# AZURE FUNCTIONS AND SERVERLESS PLATFORM SECURITY

# Executive summary

This paper explores the security of the Microsoft serverless platform and the benefits of using the serverless platform architecture. It also explores security deployment issues in serverless computing and the measures that Microsoft takes to help mitigate them.

This white paper focuses on the following areas:

- Benefits, components, and scenarios for the serverless platform
- Security issues in serverless computing
- Critical Azure serverless platform security issues and possible mitigations

## Table of Contents

# Overview

Most enterprises need a significant amount of resources and time to manage servers, which adds cost. If enterprises can use fewer resource to manage servers, they can focus on building great applications.

Serverless computing helps you do just that, because the infrastructure that you need to run and scale your apps is managed for you. Serverless computing is the abstraction of servers, infrastructure, and operating systems. Serverless computing is driven by the reaction to events and triggers which are all taking place in near real-time—in the cloud.

As a fully managed service, server management and capacity planning are invisible to the developer. The serverless framework helps you develop and deploy serverless applications by using Azure Functions. It's a command-line interface (CLI) that offers structure and automation to help you build sophisticated, event-driven, serverless architectures composed of functions and events. An Azure function is an independent unit of deployment, like a microservice. It's merely code, deployed in the cloud, that is most often written to perform a single job.

With serverless, security matters are different from what we’re used to seeing in traditional server-based applications. Moving to a cloud-hosted application back end eliminates many security concerns. One significant advantage that serverless offers is turning compromised servers and OS-level vulnerabilities into something you no longer worry about, since these worries become the responsibility of the cloud service provider. Additionally, working with serverless reduces the negative consequences of denial of service (DoS) attacks.

Despite the benefits, serverless security has its own risk factors to deal with. The serverless approach doesn’t introduce new security concerns, but it requires having an approach to existing security concerns. This white paper focuses on these security matters.

Contemporary cloud-backed applications are built as an assembly of single-purpose services. With many core capabilities (like authentication, messaging, and data) provided by many vendors, developers are left to choose the right service for the right task. The serverless compute class of services adds a key piece to this pattern—the ability to add custom logic to an app without having to reserve compute resources.

Note these important considerations about serverless computing:

- Users need to trust and have confidence in the cloud provider. Vendor lock-in is an ongoing concern.
- If a company has a large application with a lot of functions to stitch together, there is no “compiler” as in the traditional IaaS systems.
- Testing and debugging are more challenging because functions are managed individually and might be on different versions in different environments.
- IaaS vendors currently support only a limited number of programming languages, which might mean additional training for the existing team or the need to bring on new team members. It remains to be seen whether mainstream organizations will embrace this type of disjointed application platform model.

Serverless computing is still a developing technology. Microsoft Azure has a strong portfolio of services that support a serverless architecture.

# Serverless platform

The following features of serverless enable developers to focus solely on the business logic of the solution:

- Serverless computing is driven by performing some action as a response to a specific event or trigger.
- While you're building serverless apps, you don't need to provision servers or infrastructure.
- Billing is based on resources consumed or actual time that code is running.
- There's no need to build the app to run the code. Just focus on code.

## Benefits of serverless

### Pay as you go

With serverless computing, you pay for just the actual amount of compute time that your code uses. The net effect is a dramatic reduction in your monthly bill. The cost structure is changing from paying for a virtual machine (VM) for an entire period (days/month), essentially "renting" capacity, to paying for 100 msec of compute time.

### Reduced operational expenses

Serverless computing scales to handle tens of thousands of concurrent functions almost instantly (within seconds), to match almost any workload and without requiring scale configuration. It reacts to events and triggers in near real time.

### Faster time to market

Serverless computing is event-driven. Resources are allocated as soon as an event triggers them. You're charged only for the time and resources it takes to execute your code—through subsecond billing.

## The Microsoft approach

Part of the Azure PaaS portfolio, Azure Functions is an open-source serverless computing offering from Microsoft. As a solution for easily running functions in the cloud, Azure Functions is built for enterprise-level development and Internet of Things (IoT). Azure Functions also enables developers who are already working with the Azure platform to integrate functions easily into their existing Azure-based workflow.

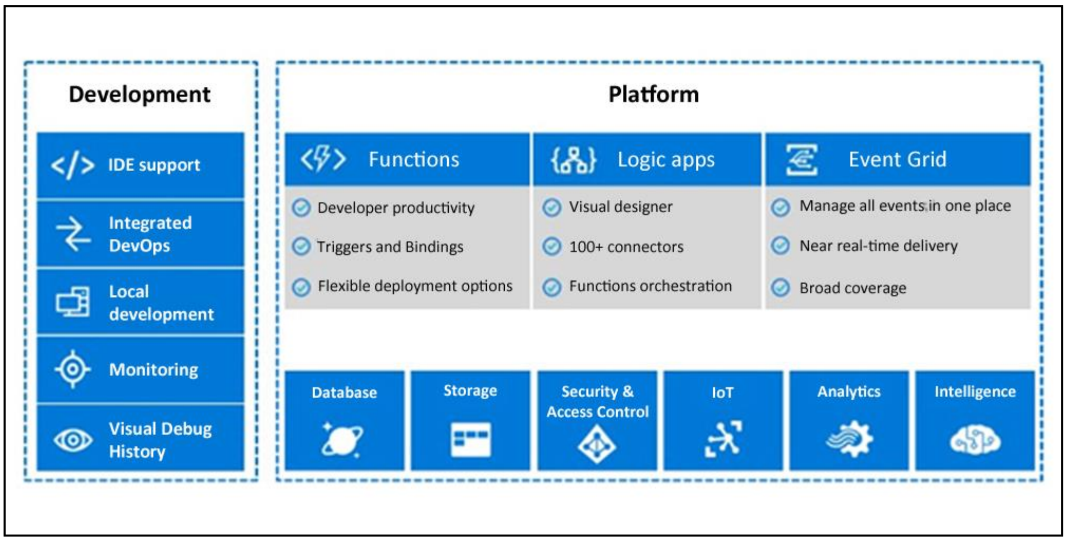
Microsoft aims to make Azure Functions portable and convenient. It runs on a user's own infrastructure, whether on-premises or on other public clouds.

## Serverless architecture and application platform components

You can think of the concepts that support a serverless architecture as three layers that sit atop existing compute, network, and storage resources:

- The serverless *fabric* removes infrastructure and operations concerns from the developer's responsibilities. The developer can write code and build a working application without managing servers.
- The event-driven programming model provides a *framework* for the creation of that code.
- *Functions* provide the packages, patterns, and reference architectures that are needed to assemble the application.

A serverless architecture includes the platform, related services, and development tools:



The term "serverless apps" or "serverless architecture" implies a fully managed set of services to build a solution. Most fully managed solutions expose specific functionality (mostly via REST APIs) and can scale to handle large loads. You can use an Azure Resource Manager template to deploy a function app.

For example, Example Azure Resource Template of a Serverless App (Logic App and Azure Function) describes how to deploy a serverless application in Microsoft Azure by using Resource Manager templates.

However, many such services don't include the ability to run user code or react to events and triggers. This ability is critical for creating a serverless application.

Examples of Microsoft Azure serverless services include Azure Cosmos DB, Storage, IoT Hub, Stream Analytics, Search, and Microsoft Cognitive Services.

Let's take a closer look at the components of serverless computing:

- Azure Functions
- Azure Logic Apps
- Azure Event Grid

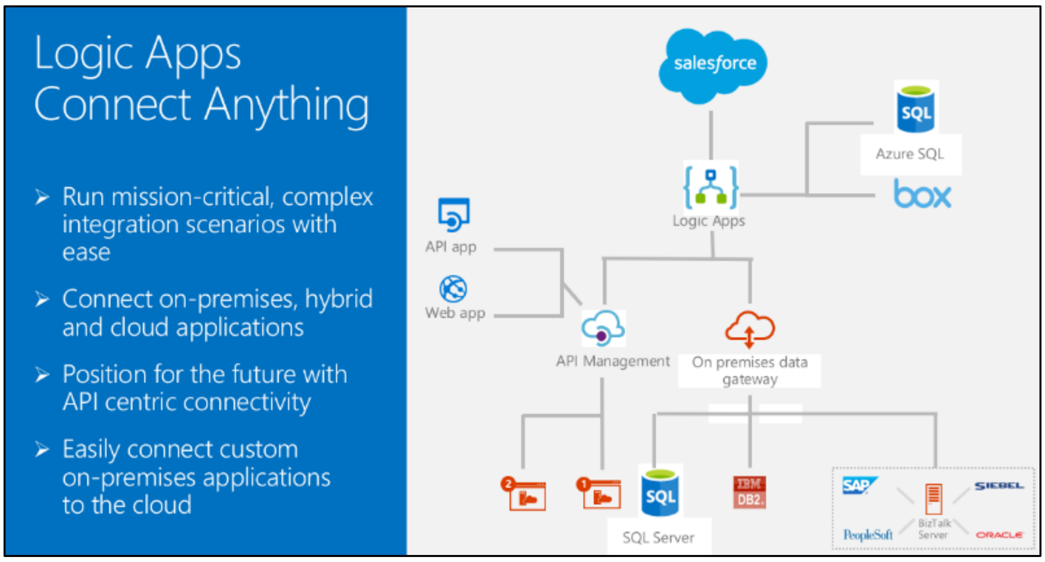
## Functions

Azure Functions is a serverless compute service that enables you to run code on demand without having to provision or manage infrastructure. You can see Azure Functions run a script or piece of code in response to a variety of events.

Azure Functions is a solution for running small pieces of code, or *functions*. You write just the code that you need for the problem at hand, without worrying about an entire application. Use the development language of your choice: C#, F#, Node.js, Python, or PHP.

## Logic Apps

The Logic Apps feature of Azure App Service provides a way to implement scalable integrations and workflows. It provides a visual designer and the ability to model and automate your process as a series of steps called a *workflow*. There are many connectors across cloud and on-premises services to quickly connect a serverless app to other APIs.



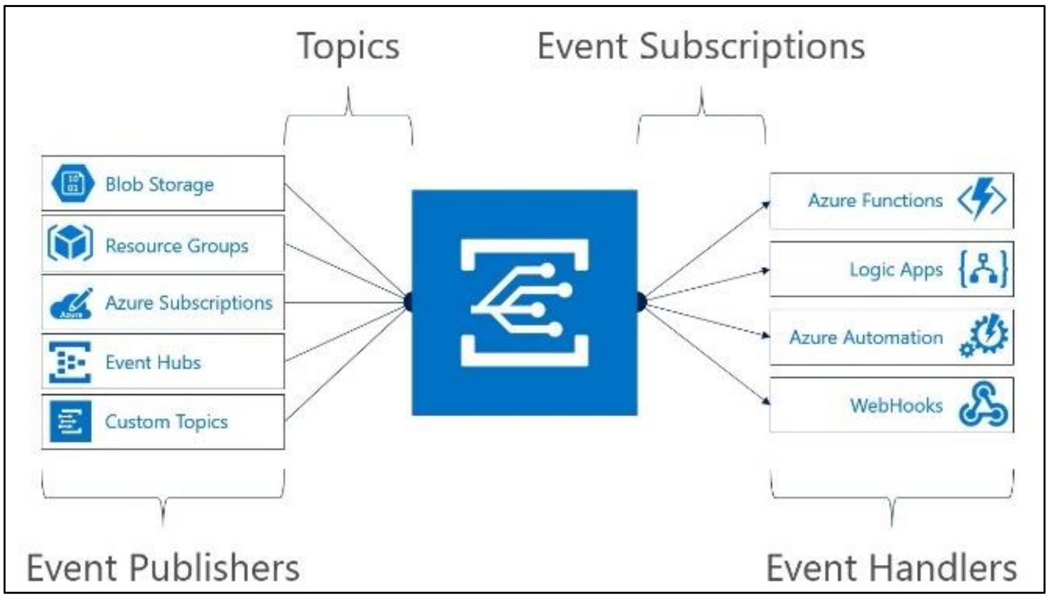
Logic Apps
Connect Anything

- Run mission-critical, complex integration scenarios with ease
- Connect on-premises, hybrid and cloud applications
- Position for the future with API centric connectivity
- Easily connect custom on-premises applications to the cloud

A logic app begins with a trigger (like “When an account is added to Dynamics CRM”) and then can begin combinations of actions, conversions, and condition logic. Logic Apps is an option for orchestrating various functions in a process—especially when the process requires interacting with an external system or API.

## Event Grid

You can use Azure Event Grid to build applications with event-based architectures. Select the Azure resource that you want to subscribe to, and give the event handler or webhook endpoint to send the event to.



Event Grid has built-in support for events coming from Azure services, like Azure Storage blobs and resource groups. Event Grid also has custom support for application and partner events, by using custom topics and custom webhooks.

## Scenarios for serverless

Azure Functions is a solution for processing data, integrating systems, working with IoT, and building simple APIs and microservices. Consider Functions for tasks like image processing, order processing, and file maintenance, or for any tasks that you want to run on a schedule.

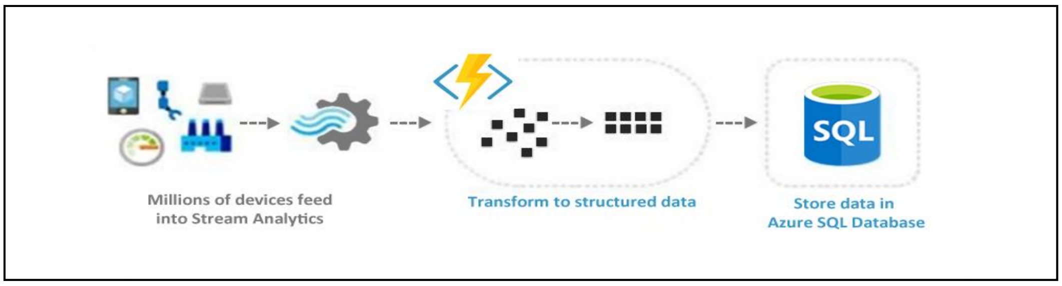
The following sections describe common scenarios.

### Real-time stream processing

Real-time stream processing:

1. Consumes messages from either queue or file-based storage.
2. Processes the messages.
3. Forwards the result to another message queue, file store, or database.

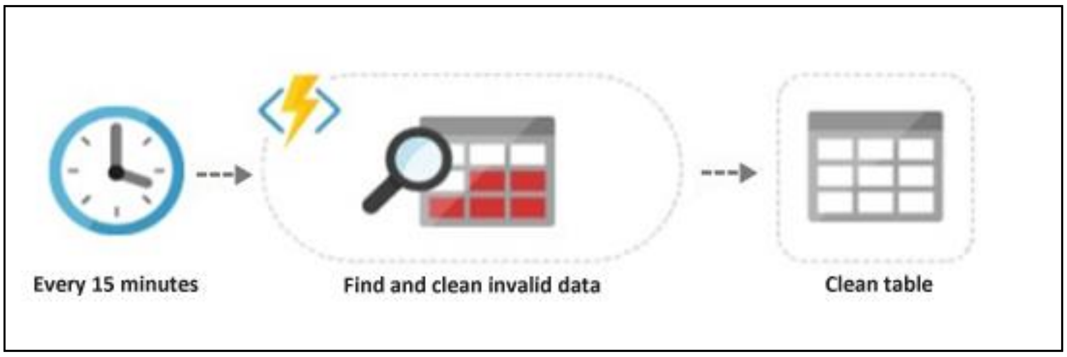
Processing might include querying, filtering, and aggregating messages. Stream processing engines must be able to consume an endless stream of data and produce results with minimal latency.


Millions of devices feed into Stream Analytics
Transform to structured data
SQL
Store data in Azure SQL Database

For example, IoT devices can send messages to Azure Stream Analytics, which then calls an Azure function to transform the message. This function processes the data and creates a new record in an Azure SQL database.

## Timer-based processing

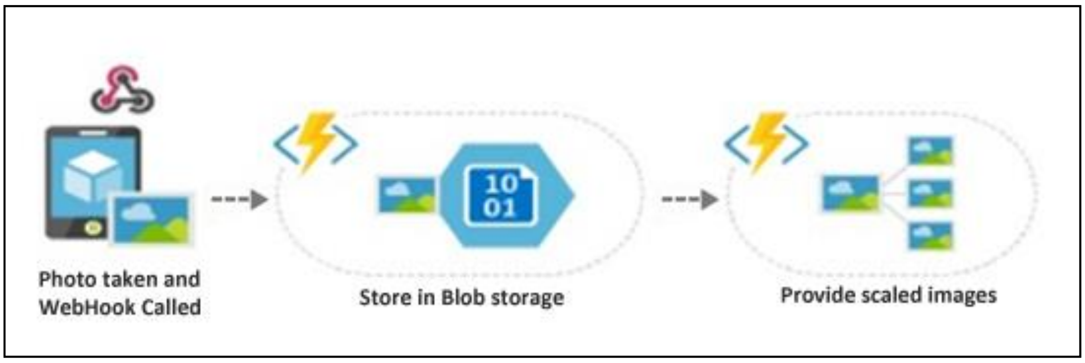
Timer-based processing is a schedule-based process that executes clean-up or other batch tasks on a predefined schedule.



[Azure functions](#) support event-based processing to accelerate overall app building and app deployment. One of the biggest advantages of Azure functions is round-the-clock processing. For example, after every 15 minutes, a function can clean the database table based on predefined business logic.

## Back ends (mobile, IoT, web)

You can build serverless back ends by using Azure Functions to handle mobile, IoT, and web apps.

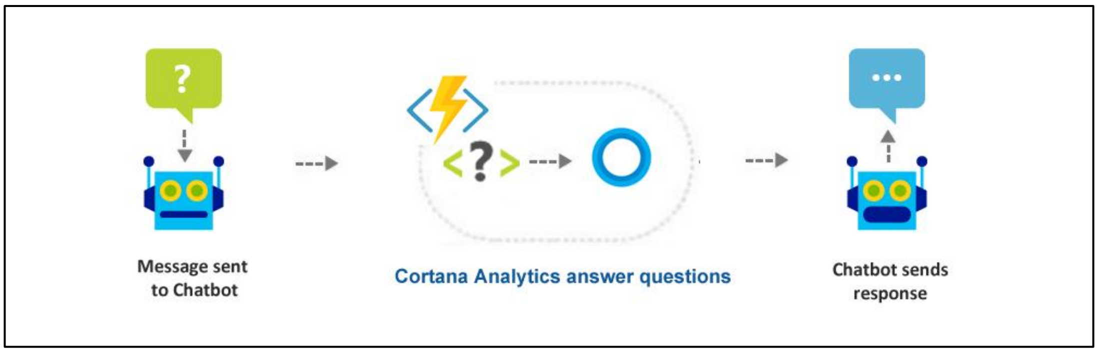


For example, IoT devices can send messages to [Stream Analytics](#), which then calls an Azure function to transform the message. This function processes the data and creates a new update in Azure Cosmos DB.

A mobile back end can be a set of HTTPS APIs that are called from a mobile client through the webhook URL. For example, a mobile application can capture an image and then call an Azure function to get an access token for uploading to Azure Blob storage. A second Azure function is triggered by the blob upload and resizes the image to be mobile friendly.

## Real-time bot messaging

You can build bots that support distinct types of interactions with users in real time. For example, bots can communicate conversationally with text, cards, or speech.



You can use Azure Functions to customize the behavior of a bot by using a webhook. For example, create a function that processes a message by using the Microsoft AI platform and calls this function by using the Microsoft Bot Framework.

# Security issues in serverless computing

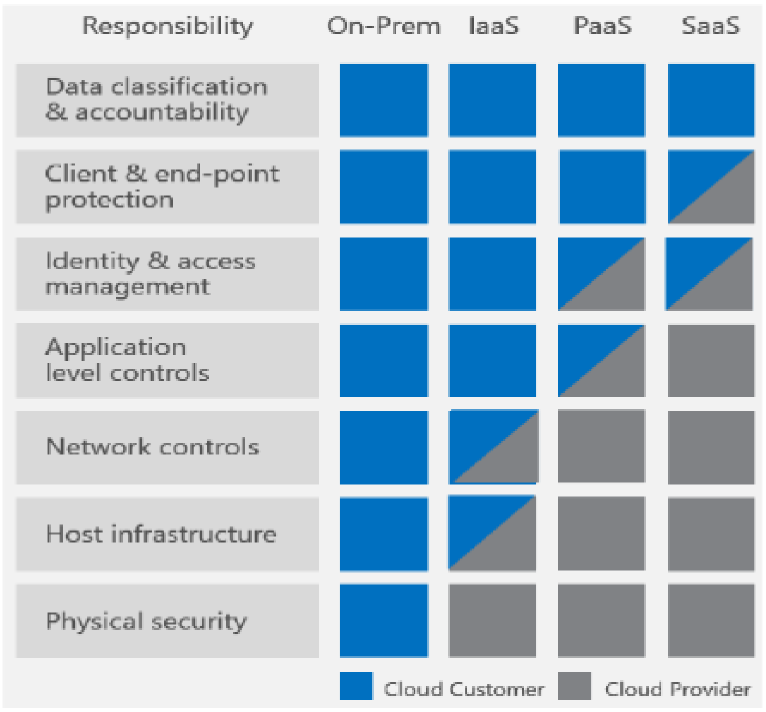
Serverless architectures are used in the enterprise to both build and deploy software and services without the need for in-house physical or virtual servers. Serverless moves the responsibility for server management from the application owner to the platform provider. This eliminates security issues like unpatched servers with known security variabilities and makes denial of service more of a billing issue than security issue.

However, there are still security issues and challenges to consider in serverless computing.

## Shared responsibility model for public cloud computing

Serverless relieves you from managing servers and security updates by shifting this responsibility to the cloud service provider.

You're still responsible for your application code. Badly written code is not secure. You're also responsible for data management, data encryption, identity management, authentication/authorization, and configuration of services and role-based access control (RBAC).



| Responsibility | On-Prem | IaaS | PaaS | SaaS |
|---|---|---|---|---|
| Data classification & accountability | 🟦 | 🟦 | 🟦 | 🟦 |
| Client & end-point protection | 🟦 | 🟦 | 🟦 | ◪ |
| Identity & access management | 🟦 | 🟦 | ◪ | ◪ |
| Application level controls | 🟦 | 🟦 | ◪ | ⬛ |
| Network controls | 🟦 | ◪ | ⬛ | ⬛ |
| Host infrastructure | 🟦 | ◪ | ⬛ | ⬛ |
| Physical security | 🟦 | ⬛ | ⬛ | ⬛ |

🟦 Cloud Customer ⬛ Cloud Provider

Application-level vulnerabilities (for example, cross-site scripting and SQL injection) continue to be severe if exploited. Mitigation techniques (for example, input validation and programmatic database access) are as critical as ever.

Application dependencies are like exploited server dependencies. They're widespread and downloaded frequently by developers. It's hard to track which packages you're using. And they are often vulnerable, with new vulnerabilities disclosed regularly.

## Increased attack surface and increase in complexity

Events like HTTPS APIs, message queues, cloud storage, and IoT device communications are the main source of data consumption for serverless functions. When such messages introduce complex protocols and message structures, the possibility of an attack increases.

Standard application-layer protections such as web application firewalls can't inspect these kinds of attacks if they can't do sophisticated application-layer inspection. And for serverless functions, you can't put an application-layer inspection device in front of the compute device that's hosting the serverless function.

Visualizing and monitoring serverless architectures is still more complex than standard software environments. To address this concern, consider each function to be its own security perimeter. Each function needs to sanitize inputs and outputs, protect its data, and secure its code and dependencies.

## Debugging chaos

There are problems related to transparency when debugging a nested service call fails. Third-party services and data "in transit" make debugging code extremely challenging.

You can use correlation IDs to track the request source, but you need to apply it in every service in a microservice architecture. There is no mitigation for this, so such correlation IDs need to be applied in every service.

### Telemetry and insight everywhere

The move to serverless computing increases application fragmentation, due to the number of moving functions and services. This fragmentation, along with the creation of new services, might lead to solutions that are hard to manage and don't provide insights and usage patterns that would otherwise be useful when you're evaluating for possible security events or indicators of compromise (IoCs).

## Critical security issues and mitigation in the context of Azure

The following sections describe the most critical security risks for serverless architectures.

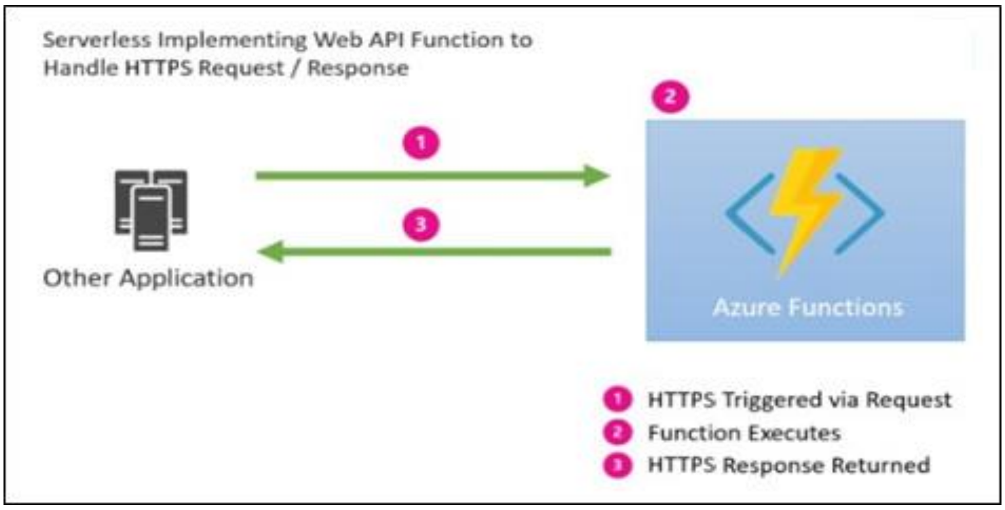
### Injection flow in Azure functions

Injection flaws occur when the untrusted input is passed directly to an interpreter and eventually gets evaluated. The event data injection is not limited to data input such as an API call, because most of the serverless architecture provides several event sources and triggers. Examples are Azure Functions, SQL, NoSQL, Azure Blob storage, Azure Cosmos DB, and serverless mobile back ends.

Functions of code hosted in serverless architecture are bound to specific events that execute them when necessary. Examples of events that you can use to trigger the execution of functions of code in a serverless architecture include:

- **HTTPS request:** Trigger the execution of your code by using an HTTPS request. For an example, see Create your first function.
- **Message queue trigger:** Respond to messages as they arrive in an Azure Storage queue. For an example, see Create a function triggered by Azure Queue storage.
- **File/storage trigger:** Process Azure Storage blobs when they are added to containers. You might use this function for image resizing. For more information, see Blob storage bindings.
- **Timer/schedule:** Execute clean-up or other batch tasks on a predefined schedule. For an example, see Create a function triggered by a timer.

The following image shows the implementation of web API functions to handle HTTPS:


Serverless Implementing Web API Function to Handle HTTPS Request / Response
Other Application
Azure Functions
1 HTTPS Triggered via Request
2 Function Executes
3 HTTPS Response Returned

Azure functions execute HTTPS requests in a three-step process:

1. HTTPS triggered via a request
2. Function execution
3. HTTPS response

When other applications try to access Azure functions, an HTTPS request is sent. When this request is received, execution of the Azure function begins. After the execution of the Azure function, an HTTPS response is returned to the application.

### Recommendations

- Do not explicitly trust input or make any assumptions about its validity.
- Use safe APIs to authorize user input or APIs . This method provides a mechanism for binding variables for the underlying infrastructure.
- Validate and authorize user input before passing directly to any interpreter.
- Check that your code always runs with least privileges required to perform its task.
- Do not assume that input can arrive only from the expected event trigger.
- Always consider all possible event types and entry points into the system when you're applying threat modeling in your development lifecycle.
- Use a web application firewall to inspect incoming HTTPS traffic to your serverless application.

## Fragmented authentication

An application built for a serverless architecture often contains hundreds or more distinct serverless functions. Each has a specific purpose. A serverless architecture promotes microservice-oriented applications, which are small, independently versioned, and scalable.

In a world of fragmented infrastructure, authentication and authorization issues and considerations are important. They provide access control and help protect relevant functions.

As an example, consider a set of public web APIs, all of which use some kind of authentication mechanism. If the applications read files from a cloud storage service, where the contents are in the form of inputs to the functions and are not authenticated to the cloud storage service, then the system will be exposed to an unauthenticated weak entry point. An attack can use this weakness to bypass application logic and manipulate its flow.

This system can lead to executing functions and performing actions that were not meant to be exposed to unauthenticated users, because the design didn't take this into consideration.

### Recommendations

- Do not build custom authentication schemes. Use authentication facilities that the serverless environment provides. For example:
  - Azure App Service authentication/authorization. Azure App Service provides built-in authentication and authorization support, so you can sign in users and access data by writing minimal or no code in your Azure functions.
  - Mobile authentication with a provider SDK.

    - Mobile authentication without a provider SDK.
    - Service-to-service authentication.
- Developers should use secure API keys, SAML assertions, client-side certificates, or similar authentication standards, because interactive user authentication is not an option.

## Serverless deployment misconfiguration

Security firms have found that incorrect settings and misconfiguration of cloud services are often the entry point for attacks against serverless architectures. The default settings that vendors of serverless architectures provide might not always be suitable for your needs. You should customize these settings for the specific needs and tasks of the surrounding environment.

A best practice for serverless architectures is to make functions stateless. Many applications built for serverless architectures rely on the cloud storage infrastructure to store and persist data between executions.

For example, an order being processed would likely have an associated state member. A function can process an order based on that state, while the function itself remains stateless.

Idempotent functions are especially recommended with timer triggers. For example, if you have something that must run once a day, write it so it can run anytime during the day with the same results. The function can exit when there's no work for a particular day. Also, if a previous run failed to finish, the next run should continue it.

### Recommendations

- To avoid sensitive data leakages from the cloud storage infrastructure, vendors can provide a set of security capabilities, like cloud storage configurations, multi-factor authentication, and encryption of data in transit and at rest.
- Organizations that use cloud storage should get familiar with the available storage security controls that their cloud vendor provides.

## Unwanted privileges and roles

Privileges should be assigned to a serverless application that requires access to a resource and should be constrained to a limited time.

Granting permissions to a user beyond the scope required for an action can allow that user to obtain or change information in unwanted ways. A careful delegation of access rights can limit attackers from damaging a system.

For example, a serverless function with Azure SQL Database will read the database and perform analysis on that data only when it's granted "read" permissions on that specific SQL resource.

### Recommendations

- Apply identity and access management (IAM) capabilities relevant to your platform, and make sure that each function has its own user role that runs with least privileges. Microsoft Azure currently does not provide per-function permissions and roles. But different Azure services offer access controls that you can deploy to reduce unnecessary privileges. For example, shared access signatures grant limited access to objects in your storage account to other clients.

- Use Privileged Identity Management with RBAC to manage, control, and monitor access to Azure resources.

## Execution flow manipulation for Azure functions

An attacker might get help to overthrow application logic by making changes in an application. By using this technique, an attacker can bypass the application's security controls.

The invocation process for multiple functions might be a target for attackers, such as those offered by Logic Apps and Azure Durable Functions.

### Recommendations

- To avoid manipulation of the function execution flow, design the system without making any assumptions about legitimate invocation flow.
- Make sure that proper access controls and permissions are set for each function.
- Durable Functions and Logic Apps are built to manage state transitions and communication between multiple functions.
- If you're not using Durable Functions or Logic Apps to integrate with multiple functions, we recommend that you use storage queues for cross-function communication. The main reason is that storage queues are cheaper and much easier to provision.
- Individual messages in a storage queue are limited in size to 64 KB. If you need to pass larger messages between functions, you can use an Azure Service Bus queue to support message sizes up to 256 KB in the Standard tier, and up to 1 MB in the Premium tier.
- Service Bus topics are useful if you require message filtering before processing.
- Event hubs are useful to support high-volume communications.

## Inadequate monitoring and logging

Real-time incident response can help protect against attacks. This is especially true when the system is successfully breached.

To avert successful attacks, organizations must have real-time monitoring and logging and eventing to gain insights into how well a system is functioning. It's also a crucial part of maintaining quality-of-service targets.

Distributed applications running in the cloud are complex in order to achieve adequate real-time security for event monitoring with a proper audit trail. Serverless developers and their SecDevOps teams are required to stitch together logging logic that will fit their organizational needs.

### Recommendations

- Adopt serverless application logic/code runtime tracing and debugging facilities to gain a better understanding of the overall system and data flow. For example, use Azure Application Insights.

## Unsecured third-party software dependencies

A serverless function is often a small piece of code that performs a single discrete task. When serverless functions rely on imported code from third-party software, such as open-source packages and libraries, vulnerabilities can pave the way for exploits.

### Recommendations

Dealing with vulnerabilities in third-party components requires a well-defined process that includes:

- Maintaining an inventory list of software packages and other dependencies and their versions.
- Scanning software for known vulnerable dependencies—especially when you're adding new packages or upgrading package versions. Vulnerability scanning should be part of your ongoing continuous integration and delivery process.
- Removing unnecessary dependencies, especially when your serverless functions no longer need them.
- Consuming third-party packages only from trustworthy resources and making sure that the packages have not been compromised.
- Upgrading deprecated package versions to the latest versions and applying all relevant software patches.

## Unsecured and sensitive storage

Secrets can be any type of sensitive information that should not be handled in plain text. It's common to have application settings that are sensitive and must be protected, such as:

- Database connection strings
- Passwords
- Cryptographic keys

You should never store these secrets in source control. It's too easy for them to leak—even if your source code repository is private. It's not just about keeping secrets from the public. On larger projects, you might want to restrict which developers and operators can access the production secrets. (Settings for test or development environments are different.)

### Recommendations

- Store all application secrets in encrypted storage. Maintain encryption keys through a centralized infrastructure for encryption key management. Most server-wide architectures and cloud vendors perform this key maintenance. Developers should provide a secure API that can be easily integrated into a serverless environment.
- A more secure option is to store these secrets in Azure Key Vault. Key Vault is a cloud-hosted service for managing cryptographic keys and other secrets.

## Denial-of-service attacks

Denial-of-service (DoS) attacks are a serious threat to the serverless architecture. They can disrupt memory allocation, duration per function, and execution limits.

A serverless architecture comes with scalability and high availability, but it has some limitations that need attention.

DoS is a collection of attack types that disrupt the availability of a target. These attacks involve a coordinated effort that uses multiple internet-connected systems to launch many network requests against DNS, web services, e-mail, and other targets.

### Causes for DoS

- An increasing number of poorly managed, connected devices allows attackers to take over them and make them members of DoS botnets.
- Default limits, poor configuration, and the distributed nature of DoS can lead to successful DoS attacks.

### Recommendations

There are several mitigations and best practices for dealing with DoS attacks:

- Always-on monitoring and automatic network attack mitigation
- Adaptive tuning based on the unique platform insights of Azure
- Application-layer protection with Azure Application Gateway web application firewall
- Integration with Azure Monitor for analytics and insights
- Protection against the unforeseen costs of a DoS attack

## Improper exception handling

Line-by-line debugging services for a serverless architecture are often more complex and limited compared to the debugging capabilities. You can see this when the serverless function is using cloud-based services that are not available when you're debugging the code locally.

Line-by-line debugging enables developers to adopt the use of verbose error messages, enable debugging environment variables, and avoid cleaning the code when it's taken into production.

Azure Functions triggers and bindings communicate with various Azure services. When you're integrating with these services, you may have errors raised that originate from the APIs of the underlying Azure services. Errors can also occur when you try to communicate with other services from your function code by using REST or client libraries. To avoid loss of data and ensure that your functions work correctly, it's important to handle errors from either source.

For most triggers, there's no built-in retry when errors occur during function execution. The two triggers that have retry support are Azure Queue storage and Azure Blob storage. By default, these triggers are retried up to five times. After the fifth retry, both triggers write a message to a special poison queue.

Take advantage of defensive measures already provided for components that you use in the Azure Functions platform. For example, see "Handling poison queue messages" in the documentation for Azure Queue storage triggers and bindings.

### Recommendations

- To prevent loss of trigger information if an error occurs in your function, we recommend that you use try-catch blocks in your function code to catch any errors.
- When an error occurs, write the information passed into the function by the trigger to the special "poison" message queue. This approach is the same one that the Blob storage trigger uses.

# Securing the Microsoft serverless platform

Azure offers several managed services that enable developers to create microservices that can run reliably and at scale. The security of the Microsoft Azure serverless platform is the main issue for both building and deploying software and services. The following sections describe some ways to help secure the serverless platform.

## Securing functions with an Azure App Service dedicated plan

When you create a function app, you must configure a hosting plan for functions that the app contains. In either mode, an instance of the Azure Functions host executes the functions. The type of plan controls:

- How host instances are scaled out.
- The resources that are available to each host.

You can run Azure Functions in one of two hosting plans: Consumption plan or an App Service plan.

### Consumption plan

The recommended hosting plan is the Consumption plan because it provides the following advantages:

- Cost is based on the time your functions are running
- You don’t have to worry about capacity planning, as the system will scale to the number of executions required by your application.

### App Service plan

An App Service plan requires more thought on your part regarding resources required to run your functions. You can automatically or manually scale resources to meet your requirements, but overall requires more management overhead on your part.

For more information details related to the Consumption plan and App Service plan, see Azure Functions hosting plans comparison.

## Securing functions with Azure Active Directory

Azure Active Directory is the identity provider that verifies the identity of users and applications that exist in an organization’s directory. It issues security tokens upon successful authentication of those users and applications. Azure AD is used to authorize the users and applications.

Azure functions are built on the same underlying core components as Azure App Service, so you can turn on some features more or less “for free” without writing extra code. Authentication is one of those features.

The authentication/authorization feature enables your application to sign in users so that you don't have to change code on the app back end. It provides a relatively simple way to protect your application and work with per-user data. And if you combine it with Azure AD, you can allow users to sign up for application and support features like password resets, two-factor authentication, and email verification.

To configure authentication and authorization:

1. In the Azure portal, browse to your App Service app. In the left pane, select **Authentication / Authorization**.
2. If **Authentication / Authorization** is not enabled, select **On**.
3. Select **Azure Active Directory**, and then select **Express** under **Management Mode**.
4. Select **OK** to register the App Service app in Azure Active Directory. This creates a new app registration. If you want to choose an existing app registration instead, click **Select an existing app** and then search for the name of a previously created app registration within your tenant. Select the app registration and select **OK**. Then select **OK** on the Azure Active Directory settings page. By default, App Service provides authentication but does not restrict authorized access to your site content and APIs. You must authorize users in your app code.
5. (Optional:) To restrict access to your site to only users authenticated by Azure Active Directory, set **Action to take when request is not authenticated** to **Log in with Azure Active Directory**. This setting requires that all requests are authenticated, and all unauthenticated requests are redirected to Azure Active Directory for authentication.
6. Select **Save**.

## Securing Logic Apps

You can use the Logic Apps feature to orchestrate and connect the functions and APIs of your application. There are many tools available to help you secure your logic apps.

### Access to the trigger

When you work with a logic app that fires on an HTTPS request (Request or Webhook), you can restrict access so that only authorized clients can fire the logic app. All requests for a logic app are encrypted and secured via TLS.

There are two ways to restrict access so that only authorized clients can fire the logic app:

- **Shared access signature**

  Every request endpoint for a logic app includes a shared access signature as part of the URL. Each URL contains these query parameters:

  - **sp**: Specifies permissions that correspond to allowed HTTPS methods.
  - **sv**: Specifies the version used to generate
  - **sig**: Authenticates access to the trigger.

  The signature is generated through the SHA256 algorithm with a secret key on all the URL paths and properties. The secret key is never exposed and published, and is kept encrypted and stored as part of the logic app. Your logic app authorizes only triggers that contain a valid signature created with the secret key.

- **Restriction of incoming IP addresses**

  In addition to the shared access signature, you might want to restrict calling a logic app only from specific clients. For example, if you manage your endpoint through Azure API Management, you can restrict the logic app to accept the request only when the request comes from the API Management instance's IP address.

You can configure this setting within the logic app settings:

1. In the Azure portal, open the logic app where you want to add IP address restrictions.
2. Select **Access control configuration** under **Settings**.
3. Specify the list of IP address ranges that the trigger will accept.

## Access to manage or edit logic apps

You can restrict access to management operations on a logic app so that only specific users or groups can perform operations on the resource. Logic apps use the Azure RBAC feature, and can be customized with the same tools. You can also assign members of your subscription to these built-in roles:

- **Logic App Contributor**: Provides access to view, edit, and update a logic app. This role can't remove the resource or perform admin operations.
- **Logic App Operator**: Can view the logic app and run history, and enable/disable. This role can't edit or update the definition.

## Access to contents of inputs and outputs for a run

All data within a workflow run is encrypted in transit and at rest. When a call to run history is made, the service authenticates the request and provides links to the request and response inputs and outputs. You can help protect this link by ensuring that only requests to view content from a designated IP address range can return the contents.

## Parameters or inputs within actions in a workflow

You might want to parameterize some aspects of a workflow definition for deployment across environments. Also, some parameters might be secure parameters that you don't want to appear when editing a workflow, such as a client ID and client secret for Azure Active Directory authentication of an HTTPS action.

To enable access to the value of a resource parameter at runtime, the workflow definition language provides a `@parameters()` operation. Also, you can specify parameters in the resource deployment template. But if you specify the parameter type as `securestring`, the parameter won't be returned with the rest of the resource definition, and it won't be accessible by viewing the resource after deployment.

## Access to services that receive requests from a workflow

There are several ways to help secure an endpoint that the logic app needs to access:

- **Using authentication on outbound requests**

  When you're working with an HTTPS, HTTPS + Swagger (Open API), or webhook action, you can add authentication to the request that's being sent. You can include basic authentication, certificate authentication, or Azure Active Directory authentication. For details on how to configure this authentication, see Get started with the HTTP action.

- **Restricting access to logic apps' IP addresses**

  All calls from logic apps come from a specific set of IP addresses per region. You can add more filtering to accept only requests from those designated IP addresses. For a list of those IP addresses, see Logic app limits and configuration.

- **On-premises connectivity**

  Logic apps integrate with several services to help provide secure and reliable on-premises communication.

- **On-premises data gateway**

  Many managed connectors for logic apps provide secure connectivity to on-premises systems, including File System, SQL, SharePoint, and DB2. The gateway relays data from on-premises sources on encrypted channels through Azure Service Bus. All traffic originates as secure outbound traffic from the gateway agent. Learn more about how the data gateway works.

- **Azure API Management**

  Azure API Management has on-premises connectivity options, including site-to-site VPN and Azure ExpressRoute integration for secured proxy and communication to on-premises systems. In the Logic App Designer, you can quickly select an API exposed from Azure API Management within a workflow, providing quick access to on-premises systems.

## Securing Event Grid resources

In simple words, Azure Event Grid is a service that raises events from anywhere to anywhere. Event Grid supports events that come from Azure services at the subscription, resource group, and resource levels. You can use Event Grid to build applications with event-based architectures. Event Grid has three types of authentication that need to be secure at each level of authentication.

Azure Event Grid offers a variety of security controls:

- **Role-based access control** is a method of regulation that's defined according to job competency, authority, and responsibility within the enterprise. Security on the Event Grid resource is managed with RBAC. Access is given to an individual user to perform a specific task, such as modify, create, or view a file that's applicable to all Azure resources.

- **Webhook validation**. Webhooks are one of the ways to receive events from Azure Event Grid. New webhooks need to be registered with Event Grid. Event Grid sends you a POST request with a simple validation code to prove endpoint ownership.

  It's important to note that only HTTPS webhooks are supported. A new subscription to an event must have the **Microsoft.EventGrid/EventSubscriptions/Write** permissions on the required resource.

- **Shared access signatures** can be used like a username and password scheme, where the client is in immediate possession of an authorization rule name and a matching key. The client needs to use SAS tokens or key authentication when they want to publish an event to a topic. You can use SAS tokens to scope the access that you grant to a certain resource in Event Grid for a certain amount of time.

# Resources

## GitHub repositories

- Azure Functions Host: The Azure Functions runtime/host.
- Azure WebJobs SDK: The "core" of the Azure Functions runtime and many bindings.
- Azure WebJobs SDK extensions: The repositories of many bindings.
- Azure Functions Core Tools: The command-line tools for Azure Functions.
- Azure Functions UX: The UX for the Azure Functions development portal.
- Azure Functions templates: The templates that appear in the Azure Functions portal, Visual Studio, Visual Studio Code, and other places.
- Azure Functions samples: The repository for some samples on how the runtime works.
- Azure Functions VS Tooling: MSBuild tasks for precompiled functions.

## Samples

- List of samples
- Azure Functions sample: webhooks
- Azure Functions sample: event processing
- Azure Functions sample: Azure services