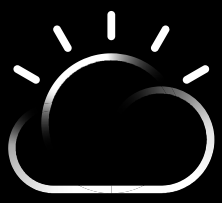


Principles for API Security

IBM

Principles for API Security

APIs allow fast and easy access to corporate assets. If you are focused on security this may be a scary thought! But the value obtained using business APIs – easing consumption of corporate assets enabling speed to market, allowing the business to reach more potential customers, and helping drive faster innovation – is significant. It is the foundation of the “API Economy” and a core component in enabling digital transformation and building digital ecosystems. Because the value provided by APIs is so high, APIs are a target for exploitation by those wishing to inappropriately access your business assets or cause damage to your enterprise. Therefore, API security is of paramount importance in gaining the promised benefits without exposure to negative consequences.

Focus on security is an ongoing effort as hackers continue to try new techniques to break into systems. It is not possible to declare security tasks completed nor should you assume your APIs are ever 100% secure. But there are principles, technologies, and techniques that can minimize the risk and provide the highest probability of success in stopping both intentional and inadvertent misuse of business assets. The goal of this paper is to focus on a set of security principles to drive the highest possible level of API protection.

Security discussions are often very technical, delving into **how** to deliver a desired security capability. Rather than focus on how, this paper focuses on **why** - highlighting the objectives that need to be achieved to have a more secure posture, and why not meeting the principle is a potential security issue. Technology is ever evolving with new technologies emerging all the time. Security principles are longer lasting. While the technical implementation of a principle may change, the principle should remain valid. Rest assured IBM has significant security skills and technologies to help in the implementation, i.e. the "how", of any and all the defined principles.

The intended audience for this paper is IT leaders, architects, and security teams who are considering API solutions and need help defining criteria and planning for API security concerns.

This paper also assumes an IT environment that is modernizing. While many businesses have historically had all their assets in one location (i.e. on-premise - inside your company), the new IT world is evolving, extending beyond that model. Enterprises are moving (at varying paces) to a hybrid-cloud and multi-cloud world in addition to maintaining on-premise environments. Companies are also using capabilities provided by others – e.g. SaaS applications or Artificial Intelligence (AI). Businesses may be forming digital ecosystems that cross multiple enterprises such as implementing Blockchain solutions across business domains. Furthermore, new and existing applications are being developed or modified to be more agile using microservice architectures. This is the modernizing IT world, and is the scope considered in this paper.

Principles are grouped in the following topic areas:

- Strategic API Economy Security Principles
- Basic API Security Principles
- API Exposure, Scope, and Positioning Principles
- API Gateway Security Principles
- Recommendations

Note: The scope of this paper is specifically about API security. The principles are related only to the API itself, not the backend implementation of business capability that is invoked by the API. In fact, mixing what should to be done at the interface, and what should be done behind the interface is an example of a security exposure discussed later. Since the paper's focus is only regarding API security, it does not attempt to cover other aspects of security such as identity management, firewalls, encryption, etc. These are all critical security elements. To the extent that these interact with API security the topics are covered, but only within the context of that relationship.

Strategic API Economy Security Principles

First let's establish the goals that drive our API security intentions. This initial group of principles establishes our overall strategy and direction for API Security.

Principle 1: Protect access to data and transactions

The overriding principle which drives all the others is the protection of the access to data and transactions. The system needs to ensure that appropriate audience(s) having been authenticated, and they are the only allowable users of our APIs. Also, it is paramount that they are using only the data and transactions that they are authorized to be using. This access needs to be allowed to occur.

Conversely, API security also dictates that the inappropriate users are denied access. Security efforts must also ensure that authenticated users cannot intentionally or inadvertently access data and transactions that are beyond their authorization. If they somehow do obtain access any activity they attempt to perform is thwarted, recognized quickly, and stopped as soon as possible.

For all access situations there is a need to log, monitor, and audit. This strengthens the ability to analyze event patterns and further to ensure any security exposure is captured as soon as possible.

Principle 2: Use business APIs as a point of control

One school of thought might be that if Business APIs are attractive as an attack point why not avoid using them? Aside from losing the business benefits, this opens a larger security issue by exposing multiple points of entry rather than using the API as a controlled point of access.

Jeff Bezos famously issued an API mandate for all Amazon's employees around 2002. Here is the text of the Bezos (Amazon) Mandate:

1. *All teams will henceforth expose their data and functionality through service interfaces.*
2. *Teams must communicate with each other through these interfaces.*
3. *There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*
4. *It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols — doesn't matter.*
5. *All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.*
6. *Anyone who doesn't do this will be fired.*

While Jeff Bezos was not primarily concerned about security when issuing this mandate, it does illuminate areas of security concern. First look at the alternatives listed for other forms of inter-process communication (point 3). There are many listed, and this is only a subset of the possibilities. Beyond this, also consider external connectivity (point 5). Trying to securely lock down so many possibilities of entry is a far greater challenge than focusing on a consistent point of control for interaction.

Note that the API approach is for both internal and external integration. If it is possible to ensure all data and functionality is exposed through a consistent paradigm (i.e. APIs) then our security efforts can concentrate on these APIs as a point of control for security, as well as logging, auditing, and monitoring. Also note that there may be multiple technologies involved (point 4), so this is not to say that there is only one integration pattern or only one possible technical implementation.

In short, there should be no 'unofficial' interfaces when communication happens between separately owned software domains. Inter-connectivity between domains should be 'deliberate' and 'governed' and APIs are currently the main mechanism for providing the light touch governance required.

Principle 3: Minimize potential exposure

It is important to mitigate risk. While our first efforts are to block inappropriate access, there are cases where the first line of defense is penetrated – e.g. someone inappropriately obtains a password, or a hacker breaks into the DMZ. The question is how to limit the damage that can be done if a bad actor gets past the first barrier? There are several techniques (covered in further detail later) that can be used to shield our data and transactions:

- Limit the API audience to only those who need access.
- Limit the data/transaction accessed by the API to only what is needed or authorized for the user to execute the desired task.
- Limit the access to data and business thru the exposed API, this allows the exposed API to be the central control point.
- Do not expose internal data structures or interfaces directly. Hide and do not allow access to additional parameters on backend systems or additional columns of data from an API that is not intended to use these mechanisms.
- Do not allow the capability to run any form of executable not performing API tasks while in the API gateway layer of the architecture.

Plan for the worst situation and determine the potential exposure. Consider what the potential accessibility is from an API. Does the API need to provide all this capability, or can it be limited to provide only a subset of data or functionality?

Principle 4: Recognize Abnormal Situations

Bad actors attempting to penetrate systems often need to make multiple attempts. Guessing passwords, trying various queries, or changing query parameters to obtain additional data may signal a bad actor trying to obtain/penetrate your systems. Recognize and prevent such activity:

- Place roadblocks in front of such attempts.
- Recognize these abnormal situations and stop them as they occur.
- Alert appropriate personnel of such attacks.
- Log all activity to determine what might have been affected.
- Scan audit logs to look for patterns that may indicate penetration or penetration attempts.
- Design the API to conform to the best practice, as those outlined by standard bodies, like OWASP.

Principle 5: Use multiple techniques for security

No single security technique can stop everything. But just because a security technique is not 100% effective does not mean it should not be used. Putting up a fence/barrier to entry does not keep all bad actors out. Despite it not being 100% effective, having a fence is still a good idea. Similarly, security through obscurity – hiding information about your API or the data or transactions is also not 100% effective. But, making this information more difficult to obtain is part of the security toolkit that can be used to thwart inappropriate activity.

Just because you have good locks on your car, does not mean you should leave your wallet on the seat. It is the combination of security measures that add up to ensuring you are not a soft target, not the individual measures.

In addition to multiple techniques, layering is another well-established technique. Multi-factor authentication is a well-known example.

Basic API Security Principles

Ask any sports team and they will tell you to start with the fundamentals and build from there. This section looks at several “getting started” security principles to provide the fundamentals for API security. Most of these may seem obvious but as with most aspects of security it is best not to assume these are already in place.

Principle 6: Know your APIs

All active APIs need to exist in a catalog. There can be no hidden APIs that are available or intended to be called without a definition in a governed catalog. The catalog provides the foundation for API governance which manages the lifecycle of the API. It also defines the API ownership and needs to be able to provide information about the target consumer audiences and data about who is consuming the API. You cannot secure what you do not know exists.

The APIs need to have a well-defined schema for what to expect as input and output. This prevents any leakage and unintended information to pass thru.

Principle 7: Ensure APIs are only visible and subscribe-able to intended audiences

API consumers, typically using an API Developer Portal, should only be shown APIs which are available for their subscription. Showing unsubscribe-able APIs to an audience lets them know that the APIs exist and may inadvertently provide hints as to how to access, break into, and use the API.

Principle 8: Know which applications are using an API

It is extremely important to know which applications are using an API. This allows the management platform to control the API usage by that application. Typically, API consumers identify the application that they are building and registers the application as a user of the API. Each application is uniquely identified by being provided an API key and possibly a secret which is used on every API invocation. With the API key an API management solution can track the application usage, ensure there is no abnormal activity, and indeed block or limit the application's usage if necessary, without affecting other consumers of the API.

Principle 9: Provide a mechanism for application user authentication

For some scenarios it is important to know who the user of the application is, not just identify the application that is consuming the API. For example, finding a bank ATM location may not need to know who the user of the application is, but accessing a specific individual's bank account information does require knowledge of the application user.

There are several techniques for user authentication including basic authentication which prompts for a User ID and password or OAuth and Open ID which allow the user identity to be passed through the application without providing the personal identifying secret information to be seen by the application itself. Most businesses are focusing on the latter options due to the higher level of security for the application user in that their credentials are not visible to the application. OAuth provides an added benefit of tying the user's consent to the application. This is helpful with the auditing aspect of their usage.

Principle 10: API Security must integrate with existing security mechanisms

While API Authentication may introduce some new techniques for end user authentication, these must integrate with the current authentication and authorization mechanisms inside the company. Creating a full second set of identity and access management is not an option. So, API authentication must integrate with the current authentication capabilities (e.g. LDAP or User Registry, or with Json Web Token or JWT for user's identity assertion) and the identity must be able to be passed along for traditional authorization mechanisms to ensure appropriate access is allowed and inappropriate access denied.

Principle 11: Control the level of API usage

Aside from security there are many reasons to control the level of API usage including protecting backend systems from too much load or for direct monetization models that are based on usage. Security is also a key reason to control usage levels. Rate limits are defined as a number of API calls per unit of time. All API consumers should be given a defined rate limit for their usage. This can be a soft limit which issues an alert if exceeded, or a hard limit which stops additional use for a period. Users invoking an API in larger quantities than the rate limit may signify a security issue.

It is possible that use has grown over time and perhaps a new limit is required. But it is also possible that the user is attempting to overload the API (denial of service attack) or penetrate the API's intended use.

Burst limits can also be used to ensure that there is no concentrated attack over a short period of time even if the rate limits are over a longer period. For example, if a consumer has a rate limit of 100 API calls per day, they may also have a burst limit that says they can have no more than 10 API calls per minute.

Principle 12: Message integrity and confidentiality must be ensured

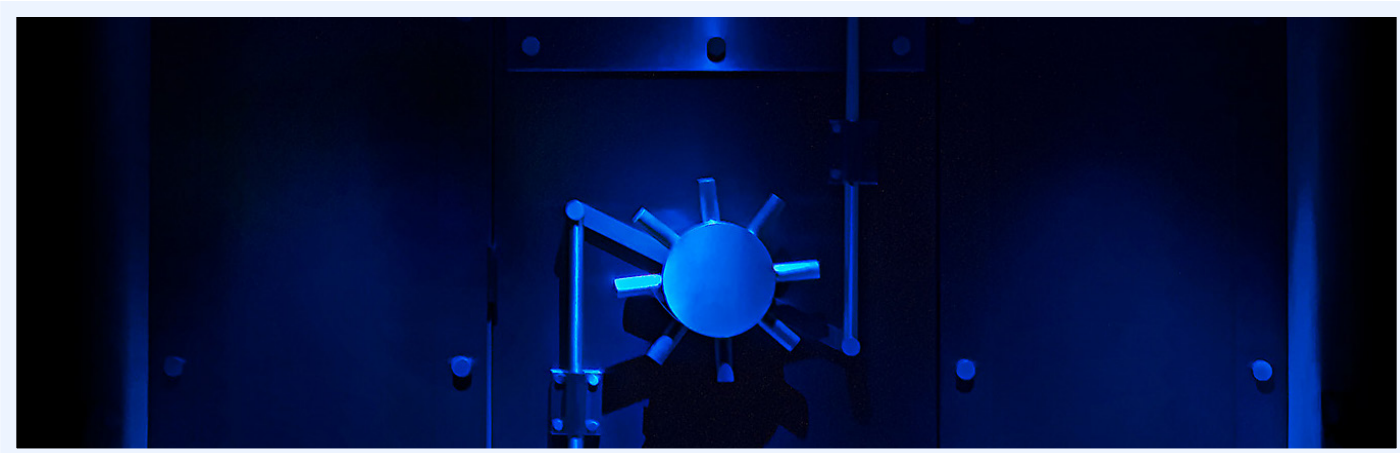
API calls must not be able to be modified between the calling application and the API end point. Typically, this is done through digital signatures and transport level protection. Digital signatures prove that the data is unchanged and that it is from the person you think it is from. These signatures used in combination with encryption of the keys known only to certain parties provide the required message integrity. In addition, it should be possible to encrypt data at rest to ensure personal or private information is always protected.

Principle 13: Provide visibility into security management

It must be possible for security or operations personnel to monitor API traffic and view any areas of potential concern. Monitoring dashboards should be available. Also, all API traffic must be able to be logged. In case of an exposure found later logs can provide the details as to what data and transactions were accessed by which API calls.

Principle 14: Enable processes for reporting potential exposure

Our first option is always to stop a security attack before it penetrates the systems. The second option is to catch it while occurring and stop the bad actor immediately. But if the bad actor does get through our security defenses, it is important to enable all channels to inform the API team that an issue has occurred. Provide a mechanism where API consumers, backend system owners, and others can report either real issues they are seeing or potential security issues they may have uncovered. The system also needs to provide a mechanism to lock out the bad actor for any future invocation.



API Exposure, Scope, and Positioning Principles

Now that the basics are covered, let's look at more advanced topics. This next set of principles are related to unnecessary exposure of the API or API parameters, the expansion of the role of an API beyond its intended purpose, and incorrect positioning of the API in the architecture.

Several of these principles are related to one another, but for simpler understanding the issues are split across multiple principles to address each principle independently.

Principle 15: Complete API Security Validation Before Invoking Business Logic

An API is an interface providing easy consumption of a business capability for an application developer. When these APIs are managed, they also provide security and analytics for the API provider. The runtime security aspect or enforcement layer in the architecture is commonly referred to as the API gateway. Behind the API that is running on the gateway (i.e. called by the API) is the programming logic that provides the desired business capability. The implementation of this backend is abstracted and invisible to the consumer and is only invoked after the API security layer is complete. The previous section covered many of the basic security principles that are handled by APIs. The API gateway is the architectural point that provides the runtime enforcement security principles that were described.

Security issues arise when the API definition expands beyond the interface management and includes business logic in the API. While APIs almost always invoke business logic in some form – backend applications, databases, SaaS applications, microservice applications, etc. –that logic must be executed in a different architecture layer. If business logic can be mixed with the API and executed in the same architectural component (i.e. gateway), then the gateway would need the ability to execute a general-purpose program to allow this to occur. This may allow a bad actor to take advantage of this ability and execute an inappropriate program in the gateway before security checking has completed.

In an external API call scenario, another (poor) option might be to do partial security checking in the DMZ and then pass traffic into the trusted zone to complete the security validation. This is also a bad idea. Unsecured traffic is now able to pass into the trusted zone in the architecture where it may introduce an inappropriate program or otherwise perform attacks before security checking completes. Security needs to complete before it is passed into a trusted area or into an area where general-purpose runtimes are accessible.

Including business logic in the API means that either the business logic is executing in the API layer where security is also being verified or that the security layer has not fully processed the API and passed it forward into a runtime that can execute business logic. Both scenarios are not desirable. It is best to keep the API separate and executed prior to the business logic.

Principle 16: Differentiate between integration logic and business logic

The nature of an API is that it receives a request, validates that it is secure, and then invokes business logic that is in the application layer of the IT infrastructure. But to take in a request and pass it forward there is a need to perform some integration capability in the API layer of the architecture. It is important to distinguish integration logic from general business logic. As previously stated, business logic should not be executed in the gateway. But the gateway can and should be able to perform basic integration and mapping functions in addition to its security purpose.

However, complicated integration needs that require significantly complex integration logic should also not be implemented in the API for the same reasons that business logic should not be in the API. The overall aim should be to minimize “composition” in the gateway (i.e. aim for 1:1 pass through, rather than 1: many invocations). The API can call a separate application integration capability to perform this type of task following the API layer security and any simple integration processing (e.g. a proxy).

The blurring or confusion between integration logic and business logic can cause poor design and open a security hole in the architecture. Beyond security, it may also cause issues with performance and maintainability.

Principle 17: Execute security for external APIs in the DMZ

First a definition - External APIs are APIs that are invoked by applications running outside the enterprise which need to invoke business capability located inside the enterprise. Note that external API calls can occur from APIs that are consumed by external programmers such as partner or public scenarios but also from internal programmers – employees of your company.

It is very common for internal programmers to work on applications such as a mobile app or applications that deal with social networks or third-party applications that include traffic flowing into the enterprise from outside (such as in a public cloud environment). Do not confuse internal or external consumer with internal or external API, these are separate concepts. This principle is discussing external APIs while the following principle discusses internal APIs.

In most enterprise architectures, calls from outside the enterprise pass through a demilitarized zone (DMZ) where security checking occurs before being allowed into a trusted zone in the architecture where business logic is executed. As previously described, it is important that all the security checking is performed in the DMZ prior to allowing traffic to enter the trusted zone. Unsecured traffic that enters the trusted zone may have access to runtimes where unsecured programs can be inserted to perform actions that should not be allowed. Allowing traffic that is not fully secured beyond the DMZ in order to complete security validation inside the trusted zone should also not be allowed.

By knowing the different type of audiences for the API, you can apply different security requirements based on the audience.

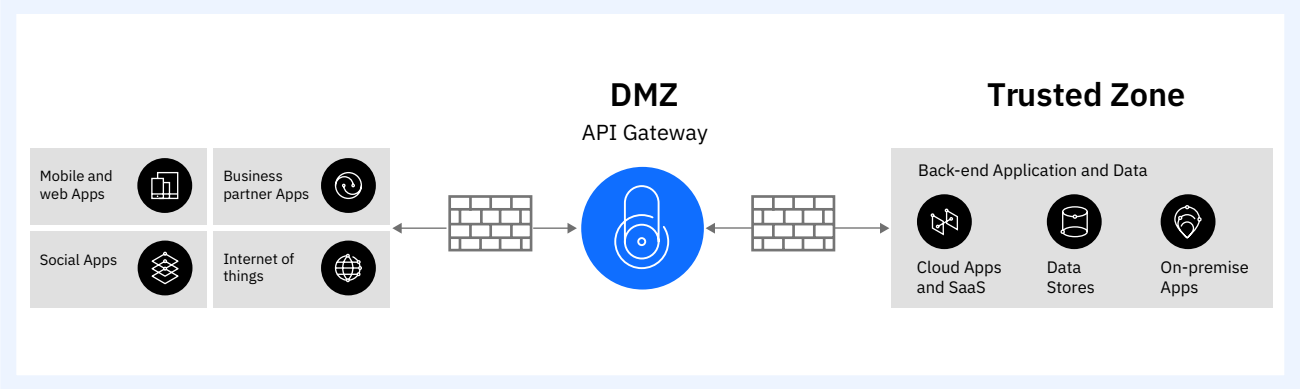


Figure 1. Securing external APIs through the DMZ

Principle 18: Internal APIs need security too

Internal access attacks are far more difficult to identify and stop. Internal security issues may occur via a rogue employee wishing to obtain/sell information, social engineering access where an outsider obtains an employees' credentials and can perform activities as if they were the employee, or inadvertent errors that expose information accidentally. The challenge is in stopping a situation where the employee is using access that they are entitled to use. A big challenge!

APIs can play a part in managing the internal security threats as well as external. Used as a control point for internal scenarios, APIs provide a mechanism to validate internal access (which may be allowable) but also restrict the amount of data returned, enforce rate limits for access, and provide an audit and logging point that can be used to understand inappropriate access by internal requestors.

An additional thought - also beware of APIs initially intended for use by internal consumers that are repurposed as external consumers later. There may be some broader access allowed in the API that was defined for internal consumers that is not appropriate for external consumers.

Principle 19: Understand when to manage an API and when not to

Not every connection needs to be done through a managed API interface. Historically applications were written in a monolithic style where internal modules were not exposed outside the application. Modules simply call one another. Control points are not needed within an application itself but could be of value between applications. In newer applications written using a microservice architectural approach the same concept still applies. Applications may be composed of many microservices and this internal application communication does not need to be via managed API. However, communication between different microservice applications may benefit by being accessible via API for both consumption and security. This is shown in the following figure.

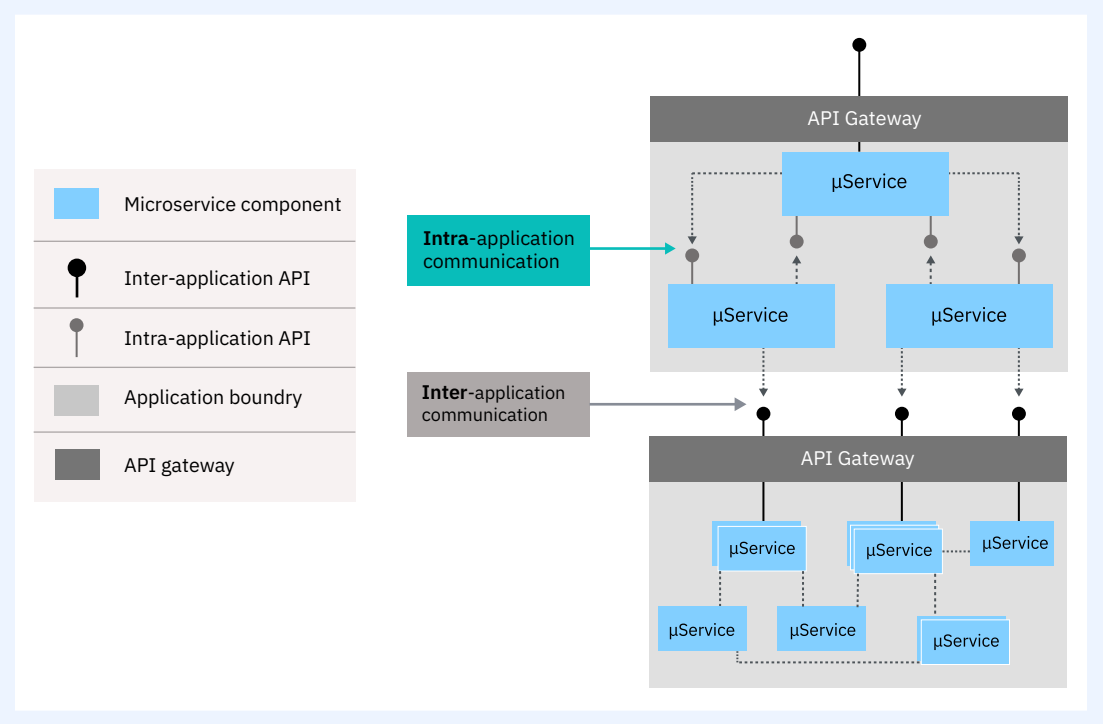


Figure 2. Inter-application vs Intra-application communication

This is sometimes referred to as “East-West” traffic for intra-application communication and “North-South” traffic for inter-application communication.

Using managed APIs between different applications (i.e. north-south) provides the security and other benefits described by API enforced control points. However, this level of security is not required for intra-application communication.

Principle 20: Know the expected request and response

An API receives a request and produces a result. The request payload needs to be well-defined, and scanned for large data, depth of the recursive elements, and active content. The request must conform to what the schema allows. The response to the API should provide just enough data it needs to account for the situation where backend services may inject additional data. It may or may not be appropriate for the new data be exposed automatically to the caller.

Principle 21: Do not expose internal data structures or program interfaces

Allowing visibility of your internal data structure or the actual set of backend parameters to invoke a transaction, provides valuable information to a bad actor looking to access these systems. Do not build your consumer interaction layer APIs based on the structure of the backend systems. Rather, focus on the functionality the API is intended to provide for the consumer. Limit the input fields to only the information the API needs to obtain from the consumer to perform its intended task. In addition to being easier for consumers to use (hiding the complexity of the backend), it also limits the ability of the consumer to use additional parameters or access other parts of databases that should not be known to valid users of the API.

Many businesses have used backend applications, the source of the desired information, to define the APIs to make this data available. This is a bad practice which very often leads to exposing more capability than is required for a consumer-oriented API.

GraphQL, a newer API technology, is used to expose data-oriented information to consumers - allowing the consumer to specify the information they wish returned from a database. Just as with other APIs, GraphQL should not expose all the columns of data contained in a database(s) or the structure of the database tables. It is best to understand the target consumer and limit their potential access. Exposing the data structures to a bad actor can enable an attack to overload or obtain information that is not appropriate for the intended audience.

Principle 22: Use APIs to meet additional security concerns

Businesses are faced with self-imposed or government-imposed regulations for consumer privacy and/or compliance with legal or other requirements. Frequently these are related to accessibility and protection of personal or confidential data elements. Using APIs as the point of control provides a secure and auditable record of protection that can be a tool used to meet the required regulations.

Principle 23: Automate security policies whenever possible

Make security as automatic as possible. Do not require API developers to remember to take API security related steps for auditing, logging, authentication, etc. Use security policies that are automatically deployed across all APIs without the API creator having to take explicit action to have them included whenever possible.

Principle 24: Retire old versions of APIs

Frequently multiple versions of an API may be active. This is normal. As you have users of an old API version and introduce a new version there is often a period allowed for migration to the new release. Once users are moved to the new release, retire the old version of the API. Leaving the old version active provides an unnecessary point of entry that can be closed. Use API management solutions that help manage the API lifecycle through API retirement.

Principle 25: Automate security testing and monitoring

Include security tests throughout the development cycle. Run automated test scripts frequently to ensure no holes have been opened. Monitor the environment and ensure responses remain as expected both from a data and performance perspective.

API Gateway Security Principles

The API Gateway is a critical component in API security which has already been mentioned several times. This section focuses specifically on principles related to the gateway itself to ensure the gateway is as secure as possible and describes how the gateway can support the requirements for API security.

Principle 26: Gateways must not contain general-purpose runtimes

Separation of concerns was discussed earlier regarding why security needs to be completed before executing business logic. This gateway principle is for the same reason but addressing the requirement from the perspective of gateway capabilities.

As the Gateway is the runtime enforcement point for security, it is important to harden the gateway to prevent as many security exposures as possible. To accomplish this, gateways should be purpose built to only execute gateway functions. This includes not having general-purpose runtimes (e.g. Java) where programs not related to gateway functionality can be executed. It also includes eliminating other full-function off-the-shelf software components that are subjects of frequent attacks (e.g. http servers), Linux or other off-the-shelf operating systems, etc. Certainly, gateways need capability to handle http requests and need to handle I/O and other operating system functions.

But the full general-purpose nature of these components needs to be made unavailable. The gateway should act like an appliance (e.g. a router) where it is a black box with no knowledge as to what is inside, and with the ability to only execute gateway related function. Many enterprises will choose a purpose-built physical hardware device for this purpose, but that is not specifically required. Software “appliances” are also reasonable options. Updates to the gateway should be executed as firmware upgrades and not rely on individual patches to the operating system and other components by system administrators as the mechanism for applying maintenance.

Principle 27: Take advantage of gateway proximity

It is a good practice to locate an API gateway near the backend data it is accessing. This is obviously good for performance but is also a benefit for security. If a business has some applications on-premise and other applications in the cloud (or multiple clouds), trying to force all requests through a single API Gateway would require that after the gateway finishes its security tasks some of the traffic would travel over the network to another location. While this can be secured, if errors occur it may open an opportunity for a bad actor to capture information.

A better option is to have gateways in multiple locations, on-premise and in multiple clouds as required, protecting the edge of the enterprise in each place. The APIs that access information from the on-premise backends can be deployed on the on-premise gateway, and the APIs that access information from cloud deployed backends can be deployed on the cloud. This minimizes the opportunity for bad actors to access the traffic.

Of course, some APIs may need to access information from multiple locations - both on-premise and in a cloud environment. In these cases, this API may call local modules directly but also access remote APIs that represent the backend applications located in another location. The remote applications are made available through an exposed API on the gateway co-located with the other backend system.

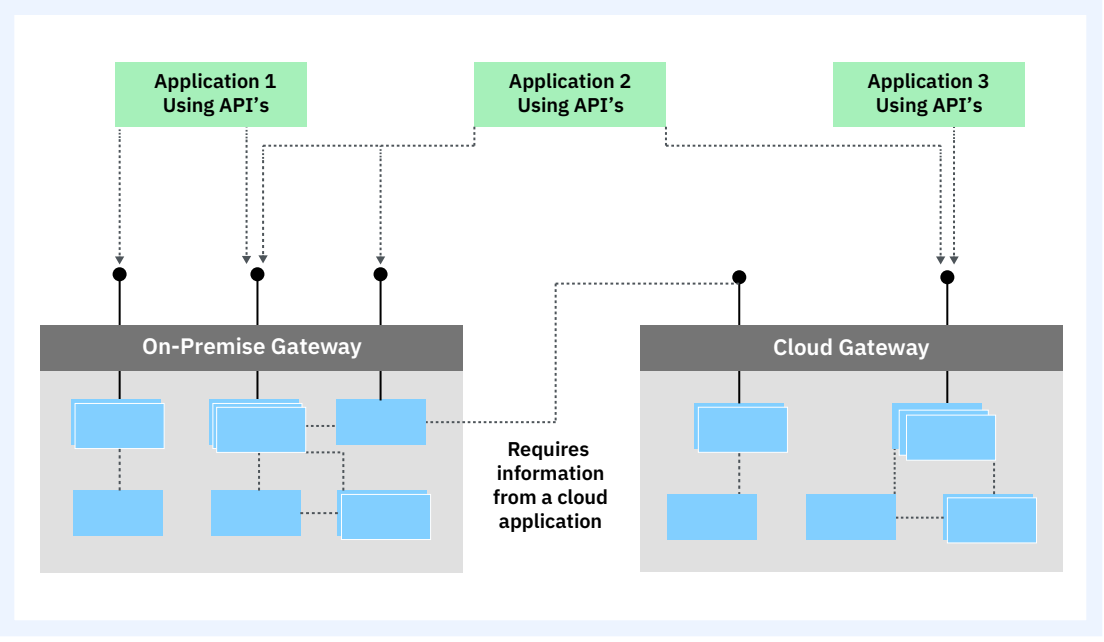


Figure 3. APIs access information from multiple locations

Principle 28: Use both internal and external gateways

The need for an external gateway to protect against security threats from outside the enterprise is very clear. Of course, internal API traffic needs security too. Like external security, the runtime enforcement for internal traffic is implemented in an API gateway. It is important to understand that it is a security exposure for internal traffic to enter the DMZ and then return directly into the trusted zone. For that reason, using the same external gateway is not appropriate for internal traffic use cases. A separate internal gateway should be deployed for internal traffic scenarios.

Some businesses are turning to a micro-gateway for security inside the trusted zone. Is this a good idea? If the micro-gateway is not able to provide the identified security principles, then probably not. The desire for a micro-gateway is for a small footprint and fast to deploy gateway, not for less functionality. Look for a fully capable API security gateway that can be deployed as a microservice in a container to obtain the deployment and management qualities of service required while still ensuring the necessary security capability.

Principle 29: Use the API gateway to handle common security threats

The API Gateway should be able to handle common security attack models including but not limited to:

- Input validation to ensure incoming calls are of the correct structure. Check for length, range, format, and type in the input fields. Verify requests for data returned are within the appropriate range.
- Injection where dangerous code is embedded into an unsecured software program to stage an attack. Most common scenarios are SQL injection and cross-site scripting. This exposure could be accomplished by transferring untrusted data into the API as part of a query or command.
- Denial of Service (DOS or DDOS) attacks where the attacker pushes an enormous amount of traffic against the API or the Gateway. The API Gateway should contribute to protecting against such attacks.
- Man-in-the-Middle attacks where the bad actor intercepts traffic between the consumer and API provider and inserts or changes information in the request. This is most often handled via encryption.

Principle 30: Monitor and analyze API traffic

What is a normal pattern for API traffic? Understanding what a typical pattern is and being able to raise an alert when the traffic pattern is not typical may signify a security exposure. Non-typical traffic may be a sign of an in-process attack or a potential security exposure. Using real-time gateway capability to recognize this situation can head off a big problem. Using analytic data and logging from the API management system can also play a part.

Recommendations

Business APIs are a primary channel to market for a growing set of companies in all industries and geographies. The estimated revenue associated with the API Economy and downstream initiatives developed on top of APIs such as digital transformation and Blockchain is incredibly large. Obtaining value through investments in this arena is extremely attractive. Therefore, it is also an attractive area for those wishing to maliciously attack these businesses using this channel. Security is critical to the success of the API Economy.

We finish with a set of recommendations to help in your security initiative:

- **Alerts** – Set up automated alerts when abnormal situations occur.
- **AI** – Artificial Intelligence can help spot abnormal or out of bounds situations and drive alerts.
- **Security champions** – Set up a group of security champions for your APIs. This group is tasked with becoming experts on current state of the art security concerns and solutions and driving security principles into the initiative.
- **Model Threats** – Take the perspective of both an external and internal attacker and model how they might attempt to penetrate your security.
- **Governance and compliance** – Work with the overall architecture team and API team on their governance and compliance tasks and ensure appropriate security principles are imbedded in their efforts.
- **Getting to "YES!"** – Shutting off access to everything provides the highest possible security but is not a viable option. The answer needs to be "yes" we can make this happen. The question needs to be how to make this happen securely.

To understand more about IBM's thoughts on Digital Business and the API Economy visit the IBM API Economy website. IBM API Connect is IBM's complete foundation to Create, Secure, Manage, Test, and Monitor APIs. You can find more information about IBM API Connect at the API Connect website. And you can also experience a trial version of API Connect.

About the author

Alan Glickenhouse Digital Transformation and API Business Strategist, IBM.

Alan Glickenhouse is a business strategist on the IBM API Connect offering management team. He joined IBM in 1981 and has held numerous positions in sales, technical sales, marketing, development and technical support. On the API Connect team, Alan assists clients in all industries, of all sizes, and in all geographies with their business strategy for Digital Transformation and APIs, understanding their business direction and existing environment (both business and technical), and helps businesses successfully adopt a strategy that fits their environment. Contact him at glick@us.ibm.com or @ARGlick.

For more information

Visit: https://www.ibm.com/cloud/api-economy
or cloud integration:
https://www.ibm.com/cloud/integration

Follow us

@IBMcloud

Facebook

Connect with us

LinkedIn

YouTube

IBM **Cloud**