Trend Research

TREND MICRO

API Security Exposed

The Role of API Vulnerabilities in Real-World Data Breaches

Alfredo Oliveira, David Fiser

Contents

Published by
Trend Research

Do organizations truly understand the intricacies of API security, and how failing to secure their systems could put the entire business at risk?

In this paper, we discuss real-world API security risks that companies face. First, we focused on two popular API gateways: APISIX and Kong. We found over 600 APISIX instances and hundreds of thousands of Kong gateways, each misconfigured, accessible online, and unprotected from attacks.

As API security does not only concern gateways alone, we also analyzed the microservices powering API backends. In our investigation of open container image registries, we found a massive 9.31 terabytes (TB) data breach affecting numerous organizations. The stolen data includes company confidential information ranging from API keys for third-party systems integrations to entire codebases– all of which were made available to the public for download.

With built-in features that enhance API security, cloud services have served as trusted tools. However, like all technologies, such systems are not entirely infallible. We found critical security flaws in Microsoft Azure services that allow attackers to take over entire clusters from just one compromised container.

Through this paper, we seek to equip Chief Technology Officers (CTOs), DevOps engineers, and employees in general with the knowledge of the full scope of API security challenges and how to handle them. As we highlight the issues, we also offer actionable and practical steps to secure API systems; for instance, by adopting an attacker mindset and ensuring secure authentication mechanisms are in place, referencing secrets, and never using default passwords even in supposedly secure environments. We describe real-world vulnerabilities that will help you anticipate cybercriminal tactics before they get a chance to infiltrate the system.

Read more of our insights and protect your systems from these threats.

API Gateway

An API gateway is often the entry point into the company's API ecosystem in the cloud-native era. The gateway usually routes incoming traffic to the appropriate backends, such as microservices running inside containers deployed within a cluster of Virtual Machines. However, API gateways offer more features than routing. While this provides more capabilities, it unwittingly allows careless users to introduce various misconfigurations that leave the entire system vulnerable[1].

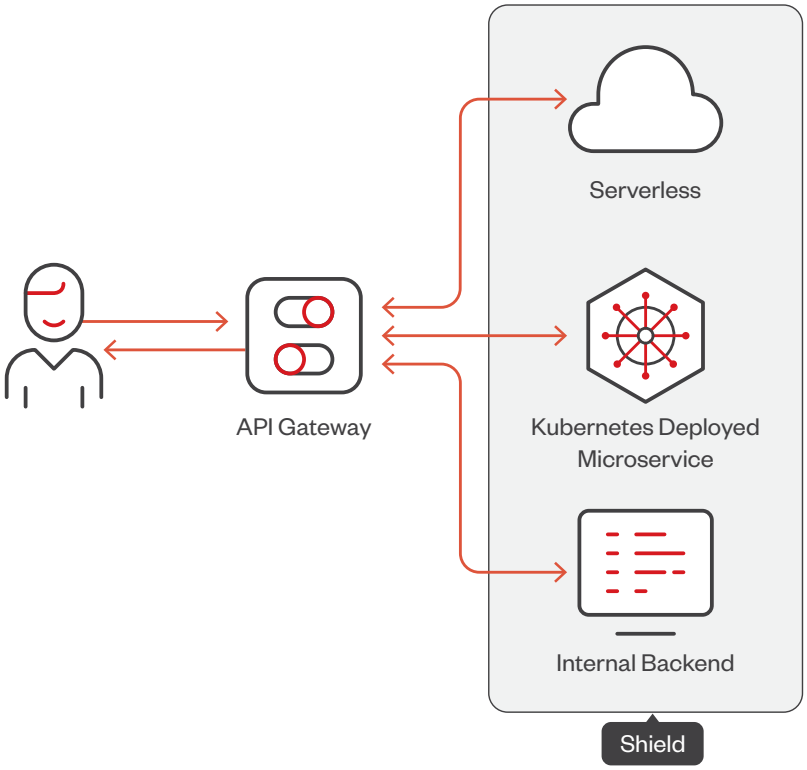

Figure 1. An example of an API gateway at work

Below are some other features of API gateways.

Authorization and Authentication

API gateways support multiple authentication mechanisms, including multi-factor authentication (MFA), to verify the identity of users or applications. Additionally, the gateways use authorization methods, such as role-based access control (RBAC) or access control lists (ACL), to define what actions users can perform on specific resources. Some features are supported natively, but others are supported via third-party identity providers . It is crucial to include the gateway administration interface in the scope.

Offloading

Offloading refers to delegating tasks such as Secure Sockets Layer (SSL) handling, request caching, and response transformations to the API gateway instead of backends, significantly reducing the load on individual services and improving overall system performance.

Transport Layer Security (TLS) Termination

TLS termination at the API gateway means the requests to the API gateway are encrypted while the requests sent to the backends are in plain and readable form.

Aggregation

API gateways can aggregate responses from multiple backend services and deliver a unified response to the client, simplifying client-side logic and reducing the number of requests needed.

Single Point of Access

This centralization can simplify security management, serving as a single entry point for all API requests.

Deployments

We can also distinguish API Gateways by how they are deployed and configured.

- Cloud – used as Cloud Service Provider (CSP)-offered service
- On-premise – custom-configured third-party API gateway software
- Hybrid solution – CSP-provided API gateway for hybrid solutions or custom-configured third-party API gateway

Each of these features and deployments has its associated risks. In this paper, we will mainly focus on on-premise and hybrid API gateways. These gateways still support cloud services but are not offered as a managed service, providing more configuration options and use cases for their users.

Risk Modeling

As a risk example, we provide a configuration where the API gateway has a route requiring an API key. However, the backend service does not require any authentication at all.

Whoever has access to the backend service inside the internal network does not need any authentication. The configuration makes the backend service vulnerable to server-side request forgery (SSRF) attacks in cases where the service is reachable within the internal network. A compromised internal device, container, or service gives threat actors free access, allowing them to carry out lateral movement attacks.

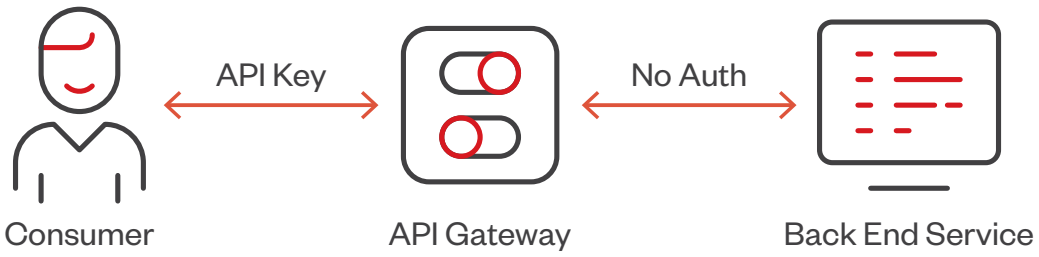

Figure 2. An example of a vulnerable authorization policy using an API gateway

When it comes to TLS termination, one of its benefits is the reduction of the performance overhead needed to send encrypted requests, decrypt them on the other end, and offload the backend. However, it comes with a price, as the secrets are sent in plain text after TLS termination. Without additional security measures on a network level, a skillful attacker will try to intercept the packets and leak out the secrets or tamper with the services, for instance launching MITM attacks in environments where it is possible. Users should be extra careful in on-premises workloads.

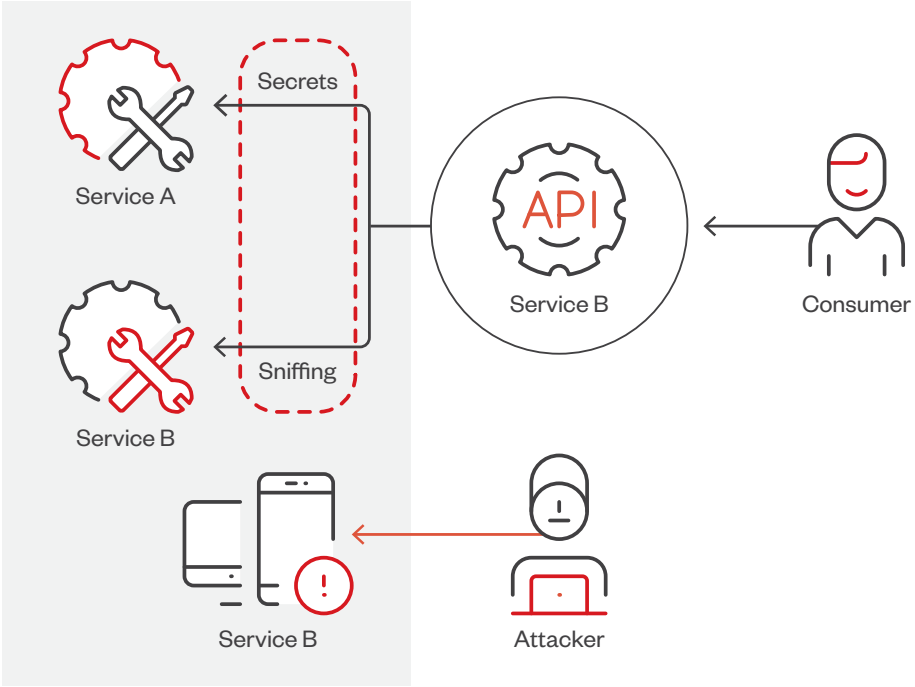

Figure 3. Example of an attack scenario

The described case gives us an idea of why access control, TLS, and proper secret storage are crucial elements in security applicable not only to the API gateway but to the whole system.

Exposure

It is no exception that users deploy custom-managed services into cloud environments. For example, they deploy an API gateway into an elastic computing instance while ultimately transferring the responsibility from the cloud service provider to themselves. We conducted our research using a publicly known CSP's IPv4 address range and looked for the fingerprints of API gateways and associated services running inside cloud environments. For this, we chose two popular gateways – APISIX and Kong.

APISIX API Gateway

APISIX is built on the NGINX webserver and the OpenResty LUA extension framework, which serve as the gateway's core components. The plug-ins extend the functionality and add features for observability, security, traffic control, etc. An optional dashboard is a separate service communicating directly with the administration API.

The APISIX core handles functions such as route matching, load balancing, service discovery, configuration management, and the provision of a management API. It also includes a plug-in runtime supporting Lua and multilingual plug-ins (e.g., Go, Java, Python, JavaScript, etc).

The user has complete configuration control. The secure configuration is his full responsibility. Misconfigurations are among the most prevalent threats for software deployments, regardless of the location.

We evaluated APISIX security defaults and found a static master password for the administration interface. That puts the default configuration of APISIX gateways at risk as users who are unaware of proper security measures might not be motivated to change the password. This unintentionally opens the gates for threat actors, providing them easy access as they already know the password.[2]

```
curl "http://127.0.0.1:9180/apisix/admin/upstreams/1" \
-H "X-API-KEY: edd1c9f034335f136f87ad84b625c8f1" -X PUT -d '
{
  "type": "roundrobin",
  "nodes": {
    "httpbin.org:80": 1
  }
}'
```

Figure 4. Example of default master API token, taken from APISIX documentation[3]

The administration API shouldn't be exposed to the public, and in more secure deployments, it will be protected by at least Web Application Firewall (WAF) and other network limitations such as IP address restrictions – however, the CVE-2022-24112[4] vulnerability affecting APISIX gateways allows its bypass. The vulnerability, combined with the default password, permits threat actors to perform remote code execution on the gateway successfully.

The vulnerability affects APISIX 1.3-2.12 gateways. The key takeaway is that the vulnerability is within the IP filter, not the default master password. The example emphasizes the need to change the default API key and not solely rely on private or limited network isolation.

APISIX Dashboard

The APISIX dashboard has direct access to administration API, so users should be extra careful in cases where admin API is shielded from the outside world while the dashboard remains accessible. The dashboard application is protected by username and password credentials, and they are often set to "admin" by default, especially in containerized applications. Using a simple POST request, we can distinguish if the dashboard instance uses default credentials, as it would send us back a valid token instead of an error response.

Configuration Traps

As previously described by the API gateway use cases and its nature, the security-related components of API gateway are:

- Secrets management for accessing upstreams
- Token issuing and gateway authentication mechanism and identity provider (IdP) settings
- Logging and secrets stripping
- Plugins and script creation

APISIX does not have any sandbox for executing Lua code. Thus, users should be extra careful with plug-ins or scripts using user-provided Lua code, as the user input might introduce a vulnerability in the gateway itself. For example, processing a HTTP header by vulnerable Lua code.

An API gateway serving as a central access point for company APIs is an ideal attack target and a perfect candidate for data breaches upon compromise, leaking the upstream configuration or IdP secrets. Data which will make threat actor to think about user impersonation and token harvesting attacks.

Secrets Management

By default, APISIX relies on etcd, a well-known storage for Kubernetes users. The configuration is stored unencrypted by default, allowing everyone accessing the etcd instance to read stored secrets, such as static API keys.

```
                    ~ % curl -i -X GET "http://192.168.10.79:9080/api"
HTTP/1.1 401 Unauthorized
Date: Wed, 20 Sep 2023 09:55:12 GMT
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Server: APISIX/3.2.2

{"message":"Missing API key found in request"}
                    ~ % curl -i -X GET "http://192.168.10.79:9080/api" -H "apikey: top-secret-key"
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 22
Connection: keep-alive
Date: Wed, 20 Sep 2023 09:55:28 GMT
Server: APISIX/3.2.2

{"msg":"Hello World"}
```

Figure 5. API endpoint protected by API key

```
/apisix/consumers/teddy
{"username":"teddy","plugins":{"key-auth":{"_meta":{"disable":false},"key":"top-secret-key"}},"create_time":1695108609,"update_time":1695108609}
/apisix/data_plane/server_info/2206f016-4642-4afd-99f7-2a6226464685
```

Figure 6. Example of plaintext API Key

Data on Publicly Exposed Systems

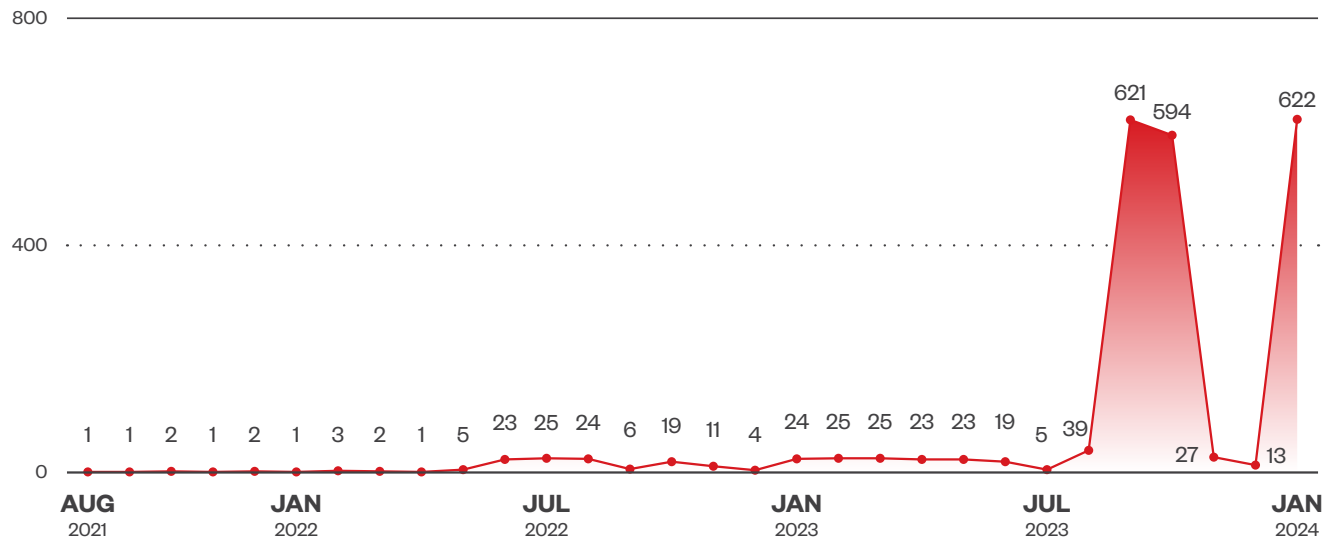

Figure 7. The number of exposed APISIX services found on Shodan

In January 2024, a Shodan search revealed 622 exposed unique IP addresses running some version of APISIX services in different ports.

When we checked the contents with the provided default secrets, at least 39 were entirely open, exposing sensitive data.

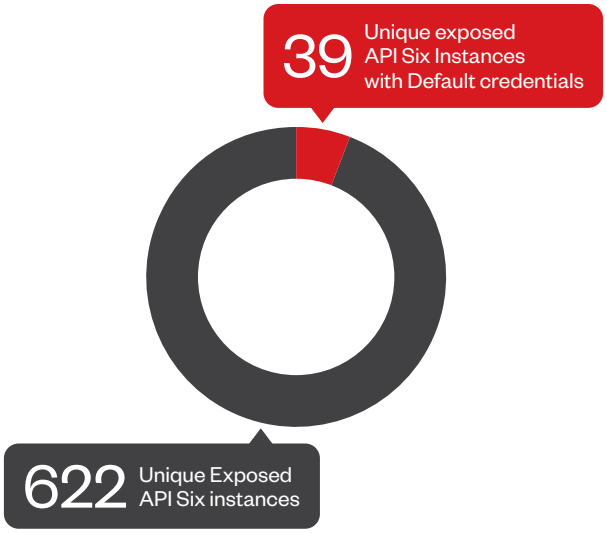

Figure 8. Instances of exposed APISIX

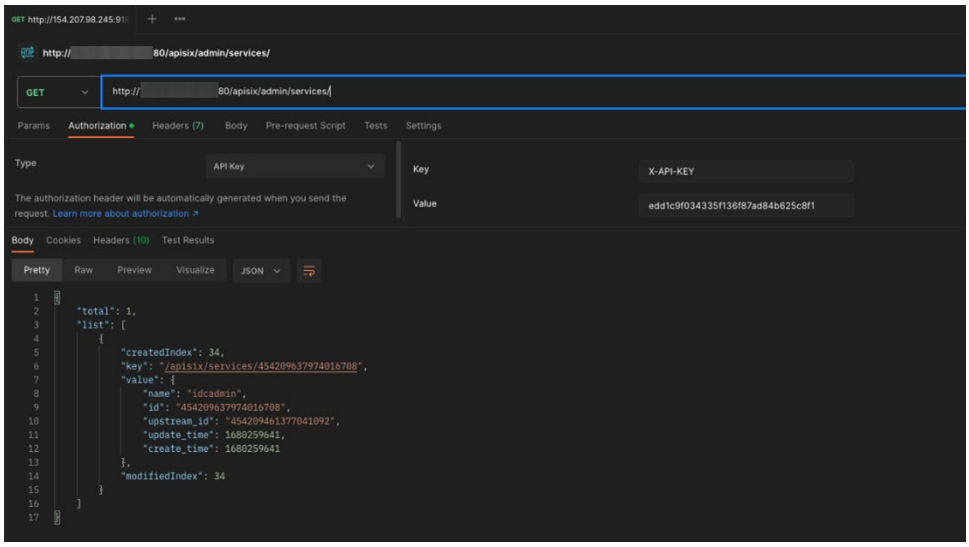

Figure 9. Example of publicly exposed APISIX instances with default APIKEY

The good news is since APISIX version 3.1,[5] users can use an encrypted configuration plug-in for storing secret configurations. Sadly, it is not a default setting, and additional configuration is still needed.

Fortunately, there are other ways to store secrets; APISIX supports storing secrets inside a vault and referencing them within the configuration. We emphasize using vaults for storing secrets. However, users should note that they most likely need at least one secret to access the vault and thus ensure its security.

APISIX currently supports storing secrets in the following ways:

- Environment Variables
- HashiCorp Vault

You can use APISIX Secret functions by specifying format variables in the consumer configuration of the following plugins, such as `key-auth`.

::: note

If a configuration item is: `key: "$ENV://ABC"`, when the actual value corresponding to $ENV://ABC is not retrieved in APISIX Secret, the value of the key will be "$ENV://ABC" instead of `nil`.

:::

Figure 10. Vault types according to APISIX's documentation

We don't recommend storing secrets inside environment variables. In a previous report, we analyzed the dangers of using environmental variables for keeping secrets.[6] Environmental variables are often misused, introducing security issues as we described in our findings exposed container registries revealing stored hardcoded secrets to be used as environmental variables and found inside container images.

We acknowledge that there is never a 100% secure system and even environmental variables can be used in a "safer" manner. However, that requires a complete understanding of its underlying implementation and ensuring no security boundary will be crossed. For example, injecting them on runtime only (no hardcoding) and being aware of default inheritance of environmental

variables to child processes, where it does not have a valid use except as another security risk, as we further elaborated in a previous entry on enhancing the security of Azure serverless using custom container images.[7]

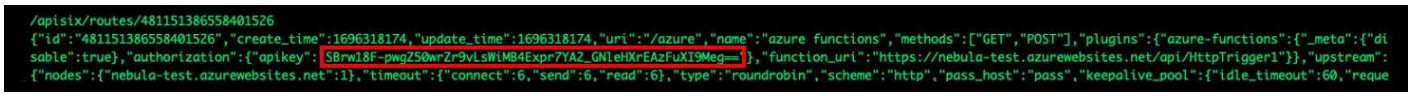

Figure 11. Example of plaintext API key bind to Azure Functions

Logging

Logging system activities could be a lifesaver when troubleshooting an issue. It allows administrators and developers to simulate and understand where the problem lies.

However, excessive logging not only fills the storage faster, it also provides additional attack surface especially as it logs sensitive information. It is also typical to discuss pending issues with peers and upload logs for consultation.

How can we know if sensitive information is logged?

The answer to that question is dependent on multiple factors. For instance, where is it passed through the request (whether inside the URL, the HTTP header, or the body), how thoroughly are activities logged, and of course, how sensitive is the information?

Let's assume we pass a token as a parameter in a GET request. Our questions would be: Is it a valid token, not an expired one? What are the token's permissions? Overly permissive tokens with extended validity are ticking bombs waiting to explode.[8]

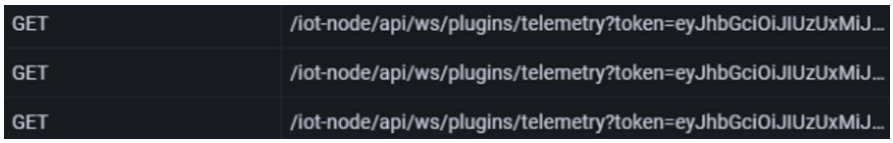

Figure 12. An example of token logging in URL

We have tested an HTTP-logger plug-in with default settings; we only configured the HTTP logging server URL. All the headers, including the authorization header, were present by default.

```
"consumer": {
   "username": "teddy"
},
"apisix_latency": 6.2942504882812e-05,
"service_id": "479160274549476342",
"upstream": "192.168.10.79:8956",
"server": {
   "hostname": "shellplayground",
   "version": "3.2.2"
},
"latency": 9.0000629425049,
"client_ip": "10.10.11.3",
"upstream_latency": 9,
"request": {
   "uri": "/api?specialToken=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxw
   "method": "GET",
   "headers": {
      "postman-token": "18204ddc-d3ce-4412-b633-b54d2b1f7cae",
      "accept-encoding": "gzip, deflate, br",
      "accept": "*/*",
      "connection": "keep-alive",
      "user-agent": "PostmanRuntime/7.33.0",
      "apikey": "top-secret-key",
      "authorization": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImlhdCI6MTY5MzMwMTkyNCwiZXhwIjoxNjkzNDAxMzUzfQ.Eaxu5mzldkUPCRdOhH7
      "host": "192.168.10.79:9080"
   },
   "url": "http://192.168.10.79:9080/api?specialToken=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0
   "querystring": {
      "specialToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeK
   },
   "size": 565
},
```


Figure 13. An example of default setting APISIX HTTP-logger

However, what does this mean in terms of security? We could say that with great power comes great responsibility, as the configuration is entirely in the user's hands. First, we should ask ourselves the following questions: What do we want to monitor? Do we genuinely need to log tokens? If so, can we strip, substitute, or extract sensitive information so that it's not valuable for threat actors?

For instance, if we follow an example from APISIX documentation, the APISIX provided example logs with way less information. As we have proven again, relying on defaults is a dangerous practice. Users should know what information they are logging as well as its purpose.


```
192.168.10.79 - - [04/Oct/2023 07:01:56] "POST /log HTTP/1.1" 200 -
[
   {
      "route_id": "100",
      "host": "192.168.10.79",
      "client_ip": "10.10.11.3",
      "@timestamp": "2023-10-04T07:10:20+00:00",
      "service_id": "479160274549476342"
   }
]
```


Figure 14. Log based on documentation of APISIX

In any case, if we are logging sensitive information for whatever reason, we should secure the sensitivity of that information by ensuring its secure transfer via Transport Layer Security (TLS) and securing its storage (which might be problematic, especially on logging platforms). Otherwise, we are just increasing the risk of sensitive information being leaked.

Risks Associated with APISIX API Gateway Plug-Ins

Community plug-ins can be a great source of vulnerabilities, especially when they are managed by multiple parties with different levels of experience. In some cases, community plug-ins can even be integrated into a product together with their security issues. As such, plug-in usage should be considered not only for its given functionality but also for its security.

APISIX's API gateway splits its plug-in set into several categories: transformation, authentication, security, traffic, observability, and serverless. Each of these categories has different risks associated with them. For instance, the transformation plug-in's risk is related to processing user input in the form of API requests, their responses, and the associated parsing issues. Meanwhile, authentication plug-ins are prone to unsecure sensitive information storage and incorrect verification implementations. On the other hand, observability plug-ins can have issues related to excessive logging of sensitive information, as previously described.

One of the authentication plug-ins is a basic-auth implementing RFC-7617,[9] which conducts authentication using usernames and passwords. Even though this mechanism is deprecated in serious use cases nowadays, it perfectly demonstrates the security posture of API gateways related to secrets. After setting up this plug-in, a valid username and password that are encoded in an authorization header are required for successful authentication.

```
/apisix/consumers/foo
{"update_time":1695816210,"create_time":1695816210,"plugins":{"basic-auth":{"password":"bar","username":"foo"}},"username":"foo"}
```

Figure 15. An example of basic-auth plug-in credentials' storage

The figure above clearly demonstrates that by default, the credentials, including the password, are stored only in plaintext. Even though it is possible to enable encryption or vault reference on the password, the real question should be why the password is even stored and why the API gateway stores such secrets. Storing secrets makes sense if their raw form is necessary for third-party service authentication and when they are not passed from the initial request.

Storing a raw password is unnecessary for authentication purposes on the API gateway level. In fact, it is a security vulnerability that allows threat actors to exploit the password when a breach happens.

A secure way to store credentials for verification purposes is to create a salted hash and store the hash instead of storing a raw password. When proper hashing is used, in case of a breach, threat actors will only have access to the salted hash. Given the salted hash's properties, they won't be able to recover the plain-text password needed for authentication, making the whole ecosystem more secure.

How Threat Actors Can Abuse API Gateways

The first thing we should consider is a threat actor's motivation – how to leverage API gateway security misconfigurations for nefarious purposes.

Clearly, abusing a vulnerable computing resource that's free of charge is a valid motivation and reason to deploy crypto-jacking malware that can generate revenue, including a bot joining a large-scale network used in distributed denial of service (DDoS) attacks.

Another motivation is malware distribution by creating their own endpoints serving malware. Launching targeted attacks might be more complicated given the fact that API gateways serve as a backend service gateway and it's unlikely to serve as a pure HTML resource. However, we can imagine an API service that's returning software download links. In such cases, exploited API gateway instances can be configured to serve as malicious links, making it a supply chain attack.

In more targeted attacks, threat actors would use API gateway functionality to gather credentials, perform user impersonation, or get critical company information to sell or exploit in future attacks.

Honeypot Data

Our honeypot, which was active for three months, was designed to monitor and analyze attacks on an API gateway connected to a website. The website was hosted on port 80, while the API gateway was hosted on its default port (9080). The link to the API gateway was present inside the website's source code.

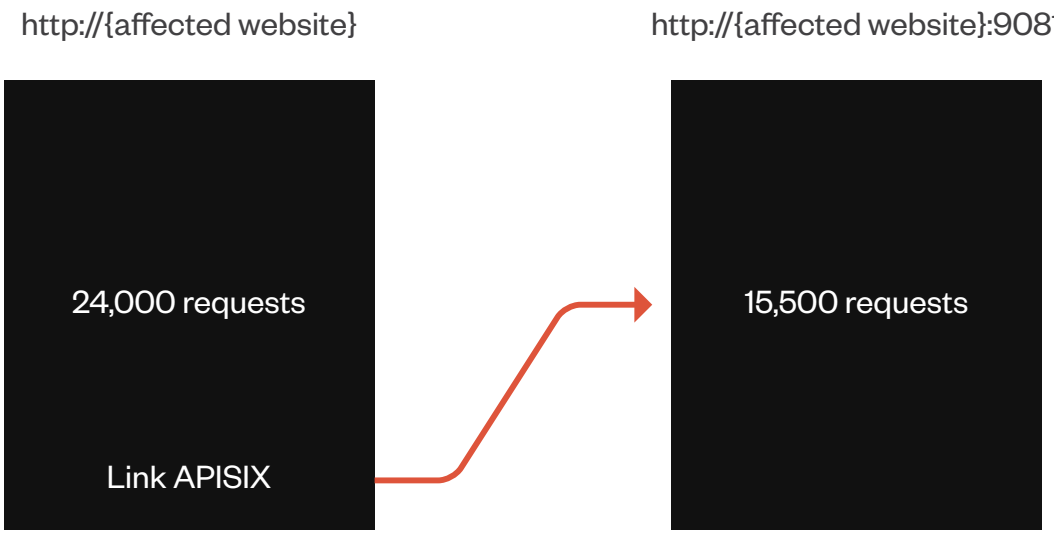

Figure 16. The honeypot accesses data from the webpage to the API

The website received about 24,000 requests during this period, while the API endpoint saw around 15,500. Excluding the root "/" requests, primarily from bots, we identified approximately 11,500 attempts using well-known generic exploits.

Figure 17. APISIX-specific requests

The setup gave no clear indication of specific API gateway presence. The attackers needed to interact with the service to discern if it is APISIX. We identified 14 targeted attacks on APISIX, which were characterized by specific URLs, additional request parameters, and attempts to access the administration dashboard using default username-password combination.

```
POST /apisix/admin/user/login HTTP/1.1
Host:          :9080
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2049.0 Safari/537.36
Connection: close
Content-Length: 39
Accept: application/json
Authorization:
Content-Type: application/json;charset=UTF-8
Accept-Encoding: gzip

{"username":"admin","password":"admin"}
```

Figure 18. Example of default password attack

Kong API Gateway

Aside from APISIX, Kong is also a popular API gateway. It is available in two versions, namely community and enterprise. Kong's default database engine is PostgreSQL, and in its earlier version, Cassandra. Additional data stores, such as Redis, can be used for caching using community plug-ins. Security-wise, the community version lacks encryption and vault support compared to the enterprise version.

The API gateway consists of three main components: the gateway itself, the database, and the administration API. Kong allows running an API gateway without a database, which means that the configuration is directly accessed from memory. However, running Kong without a database prohibits the use of certain features.

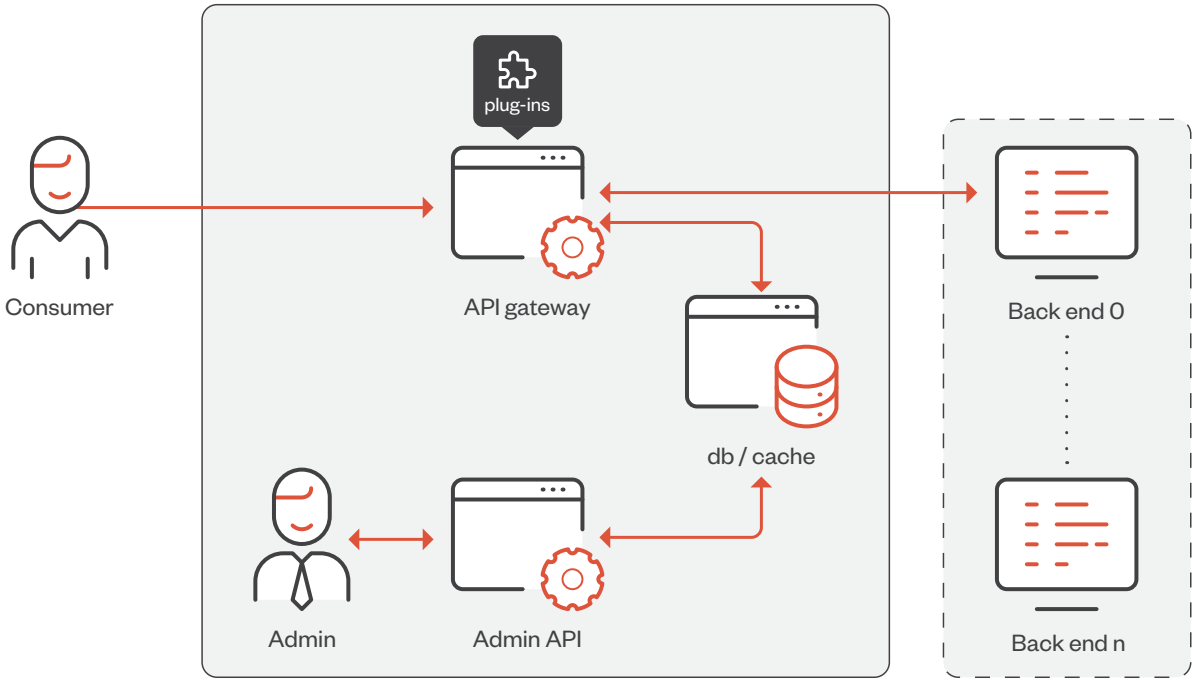

Figure 19. Kong API architecture

Like the APISIX, the Kong API Gateway is also built on NGINX.

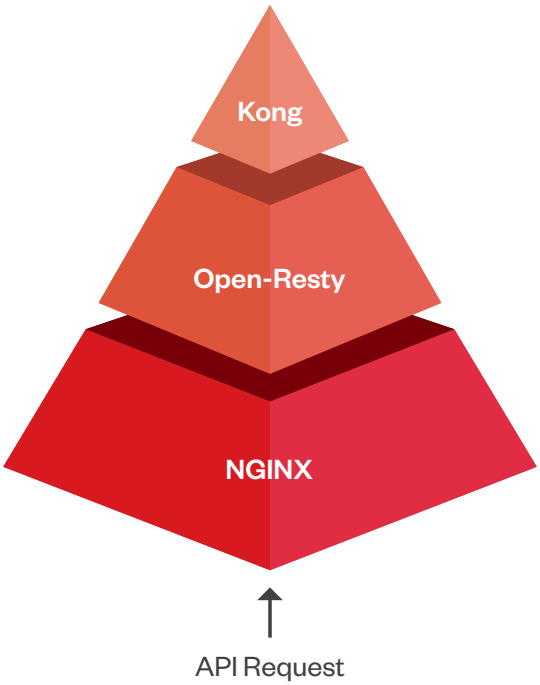

Figure 20. API request chain within the Kong deployment

We highlight the architecture to emphasize that the system is only as secure as its weakest part. Any vulnerability or misconfiguration in any of these components may lead to information disclosure, API gateway compromise, backend compromise, or supply chain attack, putting the whole organization at risk.

Similar to APISIX, Kong API gateway also allows deployment in multiple environments. Each deployment affects the security based on its application components and configuration. This is applicable no matter where the API gateway is deployed, whether on-premises, entirely in the cloud, or in a hybrid cloud. Users must ensure that only authorized entities can access crucial security components – the Administration API and the database.

Securing Kong API gateway administration API must be a top priority, since access to this component allows users to read gateway configuration in plain text and modify the configuration in various ways, such as adding routes or tampering with the authorization mechanism. Exposing the administration API to the public or even within a local network puts backend services at risk.[10]

The following sections look into common misconfigurations.

Misconfiguration No. 1: Forwarding the Admin API

In more secure deployments, the admin API should be assigned to localhost only, not forwarding administration ports nor sharing the network with the host.

Gateway administrators can set a new route as mitigation requiring credentials to access. The route can point to the administration API, making it accessible outside a container. Similarly, the enterprise version API gateway, RBAC, requiring token authentication, can be used.

Analyzing this misconfiguration, we can see how we can easily follow this scenario by using default configurations or examples found within container image repositories.

The default Kong API image configuration example on the docker hub (as shown in Figure 22) not only exposes administration API on 0.0.0.0 (=all network interfaces) but also uses weak passwords.

We can see the danger of copy-pasting without additional security thinking. We also note settings posing security risks when a database endpoint becomes available due to misconfiguration or adjacent system compromise. This provides a simple avenue to access confidential data, similar to scenarios in which no credentials were used.

```
$ docker run -d --name kong \
    --link kong-database:kong-database \
    -e "KONG_DATABASE=postgres" \
    -e "KONG_PG_HOST=kong-database" \
    -e "KONG_PG_PASSWORD=kong" \
    -e "KONG_CASSANDRA_CONTACT_POINTS=kong-database" \
    -e "KONG_PROXY_ACCESS_LOG=/dev/stdout" \
    -e "KONG_ADMIN_ACCESS_LOG=/dev/stdout" \
    -e "KONG_PROXY_ERROR_LOG=/dev/stderr" \
    -e "KONG_ADMIN_ERROR_LOG=/dev/stderr" \
    -e "KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl" \
    -p 8000:8000 \
    -p 8443:8443 \
    -p 8001:8001 \
    -p 8444:8444 \
    kong/kong-gateway
```

Figure 21. Docker hub official image example exposing port admin 8001 and 8444

Misconfiguration No. 2: Missing Firewall Rules

Imagine an IP address as a door; having a public one means anyone – even intruders – can access it anytime. It goes without saying that to promote safety, steps such as locking the door and providing keys to only trusted people are a must.

Similarly, make sure that only authorized entities can access your cloud instance. Limiting access to exposed ports only for specific IP addresses or subnets, together with an authentication mechanism, helps mitigate this scenario.

In addition, take extra care in deployments where the applications are accessed by people using dynamic IP addresses. Instead, use additional access vectors such as virtual private networks (VPNs).

Secret Storage Issues

Usually, accessing protected resources such as administration planes, API backends, and serverless endpoints requires an authentication mechanism.

One of the API gateway's use cases is simplifying the authentication using an organizational identity provider issuing valid tokens accepted by the API gateway, which then forwards a request to the protected resource using a pre-saved secret. The configuration affects the way the secrets are stored, as they might be saved in the memory of the deployed application or inside the database.

Securing these secrets is thus crucial in overall system security, and several steps can help. First, the data store should be accessible only from the API gateway. Credentials for accessing the database shouldn't be easily guessable or copied from default configurations and examples. Gateway administrators should apply TLS as protection to prevent network sniffing in on-premise deployments.

In scenarios where it is unnecessary to forward a secret, for instance, when configuring API key or token access on gateway routes, confidential information shouldn't be stored in only plaintext or encrypted plaintext, as this makes it possible to recover the secret upon leakage. Instead, hashing mechanisms and salting should be applied, making it almost impossible to recover the original secret upon database leakage. Encryption or external vault storage should be used in scenarios when it is not applicable.

Kong API gateway stores every secret in the database as plaintext by default. Specific plug-ins support additional encryption only when using the enterprise version with the configured keyring. During our research, we discovered that only specific plug-ins support the encryption of sensitive information, and users should be extra careful, especially when using third-party plug-ins.

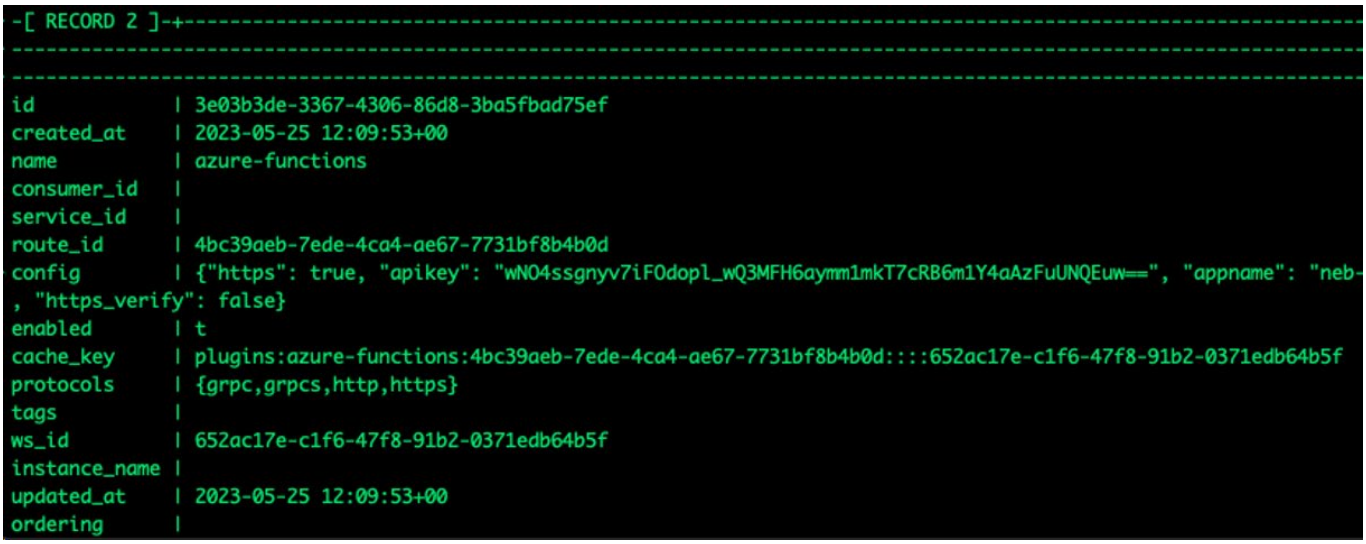

Figure 22. Example of API key for Azure Functions saved in plaintext database

Encryption support can be verified by checking the encrypted parameter within the plug-in's source code. The user can check the parameter in the source code or query the database to verify that encryption is used when configured.

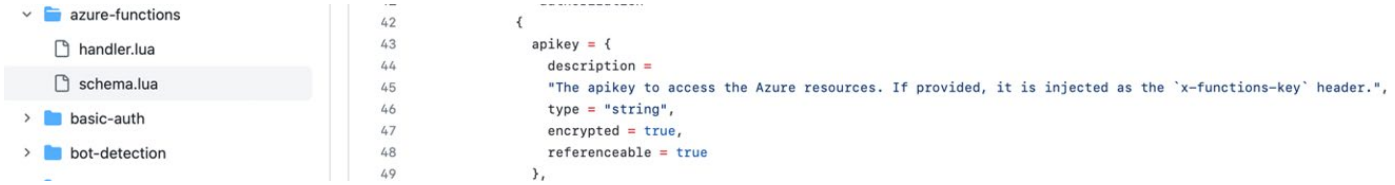

Figure 23. Encryption and vault support inside Kong API gateway plug-ins

Vaults can provide the benefits of single secret storage using a safe mechanism, making it easy to rotate. Previously, Kong listed environment variables as a vault (as seen in Figure 25). After sharing our findings with them, this has been changed.[11]

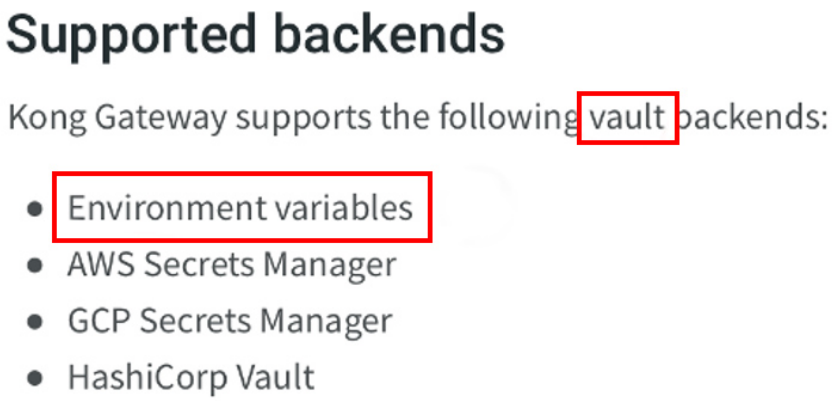

Figure 24. The environmental variables listed as a vault, though this has been changed

However, it shows the misconception in the vault terminology presented within official documentation, as environmental variables do not comply with the vault definition. We explained why storing secrets within environmental variables is a bad practice in our previous article.

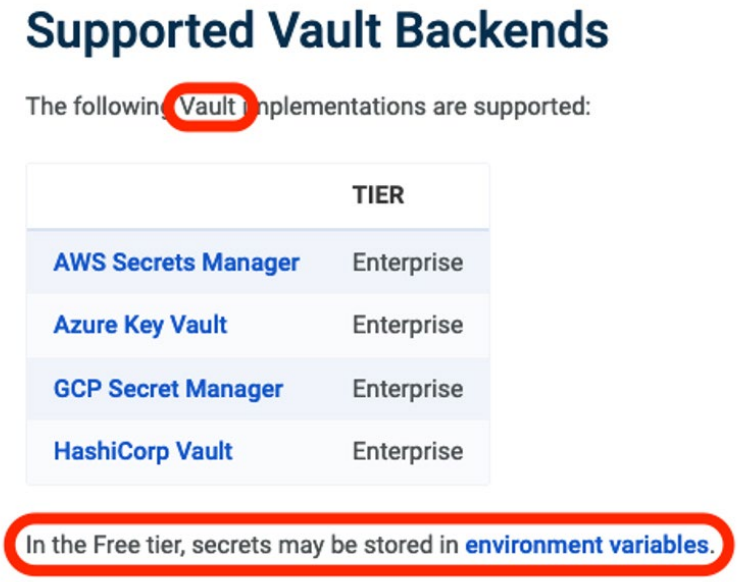

Figure 25. Updated documentation on supported vault backends

Similar to encryption support, not every plug-in supports a vault for storing sensitive information, as a referenceable argument must be set to true in the plug-in source code. Only the enterprise version of the API gateway supports external vaults.

Plug-ins provide additional functionality the community demands, often providing security challenges and bringing vulnerabilities. As most of the functionality of the Kong API gateway is provided by plug-ins, it makes it a serious concern in terms of security. In previous paragraphs, we described plug-ins that might miss encryption or vault support when required attributes are missing within the schema definition.

```
name = "key-auth",
fields = {
  { consumer = typedefs.no_consumer },
  { protocols = typedefs.protocols_http },
  { config = {
      type = "record",
      fields = {
        { key_names = { description = "Desc
            required = true,
            elements = typedefs.header_name,
            default = { "apikey" },
        }, },
        { hide_credentials = { description =
        { anonymous = { description = "An op
        { key_in_header = { description = "
        { key_in_query = { description = "It
        { key_in_body = { description = "If
        { run_on_preflight = { description =
      },
  }, },
},
}
```

Figure 26. Example of missing vault and encryption support of key-auth plug-in

Whenever a user inputs data or code, it should be considered a security risk, including plug-ins. For example, bugs in request/response- modifying policies are source of denial of service or, worse, remote code execution vulnerabilities.

Kong API gateway also supports "serverless" code execution; in the Kong language, the administrator has the ability to configure the gateway to execute a custom Lua code upon processing a request using the serverless plug-in. The good news for security is the code is run inside a Lua sandbox by default.

However, the user can still bring an issue when the configuration is altered, the sandbox is disabled, or dangerous imports are allowed within the configuration file.

These misconfigurations may allow vulnerable or malicious execution on the API Gateway. The threat actors will get complete control of the API gateway, consequently leaking all the secrets, eventually leading to a successful supply chain attack and putting the whole organization at risk.

Users should also be careful with an older gateway version that might allow sandbox escapes. We suggest turning off the functionality when not explicitly needed and using the up-to-date version of the Kong API gateway.

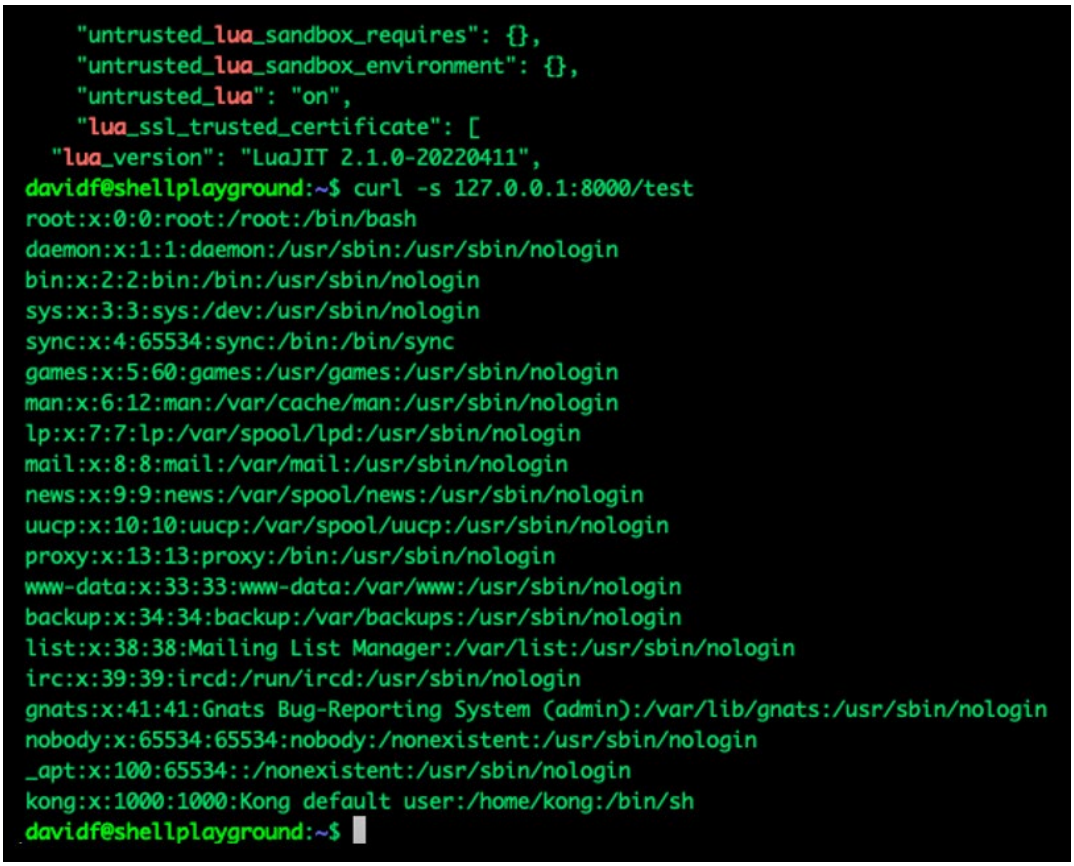

Figure 27. Example of possible Remote Code Execution when untrusted Lua is enabled

Accessing APIs and web applications is tightly linked with authentication and authorization. OAUTH 2.0 and OpenID connect became industry standards, and their support has been implemented into API gateways. Correct implementations of these flows are crucial elements of API security.

In that regard, Kong API gateway supports issuing its tokens to third parties and using the third-party OAUTH 2.0 mechanism as an authorization mechanism for routed resources.

Security-wise, API gateway administrators should validate that a secure configuration is applied, avoiding redirection attacks and allowing only trusted Identity Providers.

Failing in those elements may lead to successful user impersonation and unauthorized access, which may be hardly detectable. Issued tokens and authorization flow settings are saved within the database or in memory configuration; failing to secure the administration API or database may have the same consequences.

API Gateways Trend/Data

Unfortunately, the Shodan search engine still shows some publicly exposed Kong administration API instances that are open to the internet.

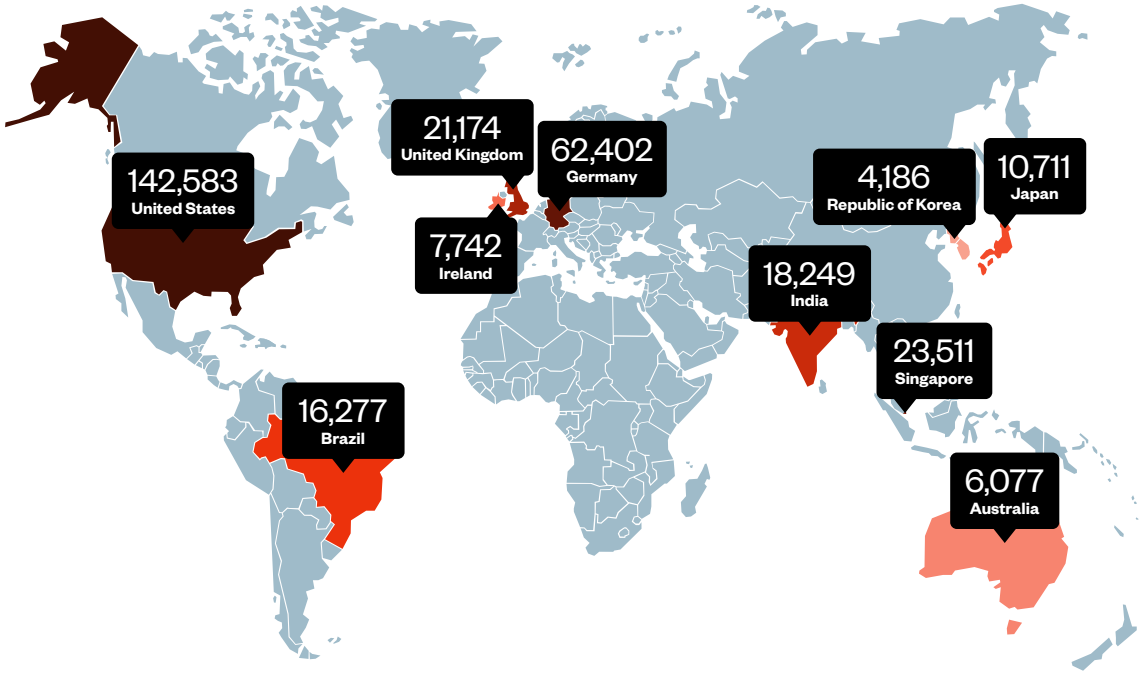

Figure 28. Global distribution of publicly exposed Kong API gateways

The first indexations started in late 2021, and from then we have observed a rising trend in the number of exposed gateways.

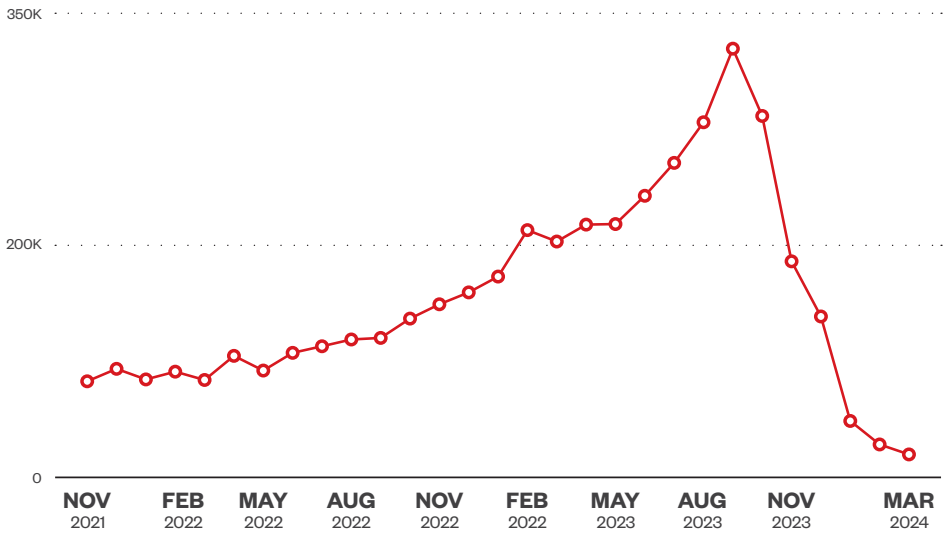

Figure 29. Publicly exposed Kong API gateways arranged by time

It should be noted that not all exposed gateways necessarily represent actual Kong API deployment, as some might be honeypots.

Container Registries and API

Containers have revolutionized the way we develop, deploy, and manage applications. By offering isolation, portability, and consistency, they allow for rapid scaling and efficient integration in cloud environments. Containers have driven the growth of cloud-native technologies, fundamentally changing how many modern applications are designed, deployed, and managed.[12]

What some might not realize is that there is a whole API ecosystem behind containers. For instance, spawning a docker container requires an API request to docker daemon. The same applies to deploying a container within a cluster using orchestrators. However, the image itself is the core component that makes the container run.

An essential aspect of working with containers is storing and managing the container images used to encapsulate applications. Since these images have to be stored somewhere, cloud service providers (CSPs) have started offering registry services that act as image repositories and allow developers to share and deploy applications across various environments. In addition, companies can choose to run their container image registries, either on-premises or inside cloud computing instances (Workloads).

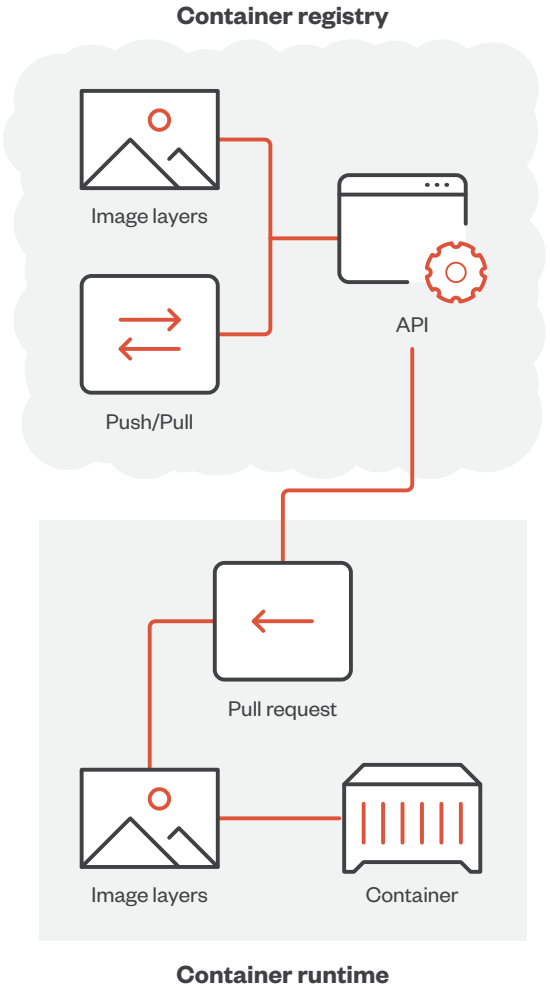

Figure 30. Container registry API actions

Container image registries are more than just repositories; they are the lifeblood of any containerized system. These registries store container images, which are static files containing layers with pre-installed applications and their dependencies and environmental setups. The setups include application codes, libraries, system tools, and settings.

Each layer of a container image represents a modification to the image, like adding a file or a configuration change. These layers are stacked on top of each other to form the final image. Only the changed layers are updated when an image is built, making container images efficient.

The creation of a container typically begins with the building of an image for an application. This process involves compiling application code, bundling it with the necessary dependencies, and configuring the runtime environment. The resulting image is a standalone executable package that encapsulates everything needed to run an application.

Once the image is built, it is then pushed to a registry. This registry is either a private repository (if configured correctly, restricted, and accessible only to authorized users or organizations) or a public one (accessible to the general public). CSPs can offer either or both, depending on the need and configuration.

The registry acts as a distribution point, storing the image and allowing it to be pulled when needed. This is a critical aspect of the container life cycle, as it ensures that different versions of applications can consistently and reliably be deployed across various environments.

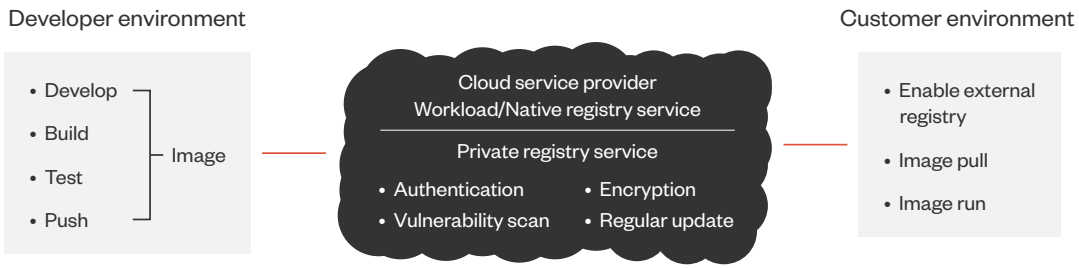

Figure 31. How container image registries work

The image stored in the registry encapsulates the application and all its software dependencies. This means the application will run the same way regardless of its deployed environment. This consistency is one of the benefits of containerization, made possible by using container image registries.

Container image registries are pivotal in deploying applications in a containerized environment and act as a centralized storage and distribution point for container images.

When a container is launched, it's the registry that the system reaches out to pull the required container image. This image is then used to instantiate a running container on the host system. Given this, the security and reliability of the registry are of utmost importance. If a registry is compromised, it can lead to the deployment of compromised images, which can have far-reaching security implications.

However, the role of a registry extends beyond just storage. Registries also manage image versions and keep track of different image variants. For example, developers might have different images for development, testing, and production environments, each with slightly different configurations. The registry keeps track of all these variants, allowing developers to pull the appropriate image as needed.

But there are also security risks in using container image registries. For instance, if malicious actors gain access, they can inject malicious code into the image. This is why securing the registry, and the individual images stored in it, is crucial.

Since container image registries are a critical component of containerized infrastructure, understanding how they work and how to secure them is crucial for maintaining the integrity and security of containerized applications.

We conducted our research on manual container registry deployments inside cloud workloads since they are inherently more prone to misconfigurations and pose a valid attack vector for threat actors.

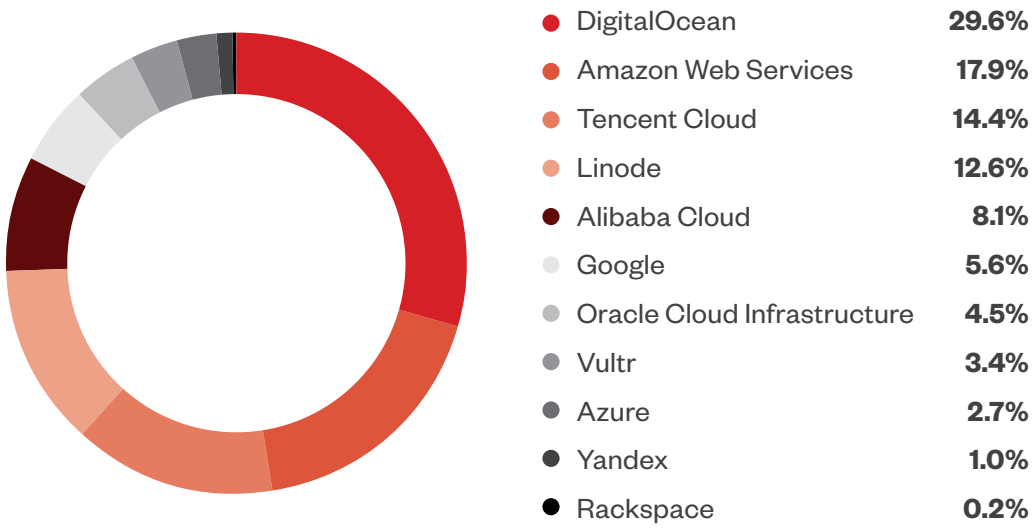

Figure 32. Distribution of publicly accessible container registries left open by users according to the cloud service provider

Open Image Registries

We analyzed open private registries and found a significant number of them hosted on a large cloud service provider. These registries had no authentication, which would permit connection and downloading of the registry content without having to utilize any brute-force methods.

Here are some key data points:

- 9.31 TB worth of downloaded images
- 197 unique registries
- 20,503 images

These figures represent a significant amount of data and an increasing number of unique registries. Registries that are available for malicious actors.

We also discovered that the ownership of these open registries was not limited to individual users and smaller companies alone. Instead, some were owned by private organizations with no other public container or code-sharing projects.

A container image is composed of code, environment variables, configuration files, libraries, and runtime functions as instructions on what components to gather, how to arrange them, and how to configure them to run the application.

We looked further into the open registries we discovered during our research and found a wealth of information, both in volume and sensitivity.[13] In the following sections, we scrutinize the contents we found to underline how problematic unsecured registries can be.

First, we noticed that some of these private registries were used as a versioning keeper: a backup of numerous versions of the same image for developed products. In our previous entry, we mentioned finding 9.31 TB worth of downloaded images, 197 dumped unique registries, and 20,503 dumped images; however, these numbers come from only the latest three versions of repositories that had versioning behavior, suggesting that actual numbers might be exponentially more significant.

Image Analysis

Another disturbing finding from our research revealed source code leakage. This severe security issue unveils configurations allowing attackers to generate valid tokens or open authorization (OAuth) flow secrets to gain unauthorized access to systems and data. The source code availability could allow a malicious actor to impersonate application users, further escalating the damage. In API language, we found third-party API usage, structure, and secret generation mechanism.

The most alarming discovery in our investigation was the leakage of sensitive information, including secrets found within the file system and container image metadata that specifies environmental variables using environment directives. The following list shows the types of sensitive data we found within the images:

CSP Access Keys

These keys could give attackers access to cloud service providers (CSPs), potentially allowing them to spin up resources at the organization's expense, access more data, or carry out attacks from the organization's cloud environment.

S3 Keys

These keys give attackers access to cloud account. Depending on permissions threat actors get information about the company, alter services settings or spin up cloud resources at the organization's expense. This allows them to access even more data, or carry out attacks from the organization's cloud environment.

Database (DB) Authentication Credentials

These credentials give an attacker databases access, allowing them to steal, modify, or delete data.

Application Credentials

These give an attacker access to private applications, allowing them to act as legitimate users or access more sensitive data.

JSON Web Tokens (JWT)

These are often used for authentication and while they remain valid the token gives attackers access to protected systems or data.

In the following section, we look closely at each file type and the risk they pose if accessed by malicious actors.

.Env

.Env files are perhaps the most sensitive files we found in this research. These files are widely used in programming environments to store environment variables and can include database credentials, API keys, and other secrets the application needs to run. However, these should not be hard coded into the application's source code.

If a .env file is exposed, an attacker gains valuable information, often allowing to access multiple resources mentioned above, which leads to unauthorized database access, unauthorized API calls, or other unauthorized actions not only on the application but also on other services associated with the project, such as cloud storage buckets and remote databases. The potential impact range from data theft to service disruption and even infrastructure takeover, depending on the permissions associated with the compromised credentials.

Dockerfile

Dockerfiles are the backbone of Docker images and containers. They contain instructions that Docker uses to build an image, which can then be used to run containers. These instructions can include the base image to be used, the commands to be run during the build process, the ports to be exposed, the files and directories to be copied to the image, and the command to be run when a container is started from the image. Dockerfiles can also contain sensitive information, such as environment variables with API keys, database credentials, or other secrets.

If a Dockerfile is exposed, it reveals how the container images is constructed, including its dependencies, revealing security vulnerabilities in the software stack. It also provide an attacker with information about the internal structure of the Docker image: the locations of important files or directories, the commands used to run the application, and any exposed ports.

Figure 33. Example of Dockerfile including AWS cloud-related credentials

Application_default_credentials.json

These files are used in Google Cloud Platform (GCP) environments and contain service account credentials. These credentials authenticate applications running on GCP, allowing them to interact with other GCP services. The file typically contains information on the type of account, the client ID and client secret, the authorization URI, the token URL, the authentication provider x509 certificate URL, and the client x509 certificate URL.

If this file is exposed, an attacker will try to access the associated Google Cloud services without authorization. He may get sensitive data stored in GCP services, perform unauthorized modifications to GCP resources, or unauthorized actions such as starting or stopping services. The impact range from data theft to service disruption or even infrastructure takeover, depending on the permissions associated with the compromised service account.

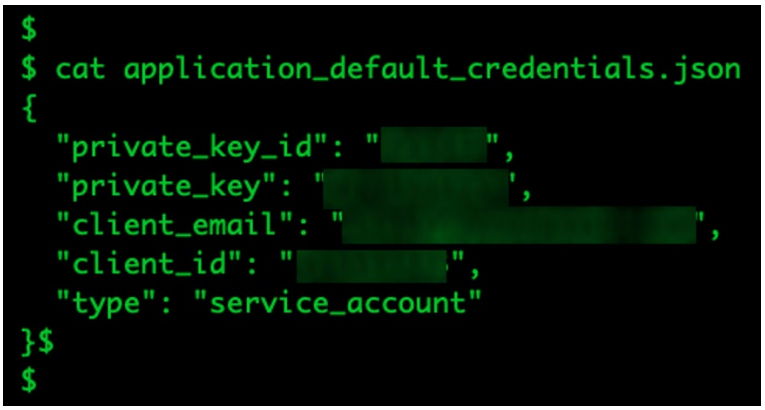

Figure 34. Sensitive information exposed in clear text on "application_default_credentials.json" files

The list does not end here; we can enumerate more specific cases, at the risk of digressing from this paper's primary topic. The main thing we wanted to point out here is that API security does not end with securing endpoints using WAF or another security product; API security is a process in itself. It is not only about API authentication or authorization but also about source codes, build scripts, deployments, continuous integration and continuous delivery/continuous deployment (CI/CD) chain, third-party usage, and secrets storage. A single mistake, putting the whole API deployment at risk.

Security Recommendations for Container Image Registries

In this case, the main security issue started with open container registries. Nonetheless, thoroughly investigating them revealed associated risks that development operations should be aware of.

We recommend the following to mitigate security risks:

- Do not hard-code secrets within files inside container images.
- Use vaults to store secrets and reference them.
- When using environmental variables for secrets, do not store them within Dockerfiles or .env files. Instead, inject them at the container runtime.
- Make sure your private container registries are not accessible to the public.
- Encrypt container images.

Container image registries should be regularly checked for misconfigurations and constantly scanned for vulnerabilities, malware, and secrets. For a lightweight and secure way to run applications, go distroless[14] and use a secrets manager to inject secrets into container runtimes.

By understanding these challenges and taking proactive steps to mitigate them, developers and security professionals can harness the power of containers without exposing their applications to unnecessary risk.

Internal APIs

APIs have been adopted as internal communication mechanisms between different software layers and microservices and might not necessarily be intended to be called by the user.

Azure WA Agent and Service Fabric

Azure Service Fabric is a distributed platform for deploying, managing, and scaling microservices and container applications.[15] It is available for Windows and Linux platforms, providing multiple options for application deployment. Azure offers two types of Service Fabric services: managed and not managed. Service Fabric's managed service puts the responsibility for the configuration and maintenance of nodes on the cloud service provider.

In a non-managed cluster, the user is responsible for maintaining the nodes and their proper configuration and deployment settings.

Service Fabric is made of virtual machines (VMs) referenced nodes. Each node uses Docker as a container engine, spawning the deployed applications inside a container.

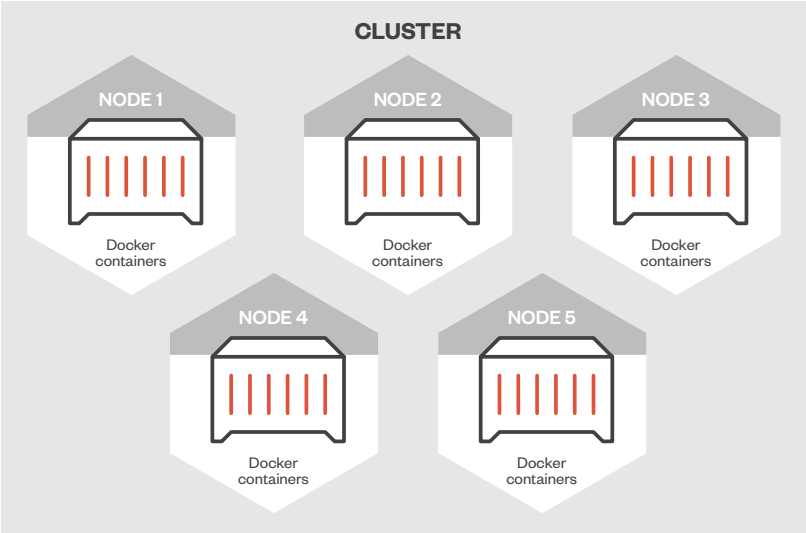

Figure 35. A simplified diagram of Service Fabric's cluster deployment

To establish communication with the cluster, a user must authenticate using a client certificate generated for the cluster. This certificate is used for accessing the dashboard and deploying CLI applications. It is essential to ensure the confidentiality of this certificate, as its exposure would compromise the entire cluster.[16]

WA Agent

WA agent is the default component allowing remote management of Azure VM instances and thus is installed on every Azure VM by default. The deployment also includes Services Fabric, as its cluster consists of Azure VMs. The agent communicates with a "wire server" using its HTTP endpoints.

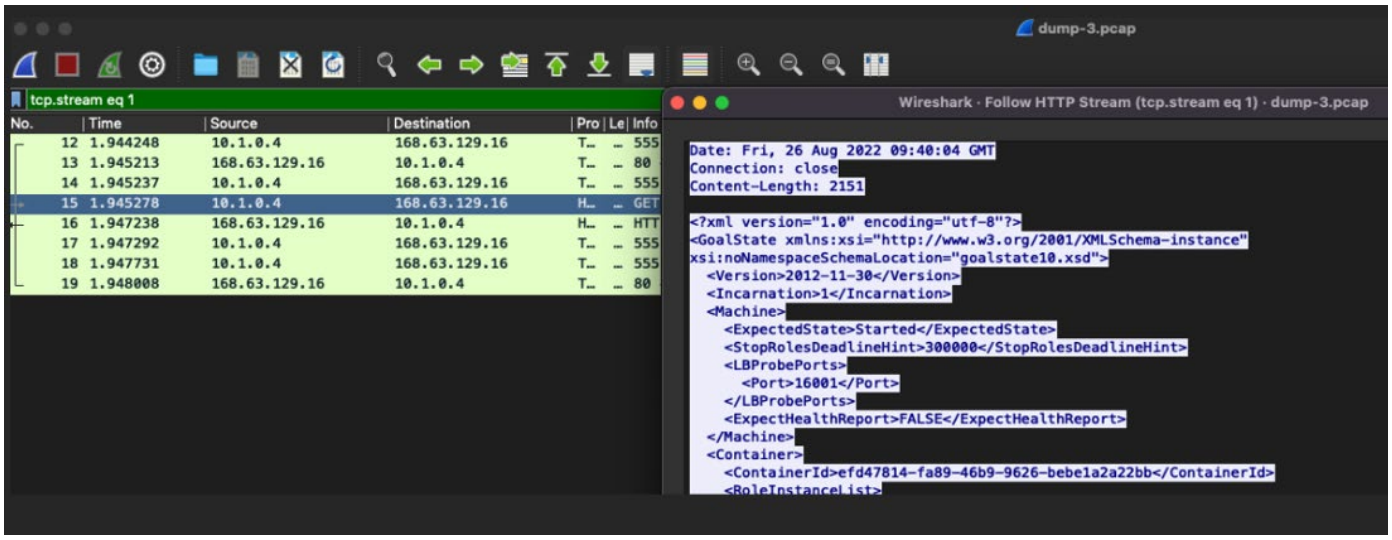

Figure 36. Example of http traffic between WA Agent and wire server

Previously, Intezer researchers[17] found a privilege escalation vulnerability allowing unprivileged users to promote themselves to the root. Microsoft later assigned it CVE-2021-27075 (Azure Virtual Machine Information Disclosure Vulnerability).

The exploitation was possible due to leaking associated certificates used to encrypt sensitive information within WA Agent extensions, which allowed remote execution of commands, adding users, and other consequential actions. This vulnerability should have been fixed by the time of disclosure (May 2021). However, since the scope of the vulnerability was un-privileged users, promoting itself into root on Azure VM, no one thought about Service Fabric implications, which led us to the discovery of CVE-2023-21531 (Azure Service Fabric Container Elevation of Privilege Vulnerability).[18]

CVE-2023-21531

Using the same principle as Intezer researchers, we provided our certificate to export Service Fabric cluster certificates. These certificates offered complete control of the cluster from the attacker-controlled container with root permissions.

It should be noted that container root permissions are not equal to host permissions, which effectively crosses the security boundary.

```
root@Type860000000:/home/fabric1# iptables -l -t security
iptables -l -t security
iptables v1.6.1: unknown option "-l"
Try `iptables -h' or 'iptables --help' for more information.
root@Type860000000:/home/fabric1# iptables -L -t security
iptables -L -t security
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             168.63.129.16        tcp dpt:domain
ACCEPT     tcp  --  anywhere             168.63.129.16        owner UID match root
DROP       tcp  --  anywhere             168.63.129.16        ctstate INVALID,NEW
root@Type860000000:/home/fabric1#
```

Figure 37. Original fix of CVE 2021-27075

After further investigation, we found that the original fix was only limited to the root user, which also applies to the root inside the container.

Internal API Security

This scenario demonstrates the importance of internal API security, especially when they are available inside external products. Users should be aware that a lack of authentication or authorization leads to a similar outcome as everyone is able to perform the API request and exploit internal implementation, like in the case of the wire server.

The other problem lies in the fix. Instead of implementing a proper authorization mechanism on the server side, a shortcut has been used by creating a firewall rule on the host, allowing exploitation in Service Fabric.

Managed Identities and Token Scopes

Managed Identities represent another example of an internal API security risk we found inside Azure Cloud. Managed identities allow users to authenticate services available within Azure cloud. Managed identities provide an interface to obtain a token dynamically that is then used for authentication.[19]

The user sends an API request to a Managed Identity service to obtain a token. For instance, an HTTP request is made to the MSI_ENDPOINT environment variable, authenticating using the MSI_SECRET. In a serverless environment, the requests were sent to the internal API service listening on localhost.

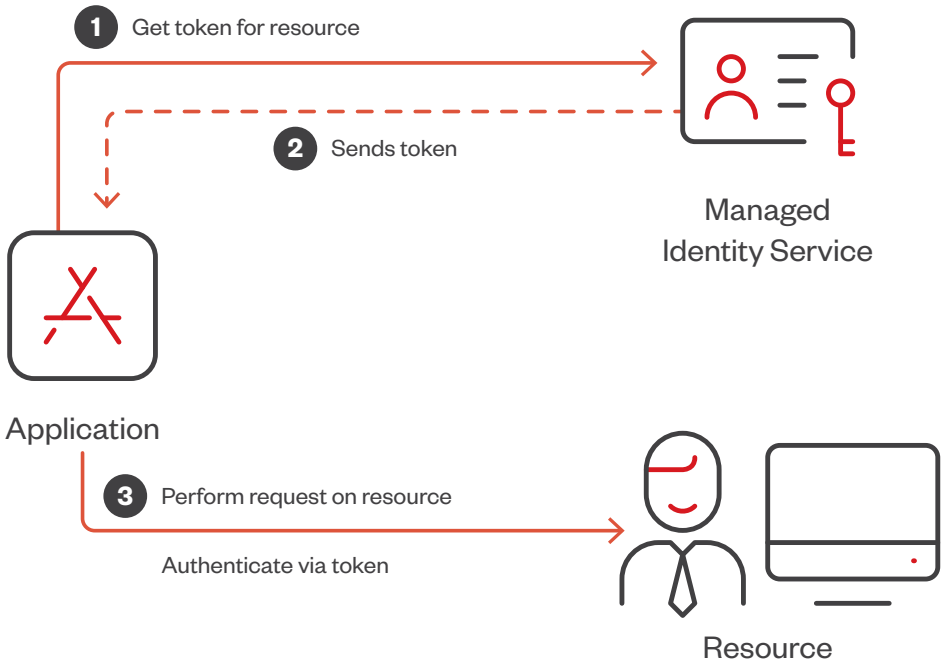

Figure 38. Internal API for managed identities

We analyzed the service binary and found that the request is forwarded to the publicly accessible endpoint and authenticated using a client certificate. The certificate itself, together with other necessary parameters, is injected using environmental variables and used by the binary.

While the certificate itself is hidden within the encrypted blob, the problem is that the decryption key was also present inside environmental variables in plain form. This allowed us to decrypt the whole encrypted context, including the client certificate and all necessary parameters for the public proxy. We found out that none of these variables were necessary in the environment.

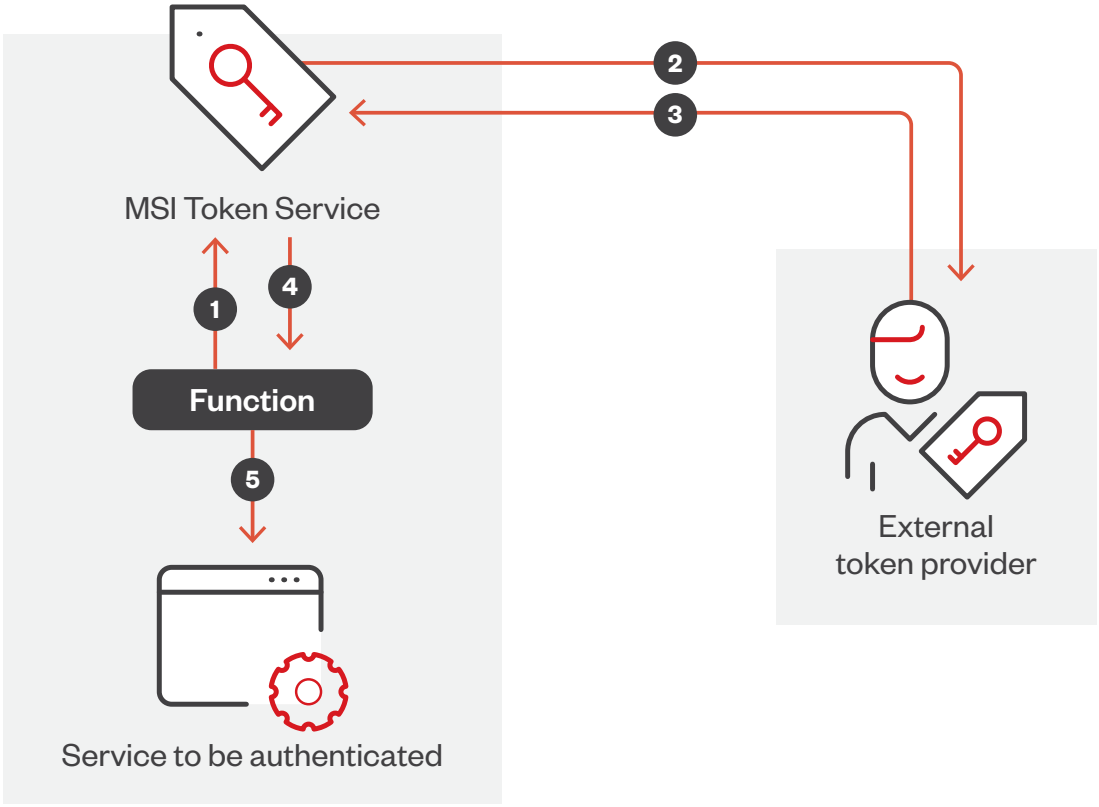

Figure 39. Internal API forwarding to the public endpoint

Their leakage allowed us to obtain a valid Managed Identity token outside the Azure environment. We crafted a request to the public endpoint by leaking the environment variables. We received a reply with a valid token and verified it works outside Azure.

We found a similar issue in Azure Machine Learning service.[20] Using a pair of certificates and its private key, we fetched Storage Account credentials, AML workspace metadata, and Managed Identity tokens off the assigned resource; in our case, a Compute Instance.[21] Additionally, the logs generated from a malicious activity are not differentiable from regular activities as they are missing actionable indicators for the location (e.g., IP address) to determine where the request originated from. Such untested edge cases in internal cloud agents APIs lead to the surfacing of bugs that allow attackers to persist in cloud environments without detection, even when cloud native logging is enabled.

The Importance of Token Validity

This is another example of improper internal API usage posing security risks. Information disclosure of environment variables revealed internal API implementation details and further security issues. As we can see, token validity matters, and thus, we should think of what scope the token is valid. For instance, it is not a good idea to issue a token valid to internal infrastructure that can be used from outside services.

Reinventing the Wheel – Implementing HTTPS over HTTP

Azure DevOps server provides an alternative to existing CI/CD solutions especially attractive when using Visual Studio for development. We analyzed the default security of Azure DevOps 2020 on-premise server in 2021.[22]

One of the tasks of the CI/CD pipeline is to execute jobs such as pulling source code from the source code management (SCM) and executing various pre- and post-build tasks. The tasks are executed on connected working nodes running Azure DevOps agent.

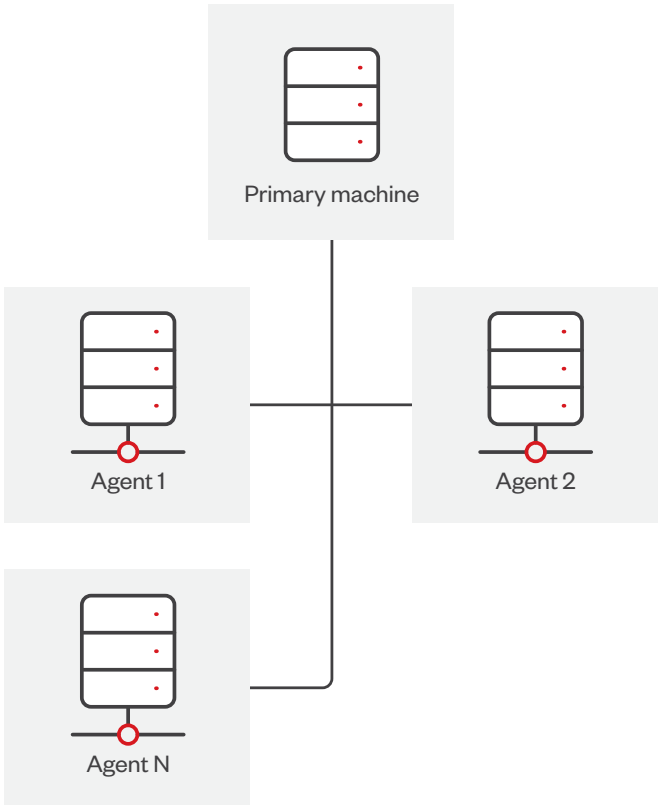

Figure 40. The main server job agents' architecture of Azure DevOps 2020

To be able to execute such tasks, an API is used. The agent creates an HTTP(s) connection to the server. The HTTP protocol is used by default, meaning the traffic is transferred unencrypted, allowing its capture, especially in local area networks in on-premise deployments.

Knowing this is a security issue, Microsoft encrypted confidential data such as the task execution parameters inside the HTTP request. During initial configuration, the agent generates a new RSA public-private key pair and sends it with other agent metadata to the server, storing it in its database. However, there is a problem: the server doesn't generate a key pair for the agent to be able to verify its responses (this will play a significant role later).

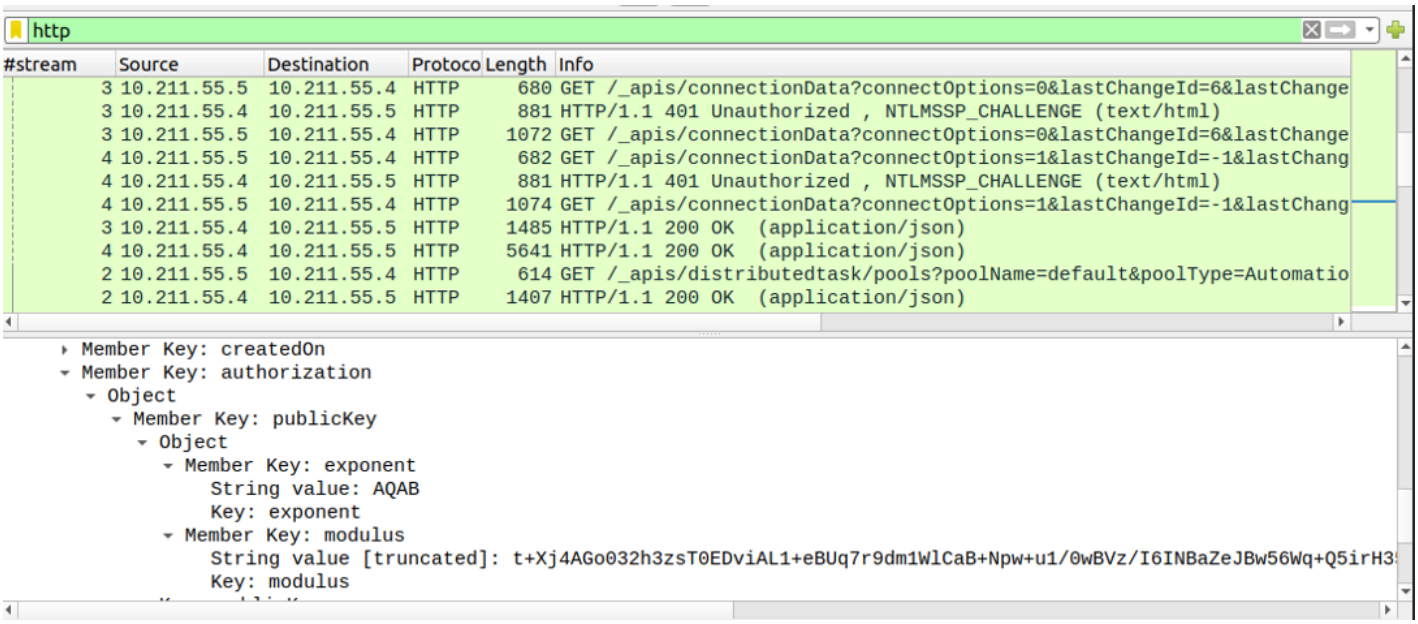

Figure 41. The transmitted public key during agent configuration

Once the agent is running, it will connect to the server through a series of HTTP connections that are kept active and wait for the jobs.

However, when the agent connects to the server, a session is established, during which the server generates an Advanced Encryption Standard (AES) key, which is encrypted by the agent's public key by default. This means that only the agent can decrypt the encrypted message's content, which contains the build definition. As this might seem like bulletproof security at first glance, it is imperative to point out the existence of the "encrypted" property in the session initiation response.

This property is connected to the AES key generated by the server. Upon further analysis of the agent binaries, we discovered that it is possible to send an unencrypted version of the AES key.

```
private ICryptoTransform GetMessageDecryptor(Aes aes, TaskAgentMessage message)
{
    if (this._session.EncryptionKey.Encrypted)
    {
        using (RSACryptoServiceProvider key = base.HostContext.GetService<IRSAKeyManager>().GetKey())
        {
            return aes.CreateDecryptor(key.Decrypt(this._session.EncryptionKey.Value, RSAEncryptionPadding.OaepSHA1), message.IV);
        }
    }
    return aes.CreateDecryptor(this._session.EncryptionKey.Value, message.IV);
}
```

Figure 42. Decryption routine allowing unencrypted key

After our discovery, we checked to see if performing a remote code execution (RCE) attack on the agent is possible when an attacker is on the same network segment or has access to a network device that is on a path between the server and the agent.

We have confirmed that one of the supply chain attack scenarios would consist of the following steps:

- Performing Address Resolution Protocol (ARP) spoofing, an attack that allows a malicious actor to send fraudulent ARP messages over a local area network to link the malicious actor's media access control address with the server's IP address. This will permit a malicious actor to intercept connections and pretend to be a legitimate server. Simulating server replies to agent requests forces the agent to execute arbitrary commands.
- Execute a successful attack on an Azure DevOps Server agent.

Takeaway

Instead of reinventing the wheel and implementing a custom encryption layer over non-encrypted channels, it makes more sense to enforce already verified and well-implemented solutions such as TLS on HTTP, also known as HTTPS, and avoid security issues when the default HTTP configuration is used.

The main problems within the implementation are:

- The agent does not verify if the reply is from the legitimate server or otherwise.
- The AES key that is used for encrypting a job specification can be successfully replaced by a custom AES key by setting the "encrypted" property to "false," forcing the agent to accept the unencrypted custom version of the AES key. Even if this feature isn't present, the attacker can sniff out a public key transmission while the agent is being configured and use it for encrypting its own AES key.

Endnotes

1 D. Fiser and A. Oliveira. (Dec. 14, 2023). *Trend Micro*. "Threat Modeling API Gateways: A New Target for Threat Actors?" Accessed on Aug. 8, 2024, at: Link.

2 D. Fiser and A. Oliveira. (Mar. 8, 2024). *Trend Micro*. "Apache APISIX In-the-wild Exploitations: An API Gateway Security Study." Accessed on Aug. 8, 2024, at: Link.

3 Apache APISIX. (n.d.). *Apache APISIX*. "Apache APISIX documentation." Accessed on Aug. 8, 2024, at: Link.

4 CVE Program. (Feb. 11, 2022). *CVE.org*. "CVE-2022-24112." Accessed on August 8, 2024, at: Link.

5 Z. Luo and Sylvia. (Dec. 30, 2022). *Apache APISIX*. "Release Apache APISIX 3.1.0." Accessed on Aug. 8, 2024, at: Link.

6 D. Fiser and A. Oliveira. (Aug. 17, 2022). *Trend Micro*. "Analyzing the Hidden Danger of Environment Variables for Keeping Secrets." Accessed on Aug. 8, 2024, at: Link.

7 D. Fiser and A. Oliveira. (Sept. 29, 2022). *Trend Micro*. "Stronger Cloud Security in Azure Functions Using Custom Cloud Container." Accessed on Aug. 8, 2024, at: Link.

8 N. Surana. (Aug. 17, 2023). *Trend Micro*. "Uncovering Silent Threats in Azure Machine Learning Service Part I." Accessed on Aug. 8, 2024, at: Link.

9 J. Reschke, (Sept. 2015). *Internet Engineering Task Force*. "RFC-7617 The Basic HTTP Authentication Scheme." Accessed on Aug. 8, 2024, at: Link.

10 D. Fiser and A. Oliveira. (May 21, 2024). "Kong API Gateway Misconfigurations: An API Gateway Security Case Study." Accessed on Aug. 8, 2024, at: Link.

11 Kong. (n.d.). Kong. "Supported Vault Backends." Accessed on Aug. 8, 2024, at: Link.

12 D. Fiser and A. Oliveira. (Sept. 26, 2023). *Trend Micro*. "Exposed Container Registries: A Potential Vector for Supply-Chain Attacks." Accessed on Aug. 8, 2024, at: Link.

13 D. Fiser and A. Oliveira. (Oct. 9, 2023). *Trend Micro*. "Mining Through Mountains of Information and Risk: Containers and Exposed Container Registries." Accessed on Aug. 8, 2024, at: Link.

14 A. Oliveria and R. Bottino. (Sept. 7, 2022). *Trend Micro*. "Enhancing Cloud Security by Reducing Container Images Through Distroless Techniques." Accessed on Aug. 8, 2024, at: Link.

15 tomvcassidy et al. (July 15, 2022). *Microsoft*. "Overview of Azure Service Fabric." Accessed on Aug. 8, 2024, at: Link.

16 D. Fiser. (June 21, 2023). *Trend Micro*. "Gaps in Azure Service Fabric's Security Call for User Vigilance," Accessed on Aug. 8, 2024, at: Link.

17 P. Litvak. (May 11, 2021). *Intezer*. "CVE-2021-27075: Microsoft Azure Vulnerability Allows Privilege Escalation and Leak of Private Data." Accessed on Aug. 8, 2024, at: Link.

18 CVE Program. (Jan 10, 2023). *CVE.org*. "CVE-2023-21531." Accessed on Aug 8, 2024, at: Link.

19 D. Fiser, (June 14, 2022). *Trend Micro*. "An Analysis of Azure Managed Identities Within Serverless Environments." Accessed on Aug 8, 2024, at: Link.

20 N. Surana and David Fiser, (Apr. 22, 2024). *Trend Micro*. "You Can't See Me; Achieving Stealthy Persistence in Azure Machine Learning." Accessed on Aug. 8. 2024, at: Link.

21 vijetajo et. al. (Jan. 17, 2024). *Microsoft*. "What is an Azure Machine Learning compute instance?" Accessed on Aug. 8, 2024, at: Link.

22 D. Fiser. (Apr. 13, 2021). *Trend Micro*. "HTTPS over HTTP: A Supply Chain Attack on Azure DevOps Server 2020." Accessed on Aug. 8, 2024, at: Link.