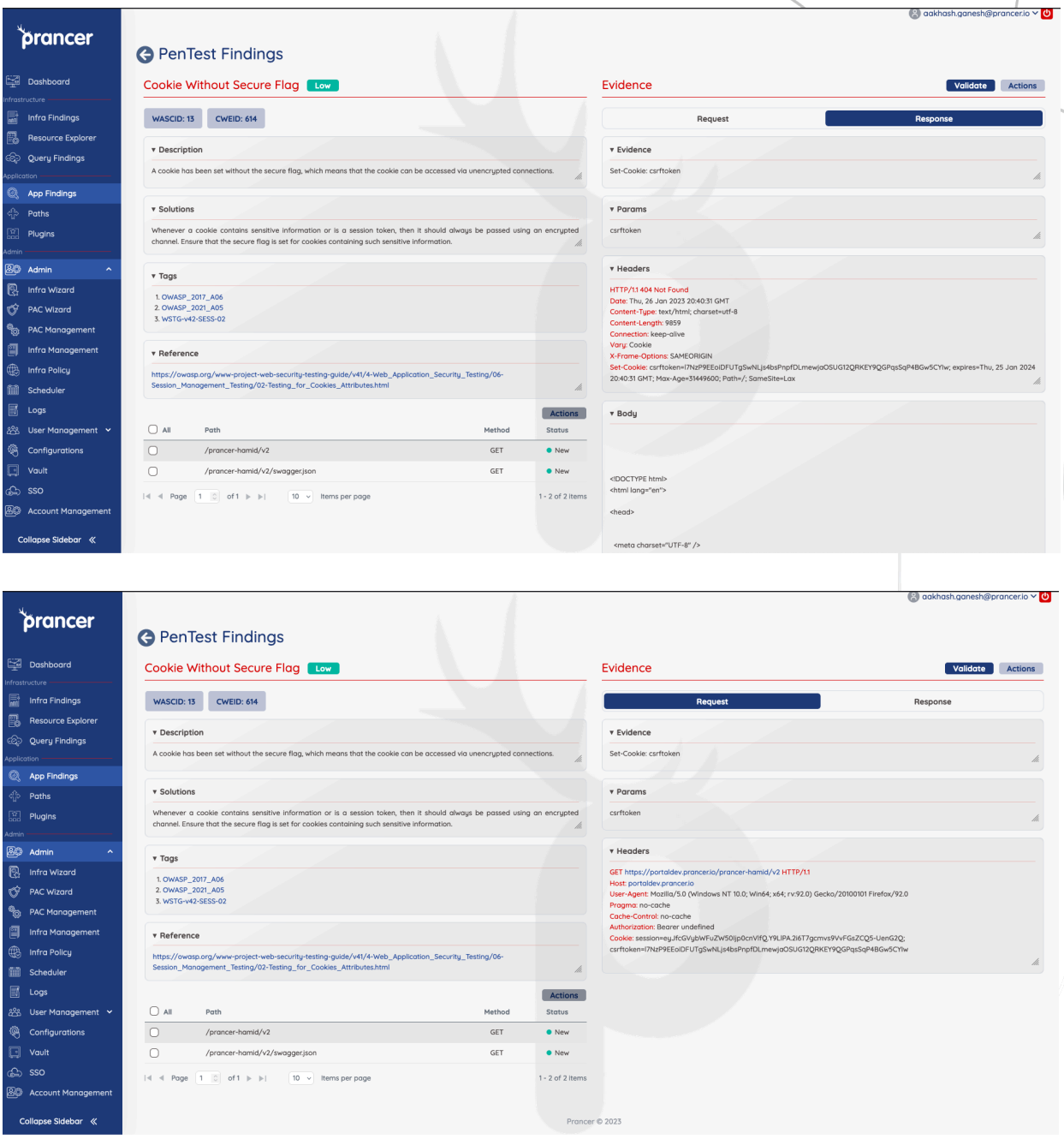




The result shown above is for a Cross-Domain JavaScript Source File Inclusion vulnerability. It's classified as a Broken Access Control vulnerability because it allows an attacker to bypass security restrictions and gain unauthorized access to sensitive data and functionality. This can occur when the application allows the inclusion of untrusted and potentially malicious JavaScript code from external domains, potentially allowing an attacker to inject malicious code into the application and gain access to sensitive data and functionality.

Broken Authentication

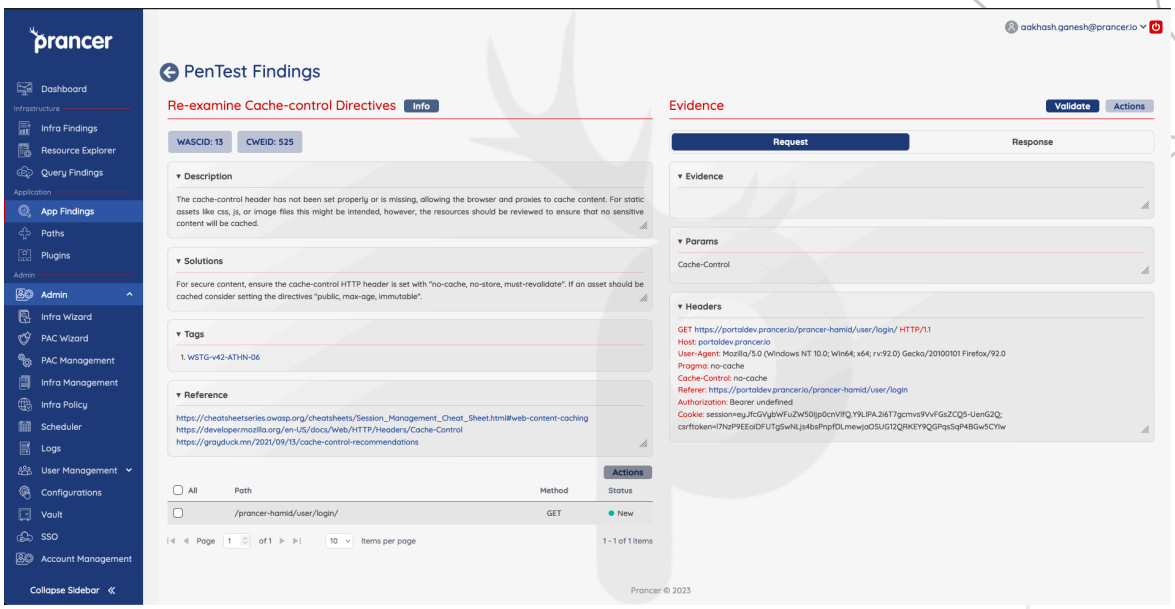
A sample screenshot of a Broken Authentication vulnerability alert is shown below:

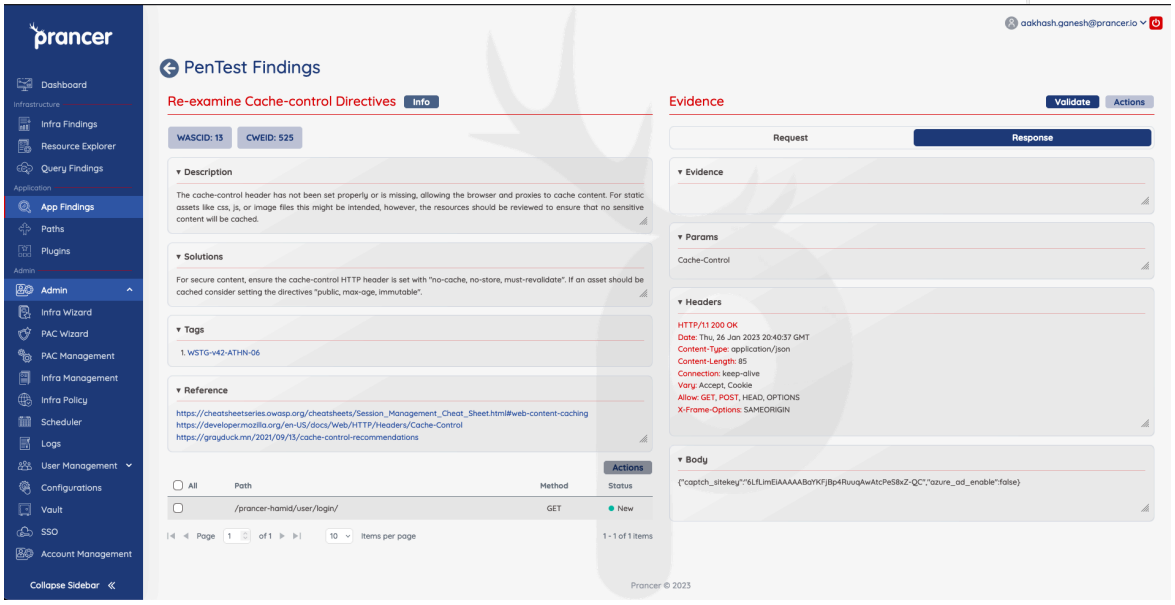


The result shown above is for a Cookie Without Secure Flag vulnerability. It's classified as a broken authentication vulnerability because it can allow an attacker to intercept and steal sensitive information transmitted in the cookie, such as user credentials and other sensitive data. The secure flag is a security setting that instructs the browser to only send the cookie over an encrypted connection, such as HTTPS. If the secure flag is not set, the cookie can be transmitted in clear text, making it vulnerable to interception and theft by an attacker who is able to sniff the network traffic.

Improper Asset Management

A sample screenshot of an Improper Asset Management vulnerability alert is shown below:

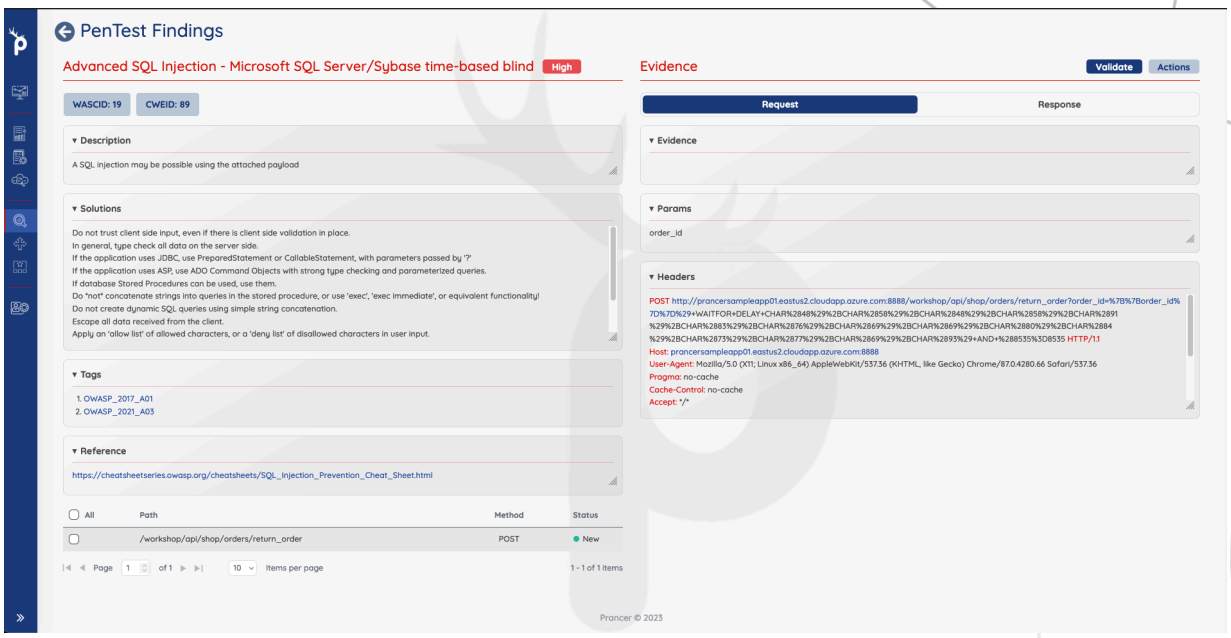


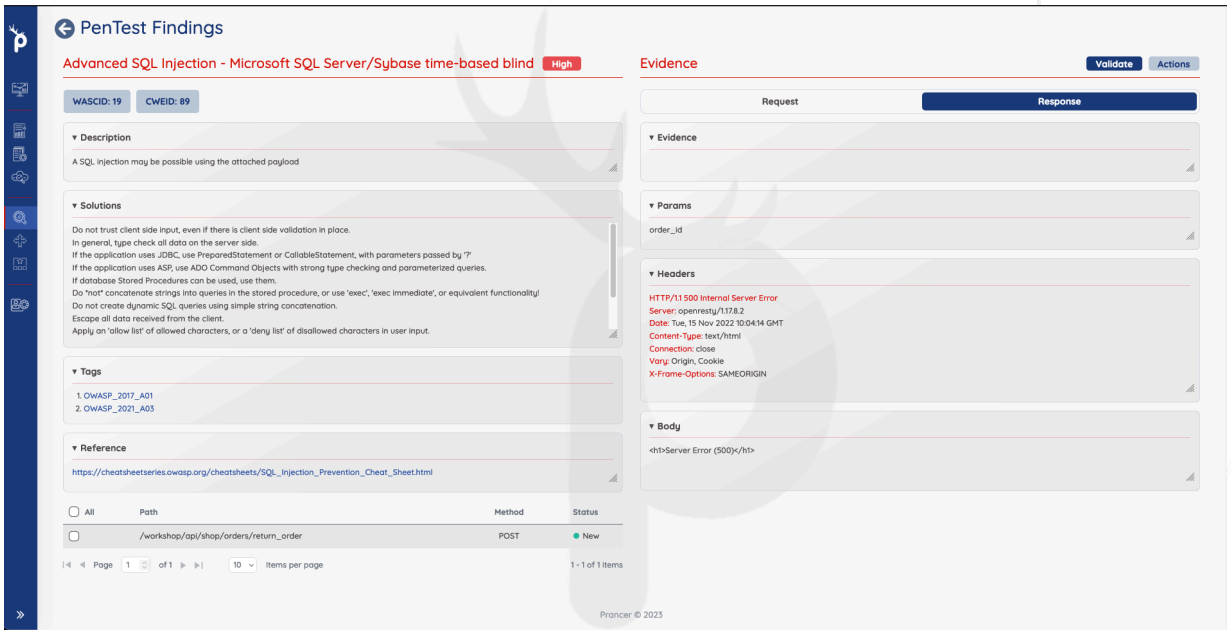


The result shown above is for a Re-Examine Cache-Control Directives vulnerability. It's classified as a broken authentication vulnerability because it can result in sensitive information being cached and exposed to unauthorized parties. This can occur when the application does not properly manage cache control directives, allowing sensitive information, such as user credentials, session tokens, and other confidential data, to be cached on the client side and potentially exposed to attackers. Improper asset management refers to the failure of the application to properly manage and secure its assets, including sensitive data, resources, and more, potentially leading to security breaches.

Injection

A sample screenshot of an Injection vulnerability alert is shown below:





The result shown above is for an SQL injection vulnerability. It's classified as an Injection vulnerability because it allows an attacker to inject malicious code into the application's SQL queries, potentially compromising the database and its contents. This can occur when the application does not properly validate user-supplied input, allowing an attacker to enter malicious SQL code into the application's queries, which can then be executed by the database.

Conclusion

At Prancer, we provide automated security validation services based on the OWASP API Security Project standards. Our cutting-edge technology ensures that your APIs are thoroughly checked for vulnerabilities and risks in a matter of minutes. And that's not all, our expertise extends to automating your infrastructure and cloud security validation as well. Reach out to us today and let us show you how our automated security validation services can benefit your business. Give us a try and see the difference for yourself.

Additional Resources

https://owasp.org/www-project-api-security/

https://cloudsecurityalliance.org/blog/2021/05/11/understanding-the-owasp-api-security-top-10/

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

https://cloudsecurityalliance.org/blog/2021/05/11/understanding-the-owasp-api-security-top-10/

Prancer White Paper on PAC

https://prancer.io/automated-penetration-testing-whitepaper

Prancer Links

https://www.prancer.io/api-security-validation-at-scale/

https://docs.prancer.io/web/