OWASP API security top 10 explained

Table of Contents

Executive Summary

APIs have evolved significantly since their early days when just a few companies used them to address a limited set of needs. Their use has exploded in the past couple years, since APIs are critical to digital transformation and automation efforts. Application environments of all types rely on them, as do businesses across all industries and size, for a broad set of use cases.

Given that APIs are expressly used to connect critical services and data, hackers have honed in on APIs as a primary attack vector. High-profile breaches and "leaky APIs" have plagued companies from Peloton and Experian to Facebook and Panera. Gartner predicts that "By 2022, API abuses will move from an infrequent to the most-frequent attack vector, resulting in data breaches for enterprise web applications."

Seeing the increase in API-related security incidents and breaches, the Open Web Application Security Project (OWASP) released the API Security Top 10 to raise awareness about the most common API security threats organizations need to guard against.

This paper provides a detailed review of each threat outlined in the OWASP API Security Top 10, including examples and insight to help you understand how to protect your organization from the threats targeting APIs and API-based applications.

API1:2019 Broken Object Level Authorization

Description

APIs often expose endpoints that handle object identifiers, creating a wide potential attack surface. Object level authorization is an access control mechanism usually implemented at the code level to validate a user's ability to access a given object. Authorization and access control mechanisms in modern applications are complex and wide-spread. Even if an application implements a proper infrastructure for authorization checks, developers often forget to apply these checks before accessing an object.

Attackers can easily exploit API endpoints that are vulnerable to broken object level authorization (BOLA) by manipulating the ID of an object that is sent within an API request. These vulnerabilities are extremely common in API-based applications because the server component usually does not fully track the client's state. Instead, the server component usually relies on parameters like object IDs sent from the client, to decide which objects can be accessed.

Any access of unauthorized data is severe, regardless of its data classification or data sensitivity. These types of authorization flaws are also not easily detectable with automated static or dynamic testing.

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. These checks should be made continuously throughout a given session to validate that the logged-in user has access to perform the requested action on a requested object.

Potential Impact

Failure to enforce authorization at the object level or broken improper object level authorization can lead to data exfiltration as well as unauthorized viewing, modification, or destruction of data. BOLA can also lead to full account takeover such as in cases where an attacker can compromise a password reset flow and reset credentials of an account they aren't authorized to.

Example

Legitimate – userId matches in the query parameter and request	**Attack** - Attacker changes the userId in the query parameter
Request: GET /v1/customers/15981?userId=207939055 HTTP/1.1 **Authorization:** Bearer gwwh1Y4epjv9Y **Cookie:** _ga=GA1.3.630674023.1502871544; _gid=GA1.2.1579405782.1502871544;userId= 207939055**Host:** payments-api.dnssf.com **X-Forwarded-For:** 54.183.50.90	**Request:** GET /v1/customers/15981?userId=207938044 HTTP/1.1 **Authorization:** Bearer gwwh1Y4epjv9Y **Cookie:** _ga=GA1.3.630674023.1502871544; _gid=GA1.2.1579405782.1502871544;userId= 207939055**Host:** payments-api.dnssf.com **X-Forwarded-For:** 54.183.50.90

Response:
200 OK
 	
{
userId: 207939055,
firstName: "John",
lastName: "Smith",
email: "john.smith@acme.com",
phoneNumber: "+1650123123"
}
	Response:
	200 OK
	
	{
	userId: 207938044,
	firstName: "David",
	lastName: "Miller",
	email: "david.miller@example.com",
	phoneNumber: "+1912456456"
	}

In this example, the backend logic of the application queries the database with the **userId in the query parameter** while verifying the authorization with the **userId in the cookie**. Under normal conditions these two values should match, however, an attacker could simply modify the **userId value in the query parameter** in order to access unauthorized data.

The attacker (John Smith) is logged in with userId 207939055. When the attacker changes the userId in the query parameter to userId 207938044 the application does not validate that the userId of the authenticated user matches that of the record being requested in the query parameter or whether the authenticated user is authorized to view that given record. As a result, the database backend returns the record for David Miller as opposed to John Smith.

If the **userIds** are sequential the attacker can simply enumerate the query parameter **userId** value to scrape, or exfiltrate, large amounts of data, particularly if rate limits aren't enforced.

Real World Example

How I could have hacked your Uber account
In 2019 a security researcher disclosed a BOLA vulnerability that would have enabled an attacker to take over any user account on Uber. By exploiting the vulnerability, the attacker could access another user's account to track the target user's location, take rides, and more. The attacker could also exploit the BOLA vulnerability to harvest Uber mobile app access tokens, and then use those access tokens to take over Uber Driver and Uber Eats accounts. The Uber application userId could be easily enumerated by supplying a user's phone number or email address in another API request.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways miss these types of attacks because they don't understand API context and don't baseline normal API usage. In this case, these tools do not know that the **userId** in the query parameter and the **userId** in the cookie should match. Also consider that since this is not a known, predictable attack pattern like a code injection where basic pattern matching and message filtering can be employed, it won't be identified by the signatures used by a WAF or API gateway.

How to Protect Against BOLA Attacks

In order to prevent BOLA attacks an API security solution must be able to learn the business logic of an API and detect when one authenticated user is trying to gain unauthorized access to another user's data. In this particular case, two objects should match and that the authenticated user is authorized to access the requested object. This kind of detection requires the analysis of large amounts of API traffic in order to gain context and understand the normal usage for each API. A solution with a baseline of normal usage can identify abnormal behavior like an attacker manipulating the **userId** in a query parameter in *GET requests*, or a **userId** variable within a message body of *POST requests*.

API2:2019 Broken User Authentication

Description

Authentication in APIs is a complex and confusing topic. Software and security engineers might have misconceptions about what the boundaries of authentication are and how to correctly implement it. Prompting users or machines for credentials and additional authentication factors may also not be possible in direct API communication. In addition, authentication mechanisms are easy targets for attackers, particularly if the authentication mechanisms are fully exposed or public. These two points make the authentication component potentially vulnerable to many exploits. Advanced attacks that target authentication include brute-forcing (of authentication), credential stuffing and credential cracking.

Authentication in APIs has two sub-issues:

1. Lack of protection mechanisms - API 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 or implemented without considering the attack vectors, or the mechanism is not appropriate for the use case. As an example, an authentication mechanism designed for IoT devices is typically not the right choice for a web application like an eCommerce site.

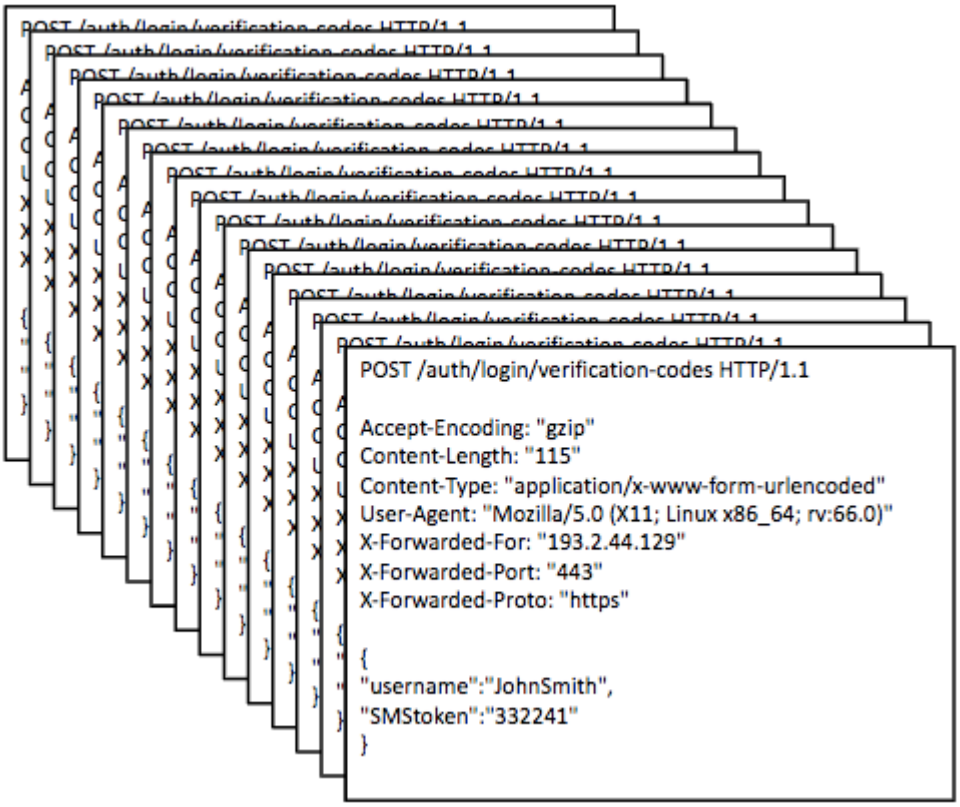
Technical factors leading to broken authentication in APIs are numerous and include:

- Weak password complexity
- Short or missing password history
- Excessively high or missing account lockout thresholds
- Failure to provision unique certificates per device in certificate-based authentication
- Excessively long durations for password and certificate rotations
- Authentication material exposed in URLs and *GET requests*
- Authentication tokens with insufficient entropy
- Use of API keys as the only authentication material
- Failure to validate authenticity of authentication material
- Insecure JSON Web token (JWT) configuration such as use of weak digital signature algorithm or missing signatures
- Use of small key sizes in encryption or hashing algorithms
- Use of weak or broken ciphers
- Use of algorithms that are inappropriate for the use case, such as use of hashing algorithms rather than password-based key derivation functions (PBKDF).
- Failure to step-up authentication if authentication flows are being targeted, such as dynamically challenging with CAPTCHA or second factor authentication (2FA) material.

Potential Impact

An attacker who is able to successfully exploit vulnerabilities in authentication mechanisms can take over user accounts, gain unauthorized access to another user's data, or make unauthorized transactions as another user. Similarly, APIs may be designed explicitly for machine communication, or direct API communication. An attacker who compromises that authentication mechanism or authenticated session can potentially gain access to all of the data that machine identity is entitled to access. There are also variants of this type of attack in cloud-native design with compromises of workload authentication and server-side API metadata services.

Example



Common examples of attacks targeting broken user authentication include API enumeration and brute-forcing attacks that make high volumes of API requests with minor changes. These attacks may also target broken or weak authentication.

As an example, password recovery mechanisms often send an SMS to a user's phone with a reset token consisting of a series of numbers. An attacker can initiate a password reset, and if the API does not implement rate limiting, the attacker can enumerate (or "guess") the password reset token until they get a successful response. Depending on the throughput of the target API endpoint, an attacker may be able to iterate through thousands or millions of different combinations within a few minutes.

Real World Example

Unpacking the Parler Data Breach

In 2021, Salt analysis of the Parler data breach and the general consensus of media outlets and hacktivists found that Parler's authentication was at least partially absent. This flaw, along with other security flaws in the Parler platform, enabled the scraping of at least 70TB of data. Based on what the hacktivist shared publicly, at least one endpoint was available without authentication which provided access to user data without requiring authentication. In Parler's case, these APIs likely were not intended to be anonymous, public APIs. The APIs allowed direct access to Parler user profile information and user content, including message posts, images, and videos. It is unlikely that Parler would have intended or configured these APIs and pages to be accessible without authentication.

Some reports indicated there was a security misconfiguration as a result of Twilio integration that was later decommissioned. Allegedly, some of the archivists used this to bypass multifactor (MFA) authentication during account creation and extract data. The issue was later disputed by the hacktivist, and Twilio representatives have also stated it was false. An MFA misconfiguration would further fuel the debate whether the Parler data was truly public and Parler APIs were lacking authentication.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs don't typically enforce authentication at a granular level and may only verify presence of a session identifier or authentication token in a given request. API gateways may enforce authentication as part of API management access control policies, but that presumes owning teams have defined policy appropriately. There is often an operational breakdown between teams creating APIs, teams publishing APIs, and teams securing APIs. Even still, API gateways lack understanding of what authentication is proper for an API in a given use case. Traditional security controls also lack capabilities to track attack traffic over time, which is necessary to decipher the different forms of advanced attacks targeting authentication such as credential stuffing and credential cracking. They will often rely on excessive API consumption rates to identify basic brute-force attack attempts.

How to Protect Against Broken User Authentication Attacks

In order to protect against broken user authentication attacks, an API security solution must be able to profile the typical authentication sequence for every API flow. The solution can then detect abnormal behavior such as missing credentials, missing authentication factors, or authentication calls that are out of sequence. Determining the baseline and identifying abnormal behavior can only be done by analyzing large amounts of production API traffic. This form of analysis is critical for mitigating advanced attacks that target authentication such as credential stuffing and credential cracking.

API3:2019 Excessive Data Exposure

Description

Exploitation of Excessive Data Exposure is simple, and is usually performed by sniffing the traffic to analyze the API responses, looking for sensitive data exposure that should not be returned to the user.

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. Traditional security scanning and runtime detection tools will sometimes alert on this type of vulnerability, but they can't differentiate between legitimate data returned from the API and sensitive data that should not be returned. This requires a deep understanding of the application design and API context.

Potential Impact

APIs often send more information than is needed in an API response and leave it up to the client application to filter the data and render a view for the user. An attacker can sniff the traffic sent to the client to gain access to potentially sensitive data that can include information such as account numbers, email addresses, phone numbers, and access tokens.

Example

Legitimate request – user retrieving stored credit card information	**Response with data exposure** - HTTP response to the API call contains sensitive data in the message body
Request: POST /payments/storedcard/json HTTP/1.1 Host: payments.host.com Connection: close Content-Length: 78 Cache-Control: max-age=0 Origin: https://payments.host.com Content-Type: application/x-www-form-urlencoded User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36	**Response:** HTTP/1.1 200 OK Cache-Control: no-cache, no-store, max-age=0, must-revalidate Pragma: no-cache Expires: Mon, 01 Jan 1990 00:00:00 GMT Date: Wed, 27 Jan 2021 15:43:39 GMT Content-Type: application/json; charset=utf-8 X-Frame-Options: SAMEORIGIN X-XSS-Protection: 1; mode=block Connection: close Content-Length: 55
(KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9 Referer: https://payments.host.com/methods/ Accept-Encoding: gzip, deflate Accept-Language: en-US,en;q=0.9 Cookie: session=kjasdnfklnf01922wndlasdjkfnz-0f71k9013u18901u2mklsduhz12234d4D	[{"PAN":"4111111123454321","status":"ok","CVV":"1234"}]

In the example, the client-side code running in the user's web browser is submitting a POST request to a backend API to retrieve stored payment information. In this case, the API is retrieving stored credit card information, specifically primary account number (PAN) and card verification value (CVV) code. Within the world of credit card handling and payment processing, this type of data is deemed to be sensitive as part of PCI-DSS and must be protected appropriately. The scope of what is necessary for protection varies depending on exposure of the cardholder data environment, or where the data is stored, processed, or transmitted.

This sensitive data sharing may be intentional as part of the design or necessary for functionality. As a result, organizations augment with additional security controls such as stronger authentication or encrypted transport to ensure the data is sufficiently protected. In the example, you can see additional HTTP security headers to help protect the data, such as x-frame-options for mitigating cross-frame scripting attacks and x-xss-protection for mitigating cross-site scripting attacks. Some organizations may also mask data being returned to a client to avoid cases where someone intercepts traffic or views data outside of the intended client application. Relying on the client-side code to filter or obscure such sensitive data is typically not appropriate since attackers regularly bypass client-side web application and mobile application code and call APIs directly.

Real World Example

Flaw left user data of 2 million Bounceshare customers vulnerable to hack

In 2019 a security researcher found that by passing a phone number in an API request the Bounceshare application would return an access token and RiderId associated with the account for that phone number. An attacker could automate this process by using a phone number dump found online and a script allowing them to gain unauthorized access to multiple user accounts. Once logged in to a target user's Bounceshare account the attacker would have access to sensitive information such as their driver's license, email address, and photos. If the target user had linked their Paytm account for payments, the attacker could also see the user's balance and book rides from the target user's account.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways have no context to identify sensitive data being sent over an API and therefore do not understand the exposure risk of the data being sent. Typically, they will employ basic pattern matching and message filtering to identify sensitive data types, also referred to as regular expression (regex) patterns. While these types of filters can catch well-defined sensitive data types such as PANs or social security numbers (SSNs), they do not understand API context and business logic flows. They will flag any data that matches the pattern, regardless of whether it is necessary to block the request, encrypt payloads or obscure data. API gateways are often used to mediate API calls that contain sensitive data, and this may be necessary as part of an overarching enterprise architecture, application design or systems integration. Blocking or masking sensitive data wholesale often breaks functionality as a result leaving security teams reluctant to aggressively use these capabilities in proxies in favor of relying on the API/application layer to control exposure.

How to Protect Against Excessive Data Exposure

An API security solution must be able to identify and report on the large variety of sensitive data types that can be sent in API requests and responses. These solutions must also have the ability to baseline and track API access per endpoint and per user in order to identify excessive consumption of sensitive data. These solutions must also provide API context and a range of response actions so that not every transmission of sensitive data results in an alert or blocked request.

API4:2019 Lack of Resources & Rate Limiting

Description

API requests consume resources such as network, CPU, memory, and storage. The amount of resources required to satisfy a request greatly depends on the input from the user and the business logic of the endpoint. APIs do not always impose restrictions on the size or number of resources that can be requested by the client or user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but it also leaves the door open to brute-forcing and enumeration attacks against APIs that provide authentication and data fetching functionality. This includes automated threats like credential cracking and token cracking among others.

Potential Impact

When determining impact, it is best to break down the impact of this issue into two sub-components:

1. With respect to lack of resource limiting, an attacker can craft a single API call that can overwhelm an application, impacting the application's performance and responsiveness or causing it to become unresponsive. This type of attack is sometimes referred to as an application-level DoS. These types of attacks not only impact availability though. They may also expose the system, application or API to authentication attacks and excessive data leakage.

2. With respect to lack of rate limiting, an attacker may craft and submit high volumes of API requests to overwhelm system resources, brute force login credentials, quickly enumerate through large data sets, or exfiltrate large amounts of data.

Example

Legitimate – max_return and page_size request attributes are normal	**Attack** – Attackers modify the request to return an abnormally high response size
POST /example/api/v1/provision/user/search HTTP/1.1 User-Agent: AHC/1.0 Connection: keep-alive Accept: */* Content-Type: application/json; charset=UTF-8 Content-Length: 131 X-Forwarded-For: 10.93.23.4	POST /example/api/v1/provision/user/search HTTP/1.1 User-Agent: AHC/1.0 Connection: keep-alive Accept: */* Content-Type: application/json; charset=UTF-8 Content-Length: 131 X-Forwarded-For: 10.93.23.4

{
"search_filter":
"user_id=exampleId_100",
"max_return": "250",
"page_size": "250",
"return_attributes": [
 	
]
}
	{
	"search_filter":
	"user_id=exampleId_100",
	"max_return": "20000",
	"page_size": "20000",
	"return_attributes": [
	
	]
	}

In the example above of a lack of resource limit, the attacker has increased the max_return and page_size values for the search filter from 250 to 20,000. This increase would cause the application to return an excessive number of items in response to a query. It could also cause the application to slow down or become unresponsive for all users.

Real World Example

Checkmarx Research: SoundCloud API Security Advisory

In 2020 the Checkmarx research team found that SoundCloud had not properly implemented rate limiting for the /tracks endpoint of the api-v2.soundcloud.com API. Since no validation was performed for the number of track IDs in the ids list, an attacker could manipulate the list to retrieve an arbitrary number of tracks in a single request and overwhelm the server. Under normal conditions the request issued by the SoundCloud WebApp includes 16 track IDs in the ids query string parameter. The researcher was able to manipulate the list to retrieve up to 689 tracks in a single request causing the service response time to increase by almost 9x. According to Checkmarx "This vulnerability could be used to execute a Distributed Denial of Service (DDoS) attack by using a specially crafted list of track IDs to maximize the response size, and issuing requests from several sources at the same time to deplete resources in the application layer will make the target's system services unavailable."

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs, API gateways, and other proxying mechanisms will commonly offer basic or static rate limiting which are difficult to enforce at scale. Security teams may not know enough about the application design to know what "normal" looks like in order to enforce limits to thwart attackers while not impacting business functionality. WAFs and API gateways lack the context required to inform security teams on what a normal value should be for an API parameter, and they will miss attacks where an attacker manipulates a single API parameter value to overwhelm the application. These proxies may also only cover ingress, or inbound requests, as opposed to egress traffic, or outbound requests and responses.

How to Protect Against Lack of Resources & Rate Limiting Attacks

An API security solution must be able to identify calls to API endpoints and alterations to API parameter values that fall outside of normal usage. This would be done by analyzing all API traffic in order to create a baseline of typical behavior and identifying deviations that fall outside of that baseline.

In the example above an API security solution will have created a baseline of values for the **max_return** and **page_size** parameters and will identify that a value of 20,000 is abnormal. The solution could then alert on and block an attacker who crafts API requests that deviate from the baseline.

API5:2019 Broken Function Level Authorization

Description

Authorization flaws are often the result of improperly implemented or misconfigured authorization. Implementing adequate authorization mechanisms is a complex task, since modern applications can contain many types of roles, groups, and user hierarchy such as sub-users and users with more than one role. This is further complicated with distributed application architectures and cloud-native design. Broken function level authorization (BFLA) shares some similarity to BOLA in this regard, though the target with BFLA is API functions as opposed to objects that APIs interact with as in the case of BOLA. Attackers will attempt to exploit both vulnerabilities when targeting APIs in order to escalate privileges horizontally or vertically.

Attackers discover these flaws in APIs since API calls are structured and predictable, even in REST designs. This can be done in the absence of API documentation or schema definitions by reverse engineering client-side code and intercepting application traffic. Some API endpoints might also be exposed to regular, non-privileged users making them easier for attackers to discover.

Attackers can exploit these flaws by sending legitimate API requests to an API endpoint that they should not have access to or by intercepting and manipulating API requests originating from client applications. As an example, an attacker could change an HTTP method from GET to PUT. Alternatively, the attacker might also alter a query parameter or message body variable such as changing the string “users” to "admins" in an API request.

Potential Impact

Attackers exploiting broken function level authorization vulnerabilities can gain access to unauthorized resources, take over another user’s account, create/delete accounts, or escalate privileges to gain administrative access.

Example

Legitimate – POST method is correctly requested	**Attack** – Request is modified to send a DELETE method
POST /example/api/v1/provision/user/search HTTP/1.1 User-Agent: AHC/1.0	**DELETE** /example/api/v1/provision/user/search HTTP/1.1 User-Agent: AHC/1.0

Connection: keep-alive
Accept: */*
Content-Type: application/json;
charset=UTF-8
Content-Length: 131
X-Forwarded-For: 10.93.23.4
 	
{
"search_filter":
"user_id=exampleId_100",
"max_return": "250",
"page_size": "250",
"return_attributes": [
 	
]
}
	Connection: keep-alive
	Accept: */*
	Content-Type: application/json;
	charset=UTF-8
	Content-Length: 131
	X-Forwarded-For: 10.93.23.4
	
	{
	"search_filter":
	"user_id=exampleId_100",
	"max_return": “250",
	"page_size": “250",
	"return_attributes": [
	
	]
	}

In the example above, the attacker has changed the method from POST. to DELETE allowing them to delete the account associated with user_id=exampleId_100. Access to the DELETE method should have been restricted to users with administrative access but was allowed due to an inadequate authorization policy.

Real World Example

New Relic Synthetics users can escalate privileges to add or modify alerts

In 2018 Jon Bottarini found that a restricted user could make changes to alerts on Synthetics monitors without the proper permissions to do so. In fact, they could make changes with no permissions at all as a result of the privilege escalation weakness that was present in the product at that time. Exploitation involved submitting a legitimate request to an API endpoint that was otherwise not visible to the restricted user.

As part of his security research, Jon captured traffic of a privileged session using an intercepting proxy tool, Portswigger Burp Suite. In particular, this traffic included a POST request to an API endpoint and function that creates alerts on Synthethics monitors. He found that you could trap a GET request from the non-privileged session, retain the tokens and cookies for that restricted user, and alter the remainder of the trapped request to resemble the privileged POST request. This manipulation of API traffic to access functionality not visible in the UI (at all or to that user and their permissions) is a common technique attackers use to exploit function level authorization weaknesses and escalate privileges.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways lack context of API activity and therefore do not know that the attacker in the example above should not be able to send a DELETE method. This API call would be seen as legitimate and would pass through these security controls. WAFs and API gateways sometimes support explicit, statically defined message filters, often referred to as a positive security approach. However, these approaches can inhibit or break business functionality, and most organizations find them difficult to operationalize at scale. Restricting HTTP methods is also an easier task than restricting API parameters and values, the latter of which requires deeper subject matter expertise on the design of the API.

The activity in the Facebook example above would be missed by WAFs and API gateways for the same reason. These security controls would not know that the 3rd party applications should no longer have access to the deprecated or restricted API functions. Tuning the controls would have required appropriate knowledge transfer between development, operations, and security teams to implement an appropriate static filter in the appropriate proxy within the overall enterprise architecture.

How to Protect Broken Function Level Authorization Vulnerabilities

API security solutions must be able to continuously baseline typical HTTP access patterns per API endpoint and per user. With this baseline, API security solutions can identify calls with unexpected parameters or HTTP methods sent to specific API endpoints such as in the DELETE example above. It is critical that the solution is capable of baselining continuously, as APIs may go through a high rate of change as a result of modern development and release practices. API security solutions must be able to identify and prevent attackers or unauthorized users from accessing administrative level capabilities or unauthorized functionality as in the Facebook example above.

API6:2019 Mass Assignment

Description

Modern application frameworks encourage developers to use functions that automatically bind input from the client into code variables and internal objects in order to help simplify and speed up development within the framework. Attackers can use this side effect of frameworks to their advantage by updating or overwriting properties of sensitive objects that developers never intended to expose. Mass assignment vulnerabilities are also sometimes referred to as autobinding or object injection vulnerabilities.

Exploitation of mass assignment vulnerabilities in APIs requires an understanding of the application's business logic, objects relations, and the API structure. APIs expose their underlying implementation along with property names by design. An attacker can also gain further understanding by reverse engineering client-side code, reading API documentation, probing the API to guess object properties, exploring other API endpoints, or by providing additional object properties in request payloads to see how the API responds. APIs need to be exposed to some extent in order to enable functionality and data exchange. As a result, attackers are able to exploit mass assignment vulnerabilities more easily in APIs and API-based applications.

Objects in modern applications can contain many properties, some of which can be updated directly by the client such as user first name or address details, and other sensitive properties that should not, such as user access entitlements.

An API endpoint is vulnerable if it automatically converts client provided data into internal object properties without considering the sensitivity and the exposure level of these properties. Binding client provided data like JSON attribute-values pairs to data models without proper filtering of properties based on an allowlist usually leads to mass assignment vulnerability.

Potential Impact

An attacker exploiting mass assignment vulnerabilities can update object properties that they should not have access to allowing them to escalate privileges, tamper with data, and bypass security mechanisms.

Example

Legitimate - Client sends a legitimate request

```
PUT /api/v2/users/5deb9097 HTTP/1.1

User-Agent: Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/78.0.3904.108
Safari/537.36
X-Forwarded-For: 19.42.129.253

{
  "_id": "5deb9097",
  "address": "******, NY City, NY",
  "company_role": "Investment Services",
  "email": "******",
  "first_name": "******",
  "full_name": "******",
  "job_title": "Broker",
  "last_name": "******",
  "phone_number": "******"
}
```

Attack – Attackers sends the same request but adds the admin role in the request body

```
PUT /api/v2/users/5deb9097 HTTP/1.1

User-Agent: Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/78.0.3904.108
Safari/537.36
X-Forwarded-For: 19.42.129.253

{
  "_id": "5deb9097",
  "address": "******, NY City, NY",
  "company_role": "Investment Services",
  "email": "******",
  "first_name": "******",
  "full_name": "******",
  "is_admin": true,
  "is_sso": true,
  "job_title": "Broker",
  "last_name": "******",
  "permission_type": "admin",
  "phone_number": "******",
  "role": "admin",
  "sso_type": "admin",
  "system_user_type": "admin",
  "system_user_type_cd": 2,
  "user_type": "admin",
  "user_type_cd": 10
}
```

In the example above, the attacker has changed the API call to update their account, escalate their role and privileges to an "admin" role, and bypass single-sign on (SSO). If successful, the attacker can then perform actions within the application as an administrator.

Real World Example

Hacking rails/rails repo

In 2012, a security researcher by the name of Egor Homakov found a critical mass assignment vulnerability in GitHub's public key form update function. This mass assignment vulnerability allowed any user to associate their public key to a given GitHub public or private repo and take ownership of that repo. The attack made use of one of GitHub's public APIs to find the identifier ID for a given repo. An attacker could then pair this identifier with their own public key and submit the data to GitHub's public key form update function to exploit the vulnerability.

Egor attempted to report the issue to GitHub prior to GitHub having a responsible disclosure policy. Egor felt his report wasn't being taken seriously or being addressed quickly enough, and so he chose

to exploit the vulnerability, taking ownership of the public rails repo hosted on GitHub to prove their point. This takeover activity and resulting swarm of comments is still visible in the rails git commit history. GitHub resolved the issue within roughly an hour after Egor's exploit. The vulnerability was very simple to exploit, which may have been why it was so overlooked. It was also a catalyst for GitHub developing a responsible disclosure policy that still stands today, and which has evolved into GitHub's public bug bounty program.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways lack context of API activity and intended business logic. They can't know if the API caller in the example above should be able to send a request using the PUT method with additional parameters, failing to differentiate between a legitimate call and malicious activity. To these traditional controls, this API call looks normal. They lack the context to know that this user is not an administrator, and the user should not have access to these additional parameters. At best, a WAF or API gateway may be able to offer basic message filtering mechanisms to block this type of request wholesale. However, additional parameters may be necessary for other users and other use cases. It would also require detailed knowledge upfront from development teams on the design and intended use of the API so that operational teams can implement even basic message filters.

How to Protect Against Mass Assignment Attacks

API security solutions must be able to identify anomalous API activity where attackers send manipulated API requests with unauthorized parameters. To do this, API security solutions must be able to continuously baseline normal API behavior and identify when additional parameters are passed in API calls that fall outside of typical behavior. API Security solutions should also be able to identify attackers as they probe the API during their reconnaissance phase to gain an understanding of the API structure and business logic.

API7:2019 Security Misconfiguration

Description

This issue is a catch-all for a wide range of security misconfigurations that often negatively impact API security as a whole and introduce vulnerabilities inadvertently. Some examples of security misconfigurations include insecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, overly permissive Cross-Origin resource sharing (CORS), and verbose error messages.

Potential Impact

Attackers can exploit security misconfigurations to gain knowledge of the application and API components during their reconnaissance phase. Detailed errors such as stack trace errors can expose sensitive user data and system details that can aid an attacker during their reconnaissance phase to find exploitable technology including outdated or misconfigured web and application servers.

Attackers also exploit misconfigurations to pivot their attacks against APIs, such as in the case of an authentication bypass resulting from misconfigured access control mechanisms.

Many automated tools are available to detect and exploit common or known misconfigurations such as unnecessary services or legacy options, though where you detect them in the technology stack varies greatly. Commonly used vulnerability scanners may only scan a running server for known vulnerabilities and misconfigurations in published software, usually in the form of CVE IDs. However, they don't provide the complete picture, since misconfigurations can exist in underlying code, in third party dependencies, or in integrations with other enterprise architecture. As a result, organizations will often employ a barrage of security testing tooling in build pipelines to catch as much as possible prior to production deployment. There are certainly cases where security misconfiguration can be the result of something basic like a missing patch, but some misconfigurations are far stealthier and obscured by complex architectures.

Example

Legitimate – Client sends a legitimate request	**Attack** – Attackers modify the API request and specify an invalid identifier, resulting in a detailed exception error

GET /api/v2/network/connections/593065
HTTP/1.1
Accept: application/json, text/plain, */*
Accept-Encoding: gzip
 	
HTTP/1.1 200 OK
{
"status":"success",
}
	GET
	/api/v2/network/connections/5930aaaaa
	HTTP/1.1
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip
	
	HTTP/1.1 500 Server Error
	{
	"status":"failure",
	"statusMessage":"An error occurred while validating input: validation error: unexpected content \"593065d1\" ({com.tibco.xml.validation}COMPLEX_E_UNEXPECTED_CONTENT) at /{http://www.tibco.com/namespaces/tnt/plugins/json}ActivityOutputClass[1]/searchSvcReqsByRepReq[1]/search[1]/status[1]/aaaa[1]\ncom.tibco.xml.validation.exception.UnexpectedElementException: unexpected content \"aaaa\"

\n\tat com.tibco.xml.validation.state.a.a.a(CMElementValidationState.java:476)
\n\tat com.tibco.xml.validation.state.a.a.a(CMElementValidationState.java:270)
\n\tat com.tibco.xml.validation.state.driver.ValidationJazz.c(ValidationJazz.java:993)
\n\tat com.tibco.xml.validation.state.driver.ValidationJazz.b(ValidationJazz.java:898)
\n\tat .....

In the example above, the attacker modified the connectionId parameter of the GET request to an API, causing the application server to respond with a detailed exception error with stack trace information. These errors can include information about the application environment such as software vendor names, software packages used, software versions, and lines of code within the backend server-side code that the error resulted. All of this information is invaluable to an attacker who is performing reconnaissance in order to gain an understanding of infrastructure that serves the applications and APIs as well as the application code itself in order to discover other potentially exploitable vulnerabilities.

Real World Example

A Technical Analysis of the Capital One Cloud Misconfiguration Breach

The Capital One breach in 2019 was a chained attack, that was the result of a few issues, the primary vector being a misconfigured WAF. Through other sources, we know that ModSecurity, an open-source WAF, was likely used to protect certain Capital One web applications and APIs. The WAF was not appropriately configured or tuned for Capital One's AWS environment and was overly permissive. As a result, an attacker was able to bypass the WAF's content inspection and message filtering using a well-crafted injection that targeted the backend AWS cloud metadata service. Harvesting metadata typically only available to running workloads, the attacker was able to pivot their attack and compromise other systems within the AWS cloud environment, commonly referred to as server-side request forgery attack.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways are not able to identify the modification to the connectionId parameter in the example above since it does not match a pattern of a typical attack. These tools also lack the context to know that the modified connectionId parameter does not match typical usage for this parameter or that it would result in an application server error, and therefore would miss this attack. These tools would also not alert on the excessive data sent in the API response since these traditional security controls lack context about this information to know that it is potentially sensitive and should not be returned in error messages. It's also not uncommon for traditional security controls to only check client requests to APIs, or inbound traffic, and not the server response back to the client, or outbound traffic.

How to Protect Against Security Misconfiguration Vulnerabilities

An API security solution must be able to identify misconfigurations and security gaps for a given API and its serving infrastructure. It must suggest remediation when manipulation attempts are made, and the application server itself is not configured to reject the request or mask sensitive data in the response. An API security solution must be able to analyze all API activity and establish a baseline of typical API activity so that it can help identify excessive data and sensitive data sent in error messages. These solutions also help to identify the early activity of an attacker who is performing reconnaissance in order to look for security misconfigurations and learn more about the API structure and logic. Early detection defines the difference between a security incident, where you catch attacker behavior early in their methodology and stop it, as opposed to a breach, where an attacker is able to successfully exfiltrate data or compromise systems.

API8:2019 Injection

Description

Injection flaws are very common in the web application space, and they carry over to web APIs. Structured Query Language (SQL) injection is one of the most well-known, but there are other injection varieties that can impact a range of interpreters and parsers beyond just SQL including, Lightweight Directory Access Protocol (LDAP), NoSQL, operating system (OS) commands, Extensible Markup Language (XML), and Object-Relational Mapping (ORM).

Attackers exploit these injection vulnerabilities by sending malicious data to an API that is in turn processed by an interpreter or parsed by the application server and passed to some integrated service, such as a database management system (DBMS) or a database-as-a-service (DBaaS) in the case of SQL injection (SQLi). The interpreter or parser is essentially tricked into executing the unintended commands since they either lack the filtering directly or expect it to be filtered by other server-side code.

Potential Impact

Injection can lead to a wide range of impacts including information disclosure, data loss, denial of service (DoS), or complete host takeover. In many cases, successful injection attacks expose large sets of unauthorized sensitive data. Attackers may also be able to create new functionality, perform remote code execution, or bypass authentication and authorization mechanisms altogether.

Example

Legitimate - Client sends a legitimate request	**Attack** – Attackers sends the same request but adds an injection attempt

Request:
GET /v1/customers HTTP/1.1
 	
Authorization: Bearer gwwh1Y4epjv9Y
 	
Cookie: _ga=GA1.3.630674023.1502871544; _gid=GA1.2.1579405782.1502871544;userId=207939055
Host: payments-api.dnssf.com
X-Forwarded-For: 54.183.50.90
 	
{
userId: “207939055”
}
	Request:
	POST/v1/customers HTTP/1.1
	
	Authorization: Bearer gwwh1Y4epjv9Y
	
	Cookie: _ga=GA1.3.630674023.1502871544; _gid=GA1.2.1579405782.1502871544;userId=207939055
	Host: payments-api.dnssf.com
	X-Forwarded-For: 54.183.50.90
	
	{
	userId: “207939055’ OR 1=1”
	}

In the example above the attacker appends the **userId** and sends additional syntax which will be parsed by the SQL query interpreter. This could cause the database to return all rows in the table as opposed to just the row that matches the user's ID. That is because the SQL interpreter will evaluate both portions of the submitted SQL query. The application logic was built with the expectation that the user will provide their legitimate **userId**, which is then passed to the database service for a lookup in the backend database table or view defined in the server-side code. Normally, the SQL database engine will look for a row with the identifier that matches that of the **userId** provided by the client.

In this case, the attacker provided two components of a query through the front end web API, terminating the first part of the query with the use of a " ' " character. One query value is a **userId,** which need not even be valid. They also provided a query value that will result in a comparison of two numerical values. The value of 1 is of course equal to 1, which the SQL engine will evaluate as *TRUE*. Since the complete query string contains the OR operator, either component of the query that evaluates as *TRUE* will return *TRUE* for the final executed query. As a result, all table rows will match this SQL query string. The database service will return all rows in the table, and the data will be passed through the web API back to the attacker.

Why Existing Tools Fail to Protect You

Protecting from injection attacks is common functionality for WAFs and some API gateways since these tools can use signatures to pattern match and identify known injection types. The signatures that these tools use, however, need to be kept up to date in order to protect against the latest injection attacks. If these tools lack the latest signature updates, they will miss new attack types. Unfortunately, signatures are often built for off the shelf web and application software packages including open-source projects like Drupal and Wordpress. Software vendors and open-source web content management system (CMS) project owners will call out pitfalls of WAF signatures for covering the range of custom development or plugins in their respective ecosystems. Web CMS also serve as development platforms. Custom code that development teams build within each respective ecosystem can look wildly different than what a WAF's out of the box signatures are built for.

This is typically where WAF tuning discussions begin, or end, depending on your perspective. It can be difficult for many security teams to keep up with the rate of change of web pages, mobile apps and web APIs. The internet is also riddled with WAF evasion techniques that help attackers avoid WAF pattern matching mechanisms, commonly regex or libinjection. The situation gets worse for API gateways, which don't receive signature updates regularly if at all. API gateways often employ basic threat protection or message filters that look for known malicious characters in requests and responses, such as "= " or " ' " in the case of SQLi. This type of approach is often too basic for organizations since it catches only basic injection attacks and may break other system integrations.

Another consideration is that WAFs focus on all web traffic, of which API traffic is only a subset and tangential focus. This may result in WAFs only being deployed with a positive security model to enforce traffic against an API schema or specific HTTP traffic patterns. Rulesets such as injection protections may also not be applied to API traffic.

How to Protect Against Injection Attacks

An API security solution must be able to identify attackers probing APIs with potentially malicious data through all vectors. Injection flaws can be exploited in many parts of a request, including headers, cookies, URL query parameters, and message body variables depending how other backend application components and systems are architected. Detecting injection flaws successfully and early requires that the solution analyze all API traffic and establish a baseline of typical API behavior. From the baseline, the solution can identify anomalous and potentially malicious data in an API request such as what is seen in injection attacks. This can be done without the need for signatures or pattern matching, which eliminates the need to maintain configurations and signatures while ensuring that even injection attempts using the latest methods are identified and stopped.

API9:2019 Improper Assets Management

Description

Maintaining a complete, up to date API inventory with accurate documentation is critical to understanding potential exposure and risk. An outdated or incomplete inventory results in unknown gaps in the API attack surface and makes it difficult to identify older versions of APIs that should be decommissioned. Similarly, inaccurate documentation results in risk such as unknown exposure of sensitive data and also makes it difficult to identify vulnerabilities that need to be remediated.

Unknown APIs, referred to as shadow APIs, and forgotten APIs, referred to as zombie APIs, are typically not monitored or protected by security tools. Even known API endpoints may have unknown or undocumented functionality, referred to as shadow parameters. As a result, these APIs and the infrastructure that serve them are often unpatched and vulnerable to attacks.

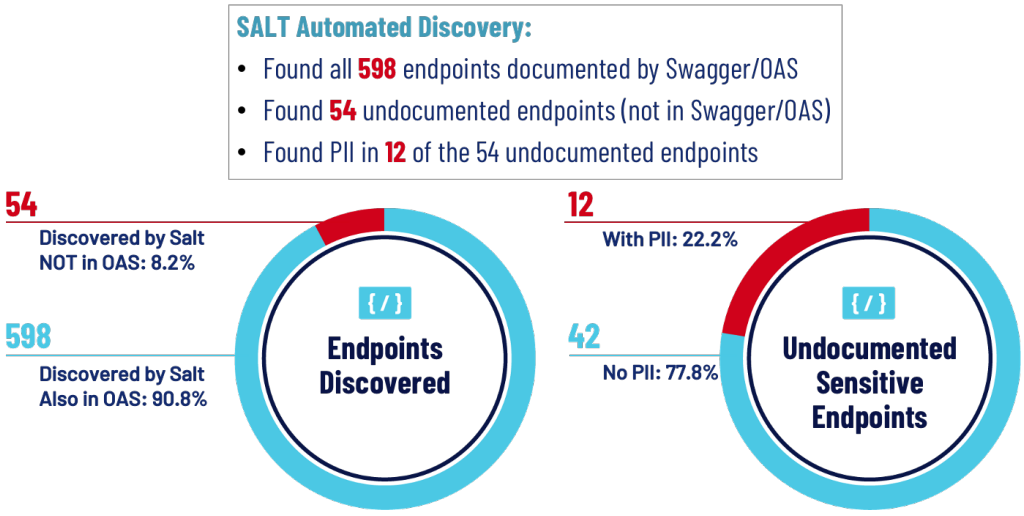
Potential Impact

Attackers may gain unauthorized access to sensitive data, or even gain full server access through old, unpatched or vulnerable versions of APIs.

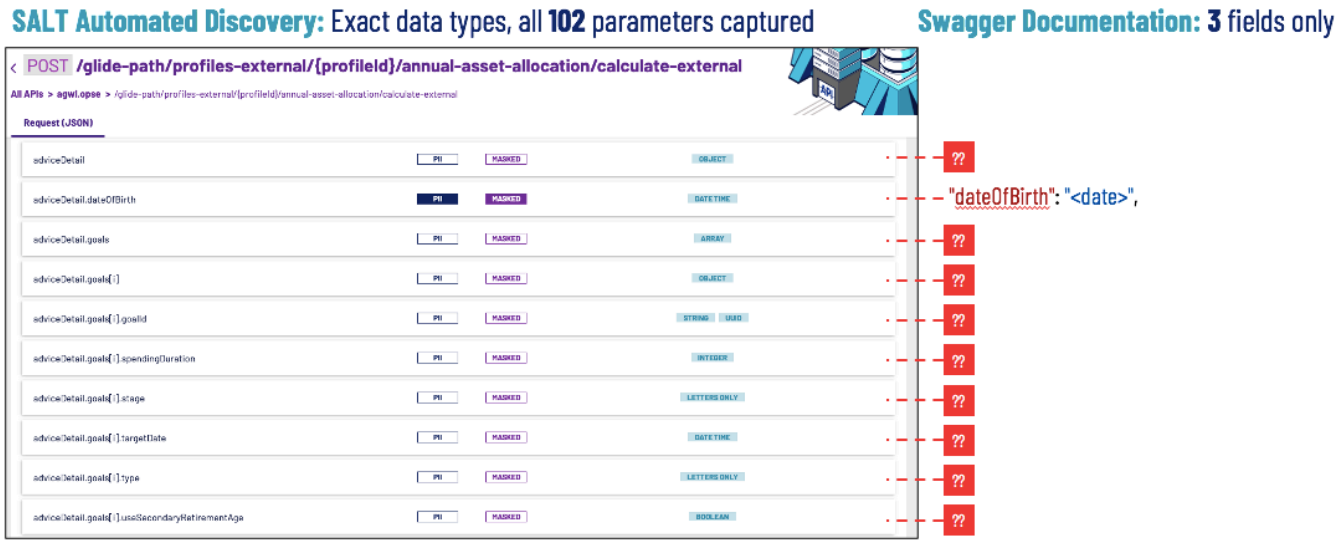
Example

Research conducted by Salt Security shows a common gap of up to 40% between manually created API documentation (or schema definitions) in the form of Open API Specification (OAS) vs. what is actually deployed in production APIs. These gaps fall into the following three categories:

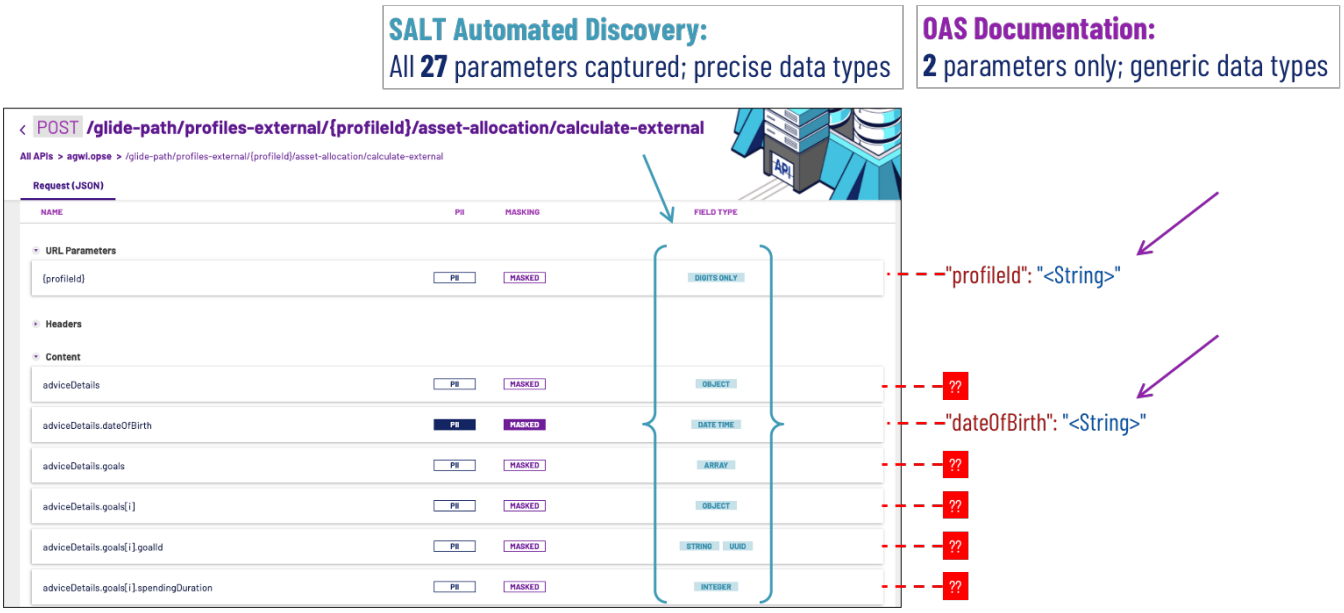
1. Shadow API Endpoints – API endpoints that are missing from the OAS or have no OAS at all. In the following example, Salt Security research found an additional 54 endpoints that were not included in the Swager or OAS documentation, and 12 of those undocumented endpoints were exposing sensitive PII data.



2. Shadow Parameters – API endpoints known to exist but whose API documentation is missing many parameters. As a result, the API documentation does not cover the majority of the attack surface – in this research, API schema definitions listed just three parameters, but the Salt Security platform identified 102 parameters for the single API endpoint.



Parameter Definition Discrepancies – in addition to many missing parameters, data types that lack needed details such as "String" instead of "UUID" or "DateTime" will leave APIs vulnerable. Message filters used by traditional security controls will allow any input through the API to be processed by the backend. These controls rely on a positive security approach and explicitly written rules and policies when enforcing requests against API schema definitions.



Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways lack capabilities to continuously discover APIs at a granular level and monitor them for changes. These security controls only know what they are configured for, requiring API schema definitions to be imported in order to gain a view of the API environment. If documentation is missing or inaccurate, as is often the case for many security teams, these traditional security controls will have an inaccurate view of the API environment.

How to Protect Against Improper Asset Management Vulnerabilities

API security solutions must be able to analyze all API traffic and continuously discover APIs. Discovery must include the ability to identify all host addresses, API endpoints, HTTP methods, API parameters, and their data types including the identification and classification of sensitive data. These solutions must provide discovery on an ongoing basis to maintain an up-to-date catalog of the API environment and accurate API documentation even as new APIs are introduced and updates are made to existing APIs.

API10:2019 Insufficient Logging & Monitoring

Description

Insufficient logging and monitoring combined with missing or ineffective integration with incident response, allows attackers to perform reconnaissance, exploit or abuse APIs, compromise systems, maintain persistence, advance attacks, and move laterally across environments without being detected. The longer an attacker is present in an environment the higher the likelihood the attack will result in a breach, brand or reputation damage, or some other negative impact to the company or its service.

Potential Impact

Without visibility over ongoing malicious activities, attackers have plenty of time to perform reconnaissance, pivot to more systems, and tamper with, extract or, destroy data.

Why Existing Tools Fail to Protect You

Traditional security controls like WAFs and API gateways provide limited logging, monitoring, alerting and incident response capabilities. These security controls alert based on every anomaly without the ability to decipher between benign and malicious abnormal behavior. This results in an overwhelming number of alerts that can be seen as "noise" by SOC and incident response teams, lead to SecOps fatigue and result in the organization missing high priority security incidents that turn into breaches.

How to Protect Against Insufficient Logging & Monitoring

API security solutions must be able to monitor and analyze all API activity and provide proper logging and incident response capabilities, such as feeding actionable security events into the organization's security information and event management (SIEM). By analyzing all API activity, an API security solution can differentiate between benign and malicious abnormal behavior, reducing false positives and low priority alerts. These solutions must also correlate event data to provide a consolidated view of attacker activity, consolidated alerts, and detailed attacker timelines to help accelerate incident response and forensic investigations.

Conclusion: Protecting APIs from the OWASP API Security Top 10 Threats

Protecting APIs from the threats outlined in the OWASP API Security Top 10 requires a new approach to security. Traditional methods of protecting web applications with only authentication, authorization, and encryption are not enough, and traditional tools including API gateways and WAFs do little to stop the top threats targeting APIs. Likewise, not all elements of API security can be addressed in code, let alone tested for and validated pre-deployment – many aspects of how attackers abuse APIs reveal themselves only in runtime.

The Salt Security API Protection Platform secures the APIs at the heart of all modern applications. The platform collects API traffic across the entire application landscape and makes use of big data, AI, and ML to discover all APIs and their exposed data, stop attacks and eliminate vulnerabilities at their source. The Salt solution enables organizations to:

- Discover all APIs and exposed data.
  The Salt platform automatically inventories all APIs, including shadow and zombie APIs, across all application environments. Salt also highlights all instances where APIs expose sensitive data like Personally Identifiable Information (PII). Continuous discovery ensures APIs stay protected even as environments evolve and change as a result of agile methodologies and DevOps practices.

- Stop API attackers.
  Pinpoint and stop threats to APIs with Salt's big data and patented artificial intelligence (AI) technology that baselines legitimate behavior and identifies attackers in real time, during reconnaissance, to prevent them from advancing. The platform correlates all activities back to a single entity, sends a single consolidated alert to avoid alert fatigue, and blocks the attacker – not just transactions.

- Improve API security posture.
  The Salt platform proactively identifies vulnerabilities in APIs even before they serve production traffic. The platform uses attackers like pen testers, capturing their minor successes to provide insights for dev teams while stopping attackers before they reach their objective.

WP-OWASP-20210527