OWASP API Security Top 10 2019

The Ten Most Critical Web Application Security Risks

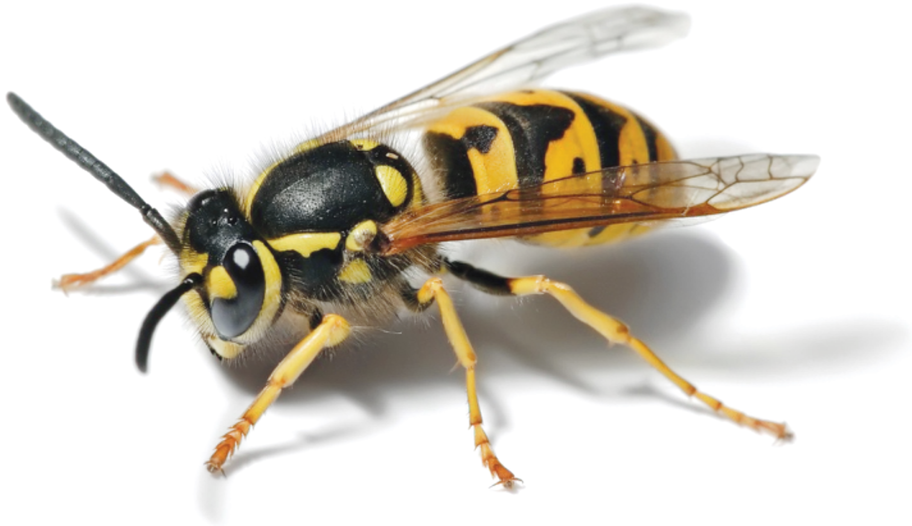


Table of Contents

About OWASP

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications and APIs that can be trusted.

At OWASP, you'll find free and open:

- Application security tools and standards.
- Complete books on application security testing, secure code development, and secure code review.
- Presentations and videos.
- Cheat sheets on many common topics.
- Standard security controls and libraries.
- Local chapters worldwide.
- Cutting edge research.
- Extensive conferences worldwide.
- Mailing lists.

Learn more at: https://www.owasp.org.

All OWASP tools, documents, videos, presentations, and chapters are free and open to anyone interested in improving application security.

We advocate approaching application security as a people, process, and technology problem because the most effective approaches to application security require improvements in these areas.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, and cost-effective information about application security.

OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. OWASP produces many types of materials in a collaborative, transparent, and open way.

The OWASP Foundation is the non-profit entity that ensures the project's long-term success. Almost everyone associated with OWASP is a volunteer, including the OWASP board, chapter leaders, project leaders, and project members.

We support innovative security research with grants and infrastructure.

Come join us!

FW Foreword

A foundational element of innovation in today's app-driven world is the API. From banks, retail and transportation to IoT, autonomous vehicles and smart cities, APIs are a critical part of modern mobile, SaaS and web applications and can be found in customer-facing, partner-facing, and internal applications.

By nature, APIs expose application logic and sensitive data such as Personally Identifiable Information (PII) and because of this, they have increasingly become a target for attackers. Without secure APIs, rapid innovation would be impossible.

Although a broader web application security risks top 10 still makes sense, due to their particular nature, an API specific security risks list is mandatory. API security focuses on strategies and solutions to understand and mitigate the unique vulnerabilities and security risks of Application Programming Interfaces (APIs).

If you're familiar with the OWASP Top 10 Project, then you'll notice the similarities between both documents: they are intended for readability and adoption. If you're new to the OWASP Top 10 series, maybe you'll be better reading the API Security Risks and Methodology and Data sections before jumping into the Top 10 list.

You can contribute to OWASP API Security Top 10 with your questions, comments, and ideas at our GitHub project repository:

- https://github.com/OWASP/API-Security/issues
- https://github.com/OWASP/API-Security/blob/master/CONTRIBUTING.md

You can find the OWASP API Security Top 10 here:

- https://www.owasp.org/index.php/OWASP_API_Security_Project
- https://github.com/OWASP/API-Security

We wish to thank all the contributors who made this project possible with their effort and contributions. They are all listed in the Acknowledgments section. Thank you!

I Introduction

RN Release Notes

This is the first OWASP API Security Top 10 edition, which we plan to be updated periodically, every three or four years.

Unlike this version, in future versions, we want to make a public call for data, involving the security industry in this effort. In the Methodology and Data section, you'll find more details about how this version was built. For more details about the security risks, please refer to the API Security Risks section.

It is important to realize that over the last few years, applications architecture has changed a lot. Nowadays, APIs play a very important role in this new architecture of microservices, Single Page Applications (SPAs), mobile apps, IoT, etc.

The OWASP API Security Top 10 was a required piece to create awareness about modern APIs security issues. It was only possible thanks to a great effort of several volunteers, all of them listed in the Acknowledgments section. Thank you!

RISK API Security Risk

The OWASP Risk Rating Methodology was used to do the risk analysis.

The table below summarizes the terminology associated with the risk score.

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impacts
API Specific	Easy: **3**	Widespread **3**	Easy **3**	Severe **3**	Business Specific
	Average: **2**	Common **2**	Average **2**	Moderate **2**	
	Difficult: **1**	Difficult **1**	Difficult **1**	Minor **1**	

Note: This approach does not take the likelihood of the threat agent into account. Nor does it account for any of the various technical details associated with your particular application. Any of these factors could significantly affect the overall likelihood of an attacker finding and exploiting a particular vulnerability. This rating does not take into account the actual impact on your business. Your organization will have to decide how much security risk from applications and APIs the organization is willing to accept given your culture, industry, and regulatory environment. The purpose of the OWASP API Security Top 10 is not to do this risk analysis for you.

References

OWASP

- OWASP Risk Rating Methodology
- Article on Threat/Risk Modeling

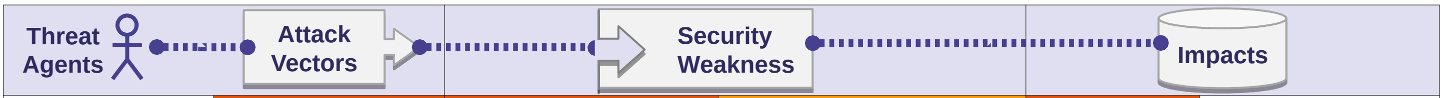
External

- ISO 31000: Risk Management Std
- ISO 27001: ISMS
- NIST Cyber Framework (US)
- ASD Strategic Mitigations (AU)
- NIST CVSS 3.0
- Microsoft Threat Modeling Tool

T10 OWASP API Security Top 10 - 2019

A1:2019 - 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 taken in mind in every function that accesses a data source using an input from the user.
A2:2019 - 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 system's ability to identify the client/user, compromises API overall security.
A3:2019 - 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 showing it to the user. Without controlling client's state, servers receive more and more filters which can be abused to gain access to sensitive data.
A4:2019 - Lack of Resources & 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 this can impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force.
A5:2019 - Broken Function Level Authorization	Complex access control policies with different hierarchies, groups and roles and a not so clear separation between administrative and regular functions tend to lead to authorization flaws. Exploiting these issues, attackers gain access to other users resources and/or administrative functions.
A6:2019 - Mass Assignment	Binding client provided data (e.g. JSON) to data models without proper properties filtering based on a whitelist usually lead to Mass Assignment. Either guessing objects properties, exploring other API endpoints or reading the documentation, providing additional object properties in request payloads, allow attackers to modify object properties they are not supposed to.
A7:2019 - Security Misconfiguration	Security misconfiguration is commonly a result of insecure 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.
A8:2019 - Injection	Injection flaws, such as SQL, NoSQL, Command Injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A9:2019 - Improper Assets Management	APIs tend to expose more endpoints than traditional web applications, what makes 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.
A10:2019 - 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 and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

A1:2019 Broken Object Level Authorization



API Specific	Exploitability: 3	Prevalence: 3	Detectability: 2	Technical: 3	Business Specific
Attackers can exploit API endpoints that are vulnerable to broken object authorization by manipulating the ID of an object that is sent in the request. This may lead to unauthorized access to sensitive data. This issue is extremely common in API based applications because the server component usually does not fully track the client's state, and instead relies more on parameters like object IDs, that are sent from the client to decide which objects to access.		This has been the most common and impactful attack on APIs. Authorization and access control mechanisms in modern applications are complex and wide-spread. Even if the application implements a proper infrastructure for authorization checks, developers might forget to use these checks before accessing a sensitive object. Access control detection is not typically amenable to automated static or dynamic testing.		Unauthorized access can result in data disclosure to unauthorized parties, data loss, or data manipulation. Unauthorized access to objects can also lead to full account takeover.	

Is The API Vulnerable?

Object level authorization is an access control mechanism that is usually implemented at the code level to validate that one user can access only objects that he should have access to.

Every API endpoint that receives an ID of an object and performs any type of action on the object should implement object level authorization checks. The checks should validate that the logged-in user does have access to perform the requested action on the requested object.

Failures in the mechanism typically leads to unauthorized information disclosure, modification or destruction of all data.

Example Attack Scenarios

Scenario #1

An e-commerce platform for online stores provides a listing page with the revenue charts for their hosted shops. Inspecting the browser requests, an attacker identifies the API endpoints used as a data source for those charts and their pattern `/shops/{shopName}/revenue_data.json`. Using another API endpoint, the attacker gets the list of all hosted shop names. With a simple script to iterate over the names in the list, replacing `{shopName}` in the URL, the attacker gains access to the sales data of thousands of e-commerce stores.

Scenario #2

While monitoring the network traffic of a wearable device, the following HTTP `PATCH` request get the attention of an attacker due to the presence of a custom HTTP request header `X-User-Id: 54796`. Replacing the `X-User-Id` value with `54795`, the attacker receives a successful HTTP response and is able to modify other user account data.

A1:2019 Broken Object Level Authorization

How To Prevent

- Implement a proper authorization mechanism that relies on the user policies and hierarchy.
- Prefer not to use an ID that has been sent from the client, but instead use an ID that is stored in the session object when accessing a database record by the record ID.
- 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.

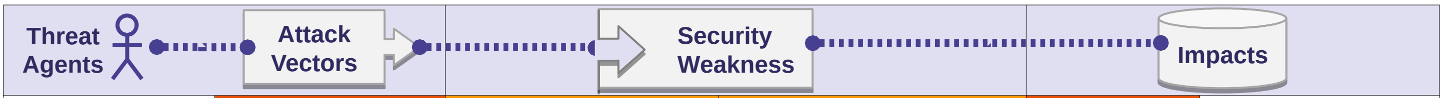
References

OWASP

External

- CWE-284: Improper Access Control
- CWE-285: Improper Authorization
- CWE-639: Authorization Bypass Through User-Controlled Key

A2:2019 Broken Authentication



API Specific	Exploitability: 3	Prevalence: 2	Detectability: 2	Technical: 3	Business Specific
Authentication in APIs is a complex and confusing mechanism. Software and security engineers might have misconceptions about what are the boundaries of authentication and how to implement it right. On top of that, the authentication mechanism is an easy target for attackers, since it's exposed to everyone. These two points, makes the authentication component a fertile ground for many exploits.		There are two sub issues: 1. Lack of protection mechanisms: APIs endpoints that are responsible for authentication must be treated differently from regular endpoints and implement extra layers of protection. 2. Misimplementation of the mechanism: The mechanism is used \ implemented without considering the attack vectors, or it's the wrong use case (e.g.: an authentication mechanism designed for IoT clients might not be the right choice for web applications)		Attackers can gain control to other users' accounts in the system, read their personal data and perform sensitive actions on their behalf like money transactions and sending personal messages.	

Is the API Vulnerable?

Authentication endpoints and flows are assets that need to be protected. "Forgot password / reset passwords" should be treated the same way as authentication mechanisms.

An API is vulnerable if:

- Permits credential stuffing where the attacker has a list of valid usernames and passwords.
- Permits attackers to perform a brute force attack on the same user, without presenting captcha \ account lockout mechanism
- Permits weak passwords
- Sends sensitive authentication details, such as auth tokens and password in the URL.
- Doesn't validate the authenticity of tokens
- Accepts unsigned / weakly signed JWT tokens ("alg":"none") / doesn't validate their expiration date
- Uses plain text, encrypted, or weakly hashed passwords
- Uses weak encryption keys / API keys

Example Attack Scenarios

Scenario #1

Credential stuffing, the use of lists of known 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 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-thread script, against the `/api/system/verification-codes/{smsToken}` endpoint to discover the right token within a few minutes.

A2:2019 Broken Authentication

How To Prevent

- APIs that have access to sensitive data should use an additional form of authentication in addition to API keys.
- API keys should have restrictions both for the applications (e.g. mobile app, IP address) and the set of APIs they are valid for.
- API keys should be stored on a secure location such as a vault.
- A Configuration Management (CM) tool should be used and a clear configuration management process should be defined.

References

OWASP

- OWASP Key Management Cheat Sheet
- OWASP Authentication Cheatsheet
- Credential Stuffing

External

- CWE-798: Use of Hard-coded Credentials

A3:2019 Excessive Data Exposure

<table>
<tr><th colspan="2">Threat Agents / Attack Vectors</th><th colspan="2">Security Weakness</th><th colspan="2">Impacts</th></tr>
<tr><td>API Specific</td><td>Exploitability: 3</td><td>Prevalence: 2</td><td>Detectability: 2</td><td>Technical: 2</td><td>Business Specific</td></tr>
<tr><td colspan="2">Exploitation of Excessive Data Exposure is simple, and is usually done by sniffing the traffic to analyze the API responses looking for sensitive data exposure that should not be returned to the user.</td><td colspan="2">APIs rely on clients to perform the data filtering. Since APIs are used as data sources, sometimes developers try to implement them in a generic way without thinking about the sensitivity of the exposed data. Automatic tools usually can’t detect this type of vulnerability because it’s hard to differentiate between legitimate data returned from the API and sensitive data that should not be returned without a deep understanding of the application.</td><td colspan="2">Excessive Data Exposure commonly leads to exposure of sensitive data.</td></tr>
</table>

Is the API Vulnerable?

The API returns sensitive data to the client by design. This data is usually filtered on the client side before being presented to the user. An attacker can easily sniff the traffic and see the sensitive data.

Example Attack Scenarios

Scenario #1

The mobile team uses the `/api/articles/{articleId}/comments/{commentId}` endpoint in the articles view to render comments metadata. Sniffing the mobile application traffic an attacker finds out that other sensitive data related to comment’s author, is also returned. The endpoint implementation uses a generic `toJSON()` method on the `User` model, which contains PII, to serialize the object.

Scenario #2

An IOT-based surveillance system allows administrators to create users with different permissions. An admin created a user for a new security guard that should have access only to specific buildings in the site. Once the security guard uses his mobile app, an API call is triggered to: `"/api/sites/111/cameras"` in order to receive data about the available cameras and show them on the dashboard. The response contains a list with details about cameras in the following format: `{"id":"xxx","live_access_token":"xxxx-bbbbb","building_id":"yyy"}` While the client GUI shows only cameras which the security guard should have access to, the actual API response contains a full list of all the cameras in the site.

A3:2019 Excessive Data Exposure

How To Prevent

- Never rely on the client side to perform sensitive data filtering.
- Review the responses from the API to make sure they contain only legitimate data.

References

OWASP

External

- CWE-213: Intentional Information Exposure

A4:2019 Lack of Resources & Rate Limiting

Threat Agents / Attack Vectors		Security Weakness		Impacts	
API Specific	**Exploitability: 2**	**Prevalence: 3**	**Detectability: 3**	**Technical: 2**	**Business Specific**
Exploitation requires simple API requests. No authentication is required. Multiple concurrent requests can be performed from a single local computer or by using cloud computing resources.		It’s common to find APIs that do not implement rate limiting or APIs where limits are not properly set.		Exploitation may lead to DoS, making the API unresponsive or even unavailable.	

Is the API Vulnerable?

API requests consume resources such as network, CPU, memory and storage and the amount of resources required to satisfy a request greatly depends on the user input and endpoint business logic. Also consider that requests from multiple API clients compete for resources. An API is vulnerable if at least one of the following limits is missing or set inappropriately (i.e. too low/high)

- Execution timeouts
- Max allocable memory
- Number of file descriptors
- Number of processes
- Request payload size (e.g. uploads)
- Number of requests per client/resource
- Number of records per page to return in a single request response

Example Attack Scenarios

Scenario #1

An attacker uploads a large image by issuing a POST request to `/api/v1/images`. When the upload is complete, the API creates multiple thumbnails with different sizes. Due to the size of the uploaded image, available memory is exhausted during the creation of thumbnails and the API becomes unresponsive.

Scenario #2

We have an application that contains the users' list on a UI with a limit of `200` users per page. The users' list is retrieved from the server using the following query: `/api/users?page=1&size=100`. An attacker changes the `size` parameter to `200 000`, causing performance issues on the database. Meanwhile, the API becomes unresponsive and unable to handle further requests from this or any other clients (aka DoS).

The same scenario might be used to provoke Integer Overflow or Buffer Overflow errors.

A4:2019 Lack of Resources & Rate Limiting

How To Prevent

- Docker makes it easy to limit memory, CPU, number of restarts, file descriptors and processes.
- 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 control the number of records to be returned in the response.

References

OWASP

- Blocking Brute Force Attacks
- Docker Cheat Sheet - Limit resources (memory, CPU, file descriptors, processes, restarts)
- REST Assessment Cheat Sheet

External

- CWE-307: Improper Restriction of Excessive Authentication Attempts
- CWE-770: Allocation of Resources Without Limits or Throttling
- "*Rate Limiting (Throttling)*" - Security Strategies for Microservices-based Application Systems, NIST

A5:2019 Broken Function Level Authorization

<table>
<tr><td colspan="2">Threat Agents / Attack Vectors</td><td colspan="2">Security Weakness</td><td colspan="2">Impacts</td></tr>
<tr><td>API Specific</td><td>Exploitability: 3</td><td>Prevalence: 2</td><td>Detectability: 1</td><td>Technical: 2</td><td>Business Specific</td></tr>
<tr><td colspan="2">Exploitation requires the attacker to send legitimate API calls to the API endpoint he should not have access to. These endpoints might be exposed to anonymous users or regular, non-privileged users. It is easier to discover these flaws in APIs since APIs are more structured, and the way to access certain functions is more predictable (e.g. replacing the HTTP method from <code>GET</code> to <code>PUT</code>, or changing the “users” string in the URL to "admins")</td><td colspan="2">Authorization checks for a function or resource are usually managed via configuration, and sometimes at the code level. Implementing proper checks can be a confusing task since modern applications can contain many types of roles or groups and complex user hierarchy (e.g. sub-users, users with more than one role)</td><td colspan="2">Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack</td></tr>
</table>

Is the API Vulnerable?

The best way to find broken function level authorization issues is to perform deep analysis of the authorization mechanism while keeping in mind the user hierarchy, different roles or groups in the application, and asking the following questions:

- Can a regular user access administrative endpoints?
- Can a user perform sensitive actions (e.g. creation, modification or erasure) that they should not have access to by simply changing the HTTP method (e.g. from `GET` to `DELETE`)?
- Can a user from group X access a function that should be exposed only to users from group Y by simply guessing the endpoint URL and parameters (e.g. `/api/v1/users/export_all`)?

Don’t assume that an API endpoint is regular or administrative only based on the URL path.

While developers might choose to expose most of the administrative endpoints under a specific relative path, like `api/admins`, it’s very common to find these administrative endpoints under other relative paths together with regular endpoints, like `api/users`.

Example Attack Scenarios

Scenario #1

During the registration process to an application that allows only invited users to join, the mobile application triggers an API call to `GET /api/invites/{invite_guid}`. The response contains a JSON with details about the invite including the user’s role and the user’s email.

An attacker duplicated the request and manipulated the HTTP method and endpoint to `POST /api/invites/new`. This endpoint should only be accessed by administrators using the admin console which does not implement function level authorization checks.

The attacker exploits the issue and sends himself an invite to create an admin account:

```
POST /api/invites/new
{“email”:”hugo@malicious.com”,”role”:”admin”}
```

A5:2019 Broken Function Level Authorization

Scenario #2

An API contains an endpoint that should be exposed only administrators - `GET /api/admin/v1/users/all`. This endpoint returns the details of all the users on the application and does not implement function level authorization checks. An attacker who learned the API structure takes an educated guess and manages to access this endpoint which exposes sensitive details of the users of the application.

How To Prevent

Your application should have a consistent and easy to analyze authorization module that is invoked from all your business functions. Frequently, such protection is provided by one or more components external to the application code.

- 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.

References

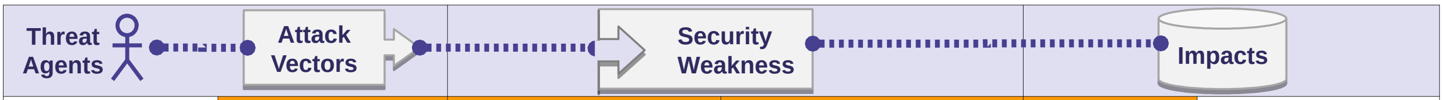
OWASP

- OWASP Article on Forced Browsing
- OWASP Top 10 2013-A7-Missing Function Level Access Control
- OWASP Development Guide: Chapter on Authorization

External

- CWE-285: Improper Authorization

A6:2019 Mass Assignment



API Specific	Exploitability: 2	Prevalence: 2	Detectability: 2	Technical: 2	Business Specific
Exploitation usually requires an understanding of the business logic, objects' relations, and the API structure. Exploitation of mass assignment is easier in APIs since by design they expose the underlying implementation of the application along with the properties' names.		Modern frameworks encourage developers to use functions that automatically bind input from the client into code variables and internal objects. Attackers can use this methodology to update or overwrite sensitive object's properties that the developers never intended to expose.		Exploitation may lead to privilege escalation, data tampering, bypass of security mechanisms and more.	

Is the API Vulnerable?

Objects in modern applications might contain many properties. Some of these properties should be updated directly by the client (e.g. `user.first_name` or `user.address`) and some of them should not (e.g. `user.is_vip` flag)

An API endpoint is vulnerable if it automatically converts client parameters into internal object properties without considering the sensitivity and the exposure level of these properties. This could allow an attacker to update object properties that they should not have access to.

Examples for sensitive properties:

- **Permission-related properties**: `user.is_admin`, `user.is_vip` should only be set by admins.
- **Process-dependent properties**: `user.cash` should only be set internally after payment verification.
- **Internal properties**: `article.created_time` should only be set internally by the application.

Example Attack Scenarios

Scenario #1

A ride sharing application provides a user the option to edit basic information for their profile. During this process an API call is sent to `PUT /api/v1/users/me` with the following legitimate JSON object:

```
{"user_name":"inons","age":24}
```

The request `GET /api/v1/users/me` includes an additional credit_balance property:

```
{"user_name":"inons","age":24,"credit_balance":10}.
```

The attacker replays the first request with the following payload:

```
{"user_name":"attacker","age":60,"credit_balance":99999}
```

Since the endpoint is vulnerable to mass assignment, the attacker receives credits without paying.

Scenario #2

A video sharing portal allows users to upload content and download content in different formats. An attacker who explores the API found that the endpoint `GET /api/v1/videos/{video_id}/meta_data` returns a JSON object with the video's properties. One of the properties is `"mp4_conversion_params":"-v codec h264"`: which indicates that the application uses a shell command to convert the video.

The attacker also found the endpoint `POST /api/v1/videos/new` is vulnerable to mass assignment and allows the client to set any property of the video object. The attacker sets a malicious value: `"mp4_conversion_params":"-v codec h264 && format C:/"`. This value will cause a shell command injection once the attacker downloads the video as MP4.

How To Prevent

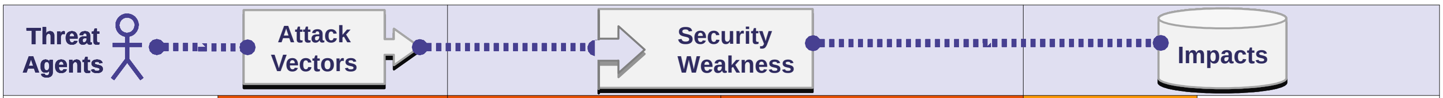
- If possible, avoid using functions that automatically bind a client's input into code variables or internal objects.
- Whitelist only the properties that should be updated by the client.
- Use built-in features to blacklist properties that should not be accessed by clients.

References

External

- CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes

A7:2019 Security Misconfiguration



API Specific	Exploitability: 3	Prevalence: 3	Detectability: 3	Technical: 2	Business Specific
Attackers will often attempt to find unpatched flaws, common endpoints or unprotected files and directories to gain unauthorized access or knowledge of the system.		Security misconfiguration can happen at any level of the API stack from the network to the application level. Automated tools are available to detect and exploit misconfigurations such as unnecessary services or legacy options.		Security misconfigurations can expose not only sensitive user data but also system details what may lead to full server compromise.	

Is the API Vulnerable?

The API might be vulnerable if:

- Appropriate security hardening is missing across any part of the application stack or if it has improperly configured permissions on cloud services.
- The latest security patches are missing or the systems are out of date.
- Unnecessary features are enabled (e.g. HTTP verbs).
- Transport Layer Security (TLS) is missing.
- Security directives are not sent to clients (e.g. [Security Headers](#)).
- A Cross-Origin Resource Sharing (CORS) policy is missing or improperly set.
- Error messages include stack traces or other sensitive information is exposed.

Example Attack Scenarios

Scenario #1

An attacker finds the `.bash_history` file under the root 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 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 (i.e 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 (profile images).

A7:2019 Security Misconfiguration

How To Prevent

The API lifecycle should include:

- A repeatable hardening process leading to fast and easy deployment of a properly locked down environment.
- A task to review and update configurations across the entire API stack. The review should include, orchestration files, API components and cloud services (e.g. S3 bucket permissions).
- A secure communication channel for all API interactions access to static assets (e.g. images).
- An automated process to continuously assess the effectiveness of the configuration and settings in all environments.

References

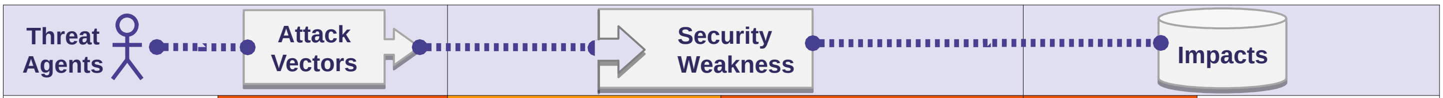
OWASP

- OWASP Secure Headers Project
- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes

External

- CWE-2: Environmental Security Flaws
- CWE-16: Configuration
- CWE-388: Error Handling
- Guide to General Server Security, NIST
- Let’s Encrypt: a free, automated, and open Certificate Authority

A8:2019 Injection



API Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business Specific
Attackers will feed the API with hostile data through whatever injection vectors are available (e.g. direct input, parameters, integrated services, etc.) expecting it to e sent to an interpreter.		Injection flaws are very common and often found in SQL, LDAP or NoSQL queries, OS commands, XML parsers and ORM. These flaws are easy to discover when reviewing the source code. Attackers can use scanners and fuzzers.		Injection can lead to information disclosure and data loss. It may also lead to DoS or complete host takeover.	

Is the API Vulnerable?

The API is vulnerable to injection flaws if:

- Client-supplied data is not validated, filtered or sanitized by the API.
- Client-supplied data is directly used or concatenated to SQL/NoSQL/LDAP queries, OS commands, XML parsers. and Object Relational Mapping (ORM)/Object Document Mapper (ODM).
- Data coming from external systems (e.g. integrated systems) is not validated, filtered or sanitized by the API.

Example Attack Scenarios

Scenario #1

Inspecting the web browser network traffic an attacker identifies the following API request responsible to start the recovery password workflow:

```
POST /api/accounts/recovery
{"username": "john@somehost.com"}
```

The attacker replays the request with a different payload

```
POST /api/account/recovery
{"email": "john@somehost.com';WAITFOR DELAY '0:0:5'--"}
```

This time the response took ~5 seconds confirming the API is vulnerable to SQL injection. Exploiting this vulnerability the attacker was able to gain unauthorized access to the system.

Scenario #2

Firmware of a parental control device provides the endpoint `/api/CONFIG/restore` which expects an appId to be sent as a multipart parameter. Using a decompiler an attacker finds out that the appId is passed directly into a system call without any sanitization:

```
snprintf(cmd, 128, "%srestore_backup.sh /tmp/postfile.bin %s %d",
         "/mnt/shares/usr/bin/scripts/", appid, 66);
system(cmd);
```

The following command allows the attacker to shutdown any device with the same vulnerable firmware:

```
$ curl -k "https://${deviceIP}:4567/api/CONFIG/restore" -F
'appid=$(/etc/pod/power_down.sh)'
```

A8:2019 Injection

Scenario #3

We have an application with basic CRUD functionality for operations with bookings. An attacker managed to identify that NoSQL injection might be possible through `bookingId` query string parameter in the delete booking request. This is how the request looks like: `DELETE /api/bookings?bookingId=678`.

The API server uses the following function to handle delete requests:

```
router.delete('/bookings', async function (req, res, next) {
  try {
    const deletedBooking = await Bookings.findOneAndRemove({_id' : req.query.bookingId});
    res.status(200);
  } catch (err) {
    res.status(400).json({
      error: 'Unexpected error occured while processing a request'
    });
  }
});
```

Attacker intercepted the request and changed `bookingId` query string parameter as below, the attacker managed to delete another user booking:

```
DELETE /api/bookings?bookingId[$ne]=678
```

How To Prevent

Preventing injection requires keeping data separate from commands and queries.

- Perform data validation using a single, trustworthy, 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 which 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.

References

OWASP

- [OWASP Injection Flaws]
- [SQL Injection]
- [NoSQL Injection Fun with Objects and Arrays]
- [Command Injection]

External

- [CWE-77: Command Injection]
- [CWE-89: SQL Injection]
- [HOW TO: Command Injection, HackerOne]

A9:2019 Improper Assets Management

<table>
<tr><th colspan="2">Threat Agents / Attack Vectors</th><th colspan="2">Security Weakness</th><th colspan="2">Impacts</th></tr>
<tr><td>API Specific</td><td>Exploitability: 3</td><td>Prevalence: 3</td><td>Detectability: 2</td><td>Technical: 2</td><td>Business Specific</td></tr>
<tr><td colspan="2">Old API versions are usually unpatched and are an easy way to compromise systems without having to fight state of the art security mechanisms that might be in place to protect the most recent API versions.</td><td colspan="2">Outdated documentation makes it more difficult to find and/or fix vulnerabilities. Lack of assets inventory and retire strategies leads to running unpatched systems and leakage of sensitive data. It’s common to find unnecessarily exposed API hosts because of modern concepts like microservices, which make applications easy to deploy and independent (e.g. cloud computing, k8s)</td><td colspan="2">Attackers may gain access to sensitive data or even takeover the server through old, unpatched API versions connected to the same database.</td></tr>
</table>

Is the API Vulnerable?

The API might be vulnerable if:

- The purpose of an API host is unclear, and there are no explicit answers to the following questions:
  - Which environment is the API running in (e.g. production, staging, test, development)?
  - Who should have network access to the API (e.g. public, internal, partners)?
  - Which API version is running?
  - What data is gathered and processed by the API (e.g. PII)?
  - What's the data flow?
- There is no documentation or the existing documentation is not updated.
- There is no retirement plan for each API version.
- Hosts inventory is missing or outdated.
- Integrated services inventory, either first- or third-party, is missing or outdated.
- Old or previous API versions are running unpatched.

Example Attack Scenarios

Scenario #1

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 user.

Scenario #2

A social network implemented a rate-limiting mechanism that blocks attackers from using brute-force to guess reset password tokens. This mechanism wasn’t implemented as part of the API code itself, but in a separate component between the client and the and the official API (`www.socialnetwork.com`). A researcher found a beta API host (`www.mbasic.beta.socialnetwork.com`) that runs the same API, including the reset password mechanism, but the rate limiting mechanism was not in place. The researcher was able to reset the password of any user by using a simple brute-force to guess the 6 digits token.

A9:2019 Improper Assets Management

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.

References

External

- CWE-1059: Incomplete Documentation
- OpenAPI Initiative

A10:2019 Insufficient Logging & Monitoring

Threat Agents / Attack Vectors		Security Weakness		Impacts	
API Specific	**Exploitability: 2**	**Prevalence: 3**	**Detectability: 1**	**Technical: 2**	**Business Specific**
Attackers take advantage of lack of logging and monitoring to abuse systems without being noticed.		Without logging and monitoring or with insufficient logging and monitoring it is almost impossible to track suspicious activities and respond to them in a timely fashion.		Without visibility over on going malicious activities, attackers have plenty of time to fully compromise systems.	

Is the API Vulnerable?

The API is vulnerable if:

- It does not produce any logs, the logging level is not set correctly or log messages do not include enough detail.
- Log integrity is not guaranteed (e.g. Log Injection).
- Logs are not continuously monitored.
- API infrastructure is not continuously monitored.

Example Attack Scenarios

Scenario #1

Access keys of an administrative API 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.

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.

A10:2019 Insufficient Logging & Monitoring

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 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 earlier.

References

OWASP

- OWASP Logging Cheat Sheet
- OWASP Proactive Controls: Implement Logging and Intrusion Detection
- OWASP Application Security Verification Standard: V7: Error Handling and Logging Verification Requirements

External

- CWE-223: Omission of Security-relevant Information
- CWE-778: Insufficient Logging

+D

What’s Next for Developers

What's Next for DevSecOps

+DAT Methodology and Data

Overview

Since the AppSec industry has not been specifically focused on most recent applications architectures, in which APIs play an important role, compiling a list of the ten most critical API security risks based on public data call would have been a hard task. Despite no public data call was made, the resulting top 10 list is still based on publicly available data, security experts contributions and open discussion with the security community.

Methodology and Data

In the first phase, publicly available data regarding APIs security incidents were collected, reviewed, and categorized by a group of security experts. Such data were collected from bug bounty platforms and vulnerability databases, within a one-year-old time frame. It was used for statistical purposes.

In the next phase, security practitioners with penetration testing experience were asked to compile their own top ten list.

The OWASP Risk Rating Methodology was used to do the Risk Analysis. The scores were discussed and reviewed between the security practitioners. For considerations on this matters, please refer to the API Security Risks section.

The first draft of the OWASP API Security Top 10 2019 resulted from a consensus between statistical results from phase one and the security practitioners lists. This draft was then submitted to appreciation and review by another group of security practitioners, with relevant experience in the API security fields.

The OWASP API Security Top 10 2019 was first presented in the OWASP Global AppSec Tel Aviv event (May 2019). Since then it is available on GitHub for public discussion and contributions.

The list of contributors is available in the Acknowledgments section.

+ACK

Ackownledgment

Acknowledgments to Contributors

We’d like to thank contributors who contribute publicly on GitHub or via other means:

- 007divyachawla
- anotherik
- caseysoftware
- Chris Westphal
- dsopas
- emilva
- ErezYalon
- flascelles
- IgorSasovets
- Inonshk
- JonnySchnittger
- jmanico
- jmdx
- kozmic
- Matthieu Estrade
- PauloASilva
- pentagramz
- philippederyck
- pleothaud
- Sagar Popat
- thomaskonrad
- xycloops123