**WISE-PaaS**

**TECHNICAL WHITEPAPER**

**WISE-PaaS**

**Platform Security**



Table of Contents

[1 Introduction 2](#_Toc497235094)

[2 Architecture Overview 2](#_Toc497235095)

[3 Azure IaaS Cloud Security 2](#_Toc497235096)

[3.1 Help protect your virtual machines from viruses and malware 2](#_Toc497235097)

[3.2 Help secure sensitive data on your virtual machines 2](#_Toc497235098)

[3.3 Encrypt your Linux and Windows virtual machine disks 2](#_Toc497235099)

[3.4 Build more compliant solutions 3](#_Toc497235100)

[4 Cloud Foundry PaaS Security 3](#_Toc497235101)

[4.1 The Industry-Standard Cloud Platform 3](#_Toc497235102)

[4.2 Understanding Cloud Foundry Security 4](#_Toc497235103)

[4.3 Understanding Container Security 10](#_Toc497235104)

[4.4 Understanding Application Security Groups 12](#_Toc497235105)

[4.5 Securing Traffic into Cloud Foundry 14](#_Toc497235106)

[5 WISE-PaaS Security 14](#_Toc497235107)

[5.1 HTTPS Security 14](#_Toc497235108)

[5.2 Database Security 14](#_Toc497235109)

[5.2.1 PostgreSQL 15](#_Toc497235110)

[5.2.2 MongoDB 21](#_Toc497235111)

[5.3 Device Connectivity Security 25](#_Toc497235112)

[6 Referenced 25](#_Toc497235113)

# Introduction

# Architecture Overview

# Azure IaaS Cloud Security

## Help protect your virtual machines from viruses and malware

Use antimalware software from major security vendors such as Microsoft, Symantec, Trend Micro, McAfee, and Kaspersky to help protect your virtual machines from malicious files, adware, and other threats. Remotely install, configure, and maintain antimalware solutions on your virtual machines through the Azure portal, Azure PowerShell, and from the command line.

## Help secure sensitive data on your virtual machines

Your data is critical to your business—and to us. So we monitor it 24/7, and build datacenters designed to shelter your data and services from unauthorized access. For extra protection, we also offer industry-leading encryption solutions from CloudLink and Trend Micro for your virtual machines and all of the data on them.

## Encrypt your Linux and Windows virtual machine disks

Azure Disk Encryption helps you address organizational security and compliance requirements, by encrypting your virtual machine disks with keys and policies that you control in Azure Key Vault. Azure Disk Encryption enables you to encrypt your virtual machine disks, including the boot and the data disks. The solution works for Linux and Windows operating systems, and it uses Key Vault to help you safeguard your disk encryption keys, manage key access policies, and audit use of your keys. All of the data in the virtual machine disks are encrypted at rest, using industry-standard encryption technology in your Azure Storage accounts. The Azure Disk Encryption solution for Windows is based on proven Microsoft **BitLocker** Drive Encryption, and the Linux solution is based on **dm-crypt**.

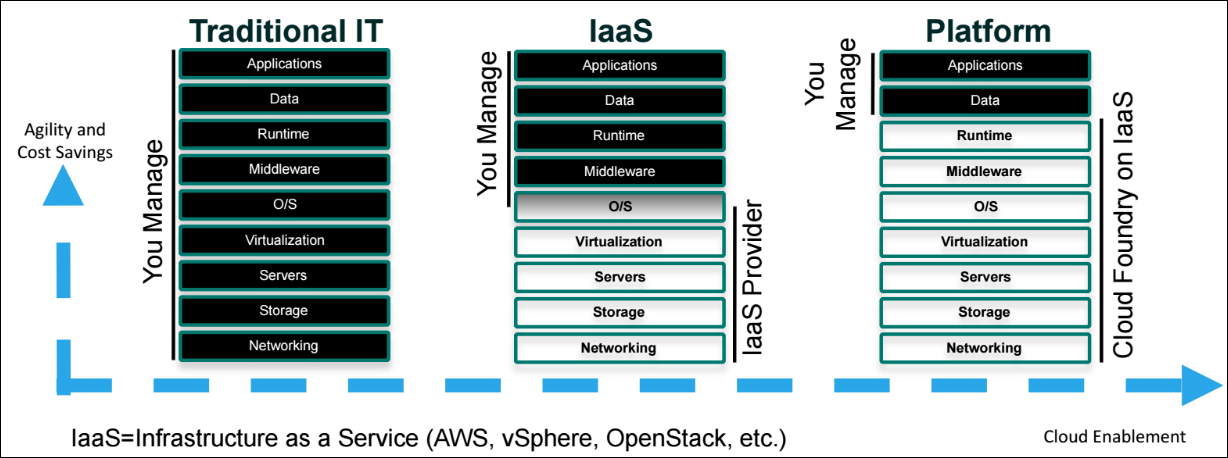
## Build more compliant solutions

Azure Virtual Machines is certified for the Federal Information Security Modernization Act (FISMA), Federal Risk and Authorization Management Program (FedRAMP), Health Insurance Portability and Accountability Act (HIPAA), Payment Card Industry Data Security Standard (PCI DSS) Level 1, and other key compliance programs—which makes it easier for your Azure applications to meet compliance requirements, and for your business to address a wide range of domestic and international regulatory requirements.

# Cloud Foundry PaaS Security

## The Industry-Standard Cloud Platform

Cloud platforms let anyone deploy network apps or services and make them available to the world in a few minutes. When an app becomes popular, the cloud easily scales it to handle more traffic, replacing with a few keystrokes the build-out and migration efforts that once took months. Cloud platforms represent the next step in the evolution of IT, enabling you to focus exclusively on your applications and data without worrying about underlying infrastructure.



Not all cloud platforms are created equal. Some have limited language and framework support, lack key app services, or restrict deployment to a single cloud. Cloud Foundry (CF) has become the industry standard. It is an open source platform that you can deploy to run your apps on your own computing infrastructure, or deploy on an IaaS like Azure, AWS, vSphere, or OpenStack. You can also use a PaaS deployed by a commercial CF cloud provider. A broad community contributes to and supports Cloud Foundry. The platform’s openness and extensibility prevent its users from being locked into a single framework, set of app services, or cloud.

Cloud Foundry is ideal for anyone interested in removing the cost and complexity of configuring infrastructure for their apps. Developers can deploy their apps to Cloud Foundry using their existing tools and with zero modification to their code.

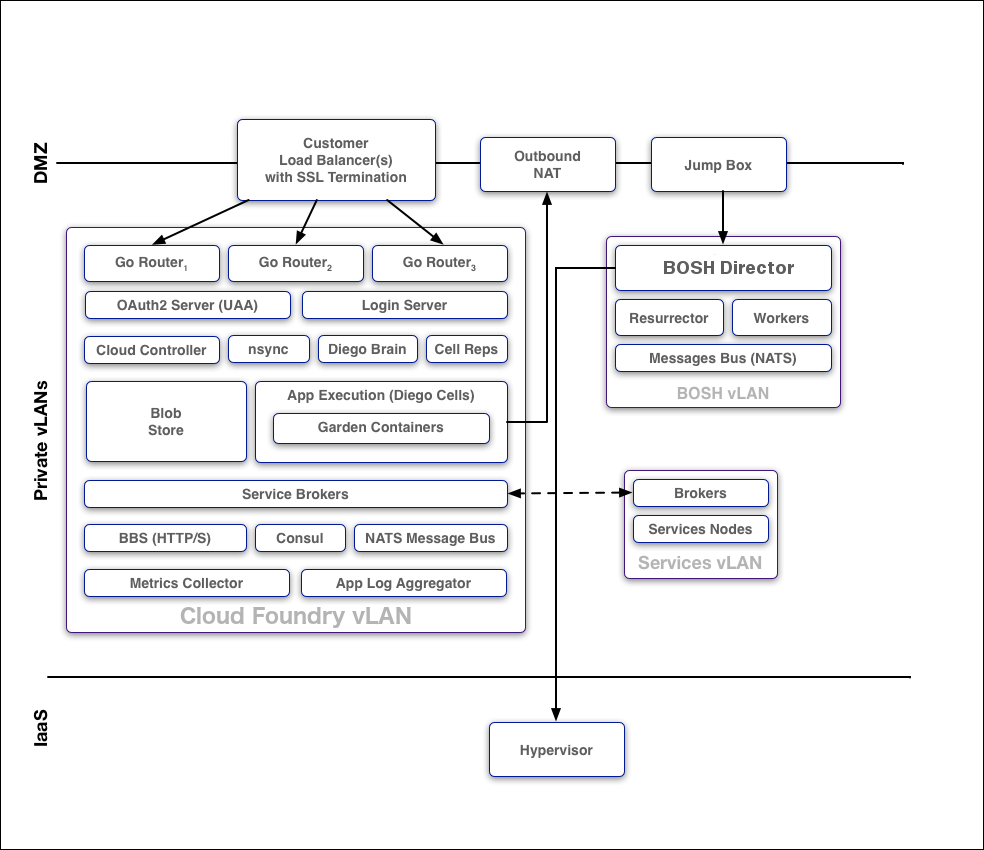
## Understanding Cloud Foundry Security

Cloud Foundry implements the following measures to mitigate against security threats:

* Minimizes network surface area
* Isolates customer applications and data in containers
* Encrypts connections
* Uses role-based access controls, applying and enforcing roles and permissions to ensure that users can only view and affect the spaces for which they have been granted access
* Ensures security of application bits in a multi-tenant environment
* Prevents possible denial of service attacks through resource starvation

System Boundaries and Access

As the image below shows, in a typical deployment of Cloud Foundry, the components run on virtual machines (VMs) that exist within a VLAN. In this configuration, the only access points visible on a public network are a load balancer that maps to one or more Cloud Foundry routers and, optionally, a NAT VM and a jumpbox. Because of the limited number of contact points with the public internet, the surface area for possible security vulnerabilities is minimized.



Protocols

All traffic from the public internet to the Cloud Controller and UAA happens over HTTPS. Inside the boundary of the system, components communicate over a publish-subscribe (pub-sub) message bus [NATS](http://nats.io/), HTTP, and SSL/TLS.

BOSH

Operators deploy Cloud Foundry with BOSH. The BOSH Director is the core orchestrating component in BOSH: it controls VM creation and deployment, as well as other software and service lifecycle events. You use HTTPS to ensure secure communication to the BOSH Director.

BOSH includes the following functionality for security:

* Communicates with the VMs it launches over NATS. Because NATS cannot be accessed from outside Cloud Foundry, this ensures that published messages can only originate from a component within your deployment.
* Provides an audit trail through the bosh tasks command. This command shows all actions that an operator has taken with BOSH.
* Allows you to set up individual login accounts for each operator. BOSH operators have root access.

Isolation Segments

Isolation segments provide dedicated pools of resources to which apps can be deployed to isolate workloads. Using isolation segments separates app resources as completely as if they were in different CF deployments but avoids redundant management components and unneeded network complexity.

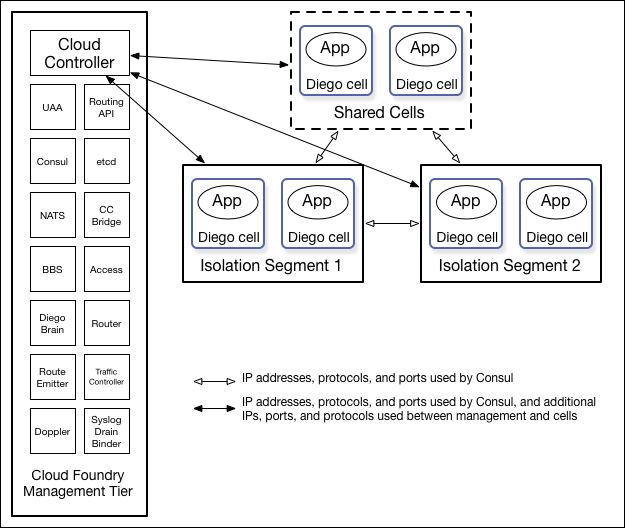
You can designate isolation segments for exclusive use by [orgs and spaces](https://docs.cloudfoundry.org/concepts/roles.html) within CF. This guarantees that apps within the org or space use resources that are not also used by other orgs or spaces.

Customers can use isolation segments for different reasons, including the following:

* To follow regulatory restrictions that require separation between different types of applications. For example, a health care company may not be able to host medical records and billing systems on the same machines.
* To dedicate specific hardware to different isolation segments. For example, to guarantee that high-priority apps run on a cluster of high-performance hosts.
* To separate data on multiple clients, to strengthen a security story, or offer different hosting tiers.

In CF, the Cloud Controller Database (CCDB) identifies isolation segments by name and GUID, for example 30dd879c-ee2f-11db-8314-0800200c9a66. The isolation segment object has no internal structure beyond these two properties at the Cloud Foundry level, but BOSH associates the name of the isolation segment with Diego cells, through their placement\_tag property.

This diagram shows how isolation segments keep apps running on different pools of cells, and how the cells communicate with each other and with the management components:



Authentication and Authorization

[User Account and Authentication](https://docs.cloudfoundry.org/concepts/architecture/uaa.html) (UAA) is the central identity management service for Cloud Foundry and its various components.

UAA acts as an [OAuth2](https://oauth.io/) Authorization Server and issues access tokens for applications that request platform resources. The tokens are based on the [JSON Web Token](http://jwt.io/) and are digitally signed by UAA.

Operators can configure the identity store in UAA. If users register an account with the Cloud Foundry platform, UAA acts as the user store and stores user passwords in the UAA database using [bcrypt](http://en.wikipedia.org/wiki/Bcrypt). UAA also supports connecting to external user stores through LDAP and SAML. Once an operator has configured the external user store, such as a corporate Microsoft Active Directory, users can use their LDAP credentials to gain access to the Cloud Foundry platform instead of registering a separate account. Alternatively, operators can use SAML to connect to an external user store and enable single sign-on for users into the Cloud Foundry platform.

Managing User Access with Role-Based Access Control

Applications that users deploy to Cloud Foundry exist within a space. Spaces exist within orgs. To view and access an org or a space, a user must be a member of it. Cloud Foundry uses role-based access control (RBAC), with each role granted permissions to either an org or a specified space. For more information about roles and permissions, refer to the [Orgs, Spaces, Roles, and Permissions](https://docs.cloudfoundry.org/concepts/roles.html)topic.

Security for Service Broker Integration

The Cloud Controller authenticates every request with the Service Broker API using HTTP or HTTPS, depending on which protocol that you specify during broker registration. The Cloud Controller rejects any broker registration that does not contain a username and password.

Service instances bound to an app contain credential data. Users specify the binding credentials for [user-provided service instances](https://docs.cloudfoundry.org/devguide/services/user-provided.html), while third-party brokers specify the binding credentials for managed service instances. The VCAP\_SERVICES environment variable contains credential information for any service bound to an app. Cloud Foundry constructs this value from encrypted data that it stores in the Cloud Controller Database (CCDB).

A third-party broker might offer a dashboard client in its catalog. Dashboard clients require a text string defined as a client\_secret. Cloud Foundry does not store this secret in the CCDB. Instead, Cloud Foundry passes the secret to the UAA component for verification using HTTP or HTTPS.

Software Vulnerability Management

Cloud Foundry manages software vulnerability using releases and BOSH stemcells. New Cloud Foundry releases are created with updates to address code issues, while new stemcells are created with patches for the latest security fixes to address any underlying operating system issues.

Ensuring Security for Application Artifacts

Cloud Foundry secures both the code and the configuration of an application using the following functionality:

* Application developers push their code using the [Cloud Foundry API](http://apidocs.cloudfoundry.org/). Cloud Foundry secures each call to the CF API using the [UAA](https://docs.cloudfoundry.org/concepts/security.html#auth) and SSL.
* The Cloud Controller uses [RBAC](https://docs.cloudfoundry.org/concepts/security.html#rbac) to ensure that only authorized users can access a particular application.
* The Cloud Controller stores the configuration for an application in an encrypted database table. This configuration data includes user-specified environment variables and service credentials for any services bound to the app.
* Cloud Foundry runs the app inside a secure container. For more information, see the [Understanding Container Security](https://docs.cloudfoundry.org/concepts/container-security.html) topic.
* Cloud Foundry operators can configure network traffic rules to control inbound communication to and outbound communication from an app. For more information, see the [Network Traffic Rules](https://docs.cloudfoundry.org/concepts/container-security.html#config-traffic) section of the [Understanding Container Security](https://docs.cloudfoundry.org/concepts/container-security.html) topic.

Security Event Logging and Auditing

For operators, Cloud Foundry provides an audit trail through the bosh tasks command. This command shows all actions that an operator has taken with the platform. Additionally, operators can redirect Cloud Foundry component logs to a standard syslog server using the syslog\_daemon\_config [property](http://docs.cloudfoundry.org/running/managing-cf/logging.html) in the metron\_agent job of cf-release.

For users, Cloud Foundry records an audit trail of all relevant API invocations of an app. The Cloud Foundry Command Line Interface (cf CLI) command cf events returns this information.

Recommendations for Running a Secure Deployment

To help run a secure deployment, Cloud Foundry recommends the following:

* Configure UAA clients and users using a BOSH manifest. Limit and manage these clients and users as you would any other kind of privileged account.
* Deploy within a VLAN that limits network traffic to individual VMs. This reduce the possibility of unauthorized access to the VMs within your BOSH-managed cloud.
* Enable HTTPS for applications and SSL database connections to protect sensitive data transmitted to and from applications.
* Ensure that the Jumpbox is secure, along with the load balancer and NAT VM.
* Encrypt stored files and data within databases to meet your data security requirements. Deploy using industry standard encryption and the best practices for your language or framework.
* Prohibit promiscuous network interfaces on the trusted network.
* Review and monitor data sharing and security practices with third-party services that you use to provide additional functionality to your application.
* Store SSH keys securely to prevent disclosure, and promptly replace lost or compromised keys.
* Use Cloud Foundry’s RBAC model to restrict your users’ access to only what is necessary to complete their tasks.
* Use a strong passphrase for both your Cloud Foundry user account and SSH keys.

## Understanding Container Security

This topic describes how Cloud Foundry (CF) secures the containers that host application instances on Linux.

* [Container Mechanics](https://docs.cloudfoundry.org/concepts/container-security.html#mechanics) provides an overview of container isolation.
* [~~Inbound and Outbound Traffic from CF~~](https://docs.cloudfoundry.org/concepts/container-security.html#networking)~~provides an~~[~~overview~~](https://docs.cloudfoundry.org/concepts/container-security.html#net-overview)~~of container networking and describes how CF administrators~~[~~customize~~](https://docs.cloudfoundry.org/concepts/container-security.html#config-traffic)~~container network traffic rules for their deployment.~~
* [Container Security](https://docs.cloudfoundry.org/concepts/container-security.html#security) describes how CF secures containers by running application instances in [unprivileged](https://docs.cloudfoundry.org/concepts/container-security.html#types) containers and by [hardening](https://docs.cloudfoundry.org/concepts/container-security.html#hardening) them.

Container Mechanics

Each instance of an app deployed to CF runs within its own self-contained environment, a [Garden container](https://docs.cloudfoundry.org/concepts/architecture/garden.html). This container isolates processes, memory, and the filesystem using operating system features and the characteristics of the virtual and physical infrastructure where CF is deployed.

CF achieves container isolation by name spacing kernel resources that would otherwise be shared. The intended level of isolation is set to prevent multiple containers that are present on the same host from detecting each other. Every container includes a private root filesystem, which includes a Process ID (PID), namespace, network namespace, and mount namespace.

CF creates container filesystems using the [Garden Rootfs](https://docs.cloudfoundry.org/concepts/architecture/garden.html#garden-rootfs) (GrootFS) tool. It stacks the following using OverlayFS:

* A **read-only base filesystem**: This filesystem has the minimal set of operating system packages and Garden-specific modifications common to all containers. Containers can share the same read-only base filesystem because all writes are applied to the read-write layer.
* A **container-specific read-write layer**: This layer is unique to each container and its size is limited by XFS project quotas. The quotas prevent the read-write layer from overflowing into unallocated space.

Resource control is managed using Linux control groups ([cgroups](https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt)). Associating each container with its own cgroup or job object limits the amount of memory that the container may use. Linux cgroups also require the container to use a fair share of CPU compared to the relative CPU share of other containers.

Container Security

CF secures containers through the following measures:

* Running application instances in [unprivileged](https://docs.cloudfoundry.org/concepts/container-security.html#types) containers by default
* [Hardening](https://docs.cloudfoundry.org/concepts/container-security.html#hardening) containers by limiting functionality and access rights
* Only allowing outbound connections to public addresses from application containers. This is the original default. Administrators can change this behavior by configuring [ASGs](#_Understanding_Application_Security).

Types

Garden has two container types: unprivileged and privileged. Currently, CF runs all application instances and staging tasks in unprivileged containers by default. This measure increases security by eliminating the threat of root escalation inside the container.

Formerly, CF ran apps based on Docker images in unprivileged containers, and buildpack-based apps and staging tasks in privileged containers. CF ran apps based on Docker images in unprivileged containers because Docker images come with their own root filesystem and user, so CF could not trust the root filesystem and could not assume that the container user process would never be root. CF ran build-pack based apps and staging tasks in privileged containers because they used the cflinuxfs2 root filesystem and all processes were run as the unprivileged user vcap.

Although all application instances and staging tasks now run by default in unprivileged containers, operators can override these defaults by customizing their Diego deployment manifest and redeploying.

## Understanding Application Security Groups

* Application Security Groups (ASGs) are a collections of egress rules that specify the protocols, ports, and IP address ranges where app or task instances send traffic. Because ASGs define **allow** rules, their order of evaluation is unimportant when multiple ASGs apply to the same space or deployment. The platform sets up rules to filter and log outbound network traffic from app and task instances. ASGs apply to both buildpack-based and Docker-based apps and tasks.
* When apps or tasks begin staging, they need traffic rules permissive enough to allow them to pull resources from the network. After an app or task is running, the traffic rules can be more restrictive and secure. To distinguish between these two security requirements, administrators can define one ASG for app and task staging, and another for app and task runtime.
* To provide granular control when securing a deployment, an administrator can assign ASGs to apply to all app and task instances for the entire deployment, or assign ASGs to spaces to apply only to apps and tasks in a particular space.
* ASGs can be complicated to configure correctly, especially when the specific IP addresses listed in a group change. To simplify securing a deployment while still permitting apps reach external services, operators can deploy the services into a subnet that is separate from their Cloud Foundry deployment. Then the operators can create ASGs for the apps that whitelist those service subnets, while denying access to any virtual machine (VM) hosting other apps.

ASG Sets

ASGs are applied by configuring ASG sets differentiated by *scope*, platform-wide or space specific, and *lifecycle*, staging or running.

Currently, four ASG sets exist in Cloud Foundry:

* Platform-wide staging ASG set, also called “default-staging”
* Platform-wide running ASG set, also called “default-running”
* Space-scoped staging ASG set
* Space-scoped running ASG set

The following table indicates the differences between the four sets.

|  |  |
| --- | --- |
| When an ASG is bound to the… | the ASG rules are applied to… |
| Platform-wide staging ASG set | the staging lifecycle for all apps and tasks. |
| Platform-wide running ASG set | the running lifecycle for all app and task instances. |
| Space-scoped staging ASG set | the staging lifecycle for apps and tasks in a particular space. |
| Space-scoped running ASG set | the running lifecycle for app and task instances in a particular space. |

Typically, ASGs applied during the staging lifecycle are more permissive than the ASGs applied during the running lifecycle. This is because staging often requires access to different resources, such as dependencies.

The Structure and Attributes of ASGs

ASG rules are specified as a JSON array of ASG objects. An ASG object has the following attributes:

|  |  |  |
| --- | --- | --- |
| Attribute | Description | Notes |
| protocol | tcp, udp, icmp, or all | Required |
| destination | A single IP address, an IP address range like 192.0.2.0-192.0.2.50, or a CIDR block that can receive traffic |  |
| ports | A single port, multiple comma-separated ports, or a single range of ports that can receive traffic. Examples: 443, 80,8080,8081, 8080-8081 | Required when protocol is tcp or udp |
| code | ICMP code | Required when protocol is icmp. A value of -1 allows all codes. |
| type | ICMP type | Required when protocol is icmp. A value of -1 allows all types. |
| log | Set to true to enable logging. For more information about how to configure system logs to be sent to a syslog drain, see the [Using Log Management Services](https://docs.cloudfoundry.org/devguide/services/log-management.html)topic. | Logging is only supported with protocol type tcp |
| description | An optional text field for operators managing security group rules |  |

About the ASG Creator Tool

The ASG Creator is a command line tool that you can use to create JSON rules files. The ASG Creator lets you specify IP addresses, CIDRs, and IP address ranges that you want to disallow traffic to, as well as the addresses that you want to allow traffic to. Based on these disallow/allow (exclude/include) lists that you provide as input, the ASG Creator formulates a JSON file of allow rules.

In turn, the JSON file is the input for the cf create-security-group command that creates an ASG.

You can download the latest release of the ASG Creator from the Cloud Foundry incubator repository on Github: <https://github.com/cloudfoundry-incubator/asg-creator/releases/latest>

## Securing Traffic into Cloud Foundry

Protocol Support

* Gorouter supports HTTP/HTTPS requests only. For more information about features supported by the Gorouter.
* To secure non-HTTP traffic over TCP Routing, terminate TLS at your load balancer or at the application.

# WISE-PaaS Security

## HTTPS Security

## Database Security

Based on 2017 database ranking, the PostgreSQL/MongoDB is Top 4 and 5. The WISE-PaaS offers relational and non-structure database for developer to process their raw data. The “Redis” (In-Memory) provide high performance with real-time. As below we would introduce how the security we adopted and enabled by database connection/authentication.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Rank** | | | **DBMS** | **Database Model** | **Score** | | |
| **Oct 2017** | **Sep 2017** | **Oct 2016** | **Oct 2017** | **Sep 2017** | **Oct 2016** |
| 1. | 1. | 1. | [Oracle detailed information](https://db-engines.com/en/system/Oracle) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 1348.80 | -10.29 | -68.30 |
| 2. | 2. | 2. | [MySQL detailed information](https://db-engines.com/en/system/MySQL) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 1298.83 | -13.78 | -63.82 |
| 3. | 3. | 3. | [Microsoft SQL Server detailed information](https://db-engines.com/en/system/Microsoft+SQL+Server) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 1210.32 | -2.23 | -3.86 |
| 4. | 4. | up arrow 5. | [PostgreSQL detailed information](https://db-engines.com/en/system/PostgreSQL) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 373.27 | +0.91 | +54.58 |
| 5. | 5. | down arrow 4. | [MongoDB detailed information](https://db-engines.com/en/system/MongoDB) | [Document store](https://db-engines.com/en/article/Document+Stores) | 329.40 | -3.33 | +10.60 |
| 6. | 6. | 6. | [DB2 detailed information](https://db-engines.com/en/system/DB2) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 194.59 | -3.75 | +14.03 |
| 7. | 7. | up arrow 8. | [Microsoft Access](https://db-engines.com/en/system/Microsoft+Access) | [Relational DBMS](https://db-engines.com/en/article/RDBMS) | 129.45 | +0.64 | +4.78 |
| 8. | 8. | down arrow 7. | [Cassandra detailed information](https://db-engines.com/en/system/Cassandra) | [Wide column store](https://db-engines.com/en/article/Wide+Column+Stores) | 124.79 | -1.41 | -10.27 |
| 9. | 9. | 9. | [Redis detailed information](https://db-engines.com/en/system/Redis) | [Key-value store](https://db-engines.com/en/article/Key-value+Stores) | 122.05 | +1.65 | +12.51 |
| 10. | 10. | up arrow 11. | [Elasticsearch detailed information](https://db-engines.com/en/system/Elasticsearch) | [Search engine](https://db-engines.com/en/article/Search+Engines) | 120.23 | +0.23 | +21.12 |

## PostgreSQL

Access to PostgreSQL is controllable via host-based access rules. Authentication is flexible and pluggable, allowing easy integration with any external security architecture. Full SSL-encrypted access is supported natively. A full-featured cryptographic function library is available for database users. PostgreSQL provides role-based access privileges to access data, by command type. PostgreSQL also provides Row Level Security for privacy, medical and military grade security

Functions may execute with the permissions of the definer, while views may be defined with security barriers to ensure that security is enforced ahead of other processing. All aspects of PostgreSQL are assessed by an active security team, while known exploits are categorized and reported at <http://www.postgresql.org/support/security/>.

The WISE-PaaS follows the principles of PostgreSQL Cook Book to enhance security.

Revoking user access to a table

*This recipe answers the question, How do I make sure that user X cannot access table Y?*

The current user must either be a superuser, the owner of the table, or a user with a GRANT option for the table.

Granting user access to a table

*A user needs to have access to a table in order to perform any action on it.*

Make sure that you have the appropriate roles defined, and that privileges are revoked from the PUBLIC role.

Granting user access to specific rows

*Recent PostgreSQL versions support granting users privileges on some rows only.*

This recipe uses row-level security (RLS), which is available only on PostgreSQL version 9.5 or later, so start by checking that you are not using an older version. As for the previous recipe, we assume that there is already a schema called someschema and a role called somerole with USAGE privileges on it. We create a new table to experiment with row-level privileges:

*CREATE TABLE someschema.sometable3(col1 int, col2 text);*

Row-level security must also be enabled on that table:

*ALTER TABLE someschema.sometable3 ENABLE ROW LEVEL SECURITY;*

Creating a new user

*In this recipe, we will show you two ways of creating a new database user, one with a dedicated command-line utility, and one using SQL commands.*

To create new users, you must either be a superuser or have the CREATEROLE or CREATEROLE privilege

Temporarily preventing a user from connecting

*Sometimes, you need to temporarily revoke a user's connection rights without actually deleting the user or changing the user's password. This recipe presents the ways of doing this.*

To modify other users, you must either be a superuser or have the CREATEROLE privilege (in the latter case, only non-superuser roles can be altered).

Removing a user without dropping their data

*When trying to drop a user who owns some tables or other database objects, you get the following error, and the user is not dropped:*

*testdb=# drop user bob;*

*ERROR: role “bob” cannot be dropped because some objects depend on it*

*DETAIL: owner of table bobstable*

*owner of sequence bobstable\_id\_seq*

*This recipe presents two solutions to this problem.*

To modify users, you must either be a superuser or have the CREATEROLE privilege.

Checking whether all users have a secure password

PostgreSQL has no built-in facilities to make sure that you are using strong passwords. The best you can do is to make sure that all user passwords are encrypted, and that your pg\_hba.conf file does not allow logins with a plain password. That is, always use MD5 as the login method for users. For client applications connecting from trusted private networks, either real or virtual (VPN), you may use host-based access, provided you know that the machine on which the application is running is not used by some non-trusted individuals. For remote access over public networks, it may be a better idea to use SSL client certificates.

Giving limited superuser powers to specific users

First, the superuser role has some privileges that can also be granted to non-superuser roles separately.

To give the bob role the ability to create new databases, run this:

*ALTER ROLE BOB WITH CREATEDB;*

To give the bob role the ability to create new users, run the following:

*ALTER ROLE BOB WITH CREATEROLE;*

However, it is also possible to give ordinary users more fine-grained and controlled access to an action reserved for superusers using security definer functions. The same trick can also be used to pass partial privileges between different users.

First, you must have access to the database as a superuser in order to delegate powers. Here, we assume the use of the default superuser named postgres. We will demonstrate two ways to make some superuser-only functionality available to a selected ordinary user.

Auditing DDL changes

This recipe shows you how you can collect the data definition language (DDL) from database logs in order to audit changes to the database structure.

Edit your postgresql.conf file to set the following:

*log\_statement = 'ddl'*

Setting it to mod or all is also OK for this. Don't forget to reload the configuration:

etc/init.d/postgresql reload

Auditing data changes

This recipe provides different ways of collecting changes to data contained in the tables for auditing purposes.

First, you must make the following decisions:

* Do you need to audit all changes or only some?
* What information about the changes do you need to collect? Only the fact that the data has changed?
* When recording the new value of a field or tuple, do you also need to record the old value?
* Is it enough to record which user made the change, or do you also need to record the IP address and other connection information?
* How secure (tamper-proof) must the auditing information be? For example, does it need to be kept separately, away from the database being audited?

Based on answers to these questions, you can select the right auditing method from the methods we present next.

Always knowing which user is logged in

In the preceding recipes, we just logged the value of the user variable in the current PostgreSQL session to log the current user role. This does not always mean that this particular user was the user that was actually authenticated at the start of session. For example, a superuser can execute the SET ROLE TO ... command to set its current role to any other user or role in the system. As you might expect, non-superusers can assume only those roles that they own.

Connecting using SSL

Here, we will demonstrate how to enable PostgreSQL to use SSL for the protection of database connections by encrypting all of the data passed over that connection. Using SSL makes it much harder to sniff the database traffic, including usernames, passwords, and other sensitive data. Otherwise, everything that is passed unencrypted between a client and the database can be observed by someone listening to a network somewhere between them. An alternative to using SSL is running the connection over a virtual private network (VPN). Using SSL makes the data transfer on the encrypted connection a little slower, so you may not want to use it if you are sure that your network is safe. The performance impact can be quite large if you are creating lots of short connections, as setting up an SSL connection is quite CPU-heavy. In this case, you may want to run a local connection pooling solution, such as PgBouncer, to which the client connects without encryption, and then configure PgBouncer for server connections using SSL. Older versions of PgBouncer did not support SSL; the solution was to channel server connections through stunnel, as described in the PgBouncer FAQ at <https://pgbouncer.github.io/faq.html>.

Get, or generate, an SSL server key and certificate pair for the server, and store these in the data directory of the current database instance as the server.key and server.crt files. On some platforms, this is unnecessary; the key and certificate pair may already be generated by the packager. For example, on Ubuntu, PostgreSQL is set up to support SSL connections by default.

**Getting the SSL key and certificate**

For web servers, you must usually get your SSL certificate from a recognized certificate authority (CA), as most browsers complain if the certificate is not issued by a known CA. They warn the user of the most common security risks and require confirmation before connecting to a server with a certificate issued by an unknown CA.

For your database server, it is usually sufficient to generate the certificate yourself, using OpenSSL. The following commands generate a self-signed certificate for your server:

*openssl genrsa 2048 > server.key*

*openssl req -new -x509 -key server.key -out server.crt*

**Setting up a client to use SSL**

The behavior of the client application regarding SSL is controlled by an environment variable, PGSSLMODE. This can have the following values, as defined in the official PostgreSQL documentation:

| **sslmode** | **Eavesdropping protection** | **MITMprotection** | **Statement** |
| --- | --- | --- | --- |
| disable | No | No | I don't care about security, and I don't want to pay the overhead of encryption. |
| allow | Maybe | No | I don't care about security, but I will pay the overhead of encryption if the server insists on it. |
| prefer | Maybe | No | I don't care about encryption, but I wish to pay the overhead of encryption if the server supports it. |
| require | Yes | No | I want my data to be encrypted, and I accept the overhead. I trust that the network will make sure I always connect to the server I want. |
| verify-ca | Yes | Depends on CA-policy | I want my data encrypted, and I accept the overhead. I want to be sure that I connect to a server that I trust. |
| verify-full | Yes | Yes | I want my data encrypted, and I accept the overhead. I want to be sure that I connect to a server I trust, and that it's the one I specify. |

MITM in the preceding table means man-in-the-middle attack; that is, someone posing as your server-perhaps by manipulating DNS records or IP routing tables-but who actually just observes and forwards the traffic. For this to be possible with an SSL connection, this someone needs to have obtained a certificate that your client considers valid.

**Checking server authenticity**

The last two SSL modes allow you to be reasonably sure that you are actually talking to your server by checking the SSL certificate presented by the server. In order to enable this useful security feature, the following files must be available on the client side. On Unix systems, they are located in the client home directory, in a subdirectory named ~/.postgresql. On Windows, they are in %APPDATA%postgresql.

| **File** | **Contents** | **Effect** |
| --- | --- | --- |
| ~/.postgresql/postgresql.crt | client certificate | requested by server |
| ~/.postgresql/postgresql.key | client private key | proves client certificate sent by owner; does not indicate certificate owner is trustworthy |
| ~/.postgresql/root.crt | trusted certificate authorities | checks that server certificate is signed by a trusted certificate authority |
| ~/.postgresql/root.crl | certificates revoked by certificate authorities | server certificate must not be on this list |

Only the root.crt file is required for the client to authenticate the server certificate. It can contain multiple root certificates against which the server certificate is compared.

**Using SSL certificates to authenticate**

This can be used as an additional security layer to use double authentication, where the client must both have a valid certificate to set up the SSL connection and also know the database user's password. It can also be used as the sole authentication method, where the PostgreSQL server will first verify the client connection using the certificate presented by the client, and then retrieve the username from the same certificate.

**Encrypting sensitive data**

*This recipe shows you how to encrypt data using the pgcrypto contrib package*

## MongoDB

For Non-Structure data, our MongoDB adopt these guide line to design, the user would trust their data safely.

User Access Management Authentication

Authentication is designed to confirm the identity of entities accessing the database. In this context, entities are defined as:

* Users who need access to the database as part of their day-to-day business function
* Administrators (i.e. sysadmins, DBAs, QA staff) and developers
* Software systems including application servers, reporting tools, and management and backup suites
* Physical and logical nodes that the database runs on. Databases can be distributed across multiple nodes both for scaling operations and to ensure continuous operation in the event of systems failure or maintenance.

**Create Security Credentials.**

Create login credentials for each entity that will need access to the database, and avoid creating a single “admin” login that every user shares.

By creating credentials it becomes easier to define, manage, and track system access for each user. Should a user’s credentials become compromised, this approach makes it easier to revoke the user without disrupting other users who need access to the database.

Developers, Administrators, and DBAs should all have unique credentials to access the database. When logins are shared it can be impossible to identify who attempted different operations, and it eliminates the ability to assign fine-grained permissions. With unique logins, staff that move off of the project or leave the organization can have their access revoked without affecting other user accounts.

It is a best practice to create separate login credentials for each application that will be accessing the database. As new applications are introduced and old applications are retired, this approach helps manage access. Some MongoDB customers create multiple logins for different services within a single application, which are then recorded in audit trails and query logs.

Authentication should be enforced between nodes. This prevents unauthorized instances from joining a database cluster, preventing the illicit copying or movement of data to insecure nodes.

**Supporting In-Database and Centralized User Access Management**

Databases should provide the ability to manage user authentication either within the database itself, typically via a Challenge/Response mechanism, or through integration with organization-wide identity management systems. Integrating MongoDB within the existing information security infrastructure enforces centralized and standardized control over user access. If, for example, a user’s access must be revoked, the update can be made in a single repository and enforced instantly across all systems, including MongoDB.

**Enforce Password Policies**

Passwords should adhere to minimum complexity requirements established by the organization, and they should be reset periodically. Low entropy passwords can be easy to break, even if they are encrypted. High entropy passwords can be compromised given sufficient time.

User Rights Management - Authorization

Once an entity has been authenticated, authorization governs what that entity is entitled to do in the database. Privileges are assigned to user roles that define a specific set of actions that can be performed against the database. Best practices include:

**Grant Minimal Access to Entities**

Entities should be provided with the minimal database access they need to perform their function. If an application requires access to a logical database, it should be restricted to operations on that database alone, and prevented from accessing other logical databases. This helps protect against both malicious and accidental access or unauthorized modification of data.

**Group Common Access Privileges into Roles**

Entities can often be grouped into “roles” such as “DBA”, “Sysadmin”, and “App server.” Permissions for a role can be centrally managed and users can be added or removed from roles as needed. Using roles helps simplify management of access control by defining a single set of rules that apply to specific classes of entities, rather than having to define them individually for each user.

**Control Which Actions an Entity Can Perform**

When granting access to a database, consideration should be made for which specific actions or commands each entity should have permission to run. For example, an application may need read/write permissions to the database, whereas a reporting tool may be restricted to read-only permission. Some users may be granted privileges that enable them to insert new data to the database, but not to update or delete existing data.

Care should be taken to ensure that only the minimal set of privileges is provided. Credentials of the most privileged accounts could compromise the entire database if they are hacked internally or by an external intruder.

**Control Access to Sensitive Data**

To prevent the emergence of data silos, it should be possible to restrict permissions to individual fields, based on security privileges. For example, some fields of a record may be accessible to all users of the database, while others containing sensitive information, such as PII, should be restricted to users with specific security clearance.

Enable Access Control and Enforce Authentication

The first step in securing your MongoDB database is to enable access control. As new apps and services are in development, it is typical that access control is not enforced. But once these applications are ready for test and QA, then it is important to specifically enable it, thus requiring all clients and servers provide valid credentials before they can connect to the database. This ensures that you are not deploying exposed instances when you launch your app into production. MongoDB 3.4 and beyond support a rolling transition to internal authentication without database downtime.Refer to the documentation for a tutorial stepping through how to configure authentication.

Ensure that MongoDB runs in a trusted network environment and limit the interfaces on which MongoDB instances listen for incoming connections. Allow only trusted clients to access the network interfaces and ports on which MongoDB instances are available. See the Security Hardening section of the documentation to learn more about reducing the risk of exposure.

MongoDB Authentication

MongoDB provides multiple authentication methods, allowing the approach best suited to meet the requirements of different environments. Authentication can be managed from the database itself, or through integration with external authentication mechanisms.

**In Database Authentication**

MongoDB authenticates entities on a per-database level using the SCRAM IETF RFC 5802 standard. Users are authenticated via the authentication command, while database nodes can be authenticated to the MongoDB cluster via keyfiles.

**x.509 Certificate Authentication**

With support for x.509 certificates MongoDB can be integrated with existing information security infrastructure and certificate authorities, supporting both user and inter-node authentication.

Users can be authenticated to MongoDB using client certificates rather than self-maintained passwords.

Inter-cluster authentication and communication between MongoDB nodes can be secured with x.509 member certificates rather than keyfiles, ensuring stricter membership controls with less administrative overhead, i.e. by eliminating the shared password used by keyfiles. x.509 certificates can be used by nodes to verify their membership of MongoDB replica sets and sharded clusters. A single Certificate Authority (CA) should issue all the x.509 certificates for the members of a sharded cluster or a replica set.

## Device Connectivity Security

The WISE-PaaS IoTHub supports multiprotocol including MQTT which the most popular for IoT (Internet of Thing) handshake protocol. MQTT has been an immense relief to IoT projects that have many modularized components communicating with each other over a range of different environments. MQTT’s simplicity and efficiency have cut down countless hours trying to get different components speaking different languages over different protocols to talk to each other.

**The WISE-PaaS IoTHub protocols**

**AMQP 0-9-1, 0-9 and 0-8, and extensions**

IoTHub was originally developed to [support AMQP](https://www.rabbitmq.com/protocol.html). As such this protocol is the "core" protocol supported by the broker. These variants are fairly similar to each other, with later versions tidying up unclear or unhelpful parts of earlier versions. We have [extended](https://www.rabbitmq.com/extensions.html) AMQP 0-9-1 in various ways.

AMQP 0-9-1 is a binary protocol, and defines quite strong messaging semantics. For clients it's a reasonably easy protocol to implement, and as such there are [many implementations](https://www.rabbitmq.com/devtools.html) available for many different programming languages and environments.

**STOMP**

[STOMP](http://stomp.github.io/) is a text-based messaging protocol emphasizing (protocol) simplicity. It defines little in the way of messaging semantics, but is easy to implement and very easy to implement partially (it's the only protocol that can be used by hand over telnet).

**MQTT**

[MQTT](http://mqtt.org/) is a binary protocol emphasizing lightweight publish / subscribe messaging, targeted towards clients in constrained devices. It has well defined messaging semantics for publish / subscribe, but not for other messaging idioms.

**AMQP 1.0**

Despite the name, AMQP 1.0 is a radically different protocol from AMQP 0-9-1 / 0-9 / 0-8, sharing essentially nothing at the wire level. AMQP 1.0 imposes far fewer semantic requirements; it is therefore easier to add support for AMQP 1.0 to existing brokers. The protocol is substantially more complex than AMQP 0-9-1, and there are fewer client implementations.

**HTTP**

HTTP is of course not a messaging protocol. However, RabbitMQ can transmit messages over HTTP in three ways:

* The [management plugin](https://www.rabbitmq.com/management.html) supports a simple HTTP API to send and receive messages. This is primarily intended for diagnostic purposes but can be used for low volume messaging without [reliable delivery](https://www.rabbitmq.com/reliability.html).
* The [Web-STOMP plugin](https://www.rabbitmq.com/web-stomp.html) supports STOMP messaging to the browser, using WebSockets or one of the fallback mechanisms supported by SockJS.
* The [JSON-RPC channel plugin](https://www.rabbitmq.com/plugins.html) supports AMQP 0-9-1 messaging over [JSON-RPC](http://json-rpc.org/) to the browser. Note that since JSON RPC is a synchronous protocol, some parts of AMQP that depend on asynchronous delivery to the client are emulated by polling.

**Authentication/Authorization**

# Referenced