Trend Micro Research

TREND MICRO

The State of Serverless Security on Microsoft Azure

David Fiser and Alfredo Oliveira

Contents

Published by
Trend Micro Research

Written by
David Fiser
Alfredo Oliveira
Trend Micro Research

Serverless computing has swiftly become a popular service among cloud service providers (CSPs) because it enables organizations to run services without needing to manage the underlying infrastructure. This means that with serverless technology, developers can upload code to a specific service without worrying about infrastructure maintenance, scalability, and availability.

In a serverless architecture, the CSP is responsible for infrastructure management. However, because this architecture allows user input, uploading untrusted code into multitenant environments poses several security challenges.

For this study, we performed exploitation simulations of user-provided code vulnerabilities among serverless services provided by major CSPs in the market. Based on our evaluation of each infected serverless environment, we found that most of the security concerns were in Microsoft Azure. Therefore, we have chosen to devote our research to this serverless environment.

In our investigation, we identified sensitive environmental variables inside the Microsoft Azure environment. When these variables are leaked, malicious actors can fully compromise the entire serverless environment. We also found a default runtime image using a master password that allows privilege escalation in most Azure App Service deployments. Based on our observations in this study, we can confirm that Microsoft Azure's default security measures would need to be supplemented by proper token management to keep Microsoft Azure serverless deployments secure.

At a glance, some of our findings might seem overrated. However, recent publicly disclosed breaches have shown that a critical cybersecurity issue could start with just a single-service compromise. To address these security gaps, organizations should use custom images or tweak CSP-provided images, which are not entirely secure by default. This can be done via the Distroless approach by following the distributed, immutable, and ephemeral (DIE) paradigm. Finally, we discuss how increased adoption of the DIE paradigm in cloud environments can enhance serverless security.

Introduction

Serverless or serverless computing is a popular service that allows developers to deploy code on systems and applications without needing to worry about the underlying infrastructure or needing high-scalability options.

Serverless functions can be used for APIs and back-end and front-end applications in various programming languages and frameworks, such as Node.js, Python, Java, PHP, .NET, and Go. Because serverless functions can have different uses and require different suitable environments, these functions might pose architectural challenges on the side of the CSPs – an aspect of the service that users have no visibility over. Serverless function execution code is often driven by an event trigger, such as HTTP access, message queues, and storage events.

These serverless functions' architectural challenges can introduce security risks and provide attackers with different scenarios where they can access the serverless environment by exploiting user input code vulnerabilities. In this research paper, we aim to shed light on the security implications of the architectural challenges posed by serverless functions. It is important to note that these implications do not serve as attack entry points. Rather, these implications provide an opportunity for attackers to move laterally within the serverless environment after an initial compromise. It is crucial for users to be aware of these implications and take responsibility for addressing any code vulnerabilities in their input, in order to minimize the risk of abuse. This report delves into uncovering hidden environmental features that could potentially have security implications. After analyzing multiple CSPs in our study, we have decided to focus on Microsoft Azure – specifically Azure App Service and Azure Functions – by virtue of our investigation's initial findings.

CSP and User Responsibilities

As serverless computing can be difficult to imagine at first, one might begin by thinking of it as code executed on third-party infrastructure. The CSP is responsible for the execution process of the serverless service, which is usually multitenant in nature and should be isolated.

Meanwhile, the user is responsible for deploying code into the serverless system, which has a zero-trust architecture for user-provided input. However, when the user-provided input is an actual source code that will be executed into the system, this becomes a security concern. This is why it is critical to design serverless environments for executing user-provided input or code that provides a certain level of isolation, such as a sandbox for executing or running untrusted applications.

If a serverless platform is not flawless in its design, breaking these isolations could have tremendous negative consequences for the CSP platform, such as CSPs being able to see customers' private data or users having unauthorized access to the cloud infrastructure itself.

Fortunately, there are several isolation techniques, such as standalone containers and micro virtual machines (micro VMs), that allow users to isolate applications. However, these isolation techniques come with performance costs.

The way isolation techniques are implemented differs depending on the CSP, the chosen service, and its runtime. Serverless services depend on a container that runs inside a VM instance on a physical server. Physical servers usually run multiple VMs and each VM can run multiple containers.

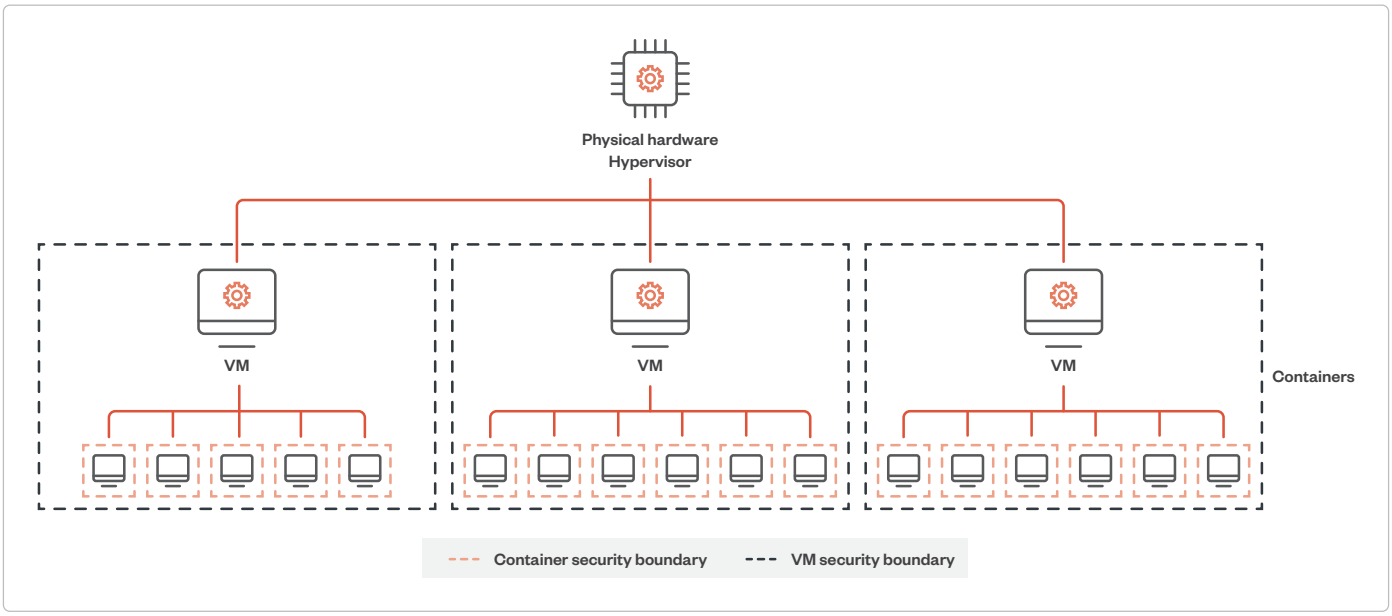


Figure 1. Container security and VM security boundaries

When it comes to security boundaries, the actual serverless code is executed inside "the lowest leaves." These are then surrounded by container isolation techniques, which are separated by VM security boundaries.

To escape these security boundaries, a malicious actor needs to exploit vulnerabilities within the container engine, kernel, or hypervisor.

CSPs are responsible for keeping these boundaries secure by ensuring that they cannot be escaped from. To mitigate this risk, CSPs can use a customized form of a kernel, such as one that does not have unnecessary modules or drivers, or does not require more security-oriented hardening. CSPs can also choose to implement an architectural design that has an additional security boundary that usually involves a trade-off between performance and costs.

An example of this second option is the use of micro VMs such as Kata Containers, which allow CSPs to use common container control interfaces. These micro VMs are lightweight in nature and can be used for additional resource isolation. This means that the code executed within the container is executed inside the hypervisor, which provides an additional sandbox between the host kernel and the container. But while micro VMs can directly run on a physical hardware host as well as inside the VM itself, this also comes with some performance cost.

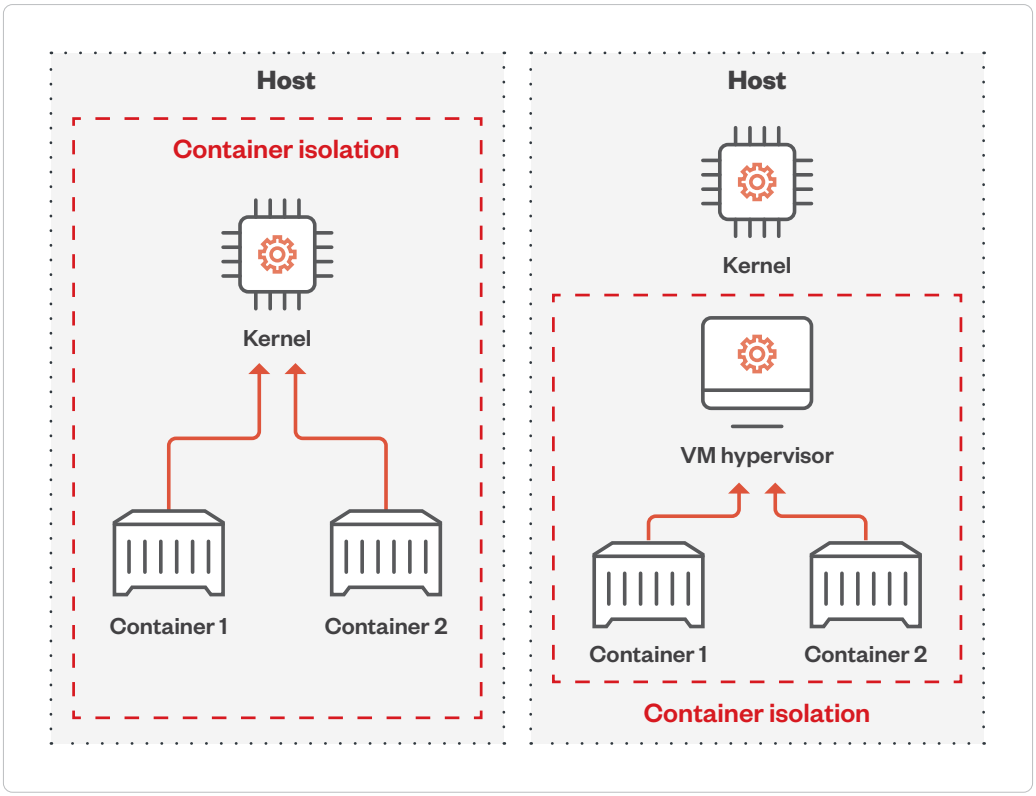


Figure 2. A comparison of standard containers and micro VM-based containers

The implementation details vary for each CSP. When the host platform is Linux, the containers used are either standalone or UVM-based. A Windows-hosted platform that has a container platform similar to that of the UVM architecture has a Host Compute System (HCS) that acts as an additional layer between the host kernel and the container.

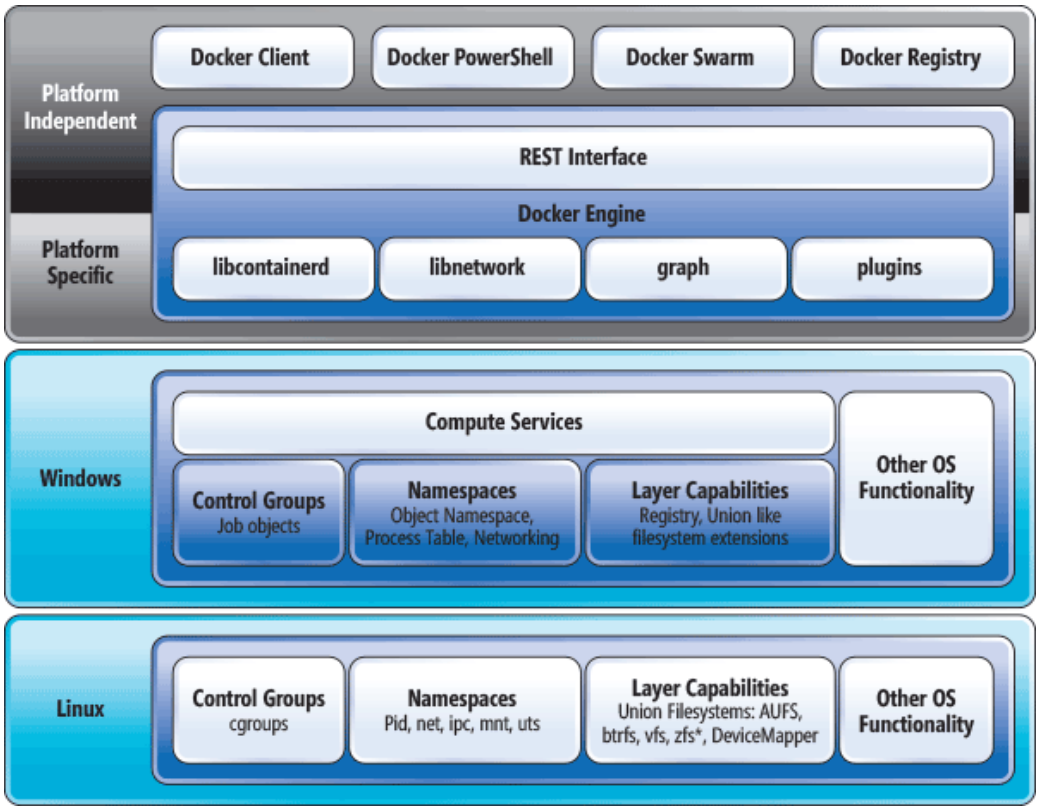


Figure 3. Host platform implementation details of Windows and Linux

Source: Microsoft Documentation[1]

Due to the high scalability of serverless environments, their containers should be executed by one of the following orchestrators:

- Kubernetes
- Docker Compose
- Service Fabric
- Custom service orchestration services

Different serverless services use different orchestrators that users do not have visibility over. Although certain serverless services can allow users to override default settings and use an orchestrator of their choice, doing so requires using alternate cloud services and advanced settings, which are beyond the scope of this paper. From a security perspective, it should be emphasized that the serverless function itself can communicate with the orchestrator or even exploit a misconfiguration.

Analyzing Serverless Security

When it comes to analyzing serverless security, the entry point that triggers the serverless function execution must first be identified. This entry point can be either a direct HTTP, an HTTPS endpoint trigger that is being called, or an indirect action, such as a certain event that provides user data that the serverless function will process.

Securing endpoints and writing secure code is the user's responsibility. This involves making sure that misconfiguration issues are not introduced into the system and that user-provided code is free of flaws or vulnerabilities.

Misconfiguration problems in cloud platforms are rampant, which is why malicious actors have started to shift their focus to abusing cloud-oriented services. It is only a matter of time before they fully set their sights on serverless environments.

In succeeding sections, we will focus on misconfiguration issues and exploring what malicious actors can do inside serverless environments once a user-provided vulnerability is exploited.

Another important aspect that needs to be identified is an organization's cloud architecture, such as what cloud services are being used and how they are being accessed. It is also critical to know if an organization will fully migrate to a public CSP or adopt a hybrid form of cloud service that will access serverless functions from private networks.

Regardless of a company's cloud architecture, the zero-trust model[2] and the Assume Breach paradigm[3] should be followed to maximize cloud security. This requires following the principle of least privilege by applying role-based access control (RBAC). Doing so would limit not only network and storage access available from the function execution context but also the execution time via timeouts. It is worth reiterating that it is the role of the CSP to provide users with sufficient tools that will allow them to create a secure configuration. And since most of the services have a default container image for the user's programming language of choice, the CSP should therefore also provide a secure default container image.

Technical Analysis

We evaluated several CSPs' serverless environments by simulating a user code vulnerability exploitation to attempt remote code execution (RCE) on the infected environment using a reverse shell that connects to our own server.

After establishing a successful connection, we ran several commands to analyze the environment, similar to what an attacker would do. We looked for specific security gaps that we could abuse once we entered the environment.

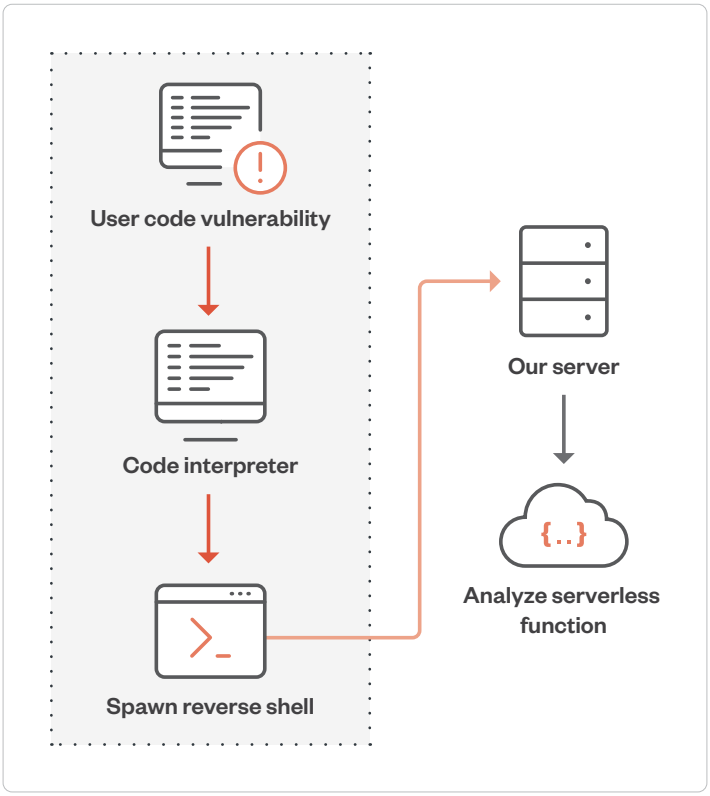


Figure 4. Attack simulation of a serverless function

Our analysis focused on gathering environmental information such as internal architecture hints, including the orchestrator or container engine used, timeouts that applied to serverless environments, execution rights and capabilities, networks, and disk access. Most importantly, we looked for secrets management tools and methods, including storage and transfer of information.

Azure App Service

According to Microsoft, the Azure App Service is used to create "enterprise-ready" web and mobile apps for any platform or device, as well as to deploy them on a scalable and reliable cloud infrastructure."[4] App Service provides ready-to-use infrastructure for applications, and from a technical perspective, it is essentially a Docker container with prepared images and language interpreters for listed runtime stacks such as Python, Node.js, Java, .NET, and Ruby.

The presence of the ".dockerenv" file inside the file system root and the "docker run" command found inside the App Service log stream confirms that a Docker container engine is used to run the service.

```
2022-03-23T14:26:35.970Z INFO  - docker run -d -p 8000:8000 --n
WEBSITE_ROLE_INSTANCE_ID=0 -e WEBSITE_HOSTNAME=nebula-test5.azu
appsvc/python:3.9_20220104.1 python3 /home/site/wwwroot/app.py
```

Figure 5. Azure log that confirms the use of a Docker container engine

Together with modern standards, the user is able to bind App Service with continuous integration and continuous delivery (CI/CD) pipelines, such as GitHub repositories and commit triggers.

.NET	Java	JavaScript	PHP	Python	Ruby
.NET 6 (LTS)	Java 8	Node 16 (LTS)	PHP 8.0	Python 3.9	Ruby 2.7
.NET 5	Java 11	Node 15 (LTS)	PHP 7.4	Python 3.8	Ruby 2.6
.NET Core 3.1		Node 12 (LTS)		Python 3.7	
ASP.NET V4.8					
ASP.NET V3.5					

Table 1. Available languages and versions for Azure App Service

Because the service is intended to be a publicly accessible web service, no default access limitation has been set. It is the user's full responsibility to limit access if needed.

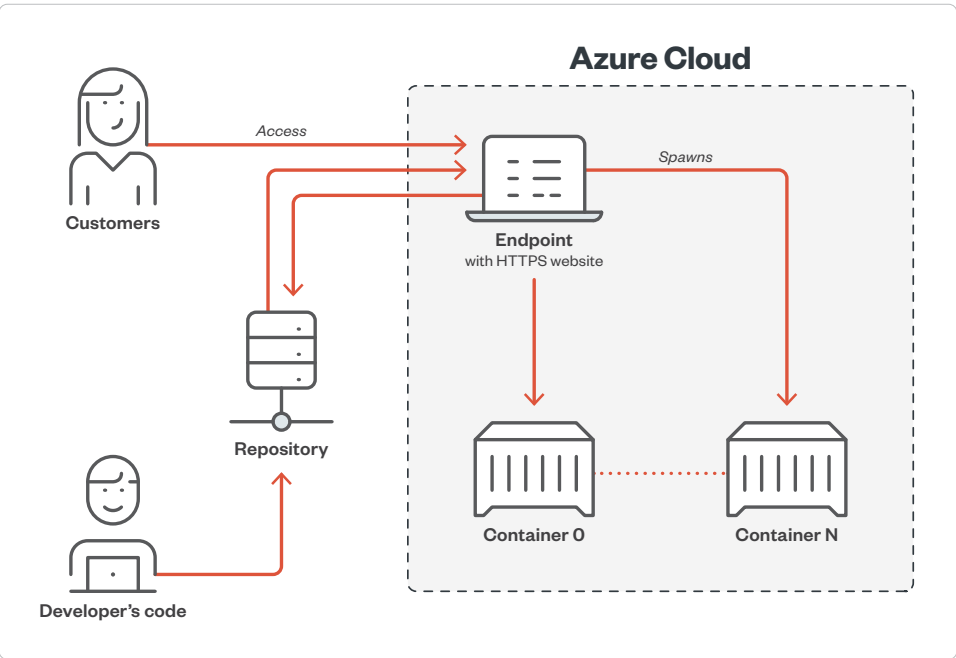


Figure 6. Azure App Service with CI/CD integration

Source: Trend Micro Security New[5]

Meanwhile, there are different timeouts for web requests and spawned processes.

Timeout for web request	Timeout for spawned process
240 seconds	Up to 10 minutes

Application and User Permissions

We also discovered that in Linux environments, the application and user permissions within an App Service application run with root privileges within the security boundary of the container. However, the container root user is mapped to a less-privileged user inside the Docker host using the user namespace remap feature,[6] which effectively lowers malicious actors' attack options upon compromise.

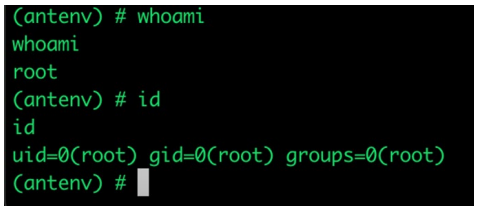


Figure 7. User permission within the App Service environment where the container is running code written in Python

A low-privileged user, such as "www-data," can be used to run an App Service application. "Www-data" is the default user that web servers on Ubuntu and Apache use, and which we used as a PHP container image in our investigation. However, malicious actors can still easily escalate privileges to run it under root within the security boundary of the running container. This is possible when malicious actors abuse the shortcomings of a security-oriented application design.

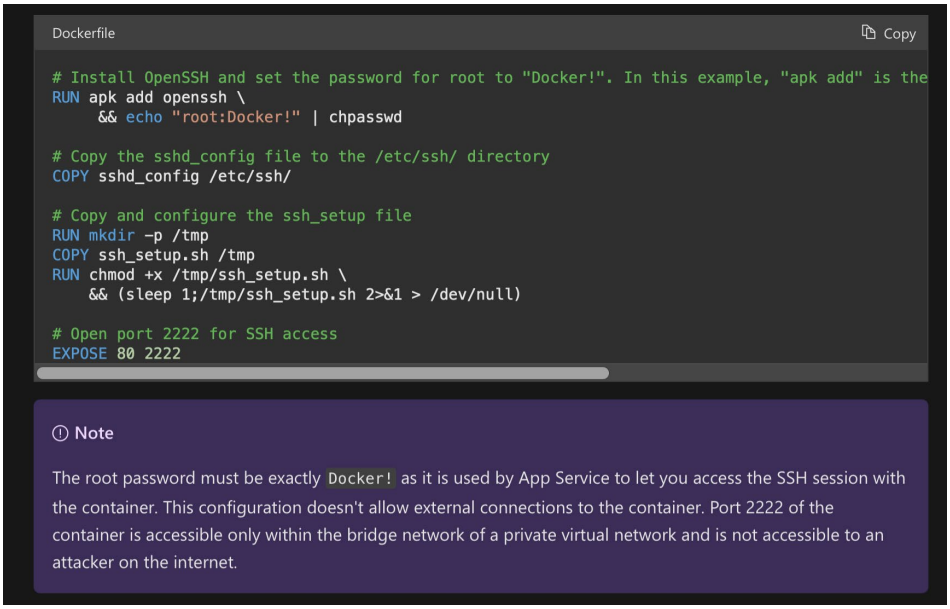


Figure 8. Screenshot of a custom container tutorial from Azure

Source: Microsoft[7]

Malicious actors can perform privilege escalation by entering the hard-coded master password, "Docker!". This password is typically used to access the container through WebSSH,[8] as it cannot be generated as an asymmetric keypair upon first execution.

```
ubuntu@ip-172-26-1-174:~$ nc -l 0.0.0.0 4242
Linux aefa66e86a8b 5.4.0-1063-azure #66~18.04.1-Ubuntu SMP Thu Oct 21 09:59:28 UTC 2021 x86_64 GNU/Linux
 10:02:50 up  2:38,  0 users,  load average: 0.35, 0.42, 0.45
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ whoami
www-data
$ su root
Password: Docker!
whoami
root
```

Figure 9. Example of privilege escalation on a container running within the Azure App Service application

However, it should be noted that the root user does not have all the capabilities of the host machine and is, in fact, limited within container isolation schemes. Simply put, the container is not running in privileged mode.[9]

The following are the available capabilities within containers implemented on Linux:[10]

- CAP_CHOWN
- CAP_DAC_OVERRIDE
- CAP_FOWNER
- CAP_FSETID
- CAP_KILL
- CAP_SETGID
- CAP_SETUID
- CAP_SETPCAP
- CAP_NET_BIND_SERVICE
- CAP_NET_RAW
- CAP_SYS_CHROOT
- CAP_SYS_PTRACE
- CAP_MKNOD
- CAP_AUDIT_WRITE
- CAP_SETFCAP

Network

Among the previously mentioned list of root user capabilities, CAP_NET_RAW allows the creation of raw sockets that give access to lower layer protocols and can be abused by malicious actors to launch low-level network attacks. The running container exposes two ports: a publicly accessible port for incoming connections that are handled by the application itself, and a port hidden behind the Azure authentication gateway for remote secure shell (SSH) access that only authenticated Azure App Service users can initiate.

Outbound network connections are enabled by default, which initiate remote connections to internet-available servers. Attackers can abuse these connections to initiate reverse shell connection attacks.

By default, a local area network (LAN) consists of a minimum of three IP addresses: one for the container network interface itself, another as the default gateway for accessing the internet, and the third for incoming SSH connections from WebSSH.

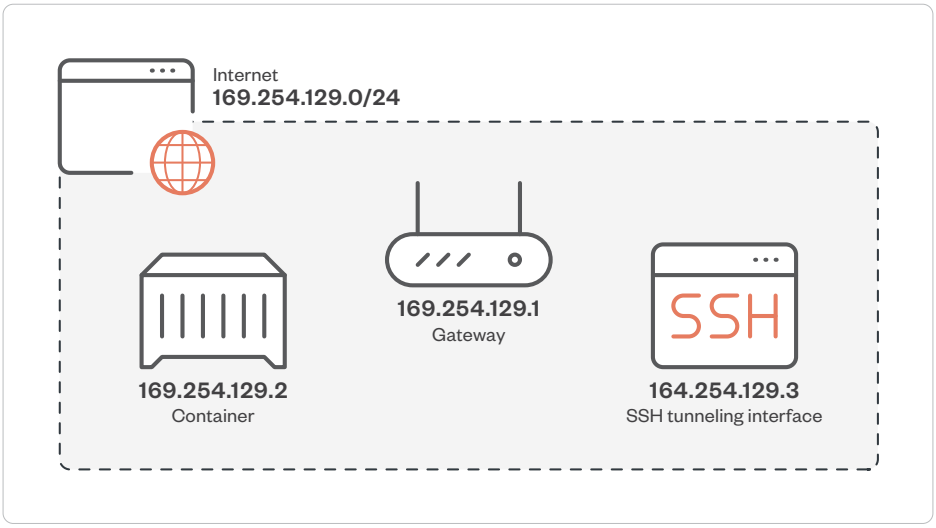


Figure 10. Default network scheme

Source: Trend Micro Security News[11]

These default LAN settings can be altered by modifying the App Service networking settings[12] on a premium subscription using virtual networks (VNETs) and hybrid connections. Because there are multiple variations and use cases for different organizational needs, we would like to emphasize applying the principle of least privilege. From the network perspective, this means denying all other traffic that is unnecessary for the application to work, especially if an organization's network consists of multiple endpoints within one VNET.

Disk Access

The source code is deployed inside the "/tmp/{build directory}" folder when a container is executed and CI/CD is configured. Because containers are mainly designed to be stateless, there is an interesting Server Message Block 3.0 (SMB3) network mount on the "/home" directory. This volume serves as a persistent storage that hosts website files when CI/CD is not configured. It should be noted that SMB servers are known for having multiple vulnerabilities in the past. One very popular example of an attack that abused an SMB vulnerability is the WannaCry ransomware.[13]

Secrets and Available Environmental Variables

Secrets are critical items that users need to secure, such as passwords, API keys, and certificates.[14] As for Azure secrets for the CI/CD pipeline, these are not located within the container, which is good for security. However, the container is architecturally designed to include environmental variables that the user does not have visibility and control over. From a security perspective, users should focus on the following variables:

- WEBSITE_AUTH_ENCRYPTION_KEY
- WEBSITE_AUTH_SIGNING_KEY

On Azure App Services, users can also customize application settings and connection strings. However, storing secrets in these locations is highly discouraged as they are exposed as environmental variables that the application or service can access at runtime.

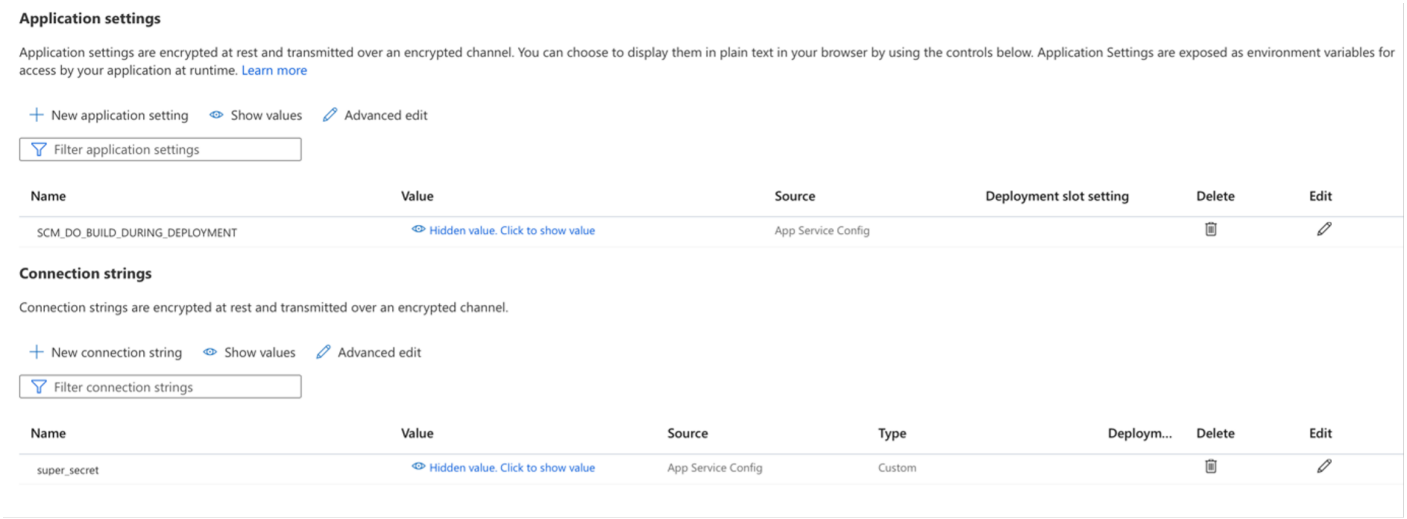


Figure 11. Screenshot of Azure App Service's application settings and connection strings

Additionally, users should not expose any secrets inside the container's persistent storage, which can happen when CI/CD is not configured, as well as when users clone a private repository using an access token. This access token can be found in plain text inside the ".git/config" file.

```
root@aa564cd5cc32:/home/site/wwwroot# cat .git/config
[core]
        repositoryformatversion = 0
        filemode = false
        bare = false
        logallrefupdates = true
        ignorecase = true
[remote "origin"]
        url = [redacted]
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
root@aa564cd5cc32:/home/site/wwwroot#
```

Figure 12. Access token information in plain text inside the ".git/config" file

App Service on Windows-Based Environments

The Windows environment is not available for all stacks, and malicious actors have limited attack options in this environment because the Azure Web App sandbox[15] restricts access to all resources. This restriction also applies to network limitations because the creation of raw sockets effectively disables ping network tests.

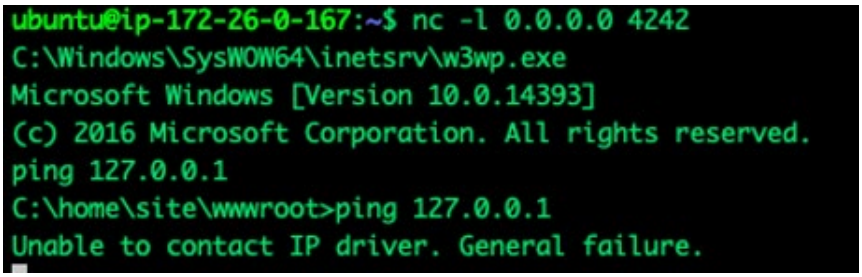


Figure 13. Example of limited functionality via the Azure Web App sandbox

However, despite these restrictions, sensitive environmental variables remain present in Windows-based App Service instances.

Figure 14. Environmental variables inside a Windows-based App Service instance

.NET	Java	JavaScript	PowerShell Core
.NET 6	Java 8	Node 16 (LTS)	7.0
.NET 3.1	Java 11	Node 15 (LTS)	7.2 (preview)
		Node 12 (LTS)	

Table 2. Available serverless stacks on Windows-based environments

Azure Functions in Linux-Based Environments

Azure Functions is a purely code-oriented serverless service primarily designed to run a small block of code with any of the following runtime stacks:

.NET	Java	JavaScript	PowerShell Core	Python	Custom
.NET 6	Java 8	Node 16 (LTS)	7.0	Python 3.9	Docker image
.NET 3.1	Java 11	Node 15 (LTS)	7.2 (preview)	Python 3.8	
		Node 12 (LTS)		Python 3.7	

Table 3. Available serverless stacks on the Linux platform

Azure Functions needs an Azure storage account to create a serverless function. The Azure storage account is where the actual code is stored in the form of a blob storage.[16] The stored blob has a .zip extension, but it is technically a compressed read-only file system for Linux called "squashfs."

Once an HTTP-triggered function is publicly deployed, by default, the uniform resource identifier (URI) endpoint requires an access token to execute the serverless function, unless an anonymous authorization level is set. The token is entered via the "code" parameter of the URL. This pre-generated token has two types:

- **A function scope token:** A token used for executing serverless code
- **An application-scope token (host key):** The default token that also acts as the master key

An application-scope token authorizes for all endpoint functions. In contrast, a function-scope token works only for defined endpoint functions. An application-scope token acts as a master key that allows administrator access to the API for environment management of the serverless function. These keys are stored inside the linked Azure storage account in encrypted form.

Figure 15. An example of an "/admin/" endpoint authorized by an application-scope token that acts as a master key

By default, endpoints can be accessed by either HTTP or HTTPS, which is why it is highly recommended for users to enforce SSL. The function access keys are a security concern when accessing endpoints from public environments. To counter this, Azure supports third-party identity providers and allows the configuration of API management authentication policies that users can set.[17]

Another available form of authentication is public key cryptography, in which either a public key or a client certificate is sent as an HTTP header. Since the validation of this authentication approach is implemented within serverless function code, the developer is fully responsible for it. Meanwhile, Azure Key Vault can be used to securely store associated secrets.[18]

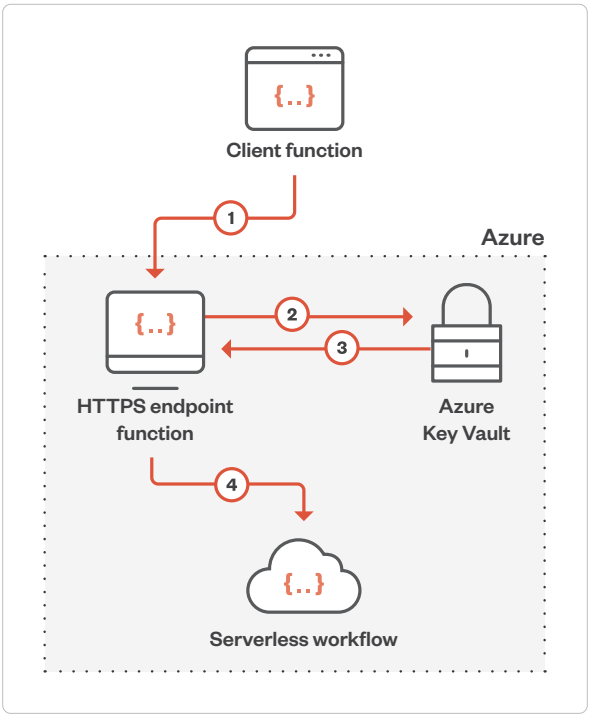


Figure 16. Azure authentication approaches

The following is how authentication is performed within the Microsoft Azure serverless environment :

1. The function client sends an authentication request. An authentication secret is sent as a header to the HTTPS endpoint function.
2. The serverless function authenticator contacts Azure Key Vault to verify if a secret can be used, such as if a fingerprint is present within the store.
3. The secret undergoes checking to determine if it is verified and valid.
4. If the validation is successful, the remaining serverless workflow is executed.

The difference between public key cryptography and shared access signature (SAS) is that with the former, multiple secrets can be used and authorized for a single app using multiple certificates. Public key cryptography is more difficult to guess, can be revoked on either the Azure Key Vault or the certificate authority (CA) level, and has a longer secret length than SAS.

Azure Functions uses its own distributed platform called Azure Service Fabric[19] as an orchestrator, which spawns an azure-functions-host container.[20] In turn, this container executes specific azure-functions-worker functions based on the chosen application stack where the actual source code is being interpreted.

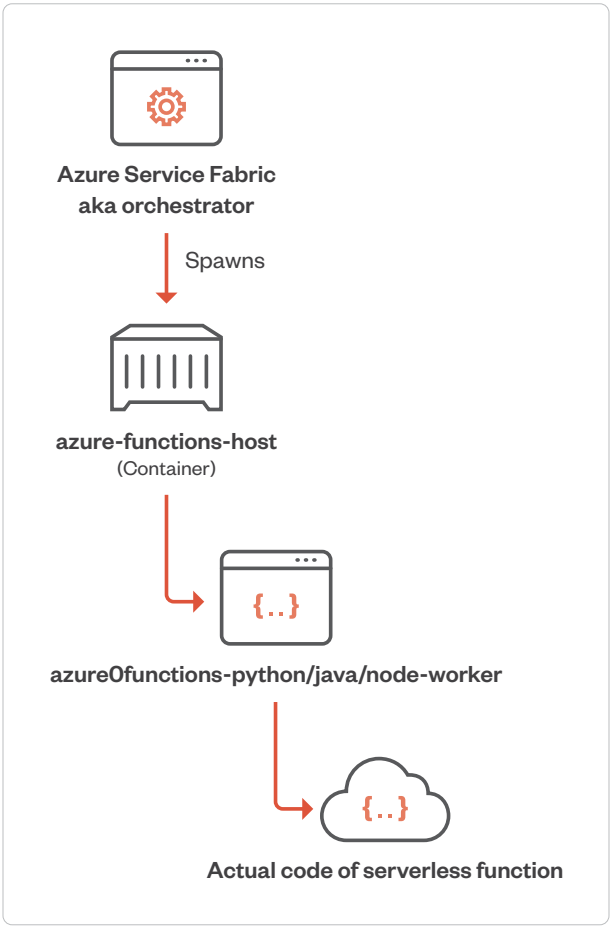
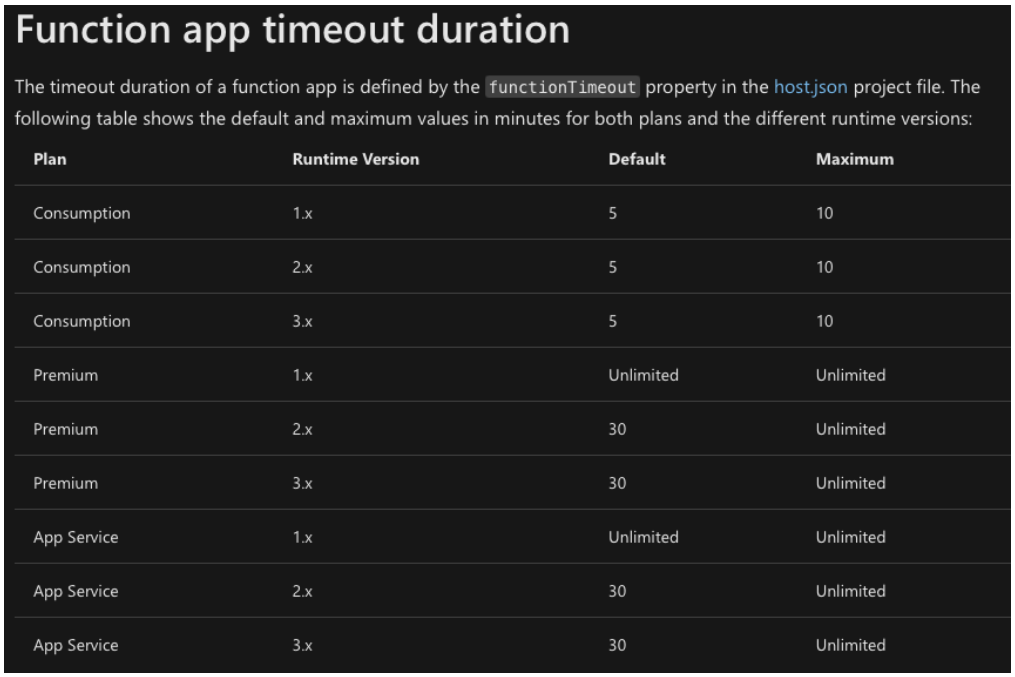


Figure 17. Azure Service Fabric's function as an orchestrator

Azure also allows developers to use Kubernetes as an orchestrator for serverless environments using Azure Kubernetes Services (AKS) and virtual nodes. However, this service is beyond the scope of this research.

Timeouts

The default maximum timeout for Azure Functions is set to five minutes. Regardless of the function app timeout setting, 230 seconds is the maximum amount of time that an HTTP-triggered function can take to respond to a request. This is because of Azure Load Balancer's default idle timeout. For longer processing times, users can consider using the Azure Durable Functions asynchronous HTTP API pattern or defer the actual work and return an immediate response. A user can specify a lower timeout to reduce the amount of time in which a potentially vulnerable service will be available to process malicious actions.



Function app timeout duration

The timeout duration of a function app is defined by the `functionTimeout` property in the host.json project file. The following table shows the default and maximum values in minutes for both plans and the different runtime versions:

Plan	Runtime Version	Default	Maximum
Consumption	1.x	5	10
Consumption	2.x	5	10
Consumption	3.x	5	10
Premium	1.x	Unlimited	Unlimited
Premium	2.x	30	Unlimited
Premium	3.x	30	Unlimited
App Service	1.x	Unlimited	Unlimited
App Service	2.x	30	Unlimited
App Service	3.x	30	Unlimited

Figure 18. A comparison of Azure applications' timeout plans

Rights and Capabilities

Default user permissions are limited and non-root. The "superuser do" or "sudo" utility is available within the environment, while write permission is only available inside the "/tmp/" folder. The content of the folder can be stateful if a cache is hit and a previous environment is used.

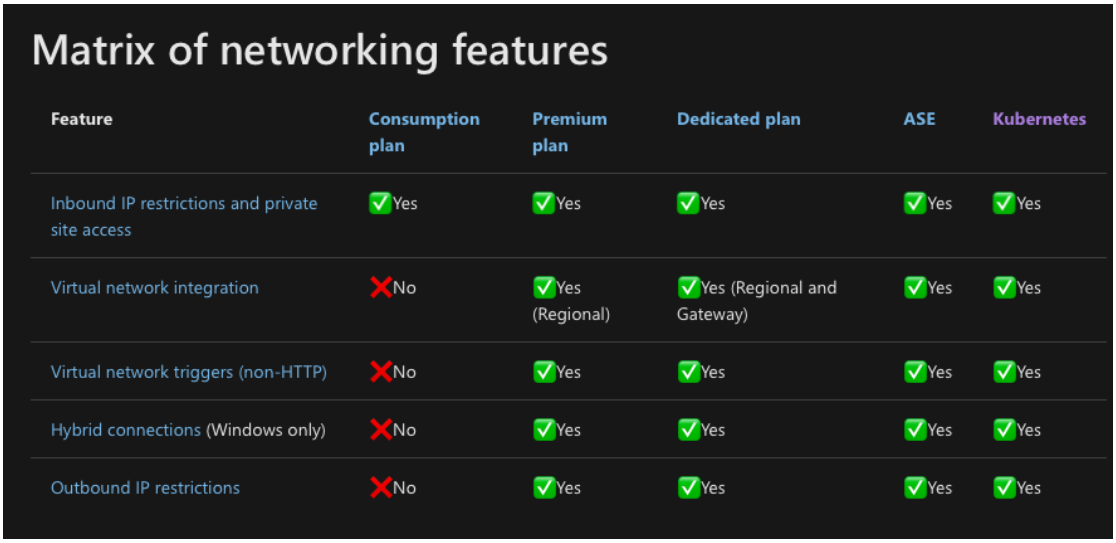
No capabilities are available for the running user.

The serverless container is running the main Azure Functions serverless service on TCP port 9091. Other internal services can also be present inside the container, such as managed identities that listen to TCP port 8081 and allow security tokens to be obtained within the Azure infrastructure.

TCP port	Definition
80	Nginx
6060	Mesh and reverse proxy
8081	Managed identities (used to obtain tokens)
9091	Azure Functions endpoints

Outgoing Connections

By default, there are no outgoing connection limitations. However, advanced subscription plans allow users to configure outgoing connections.



Matrix of networking features

Feature	Consumption plan	Premium plan	Dedicated plan	ASE	Kubernetes
Inbound IP restrictions and private site access	✅Yes	✅Yes	✅Yes	✅Yes	✅Yes
Virtual network integration	❌No	✅Yes (Regional)	✅Yes (Regional and Gateway)	✅Yes	✅Yes
Virtual network triggers (non-HTTP)	❌No	✅Yes	✅Yes	✅Yes	✅Yes
Hybrid connections (Windows only)	❌No	✅Yes	✅Yes	✅Yes	✅Yes
Outbound IP restrictions	❌No	✅Yes	✅Yes	✅Yes	✅Yes

Figure 19. A comparison of Azure networking features per available plan

Source: Microsoft[21]

Another important aspect of the serverless environment that needs to be analyzed is secrets management. This is especially true when the application needs to be authenticated in other services. Such services could be either user- or CSP-provided, and the application would usually use a form of secret for it to be authenticated.

Proper secret storage mitigates the risk of exposure and compromise. The following are key factors when evaluating secrets storage:

- Secret form of storage (whether it is plain or encrypted)
- Communication channel used for transfer
- Validity (how long secrets are valid and how often they are rotated)
- Availability (how long secrets are available in memory)

Azure stores secrets in an encrypted form by default and transfers them using a secure channel. However, users should note that they are responsible for a secret's validity and rotation. In terms of availability, there is a significant margin for improvement, as we have observed that some crucial secrets for serverless environments are stored inside environmental variables.

Environmental variables are present within every process and are inherited by default. This means that every child process spawned within a serverless execution will automatically contain environmental variables of the parent process. Thus, if parent processes contain secrets, every new process will contain them as well. This significantly increases the chance of exposure since a single vulnerability in any of these processes could lead to security concerns.

Technically, environmental variables are stored in the stack during the application execution process and are not deleted even after they are not needed anymore.

In the course of our investigation, we identified the following environmental variables as having security implications:

- APPSETTING_SCM_RUN_FROM_PACKAGE
  - This allows source code leakage when public access to the storage account is enabled.
  - To mitigate this security issue, users should disable public access to the storage account.
- APPSETTING_AzureWebJobsStorage
  - This allows read/write permission to the linked storage account.
  - This can also lead to a full RCE within the serverless environment.
  - To mitigate this security issue, users should disable storage account key access. However, it is important to note that doing this will disable Virtual Studio Code extensions and impair users' ability to upload to the storage account.
- CONTAINER_ENCRYPTION_KEY
  - This allows context decryption, which can be leaked from CONTAINER_START_CONTEXT_SAS_URI.
- CONTAINER_START_CONTEXT_SAS_URI
  - This allows encrypted container context with an initialization vector (IV) and SHA-256 of an Advanced Encryption Standard-encrypted payload.
  - This can be decrypted using CONTAINER_ENCRYPTION_KEY.The decrypted container context includes function and host keys and "MSISpecializationPayload", which has client certificate information for identity provider (IDP) services and allows the authentication of linked services that are beyond the score of Microsoft Azure.

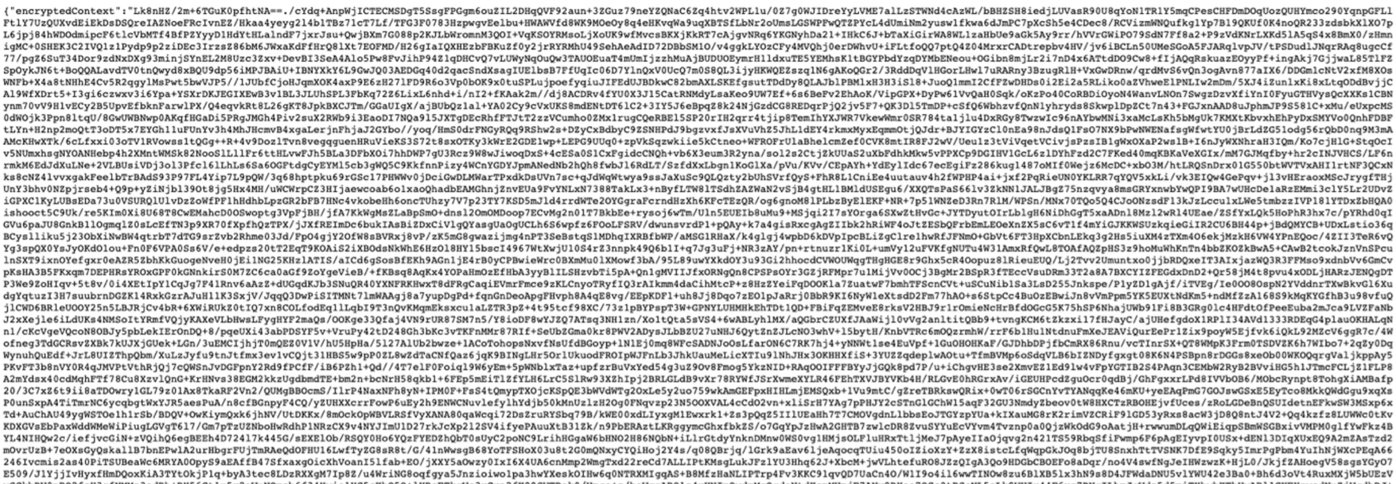


Figure 20. Encrypted container context

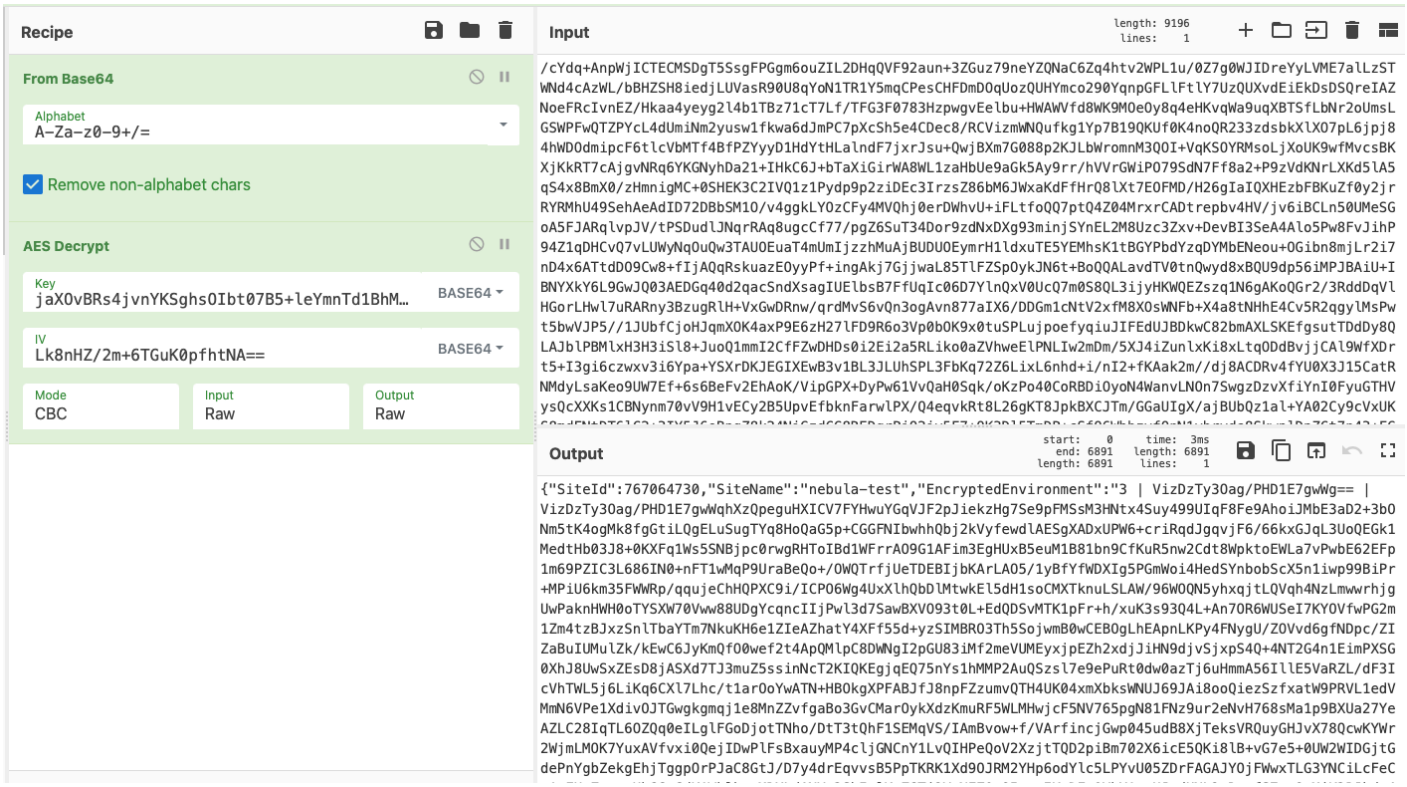


Figure 21. Decrypted container context

Figure 22. Leaked secrets from decrypted context

When configured, Azure can provide an IDP service that is accessible to serverless applications, which can allow Azure resources to obtain access tokens. Azure can distinguish between two types of managed identities:

- System-assigned
- User-assigned

A system-assigned managed identity is limited to a single resource and tied to the life cycle of this resource. Permissions can be granted to system-assigned IDPs via the Azure RBAC. Because Azure Active Directory (AD) authenticates managed identities, users do not need to store credentials in code.

The following is a list of system-assigned Azure resources that can be accessed using AD authentication via the IDP service:

- Azure Resource Manager
- Azure Data Lake
- Azure Cosmos DB
- Azure SQL
- Azure Data Explorer
- Azure Event Hubs
- Azure Service Bus
- Azure Storage blobs and queues
- Azure Analysis Services
- Azure Communication Services

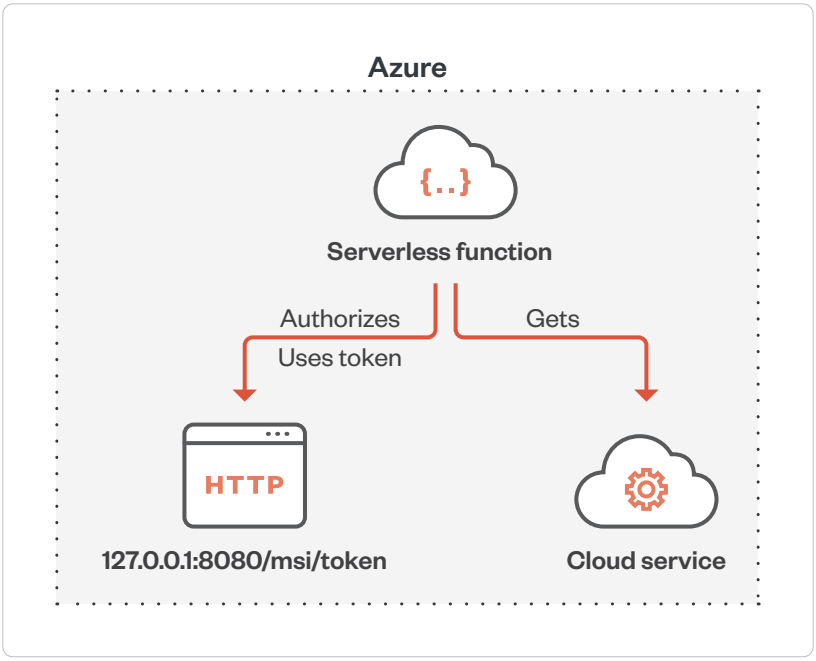


Figure 23. An example of how a system-assigned managed identity is used

Users who would need to access a specific resource in a serverless function need to configure the resource in such a way that it grants access to the function. This can be checked inside the Identity and Access Management (IAM) section of the resource.

After the resource is configured, a serverless function can obtain a token from a managed identity endpoint, which is an HTTP service running inside the container.

Figure 24. An example of getting a token from an IDP inside a serverless function

When the accessed resource is properly configured, it can be accessed using a token that was obtained via the IDP.

Figure 25. An example of an interaction with other cloud services using an IDP token for authentication

From a security perspective, it is important to discuss two access token properties: time validity and validity scope.

The time validity property is straightforward in its definition. This property simply means that when a token expires, it can no longer be used for accessing a resource. Meanwhile, the scope validity property defines whether a token can be used, as well as whether it can still be considered valid outside of the CSP boundary. For instance, the scope validity property can determine that a token is invalid in case token information is leaked. This property can invalidate a token and prevent further damage done by a malicious actors.

Based on our experiments, we discovered that the token time validity is one day, and that the access token remains valid even when it is used outside the serverless environment.

At this point, we would like to strongly emphasize the importance of implementing the principle of least privilege when configuring RBAC and IAM. There is no need to have an owner or administrator role when serverless functions only require read-only permissions.

In order for the IDP localhost service to work, it needs to contact an external service to request for an authentication token. The configuration for the external service is present inside an encrypted container context. However, this container context can be leaked and decrypted if environmental variables are accessed. Using decrypted content, a malicious user can authenticate a request to an external IDP service using a leaked client certificates. This means that when variable leakage occurs, malicious actors will be able to obtain a valid token for linked IDP-authenticated services with permission restrictions defined inside Azure. This also means that a request can be sent outside of Azure or from internet-connected devices.

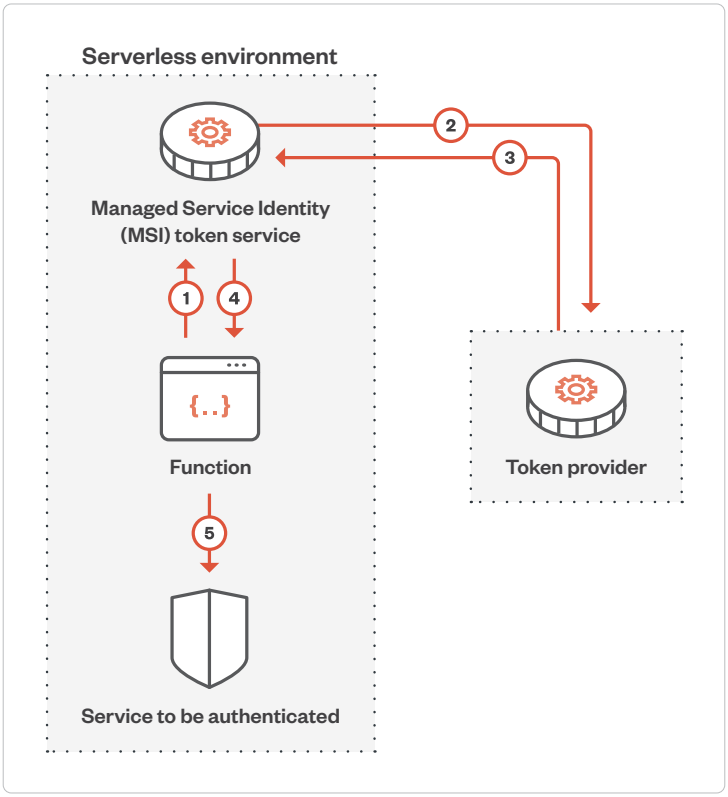


Figure 26. The architecture of a serverless IDP service

1. A serverless function sends a request to the MSI token service (*127.0.0.1:8081/msi/token*).
2. The MSI token service proxies a request to an external token provider and authenticates itself via a client certificate that has been generated for the environment.
3. The MSI token provider sends a response to the internal token service.
4. The internal token service sends a response to the serverless function.
5. The serverless function uses the token to authenticate itself into the desired resource.

The X509 client certificate is present within the encrypted context of the container and is unique per application ID (subject in certificate) with enabled managed identities. The certificate is valid for 180 days.

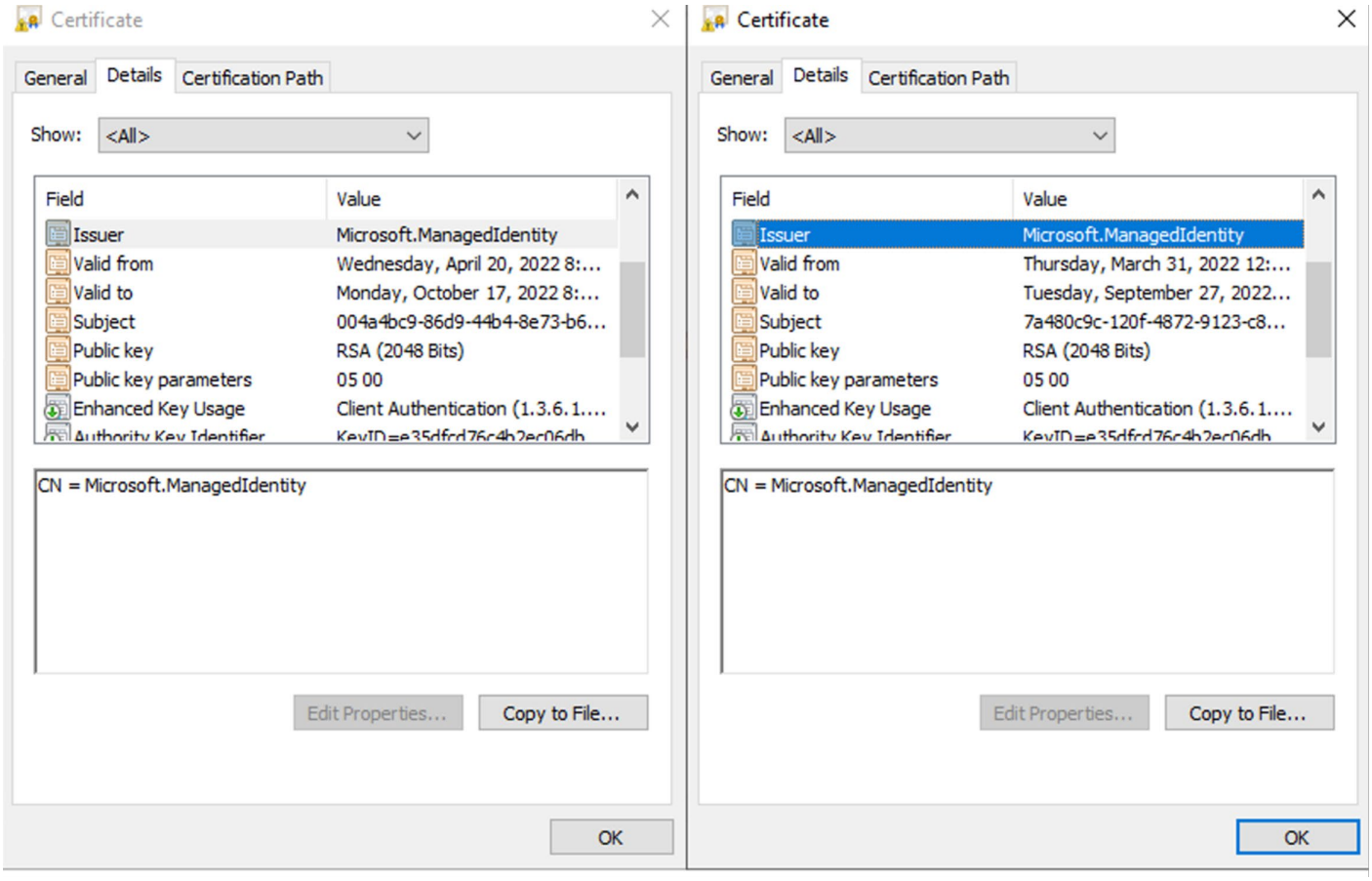


Figure 27. Details of managed identity certificates

User-defined identities are bound to a specific resource. For instance, one can bind an Azure Key Vault to a serverless function and assign read permissions to the Vault's secrets. Once this is done, the user-defined identity can then be used to retrieve secrets from the vault. The identity is identified by its "client_id" property, which is used to retrieve access tokens from the service.

The same MSI token service is used to obtain tokens for user-assigned identities and to minimize security risks. When using this service, users should employ the principle of least privilege, which prohibits the authorization of a service or an application with more rights than it needs. Users can also benefit from limiting the availability of the linked service, or from avoiding leaving the service publicly available to limit the possibility of token leakage. Notably, token leakage can be abused for authentication outside of the serverless environment's scope.

Another example of an environmental variable that poses a security risk is the linked storage account connection string. When this secret is leaked, which can happen when the "AzureWebStorage" environmental variable is leaked, it could lead to a full RCE within the serverless context. This threat scenario requires a vulnerability that would leak environmental variable content, which could be caused either by a vulnerability inside the deployed user function or the environment itself. Because the environmental variable contains a connection string, when a storage account key access is enabled, a tool such as the Azure Storage Explorer[22] can be used to manipulate the storage account.

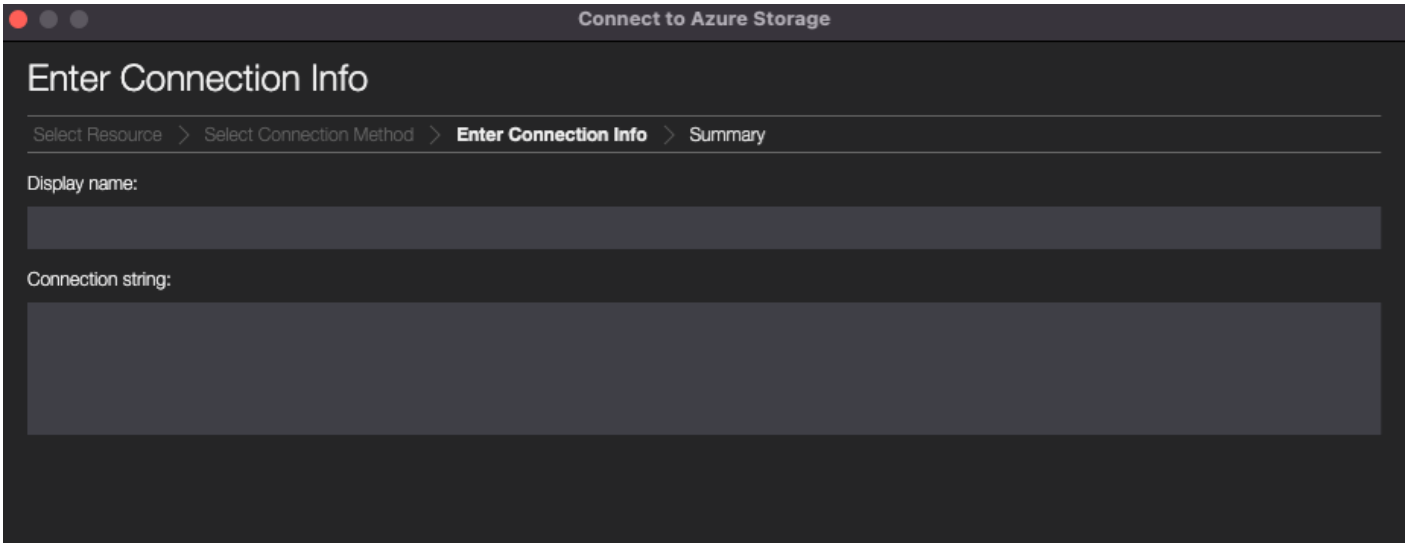


Figure 28. The Azure Storage Explorer connection string dialog where the "AzureWebStorage" variable can be entered

When connected, a user can simply delete and upload a new version of the serverless function, which can alter the serverless function itself.

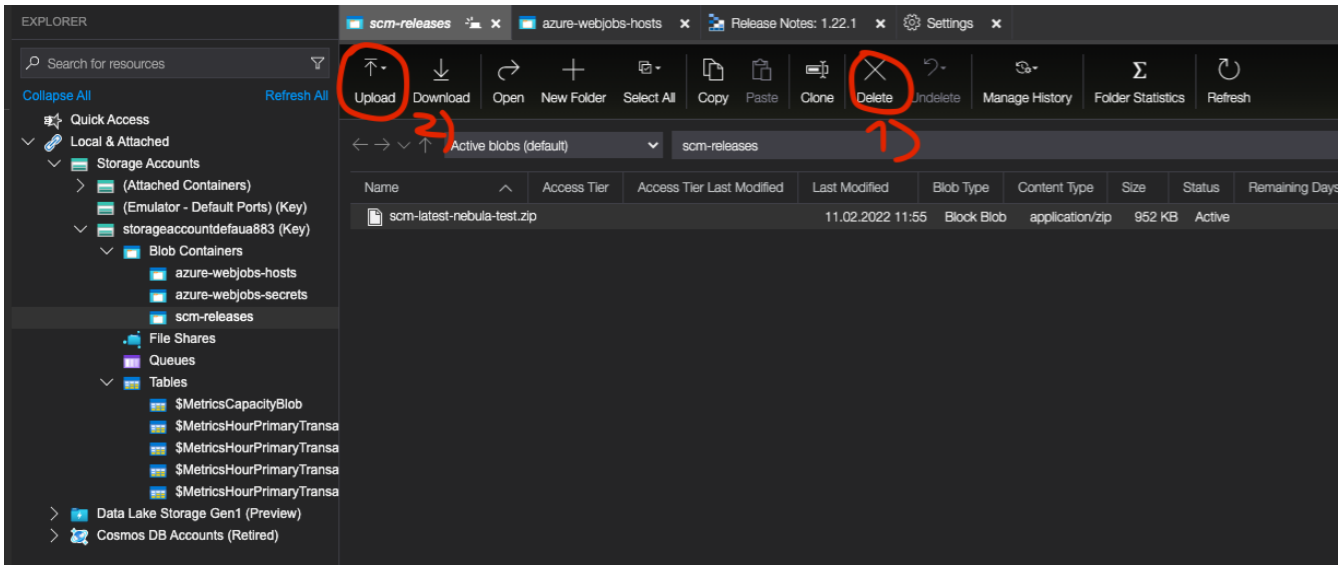


Figure 29. Altering a serverless function using the Azure Storage Explorer

This can allow the execution of an attacker-provided code within the serverless function execution context.

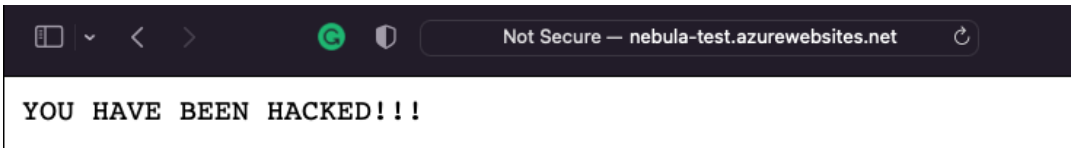


Figure 30. Execution of an altered serverless function

Azure Functions on Windows-Based Environments

Similar to the Azure App Service, the Windows-based Azure Functions environment only supports selected stacks and versions, such as .NET-powered applications, and those running on Java and Ruby. Like the Azure App Service sandbox, the Windows-based Azure Functions service also has limited capabilities.

During our investigation, we observed sensitive environmental variables in this environment. However, compared to Linux-hosted serverless applications, we did not see CONTAINER_START_CONTEXT_SAS_URI, which is the container-encrypted context linkage. What we found instead was the connection string to Azure storage, which unfortunately leads to the same threats that we have discussed regarding the Linux environment.

```
APPSETTING_FUNCTIONS_WORKER_RUNTIME=dotnet
APPINSIGHTS_INSTRUMENTATIONKEY=1d53e562-1dcc-4153-944a-98270ae1a5fc
APPSETTING_APPINSIGHTS_INSTRUMENTATIONKEY=1d53e562-1dcc-4153-944a-98270ae1a5fc
APPLICATIONINSIGHTS_CONNECTION_STRING=InstrumentationKey=1d53e562-1dcc-4153-944a-98270ae1a5fc;IngestionEndpoint=https://westeurope-
APPSETTING_APPLICATIONINSIGHTS_CONNECTION_STRING=InstrumentationKey=1d53e562-1dcc-4153-944a-98270ae1a5fc;IngestionEndpoint=https://
AzureWebJobsStorage=DefaultEndpointsProtocol=https;AccountName=nebulagroupb2b2;AccountKey=iK/22YjrZC8iLOfa61rTdDzLB+fmNtIAyh7epYy00
APPSETTING_AzureWebJobsStorage=DefaultEndpointsProtocol=https;AccountName=nebulagroupb2b2;AccountKey=iK/22YjrZC8iLOfa61rTdDzLB+fmNt
WEBSITE_CONTENTAZUREFILECONNECTIONSTRING=DefaultEndpointsProtocol=https;AccountName=nebulagroupb2b2;AccountKey=iK/22YjrZC8iLOfa61rT
APPSETTING_WEBSITE_CONTENTAZUREFILECONNECTIONSTRING=DefaultEndpointsProtocol=https;AccountName=nebulagroupb2b2;AccountKey=iK/22YjrZ
WEBSITE_CONTENTSHARE=win-nebulab8ca
APPSETTING_WEBSITE_CONTENTSHARE=win-nebulab8ca
WEBSITE_RUN_FROM_PACKAGE=1
APPSETTING_WEBSITE_RUN_FROM_PACKAGE=1
```

Figure 31. Windows-hosted environmental variables on Azure Functions

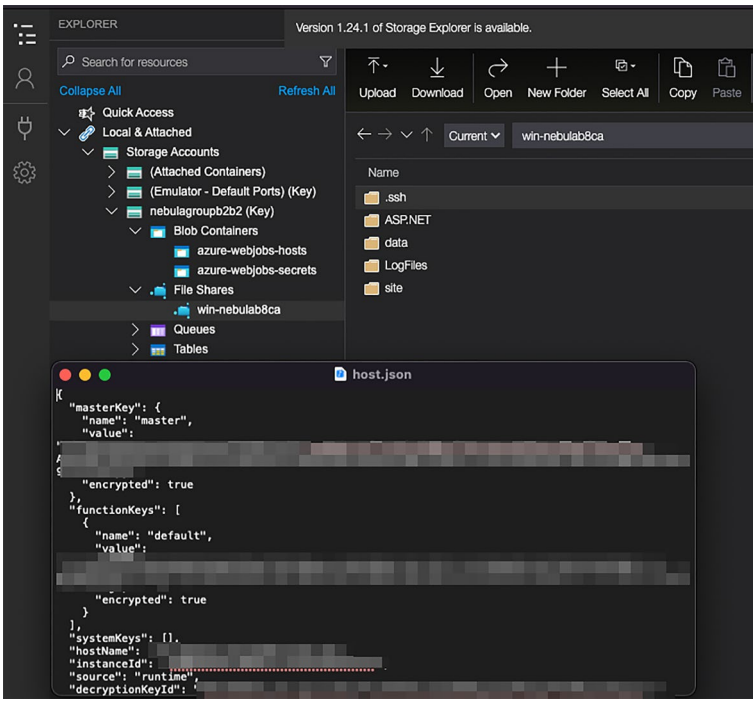


Figure 32. Connection to Azure Storage using an environmental variable connection string

Comparing Azure Services on Linux- and Windows-Based Environments

After investigating the Azure App Service and Azure Functions serverless services, we have discovered several security gaps that are mostly related to environmental variables that are used to store confidential information. We believe that environmental variables are one of the worst places to store secrets because using them for secrets storage provides malicious actors with an additional attack surface copied within every child process that can be seen in case of compromise, or when memory access vulnerabilities are abused.

Another security issue that we have identified is related to users' unwise architectural decisions. An example of this is the use of the master key for SSH access, which would then allow privilege escalation inside a container with a known password. It is therefore imperative that users choose a public key cryptography for authentication to the SSH service to keep the system secure.

Some of the sensitive environmental variables also contained publicly accessible URL endpoints together with valid tokens, which upon exposure can grant malicious actors access to other pieces of sensitive information. Such sensitive information could then be used for further attacks on cloud environments.

It is also imperative to determine whether these URL endpoints should be inherently publicly accessible. If the URL endpoints are publicly accessible, they are not limited to the cloud environment alone, meaning that tokens should be inside environmental variables.

Serverless environments are designed in such a way that the user is responsible for implementing security best practices and policies to keep them secure. Indeed, some of these settings are not enabled by default, nor are they included in all available Azure serverless packages. The application code security deployed by users is crucial because without proper security, it can serve as an entry point for attackers.

According to our analysis, applications running on Windows are more secure than those that are running on Linux. In the Windows environment, an application sandbox prevents access to all resources. Meanwhile, the Linux environment only uses Docker isolation. Even though the Docker container engine is used for running containers inside the Linux environment, it is not the only isolation mechanism; the tenants are also separated at the hypervisor level. We also observed that some sensitive environmental variables that are found in the Linux environments are not present inside Windows environments.

Improving Standard Security Using Custom Images

Both Azure App Service and Azure Functions give users the option to create their own Docker images to run serverless code in Linux environments. Because our investigation showed that sensitive information can be found inside environmental variables, we tried to enhance the official image and harden overall security while retaining maximum functionality. Overall, we were mostly successful in our efforts, except for the SSH access on Azure App Service, which would require enhancements on the side of the CSP.

Azure Functions includes the App Service plan, which guarantees physical hardware allocation that we can imagine as a VM. Inside that, we found a Docker container engine installed. This engine executes a container image that is built with the Azure-functions-host runtime. Azure-functions-host effectively manages the Azure Functions runtime, making it responsible for communication with Azure back ends.

The Azure-functions-host executes the Azure-functions-worker when the serverless function execution is triggered, which then executes the actual serverless application that has function app code provided.

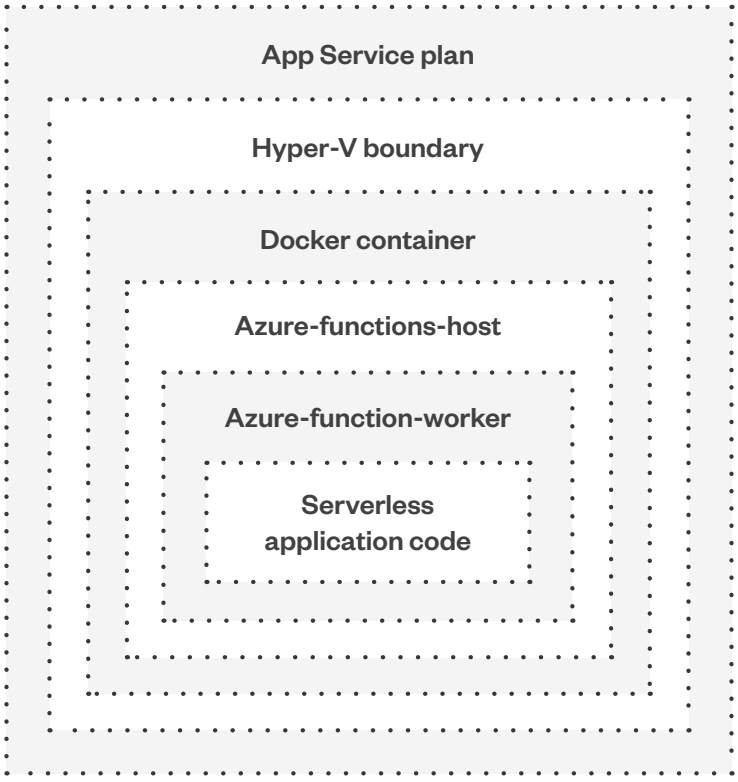


Figure 33. Azure App Service plan inclusions

The actual Docker container image could be replaced by a custom image that must contain the azure-functions-host so that it can work with Azure Functions. It is important to note that the custom container option is only available for creating function apps on the Linux platform. A premium subscription plan is also required.

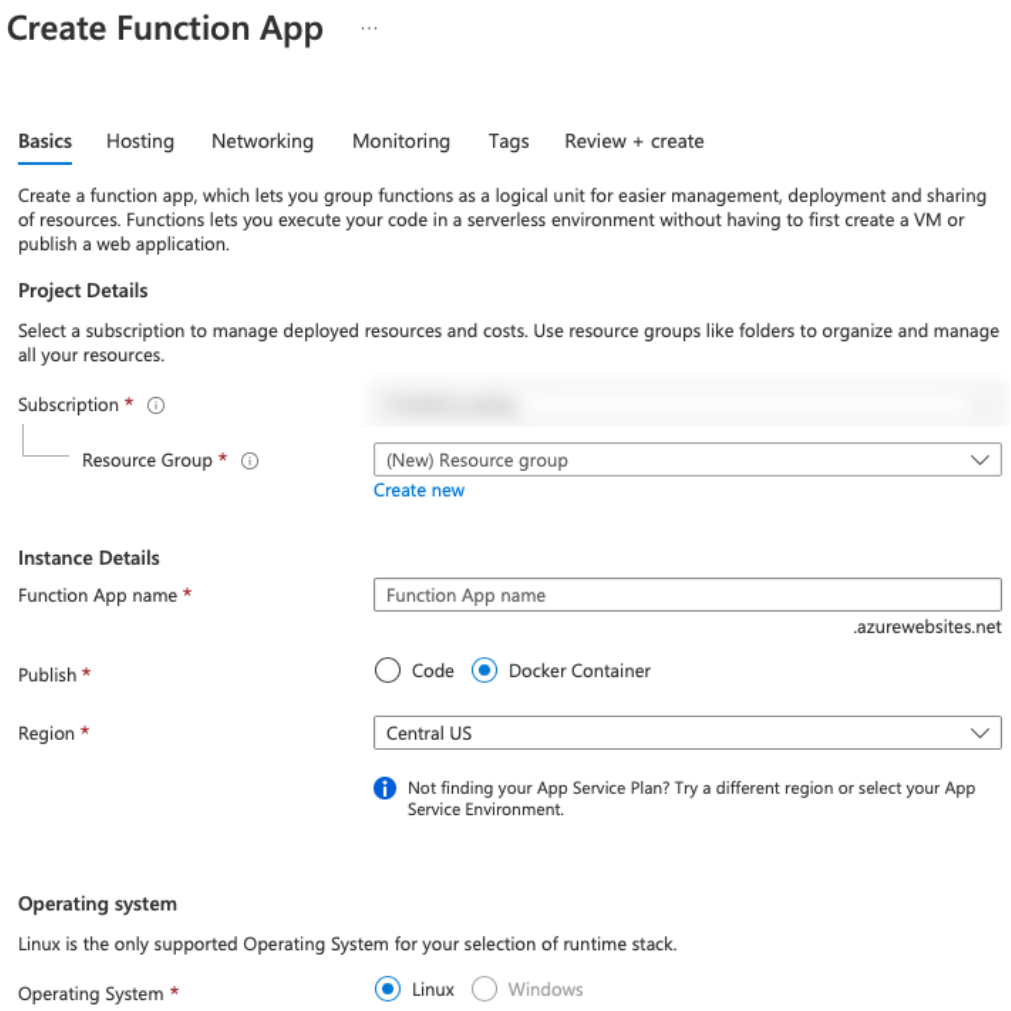


Figure 34. Creating an Azure Function app with a custom Docker container

We followed Azure documentation for creating a custom container using Python as our code interpreter.[23] However, we made a slight modification and chose a private container registry inside Azure for deployment.

Figure 35. Setting a private registry for a custom container serverless deployment on Azure

After building the container image locally, we pushed it into a private registry, which we then configured to be linked with serverless functions.

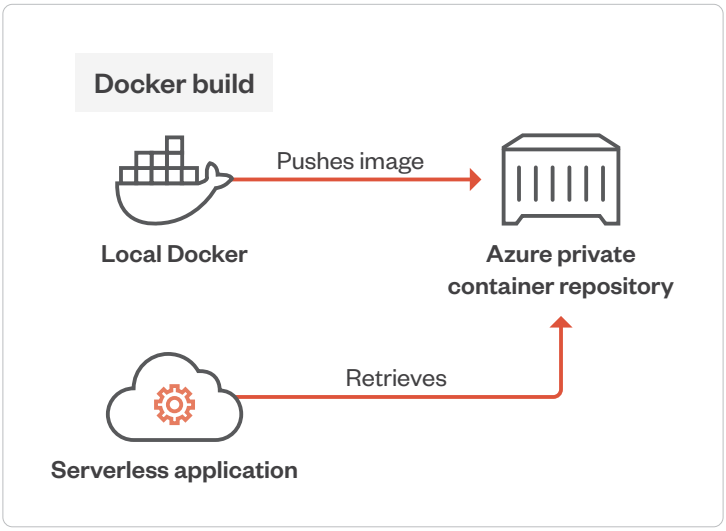


Figure 36. Deployment diagram

Building a Custom Image

We have chosen *mcr.microsoft.com/azure-functions/python:4-python3.9* as our base image from the Azure Functions base list[24] available via the Microsoft Artifact Registry.[25]

First, we must acknowledge that some environmental variables will most likely be required to have the function-host running for Azure Functions to work. However, we also wanted to ensure that our serverless application would not have access to sensitive variables.

It is also important to compare the container image differences of the chosen Python stack when creating an Azure function via *azure-functions/mesh:3.7.1-python3.9* and the Azure Functions base Python image.

The first difference is that the mesh container image executes an initialization wrapper script under the root user. The app user would then execute the *WebHost.dll* binary using the "sudo" command, wherein all the environmental variables would be passed. The azure-functions/base images would execute *WebHost.dll* binaries under root user by default. The *WebHost.dll* would then execute the python-worker, the process that will execute the serverless code itself.

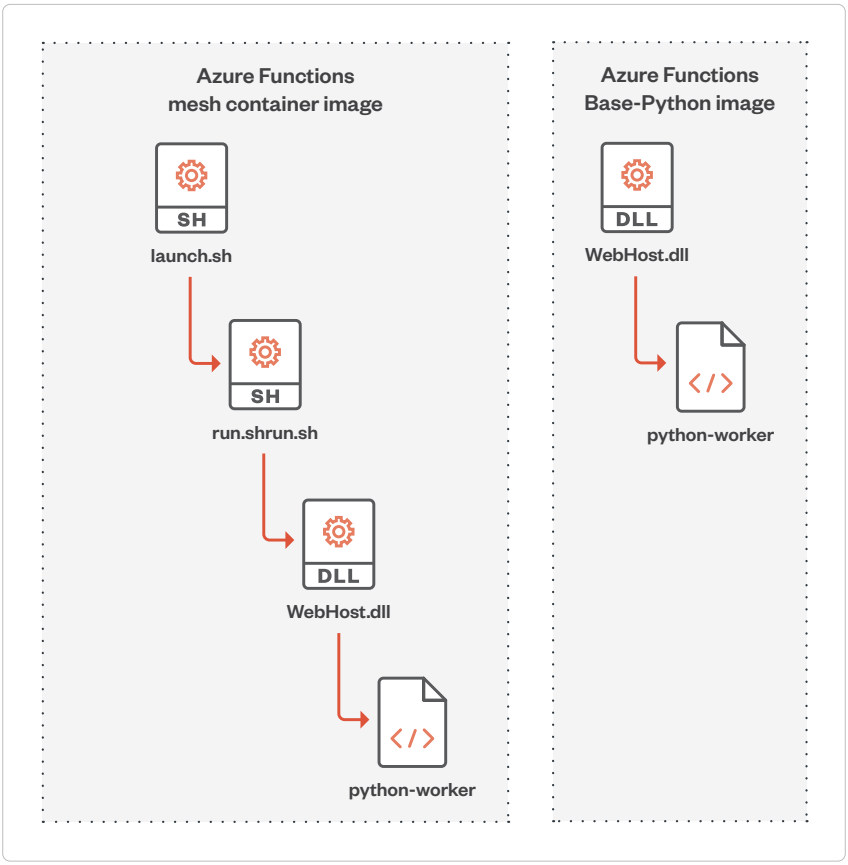


Figure 37. Comparison of container images

Custom Image Goals

Based on our investigation, we have decided to set the following goals:

- To get rid of sensitive environmental variables inside serverless applications' executing context
- To minimize required permissions and needed container images for our serverless application
- To have a minimal impact on Azure Functions' functionality

Getting Rid of Sensitive Environmental Variables

WebHost.dll needs sensitive environmental variables to run. However, because of the nature of the selected application design, they are inherited by the python-worker process, wherein serverless code is also executed. Since environmental variables are part of the process memory, our options are limited.

Another thing to consider is that other processes' environmental variables running under the same user can be printed just by using read permissions. This is because of the nature of the "/proc/" file system.

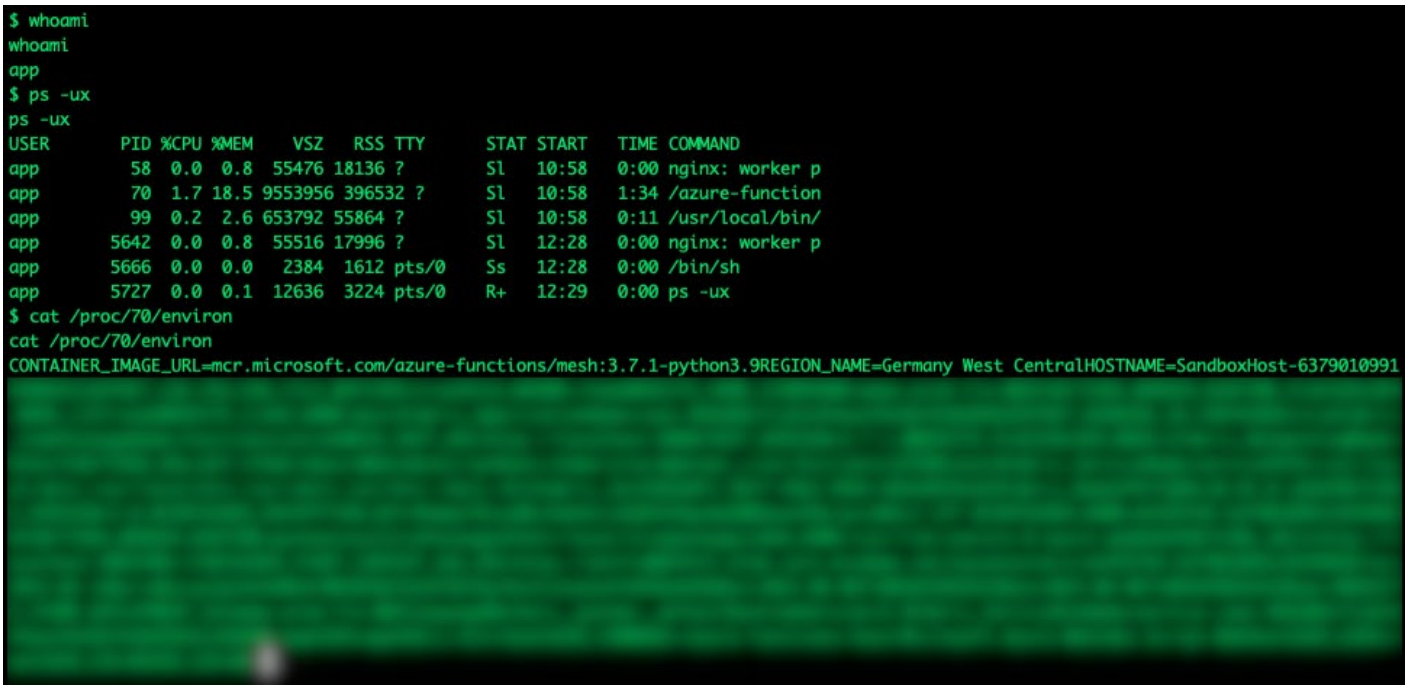


Figure 38. Accessing other processes' environmental variables

Because of this feature, our best bet was to modify the *WebHost.dll* binary or its configuration to execute the language-worker under a different user and without accessing sensitive environmental variables.

By analyzing the container image build process, we were able to investigate the best injection point to alter the execution behavior. Because our interpreter was Python, we determined that the easiest way to inject code was to alter the Python binary within the container and to replace it with a custom shell script.

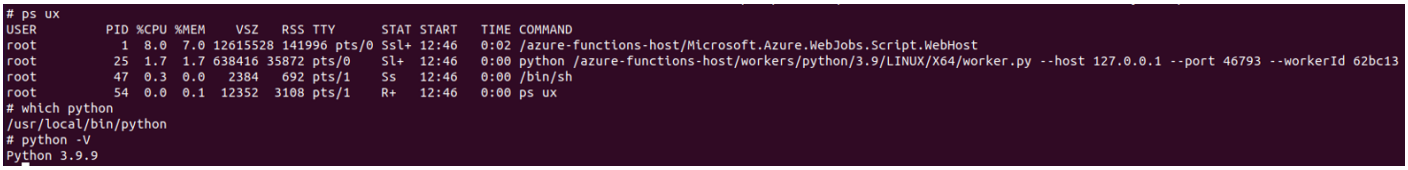


Figure 39. Environmental analysis of the container

Our custom shell script was simple. It executed the Python worker as a different user using the "sudo -u www-data" command without passing environmental variables. To pass environmental variables, sensitive ones can be suppressed by using the unset command and the -E parameter of "sudo."

```
#!/bin/sh

sudo -u www-data python-int $*
```

Figure 40. Executing the Python worker via the user "www-data" command passing all other parameters

We were able to get rid of environmental variables and limit access to sensitive variables when needed.

```
whoami
www-data
$ ps ux
ps ux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
www-data      30  0.8  1.7 638908 36020 pts/0    Sl+  12:58   0:00 python-int /a
www-data      50  0.0  0.0   2384   760 pts/1    Ss   12:58   0:00 /bin/sh
www-data      52  0.0  0.1  12352  3152 pts/1    R+   12:59   0:00 ps ux
$ env
env
SUDO_GID=0
MAIL=/var/mail/www-data
USER=www-data
HOSTNAME=b547a6ebac03
HOME=/var/www
SUDO_UID=0
LOGNAME=www-data
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LANG=C.UTF-8
SUDO_COMMAND=/usr/local/bin/python-int /azure-functions-host/workers/python/3.9/LINUX/X64/worker.py --host 127.0.0.1 --port 34355 --workerId e4143cf7-fe55-4a82-b85c-16edd5da285b --requestId 89
3a61b5-ca23-45c9-a8a8-5546c1407287 --grpcMaxMessageLength 2147483647
SHELL=/usr/sbin/nologin
SUDO_USER=root
PWD=/home/site/wwwroot
```

Figure 41. A running Python worker without additional environmental variables

```
$ ps ax
ps ax
    PID TTY      STAT   TIME COMMAND
      1 pts/0    Ssl+   0:02 /azure-functions-host/Microsoft.Azure.WebJobs.Scrip
     26 pts/0    S+     0:00 /bin/sh /usr/bin/python /azure-functions-host/worke
     28 pts/0    S+     0:00 sudo -u www-data python-int /azure-functions-host/w
     30 pts/0    Sl+    0:00 python-int /azure-functions-host/workers/python/3.9
     50 pts/1    Ss     0:00 /bin/sh
     59 pts/1    R+     0:00 ps ax
$ cat /proc/1/environ
cat /proc/1/environ
cat: /proc/1/environ: Permission denied
$
```

Figure 42. Denying access to sensitive environmental variables

We also tested if the changes we made would still allow us to run our serverless function within the Azure environment successfully.

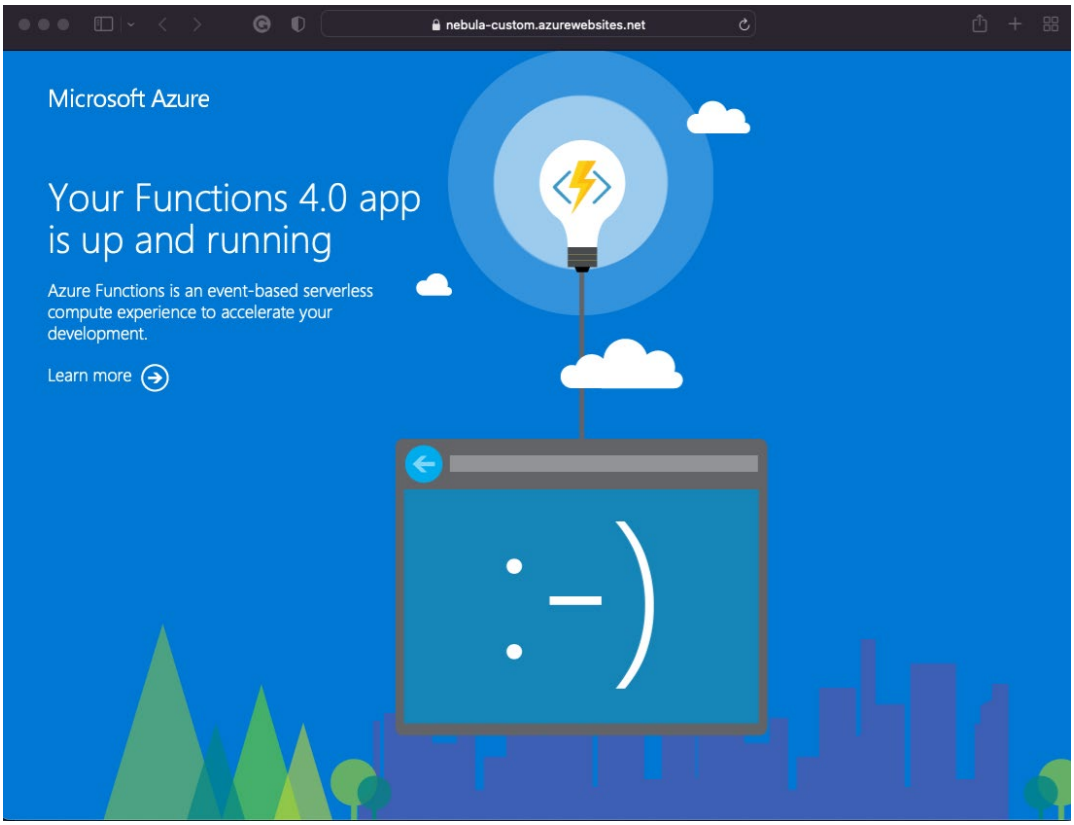


Figure 43. A custom container running on Azure without environmental variables

Minimizing Container Binaries and Permissions

The second goal was to minimize container binaries and their respective sizes to the bare minimum, which means minimizing it to include only the application and its dependencies. This is referred to as the Distroless approach.[26] The base image for our custom container would be minimized by removing binaries that are not essential for it to run. Due to the removal of inessential binaries that could also be abused by attackers upon exploitation, the container image itself will become smaller.

The binaries that we removed from the container image were all binaries from the "/bin" directory. Our custom shell was also in this directory, which is why we needed to update our environmental tweak. We also removed the "curl," "wget," and "perl" binaries located in the "/usr/bin" directory in our investigation.

```
FROM mcr.microsoft.com/azure-functions/python:4.0-python3.9

ENV AzureWebJobsScriptRoot=/home/site/wwwroot \
    AzureFunctionsJobHost__Logging__Console__IsEnabled=true

ENV environment=Production

COPY requirements.txt /
RUN pip install -r /requirements.txt

RUN apt-get install -y sudo

RUN mv /usr/local/bin/python /usr/local/bin/python-int
COPY python /usr/bin/python
COPY . /home/site/wwwroot

RUN rm -f /usr/bin/curl
RUN rm -f /usr/bin/wget
RUN rm -f /usr/bin/perl
RUN rm -f /bin/*


EXPOSE 80
```

Figure 44. An example of a Distroless container Docker file

Our custom script no longer worked because we removed the shell interpreter. To counter this, we replaced our custom script with a custom compiled binary that does the same job but uses the "execve" system function instead of a shell interpreter.[27] This function also allows the setting of environmental variables for the new process, allowing users to specify which non-sensitive environmental variables would be needed in their application. Users can also dynamically obtain variables via the "getenv" function.[28]

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char**argv) {

    int k = (argc + 5) * sizeof(char*);
    char *dargs = (char*)malloc(k);
    memset(dargs,0,k);

    char ** c = (char**)dargs;
    c[0] = "/usr/bin/sudo";
    c[1] = "-u";
    c[2] = "www-data";
    c[3] = "-E";
    c[4] = "/usr/local/bin/python-int";

    for(int i =0; i< argc-1;i++){
        c[5+i] = argv[1+i];
    }
   int ret = 0;

   char * environment[] = {"myENV=VALUE", NULL};

   ret = execve("/usr/bin/sudo",c, environment);

   free(dargs);
   return ret;

}
```

Figure 45. Custom binary proof of concept

Figure 46. A successfully deployed custom image inside the Azure serverless environment where printing environmental variables are available to the executed serverless code

Based on our analysis, we can produce a custom image on Azure App Service, have a low-privileged user execute code, get rid of sensitive environmental variables, and harden the "Docker!" root password with minimal distribution.

How to Improve the Overall State of Serverless Security

For as long as we have been studying and working in cybersecurity, we have understood that the concept of one-hundred-percent security is a myth. As cybersecurity defenders, we can only mitigate risks and impose difficulties on malicious actors, but at the end of the day, there is no such thing as an impenetrable system.

There are many security models to help achieve a good level of security. One of the more popular and widely adopted models is the confidentiality, integrity, and availability (CIA) triad,[29] a well-researched and experimented model that most industry standard guides use as their base for security development, implementation, and validation.

After we performed a deep dive into the CIA model, it became clear to us that the model weighs security against the usability of the handled data. In this model, confidentiality guarantees that the data is only accessible to those who have the right to access it. Integrity protects the data against outsiders being able to read, change, or cause damage. Meanwhile, availability requires that the data be available when it is needed and when a user passes all security checks.

Figure 47. The CIA triad model

For example, if a developer applies the CIA model for a web application with an e-commerce component, the model would help the developer find the best solutions for the following business and security concerns:

- **Confidentiality:** Before making a purchase, the customer needs to create a user account with a unique user ID. The user account also needs to be protected by a password. The web application should also offer multifactor authentication (MFA).
- **Integrity:** At the time of the purchase, the list of items or data that the customer sees must be the same as the list on the merchant side.
- **Availability:** The application's website or portal needs to be available for customers at all times so that customers can access it at their convenience and without interruption.

Although the DIE model, which is not as widely known as the CIA model, is sometimes viewed as a good competition, it is also seen as an unachievable model to use.

If we think about securing regular infrastructure, the DIE model might work. However, it might prove to be really difficult, if not impossible, to implement such a restrictive model on a physical server that runs a full operating system with several applications installed that need to communicate with each other. The DIE model can be seen as perfect in theory, but it might prove to be difficult to implement in real life.

Although DIE is not as popular as CIA, it is possible that DIE might be a good model to use with more modern forms of computing infrastructures.

The growing adoption of cloud infrastructure, containers, or a combination of both, raises the possibility of implementing the DIE model. Because of the manner of orchestration, it is easier than ever to distribute a system without needing Open Virtualization Formats (OVFs) or snapshots.

Containers that are properly developed and are only running one process can be easily immutable. All modifications needed can be done during the development of the container at runtime. There is no real need to change the system, and for better security, there might not be a need for a read/write system at all.

Once a container, a pod, or a cloud instance runs without needing to be online anymore, by its design, it can be taken off without any prejudice when needed and without delay, in order to power it back again or to use it to retrieve network configuration. Ephemerality can thus be achieved without losing anything important.

Achieving the DIE Model

The use of this particular infrastructure does not automatically mean that DIE is achieved. Based on our assessment of real-life scenarios, it was not rare to find improper implementations of DIE. These scenarios included companies that moved their whole infrastructure to a CSP but kept the same model as if they were still using a local data center and using only workload servers but on the cloud. Another example involved companies using a single container to run everything, similar to how a regular physical server would run, except without orchestration or service isolation.

These scenarios undermine not only the purpose of leveraging these more modern infrastructures but also their security implementations. Sadly, some users think that using cloud environments or containers is more secure and that there is no need to hire cybersecurity personnel or use security products or tools. This is far from true – even if users follow security best practices, it is still highly recommended that they also rely on security solutions to help keep their cloud and container environments protected against threats.

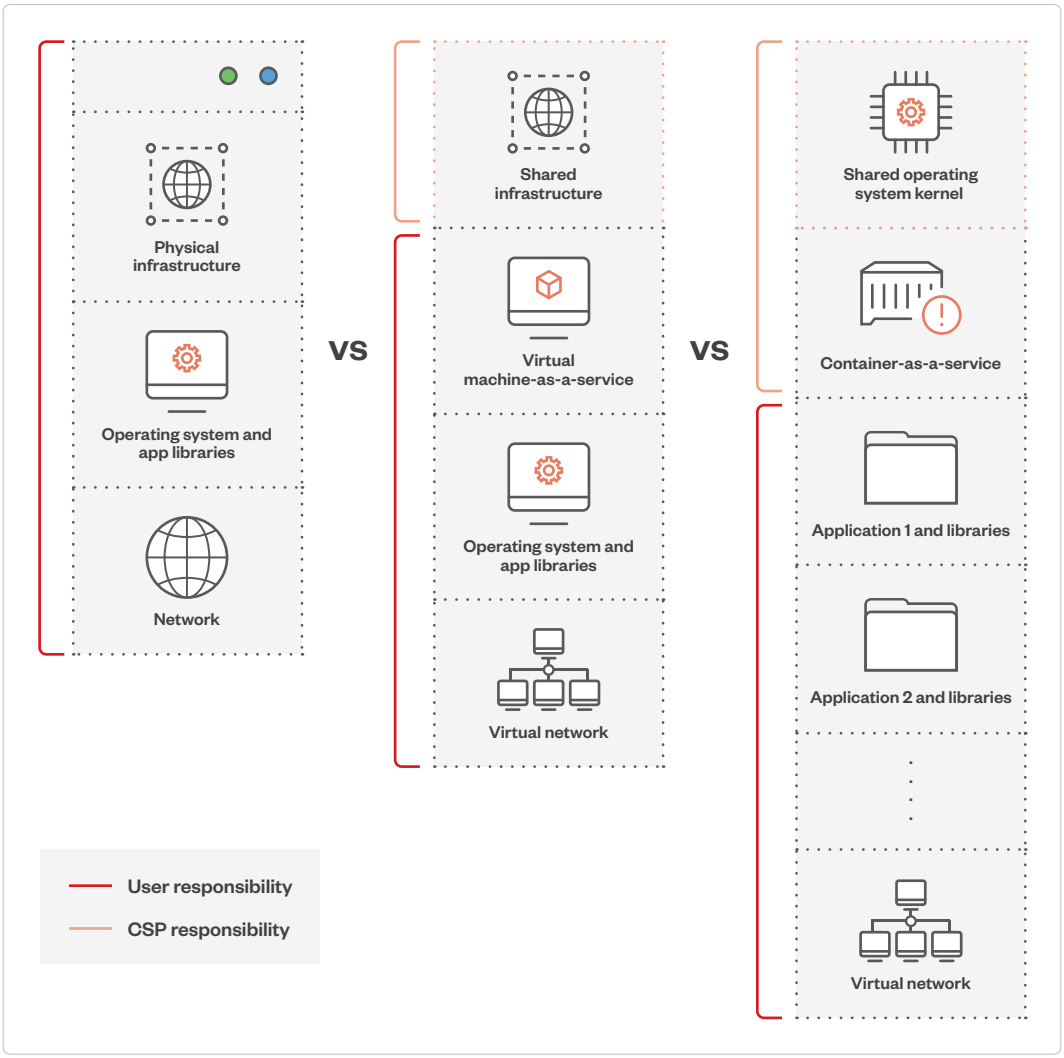


Figure 48. The shared responsibility shift over different infrastructures

Cloud platforms and containers began as a kind of a technological revolution that brought about many monetary, resource, and security advantages. As previously mentioned, applying the DIE model on cloud infrastructures would be easier than less modern infrastructures. However, a new revolution in the form of serverless technology has started gaining more ground in the last few years. Indeed, already some young companies are being born and developed using such technology.

Up until this point, we have discussed and presented what serverless technology is, how it should work ideally, as well as how it works realistically. And although it is far from being perfectly secure, in a nutshell, it works with each principle of the DIE security model:

- **Distributed:** Serverless is distributed by nature. It only runs when it is called, and a user can determine the default configuration for when, where, and how triggers are called.
- **Immutable:** Most of the serverless services we presented in our research have their file systems fully or partially read-only by default.
- **Ephemeral:** Most of the serverless services we presented in our study are also fully or semi-ephemeral by default, with a small cache being implemented for spare resources.

The Full Operating System and Native Tools

In our analysis, the framework we created relied heavily on native tools for acquiring enough information on the systems, as we would expect an attacker would do. We observed the availability of native tools without access restrictions on most of the services we checked.

In some cases, the whole operating system was present and available as a container. With such tools, it was relatively easy to map out crucial information about the environment and plan the next test phase.

Tools such as "ping," "ip,"[30] "ifconfig,"[31] "netstat,"[32] and "ss"[33] helped us understand the environment better by giving us information on whether it had internet access, what kind of network configuration it had, and if there was any other host alive on the same network in some cases.

For services which had internet access, "wget" and "curl" made it possible to download other tools for further exploration of the environment. Another very handy network tool that can be dangerous to have around is called "netcat", which allows the creation of a low-effort backdoor.

In certain cases, user promotion tools such as "sudo" or "su" were also observed. This does not necessarily mean that these tools could be directly used by the user, but these can be targets for exploitation.

As mentioned in the beginning of this section, there's no such thing as one-hundred-percent security. However, it is helpful for organizations to remove unnecessary tools – ones that malicious actors can abuse. The removal of such tools can impose a bigger challenge for malicious actors in the sense that they would need to develop their own arsenal, find a way to drop them in the system, and find an alternative way to execute them. Also, on top of developing custom tools, users would also need to consider dependencies since there is also an option not only to remove unused executables but also libraries.

Function Container

In most serverless services, the container that runs the code has all the resources of a full Debian-based operating system. Malicious actors who gain access to this environment also gain access to a great number of native tools.

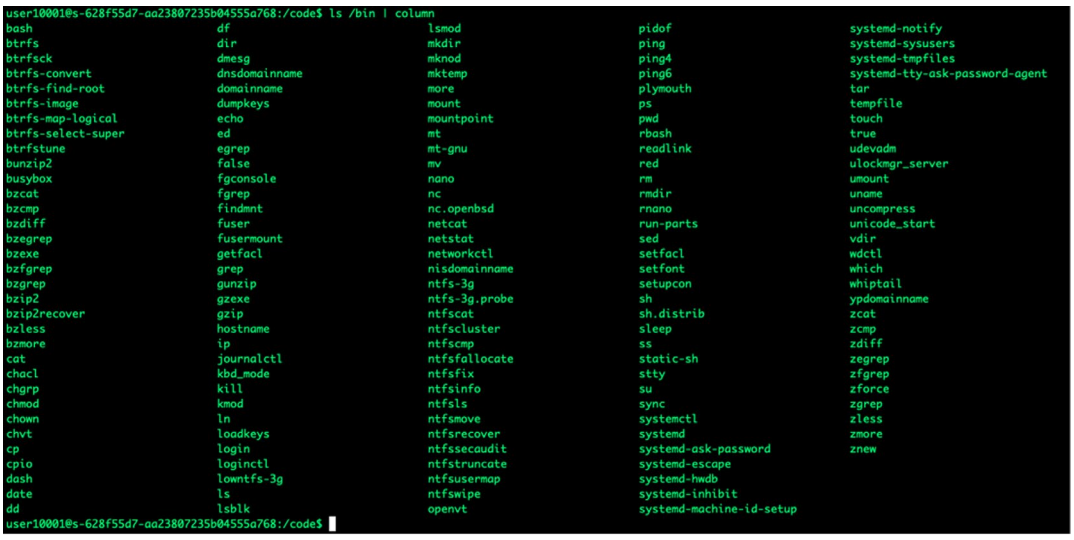


Figure 49. A list of all available default commands inside the "/bin" of the function container

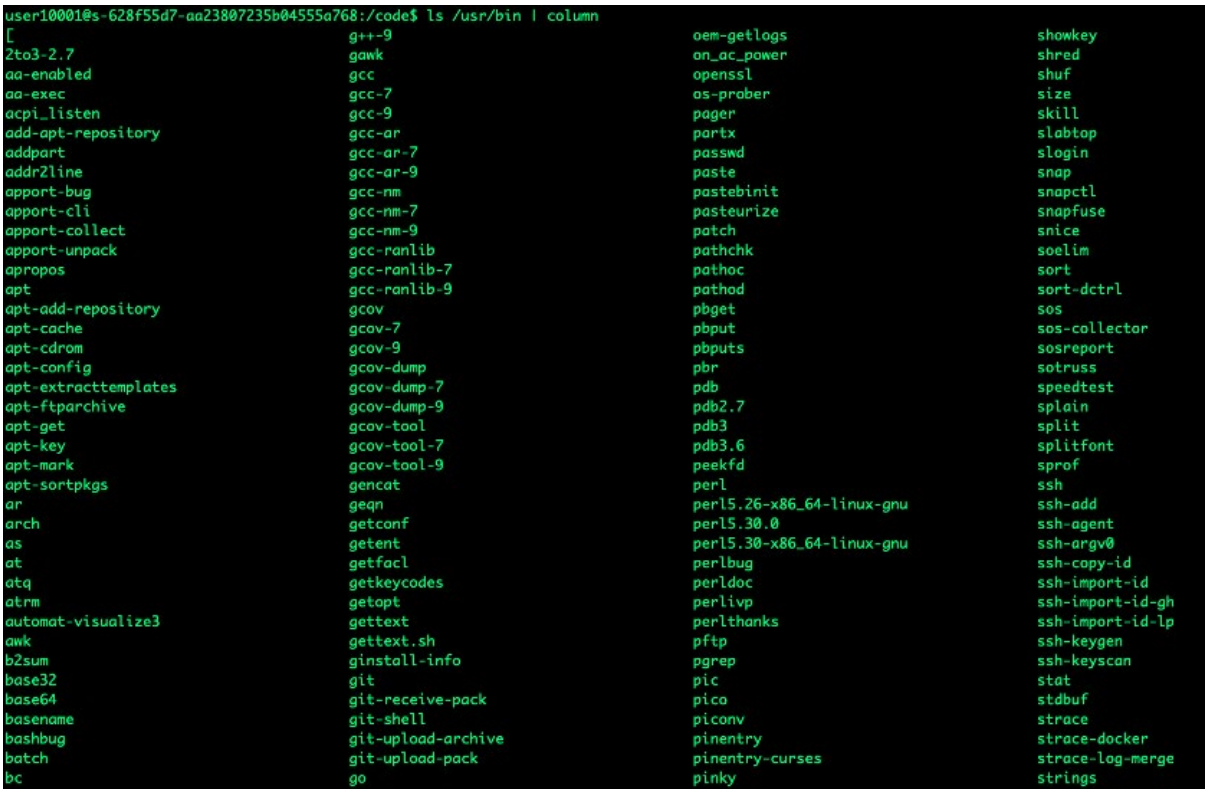


Figure 50. A list of all available default commands inside the "/usr/bin" of the function container

In case the tools are not enough, malicious actors can use "curl" and "wget" to add more tools to their arsenal for the next phase of their attack. Aside from this, an attacker can abuse native tools to get enough information about the operating system, the kernel, and the container runtime. If any of these are vulnerable, a malicious actor can download, compile, and deploy an exploit into the system. This is because of the presence of compilation tools that have execution permissions.

```
user10001@s-628f55d7-aa23807235b04555a768:/code$ which wget curl gcc sudo su ip ifconfig ss netstat ps
ifconfig ss netstat psdo su ip i
/usr/bin/wget
/usr/bin/curl
/usr/bin/gcc
/bin/su
/sbin/ip
/bin/ss
/bin/ps
user10001@s-628f55d7-aa23807235b04555a768:/code$
```

Figure 51. An example of available tools inside serverless environments that are commonly used in cloud attacks

Container Cache Removal and Lifetime

The original idea behind the use of serverless functions and containers as they pertain to runtime and data is to be able to recycle the serverless host or container as much as possible. This means that ideally, the serverless host or container should not hold any considerable amount of information.

In an ideal scenario, a container would run as a single process. If the purpose of the container does not involve storing information such as logs, or if it is not meant to serve as a database, everything inside it should be disposable.

In the serverless world, the delineation is more defined. The container or host of the function runs the code and exits, effectively killing the container or micro VM. If the container or host processes any data that must be retained, it gets forwarded to another service dedicated for that sole purpose. However, most CSPs implement a kind of a cache so that the user does not get a "cold" start every time a function is called.

That is where the line between security and usability should be drawn. Although not having a cache could impact initial performance, having it can give an attacker the opportunity to access the environment and drop different pieces of their arsenal to further expand the attack surface.

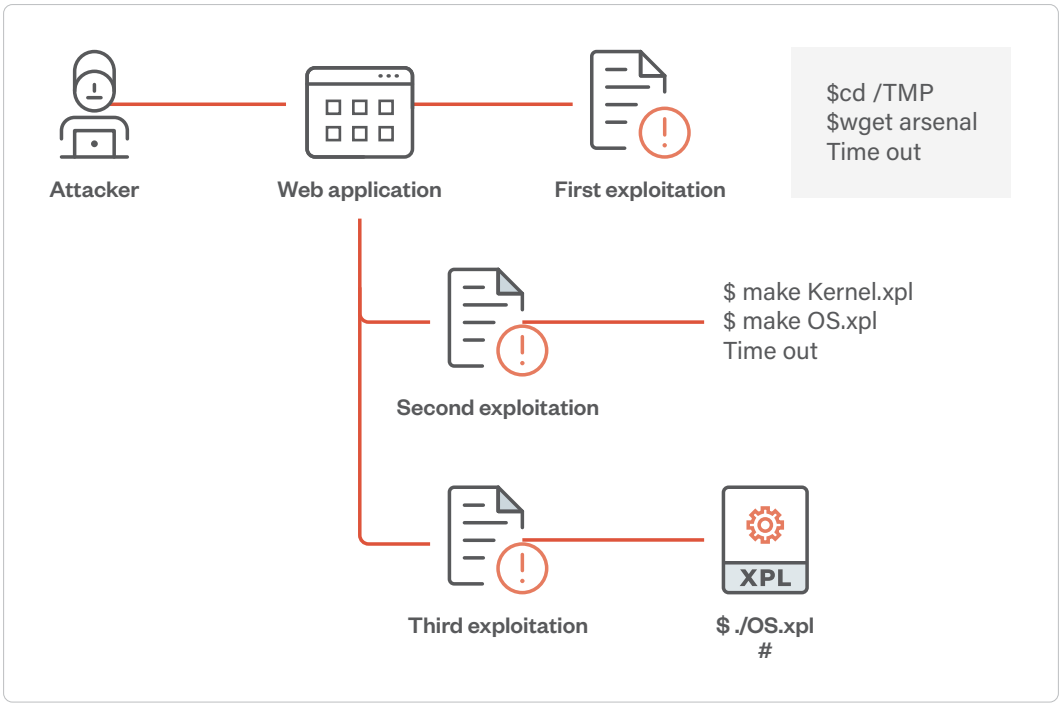


Figure 52. An example of an exploitation wherein an attacker breaks as a non-root user and downloads its arsenal under "/tmp"; the attacker then uses native building tools to compile exploits for further exploitation

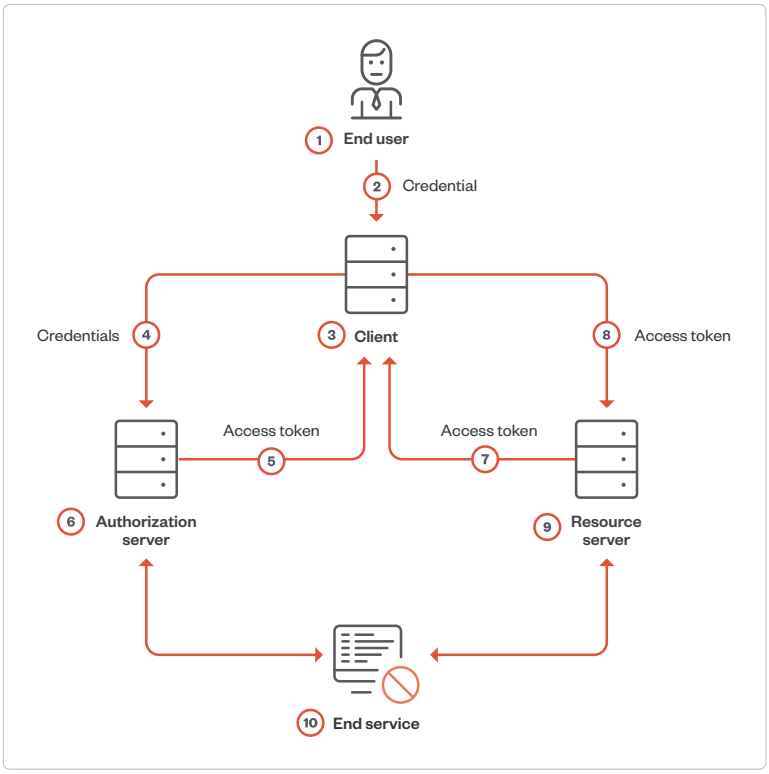
Token Management

CSPs commonly use tokens to provide a secure method of communication between the user and the server or between services. Token authentication basically works by making sure that every request to a service is accompanied by verification data that checks the authenticity of the request. Since tokens are not necessarily encrypted or hashed, the following are basic recommendations to keep their implementations secure:

- Keep tokens secure and protected.
- Rotate tokens.
- Use different tokens per application.

- Consider configuring token expiration.
- Do not use tokens out of context.

During our research, we noticed that tokens are used in two different phases of the serverless implementation. First, tokens are used on the user side. Most CSPs offer command-line interface (CLI) tools and develop application extensions[34] to manage all cloud services, including serverless ones. To start using the token in this serverless phase, the user needs to input credentials at first login or configuration. For the purpose of not having to provide it at every request, the client creates access tokens to authorize future requests.



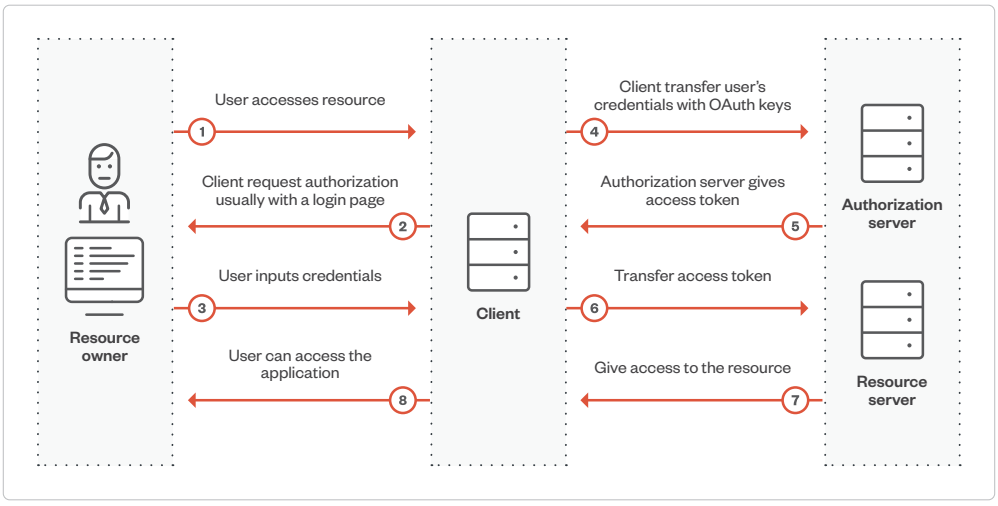


Figure 53. An ideal token management service

In some of these implementations, the tokens are stored in a plain-text format on environment variables that are meant to last for as long as the computer session lasts. The tokens can also be stored in text files with the same permissions as that of the user who initially logged in. Because tokens are easily accessible and can be used in a different context, they have become a target for cybercriminals who want to hijack cloud accounts.

Immutable Environment

Another effective security implementation – one with a high impact on threats but also a large impact on usability – is restricting changes within the environment. Once you build your system with a kernel, an operating system, and applications, nothing can be changed anymore. For a long time, having an immutable environment was very hard to achieve, because servers can have more than one purpose, or because the files inside the server cannot be static at all times and can change according to use.

Although it is hard to achieve a completely immutable system, users have plenty of options to harden their systems and achieve partial immutability on Unix-based operating systems to granulate access rights on determined directories and file systems. Based on our investigation of serverless implementations of different CSPs, users can achieve partial immutability by restricting certain mount points' writing access.

Given significant advancements in technology and assuming that containers and micro VMs have better restriction implementations, we can be more aggressive in our approach toward immutability.

The container can be configured to have read-only permissions on every mount point, and if a write permission is needed, it can be granularly given to a specific mount point, which in most cases is "/tmp," without giving it an execution permission.

During our exploration of serverless scenarios, much like ordinary infrastructures, after an attacker exploits the application and gains access to the file system, the attacker would also look for any native tool that would be available for further enumeration and exploitation. They would then download these native tools or their source code for compilation.

Given this scenario, an attacker who eventually breaks in and gets file system access would not be able to download any tool if the file system has read-only permissions. Even if they get lucky and succeed in downloading their arsenal, they would not be able to run it.

These restrictions are, again, neither new nor revolutionary and are a headache to implement. They have also been restricted to a very specific infrastructure scenario, such as embedded systems. In contrast, with containers and micro VMs, permissions can be implemented like any other security requirement.

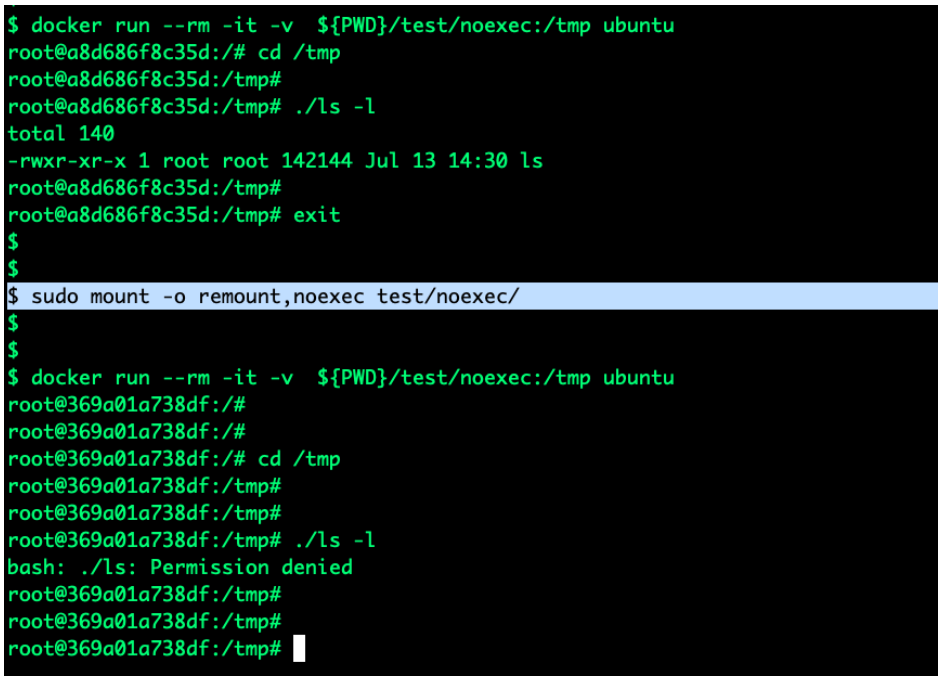


Figure 54. Comparing a container before and after removing execution permissions from a specific folder, which is a security best practice

Restricting Shell Access

In computing terms, a shell is essentially a program that takes an input from a user and forwards it to the operating system to process. Nowadays, even the most classic Unix systems have graphical interfaces. But for system administrators, the shell is still the main way to interact with the system. Not surprisingly for threat actors who target Unix systems, a shell has an important role in attacks. This is because although restricting access to it is vital for security, it can be hard to implement at times.

It has become more apparent that security implementations are becoming increasingly restrictive. Almost all the attack scenarios that we have described up to this point grant access to the file system. It is, after all, ideal for an attacker to have valid shell access to have the ability to map out the environment, use native or downloaded tools, move laterally to other services, or simply hijack sensitive data from inside the container.

Restricting the shell can also mitigate a security issue that is caused by a malicious script, as it would be applied to whatever is running inside the environment – and not only if an attacker gets access to it. Although it is not the ultimate, foolproof technique, it adds a layer of security and is easy to implement. The benefits also extend beyond a micro VM or a container environment to include any Unix system.

A restricted shell can be implemented as the default shell for any user (even specific ones) in case the environment has valid users. It should be noted, however, that this is not the best practice for serverless environments.

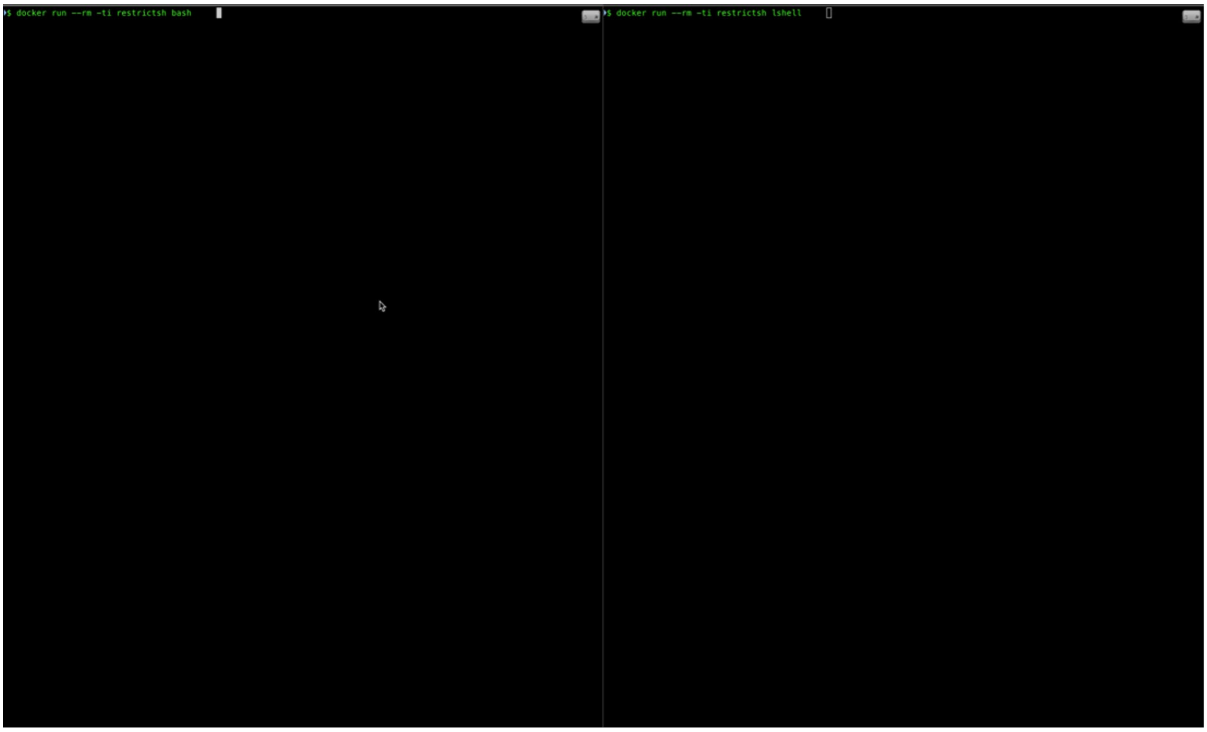


Figure 55. A video, a GIF, and a link for implementation of a restricted shell where only specific commands are allowed to run

Network Access

Given that not all of the implementations we have described so far can be implemented at once, attackers can abuse the network once they get access to the shell.

If the functions are configured in a way that they have internet access, attackers can abuse them to download and dump their own tools for further exploitation. Attackers will then map out the environment and obtain detailed information about the operating system, kernel, and running applications so that they can download exploits for specific scenarios using the outbound connection. In our investigation of different CSPs, we did not find any that block outbound traffic, nor did we find any that had an option for the user to disable outbound traffic in case their applications do not require it.

In this paper, we discussed a proof of concept in which we discovered that after the initial exploitation, we are able to download a native tool and use its own permissions to change the settings, as though it was administrative application.

We have seen a mix of internal network access implementations in our study. Some of them were very restrictive and secure, including containers that ran in an isolated network environment, which hindered attackers from mapping out other servers or services running on the same provider.

Others, for no apparent reason, implemented a full /24 network where it was possible for an attacker to run network mapping and attack tools from inside the application container.

```
Alibaba Serverless Reconnaissance
>ls -l /tmp
total 0


>id
uid=10001(user10001) gid=10001 groups=10001


>whoami
user10001


>ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc pfifo state UP group default qlen 1000
    link/ether ea:56:82:20:22:e1 brd ff:ff:ff:ff:ff:ff
    inet [redacted]248/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::e856:82ff:fe20:22e1/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 56. Serverless endpoint configuration with a single IP network

As some CSPs implementations show, it is possible to restrict internal network access even without external tools. Doing this would limit the attack surface. However, it should be noted that this option is either not available by default or is part of an upgraded plan.

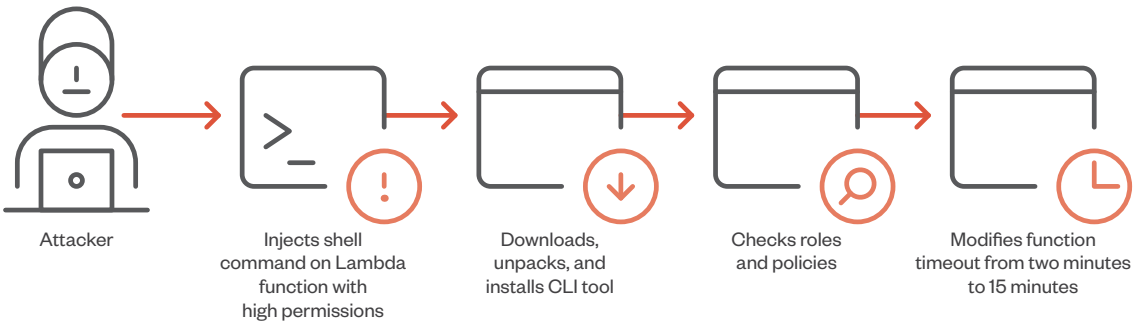


Figure 57. An AWS Lambda kill chain for a function with high permissions

Source: Trend Micro[35]

In 2020, we created a proof-of-concept video[36] that shows how poor coding practices and permissive access policies could give malicious actors the capability to alter the timeout of an AWS Lambda function.

Application Timeout

The application timeout, which is configurable for all CSPs, is a setting that users should always check, especially when they are concerned about monetary resources and security. Some CSPs use the running time of a function as a parameter to bill the customer. This means that if the running time is high, the billed amount would also be high. Though the impact of a high running time to the billed amount is not always direct, it is still part of the billing equation.

Security could serve as the extra push customers need to restrict their application's running time. The shorter the timeout, the smaller the attack surface, and this serves as another incentive for keeping the application distributed among focused and shorter functions.

The default application timeout is five minutes or less. This would depend on the CSP and on the purpose of the function. The user who will use the application and the language that will be used (since different languages need varying period lengths to load and run) also determine the application timeout.

Message from Microsoft

Microsoft Azure teams (MSRC, Azure App Service, Azure Functions) appreciate Trend Micro's efforts in looking into any possible security vulnerabilities and issues along with the education of customers about any possible threats/weaknesses.

While it is great that no unauthenticated exploit/vulnerability was identified during this research, being secure is always a moving target, and Microsoft Azure constantly evaluates best practices and how to factor them into future improvements to further harden the platform and help customers to build inherently more secure solutions easily with a "secure-by-default" mindset.

Microsoft provides security best practices for Microsoft Azure on their website.[37, 38]

Conclusion

Although the idea of going serverless shifts the responsibility over certain aspects to the CSP, giving users a feeling that their environment will be more secure, now is a good time to shine the spotlight on the security concerns associated with migrating to or starting a project using serverless services.

Security should never be neglected. Although the surface of attack is smaller in serverless deployments, it does not mean that threats only exist in the user's code, or worse, that threats no longer exist in these deployments. Planning to make the move to serverless requires ensuring the security of the application. Users need to determine how their applications will interact with the resources involved in their projects, and which data might be sensitive and should never be mishandled.

Two key factors of this threat modeling process were understanding how serverless services communicated with each other and how a user controls serverless resources. In our investigation, we considered it as a big red flag when both the user and the CSPs do not properly secure secrets and access tokens. Our team also found secrets stored in plain text inside files, or secrets that are loaded by the operating system via environment variables. This practice is not recommended for users and should be avoided. If the CSP does not provide a secure way for users to handle secrets, the user should still ensure that secrets are handled as securely as possible. When our team started investigating and shining a light on serverless security, attacks where threat actors can take advantage of unsecured tokens were just theoretical. Today, there are now multiple hacker teams that harvest CSP-specific secrets so that they can take over services and, in some cases, the whole account.[39]

There are different ways to use the cloud. Some CSP customers use the cloud in a way similar to how they would use a regular or a traditional data center: They would use workloads and at the same time, manage all the necessary infrastructure, such as server resources and networks. Meanwhile, other customers use the cloud for user application-level services, such as serverless computing, where the CSP takes care of the infrastructure. The leap from using cloud workloads to using serverless on cloud is quite challenging, as it requires users to adapt to a different manner of implementation for most aspects, including the way users deal with security. By its very nature and when properly implemented, serverless computing follows a more restrictive security model, or at the very least, it is easier to implement it in a more secure manner compared to other hard-to-apply security models.

Security features that once would have had a big impact on an application's usability, such as keeping a workload immutable and enabling safe memory cleaning, are default features of the serverless infrastructure. This makes serverless adoption more secure.

Still, it is important to understand that there is still a great need to strike a balance between security and usability. It is our hope that this study has exposed some of the blind spots and the seldomly discussed security issues of serverless services hosted on Microsoft Azure to help users make well-informed security decisions.

Security Recommendations

- **Follow CSPs' recommendations.** CSP recommendations for securing environments and projects are usually found in their respective documentations.
- **Use vaults to store keys and passwords.** This might incur additional costs to the team or organization, but it gives users and security teams an additional layer of protection for their credentials' storage.
- **Use custom images.** While default services allow for speed and efficiency for deployment and development, custom container image designs and implementations give developers more room for out-of-the-box solutions and additional security.
- **Use encrypted channels and pipelines.** Locking the values of the variables ensures that sensitive information, such as passwords and IDs, remains secret in instances of unauthorized access.
- **Achieve compliance with the Assume Breach paradigm.** Under this paradigm, users know that vulnerabilities exist. In the instance of compromise and given web vulnerabilities' prevalence in today's attacks,[40] the impact of the infiltration from the abuse of an exploit is minimized.
- **Follow the principle of least privilege.** This can be achieved by using a non-privileged user for your container and applications, using managed identities and roles, and limiting public endpoints of linked cloud services. Also, consider using safer mechanisms for generating and managing secrets such as passwords and API keys.
- **Audit and secure all employed out-of-the-box solutions.** This can be done by performing third-party reviews and following vendors' best practices for security.

Endnotes

1 Taylor Brown. (April 2017). *Microsoft*. "Bringing Docker To Windows Developers with Windows Server Containers." Accessed on Nov. 23, 2022, at: Link.

2 Trend Micro. (n.d.). *Trend Micro Security News*. "Zero Trust." Accessed on Nov. 23, 2022, at: Link.

3 Netsurion. (n.d.). *Netsurion*. "The Assume Breach Paradigm." Accessed on Nov. 23, 2022, at: Link.

4 Microsoft. (n.d.) *Microsoft*. "App Service." Accessed on Nov. 24, 2022, at: Link.

5 David Fiser. (May 4, 2022). *Trend Micro Security News*. "Crafting an Azure App Services Threat Model." Accessed on Dec. 21, 2022, at: Link.

6 Docker Docs. (n.d.). Docker Docs. "Isolate containers with a user namespace." Accessed on March 14, 2023, at: Link.

7 Microsoft. (Dec. 7, 2022). *Microsoft*. "Migrate custom software to Azure App Service using a custom container." Accessed on Dec. 21, 2022, at: Link.

8 PyPi. (n.d.). *PyPi*. "webssh 1.6.1." Accessed on Nov. 24, 2022, at: Link.

9 David Fiser and Alfredo Oliveira. (Dec. 17, 2019). *Trend Micro Research, News, and Perspectives*. "Why A Privileged Container in Docker Is a Bad Idea."Accessed on Nov. 24, 2022, at: Link.

10 Linux. (Aug. 27, 2021). *man7.org*. "capabilities(7) – Linux manual page." Accessed on Nov. 24, 2022, at: Link.

11 David Fiser. (May 4, 2022). *Trend Micro Security News*. "Crafting an Azure App Services Threat Model." Accessed on Dec. 21, 2022, at: Link.

12 Microsoft. (Sep. 16, 2022). *Microsoft*. "App Service networking features." Accessed on Nov. 25, 2022, at: Link.

13 Trend Micro. (May 13, 2017). *Trend Micro Security News*. "WannaCry/Wcry Ransomware: How to Defend against It." Accessed on Dec. 21, 2022, at: Link.

14 Microsoft. (June 3, 2021). *Microsoft*. "Azure Key Vault basic concepts." Accessed on Nov. 27, 2022, at: Link.

15 projectkudu. (Aug. 22, 2022). *GitHub*. "Azure Web App sandbox." Accessed on Nov. 27, 2022, at: Link.

16 Microsoft. (July 15, 2022). *Microsoft*. "Storage account overview." Accessed on Dec. 5, 2022, at: Link.

17 Microsoft. (June 15, 2022). *Microsoft*. "API Management authentication policies." Accessed on Dec. 5, 2022, at: Link.

18 Microsoft. (Oct. 13, 2022). *Microsoft*. "About Azure Key Vault." Accessed on Dec. 5, 2022, at: Link.

19 Microsoft. (July 15, 2022). *Microsoft*. "Overview of Azure Service Fabric." Accessed on Dec. 5, 2022, at: Link.

20 Microsoft Azure. (Dec. 3, 2022). *GitHub*. "azure-functions-host." Accessed on Dec. 5, 2022, at: Link.

21 Microsoft. (May 6, 2022). *Microsoft*. Accessed on Dec. 5, 2022, at: Link.

22 Microsoft. (n.d.). *Microsoft*. "Azure Storage Explorer." Accessed on Dec. 6, 2022, at: Link.

23 Microsoft. (Oct. 6, 2022). *Microsoft*. "Create a function on Linux using a custom container." Accessed on Dec. 12, 2022, at: Link.

24 Microsoft Artifact Registry. (Dec. 12, 2022). *Microsoft Artifact Registry*. "Azure Functions Base." Accessed on Dec. 12, 2022, at: Link.

25 Microsoft Artifact Registry. (n.d.). *Microsoft Artifact Registry*. "Microsoft Artifact Registry." Accessed on Dec. 12, 2022, at: Link.

26 Alfredo Oliveira and Raphael Bottino. (Sept. 7, 2022). *Trend Micro*. "Enhancing Cloud Security by Reducing Container Images Through Distroless Techniques." Accessed on Jan. 20, 2023, at: Link.

27 Die.net. (n.d.). *Die.net*. "execve(3) - Linux man page." Accessed on Dec. 14, 2022, at: Link.

28 Cplusplus.com. (n.d.). *Cplusplus.com*. "getenv." Accessed on Dec. 14, 2022, at: Link.

29 Wesley Chai. (June 2022). *TechTarget*. "confidentiality, integrity and availability (CIA triad)." Accessed on Dec. 14, 2022, at: Link.

30 Man7.org. (n.d.). *Man7.org*. "ip(8) — Linux manual page." Accessed on Dec. 14, 2022, at: Link.

31 Man7.org. (n.d.). *Man7.org*. "ifconfig(8) — Linux manual page." Accessed on Dec. 14, 2022, at: Link.

32 Man7.org. (n.d.). *Man7.org*. "netstat(8) — Linux manual page." Accessed on Dec. 14, 2022, at: Link.

33 Man7.org. (n.d.). *Man7.org*. "ss(8) — Linux manual page." Accessed on Dec. 14, 2022, at: Link.

34 David Fiser. (March 4, 2020). *Trend Micro Research, News, and Perspectives*. "Security Risks in Online Coding Platforms." Accessed on Dec. 14, 2022, at: Link.

35 Alfredo Oliveira. (Aug. 11, 2020). *Trend Micro*. "Securing Weak Points in Serverless Architectures: Risks and Recommendations." Accessed on Dec. 21, 2022, at: Link.

36 Trend Micro. (Aug. 11, 2020). *Trend Micro*. "Weak Points in Serverless Architecture: A Proof of Concept." Accessed on Jan. 20, 2023, at: Link.

37 Microsoft. (March 4, 2023). *Microsoft*. "Security recommendations for App Service." Accessed on March 14, 2023, at: Link.

38 Microsoft. (Dec. 16, 2022). *Microsoft*. "Securing Azure Functions." Accessed on March 14, 2023, at: Link.

39 David Fiser and Alfredo Oliveira. (Aug. 17, 2022). *Trend Micro Research, News, and Perspectives*. "Analyzing the Hidden Danger of Environment Variables for Keeping Secrets." Accessed on Dec. 21, 2022, at: Link.

40 Magno Logan and Pawan Kinger. (Aug. 23, 2021). *Trend Micro Security News*. "Linux Threat Report 2021 1H: Linux Threats in the Lcoud and Security Recommendations." Accessed on Dec. 15, 2022, at: Link.

For more information visit **trendmicro.com**