|  |
| --- |
|  |
| Devops Control Center Specification  by Luis Miguel Huapaya |
| INTERNAL DOCUMENT --- CONFIDENTIAL  This document contains proprietary information which is considered confidential and for internal purposes only. This document should not be reproduced or transferred to other documents or mechanisms and shall not be disclosed to others outside the original list of recipients.  Secure AI Labs, Inc. Proprietary and Confidential information is exempt from public disclosure under the Freedom of Information Act [5 USC 552 (b) (4)], the Arms Export Control Act [22 USC 2778 (e)], the Export Administration Act [50 USC APP. 2411 (c)] and the Trade Secrets Act [18 USC 1905]  Copyright © 2022 Secure AI Labs, Inc.  All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or in any other form or by any means, including electronic, mechanical, photocopying, recording or otherwise, without prior written permission of Secure AI Labs, Inc. |

Table of Contents

[Revision History 3](#_Toc101316610)

[Introduction 4](#_Toc101316611)

[DEVCC In A Nutshell! 4](#_Toc101316612)

[Basic Architecture 5](#_Toc101316613)

[Basic Operations and Interactions 5](#_Toc101316614)

[Example 7](#_Toc101316615)

[DEVCC Initialization 9](#_Toc101316616)

[Plugins 10](#_Toc101316617)

[Platform Provisioning Plugin 12](#_Toc101316618)

[SCN Monitoring Plugin 14](#_Toc101316619)

[Platform Monitoring Plugin 15](#_Toc101316620)

[Microsoft Azure Sentinel Monitoring Plugin 16](#_Toc101316621)

[All Other Plugins and Components 17](#_Toc101316622)

[Topics of Discussion 18](#_Toc101316623)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Comment** |
| April 20th, 2022 | Luis Miguel Huapaya | Initial draft |

# Introduction

The Devops Control Center (DEVCC) is a command and control payload which runs within an Azure VM (confidential VM) and is designed to provision and initialize all other Azure based resources in the SAIL Platform. The primary user of the DEVCC will be the SAIL devops engineer, but the DEVCC will also publish (internally) a REST API portal which can be used by the SAIL Platform Services to provision, initialize and monitor SCN payloads.

Core Goals

* centralized
* secure
* implemented using small easy to maintain micro-services
* easy to use
* Reusable

# DEVCC In A Nutshell!

The Devops Control Center is:

1. A plurality of plugins, UI components, and a REST API portal, which interact with each other within the confines of a single Azure Confidential VM.
   1. Whereas each of the components communicate with each other using JSON blobs written to individual files in a predefined hierarchy on disk.
   2. Whereas each of the plugins, UI components and the REST API Portal are individual and separate processes (i.e. micro-services) running simultaneously within the same OS instance (Ubuntu 20.04 LTS).
      1. Each component can be implemented using C++, Python or Rust.
2. Runs within a predefined Azure resource group, virtual network and network security group within an Azure subscription.
3. Provisioned and initialized by a Windows desktop application running on a remote desktop.
   1. Remote desktop user will need authenticated VPN access to the target virtual network that will host the DEVCC. This is required since after provisioning, the desktop application will initialize the DEVCC (i.e. push initialization vectors into the virtual machine) and this can only be done if the desktop application is able to address a network port on the virtual network.
   2. Windows desktop application will require the subscription ID, application ID, tenant ID and the secret required to access the Azure subscription and use API calls into it. However, the application will also be given a series of other initialization vectors as well.
4. A command-and-control platform which requires some form of data persistence (i.e. database) in order to maintain consistent state, even if the virtual machine hosting the DEVCC were to stop working and a new instance of a DEVCC was required.
   1. Some sort of persistent data storage is required in order to ensure that data about the state of the SAIL Platform can survived an unexpected collapse of the DEVCC, and can be recovered by the subsequent re-initialization of a new instance of the DEVCC.
5. Hosted within an Azure Confidential Computer virtual machine. This is required, since the DEVCC will have access to high value cryptographic keys.

# Basic Architecture

The basic architect of the DEVCC is:

Diagram

Description automatically generated

The full list of plugins (at this time) is:

* SCN Provisioning Plugin (to be implemented by Hanu)
* Platform Provisioning Plugin (to be implemented by Hanu)
* SCN Monitoring Plugin (to be implemented by Hanu)
* Platform Monitoring Plugin (to be implemented by Hanu)
* Microsoft Azure Sentinel Monitoring Plugin (to be implemented by Hanu)
* Sail Audit Event Monitoring Plugin (to be implemented by Sail)
* Authentication Plugin (to be implemented by Sail)
* Cryptography Plugin (to be implement by Sail)
* Load Balancer Management Plugin (to be implemented by Hanu)
* MongoDB Atlas Management Plugin (Future, Sail)
* Audit recording plugin (Future, Sail)

The CLI Tools are basically the UI/UX frontend of the DEVCC, used by devops engineers to monitor and control the SAIL platform. The first incarnation of the DEVCC will use CLI based apps accessed through SSH, but a future version of the DEVCC will implement a web front end which will replace the CLI tools and the SSH requirement. Unlike the plugins and the API Portal, the CLI tools only run when they are instantiated on the command line, and once the transaction is done, they stop (i.e. CLI tools do not necessarily run continuously).

# Basic Operations and Interactions

The goal of this section is to talk briefly about the basic flow of operations in the DEVCC, and how different components will interact with one another. The transactional flow of operations within the DEVCC is as follows:

1. API portal or CLI Tool generates a request. This takes the form of a JSON blob written to a predefined location on disk (i.e. special folder).
   1. The location on disk decides the target plugin that will consume the request. Each plugin has its own ‘special folder’.
   2. The location on disk is within an ephemeral virtual encrypted disk, ensuring confidentiality of transactions within the DEVCC. All contents of that virtual disk are lost/shredded once the VM terminates.
2. Target plugin will consume (i.e. read and then delete) the JSON blob. All plugins should have a tread which continuously monitors their ‘special folder’. The target plugin will then spawn a thread to handle the request (for each request).
   1. Note that all requests are given a transaction identifier and a point of origin (i.e. who submitted the request and where to write the response). When responding, the plugin should embed the transaction identifier in the response that it writes as a JSON file on disk.
   2. All plugins should be designed to be able to handle multiple simultaneous requests at once.
3. In cases where the handling of the request is time consuming/asynchronous, then the plugin will generate an ACK response before the request is completed.
   1. Currently, there is only one request which will be handled like this, and this is specifically the ‘Provisioning and Initialization of SCN’.
      1. Once the requestor receives an ACK for the original request to create an SCN, the requestor can poll the status of the SCN until the SCN is fully deployed and running.
   2. Provisioning and initialization of platform payloads will be blocking in nature (i.e. the response to the request will not be generated until the platform payload is fully deployed and running).

Plugins will never generate data (i.e. JSON blobs) unless there is an associated request.

The hierarchy of the special folders (based on a dedicated root folder) will be as follows:

./ --+-- /ScnProvisioning  
 |  
 +-- /PlatformProvisioning  
 |  
 +-- /ScnMonitoring  
 |  
 +-- /PlatformMonitoring  
 |  
 +-- /SentinelMonitoring  
 |  
 +-- /Authentication  
 |  
 +-- /LoadBalancerManagement  
 |  
 +-- /DatabaseManagement  
 |  
 +-- /SailAuditMonitoring  
 |  
 +-- /ApiPortalResponses  
 |  
 +-- /CliTool1Responses  
 |  
 +-- /CliTool…Responses

Whenever any components generate a request, the request will contain:

1. Randomly generated transaction ID which is basically a UUID.
2. Path to folder where response needs to be persisted.

Whenever a component generates a response, it needs to embed the transaction ID into the response JSON, and the JSON blob (since JSON per file) needs to be written in the target folder specified within the original request.

## Example

Consider a request to provision and initialize a Secure Computational Node (SCN), which originates from the SAIL Platform Services and is routed through the API portal component. Upon validation of the incoming request from the SAIL Platform Services, the API portal will post a request (i.e. JSON blob to file) in the target ‘special folder’ which belongs to the `SCN Provisioning` plugin. The JSON request blob will look like:

*{  
 “TransactionIdentifier”:”56491337-faa0-4b3c-9534-984784e93f43”,  
 “RequestorResponseTarget”:”/mnt/SailEncryptedVolume/Sail/Devcc/Ipc/ApiPortal”,  
 “InternalUtcEpochTime”:1650401435,  
 “OriginIdentifier”:”b1da6e37-3d4a-49b6-b97d-59b31494e30a”,  
 “RequestHmac”:”5308b6b4cb77caa8e1eb9b12dcf719eef460c172a6f7efd379b520d3888318fe”,  
 “Request”: {  
 “UtcEpochTime”:1650401427,  
 “VirtualMachineType”:”* *Standard\_EC32as\_v5”,  
 “ResearchOrganizationIdentifier”:”c9e8b975-2793-4e2e-b15f-b03f2ae05192”,  
 “DataOwnerOrganizationIdentifier”:”a05b9c78-e52f-4e6e-834d-d54bbcceb87d”,  
 “DigitalContractIdentifier”:”c69a4c91-1190-4cb0-bcca-41ed6e797bde”,  
 “DatasetIdentifier”:”78b372e8-87b3-4a06-8496-7eaa16441da9”,  
 “Region”:”USWest2”  
 }  
}*

Where:  
  
*TransactionIdentifier*: is generated by the API portal and added to the incoming request.  
  
*RequestorResponseTarget*: is generated by the API portal. This is the target folder where the response should be written to. In this case, the ‘special folder’ associated with the Api Portal.  
  
*InternalUtcEpochTime*: is generated by the API portal and added to the incoming request. This represents the UTC Epoch time when the transaction is received by the API portal. This value is used to computer internal transactional performance.  
  
*OriginIdentifier*: provided by the caller (in this case, SAIL Platform Services). This represents the identifier of the virtual machine payload which generated the request. Each SAIL payload (i.e. VM) is given an identifier. This allows the API Portal to figure out which HMAC key to use when verifying the *RequestHmac*.  
  
*RequestHmac*: provided by the caller (in this case, SAIL Platform Services). This represents the HMAC of the Request body. The Api Portal will be able to derive an HMAC key from the *OriginIdentifier* and then verify the HMAC value against what is submitted. This acts as a form of authentication using cryptography. Only a properly provisioned and initialized SAIL payload would have the cryptographic key required to generate the HMAC properly. Any importer would be unable to spoof this value.  
  
*Request*: this is the actual request provided by the caller (in this case, SAIL Platform Services).  
  
The entire *Request* component is passed as-is as part of the package of initialization vectors which are passed into the provisioned VM that will host the SCN.

The response to this transaction will look something like:  
The response will contain the standard *TransactionIdentifier*, *Status*, *InternalUtcEpochTime,* and the original *UtcEpochTime* along with a transaction specific *VirtualMachineIdentifier* which is passed back to the caller (in this case, SAIL Platform Services).

*{  
 “TransactionIdentifier”:”56491337-faa0-4b3c-9534-984784e93f43”,  
 “Status”:200,  
 “InternalUtcEpochTime”:1650401435,  
 “UtcEpochTime”:1650401427,  
 “VirtualMachineIdentifier”:”2206d262-24bb-4903-85e5-e7d366fdd182”  
}*

The *VirtualMachineIdentifier* allows the original caller (in this case, SAIL Platform Services) to poll for the status of this particular payload until its status is ‘Ready’. Polling for the Status of virtual payloads is done by the “SCN Monitoring” plugin. The *UtcEpochTime* that was original included in the incoming transaction is mirrored back, as-is, so that the caller can compute the total about of time require to execute the transaction, including network latency. The API Portal will use the *InternalUtcEpochTime* to generate and insert a value called “InternalLatency” in the JSON response being returned to the caller, so that the caller has an idea how much time was spent processing the transaction inside the DEVCC, and how much time elapsed in total, thus allowing the caller to figure out network latency. The result response back to the caller will look like:

*{  
 “Status”:200,  
 “InternalLatency”:10,  
 “UtcEpochTime”:1650401427,  
 “VirtualMachineIdentifier”:”2206d262-24bb-4903-85e5-e7d366fdd182”  
}*

It should be noted that all UTC Epoch Time values will be in milliseconds, not in seconds (i.e. the provided example actually uses Epoch time in seconds, which is not representative of reality).

DEVCC Initialization  
In order to understand how the DEVCC works, it is important to understand how it gets provisioned and initialized and what kind of resources it will need access to in order to run.

1. A Windows Desktop application, called the Windows Azure Initializer, is used to provision and initialize the DEVCC.
2. The VM image of the DEVCC is identical to ALL other virtual payloads in the SAIL ecosystem. It consists of a hardened Ubuntu 20.04 LTS image which contains just a single SAIL developed binary, namely the ‘Bootstrapper Program’.
3. As soon as the virtual machine instance is started, the ‘Bootstrapper Program’ is executed. It waits for a single incoming connection which will upload initialization vectors. In the case of the DEVCC, the IV’s are:
   1. Compressed archive containing the DEVCC binaries. The ‘Bootstrapped Program’ will decompress and deploy the DEVCC binaries automatically.
   2. Virtual Machine identifier (generated by the Windows Azure Initializer and passed into the ‘Bootstrapped Program’).
   3. Azure identifier of the Azure Key Vault which contains the keys required to decrypt the compressed archives containing binaries (see a.). This is required since the compressed archives stored in the Artifact Store (i.e. Azure Blob Storage) are encrypted at the source. The DEVCC will need the ability to decrypt the compressed archives before re-encrypting them using an ephemeral key (AES) negotiated between the DEVCC and the ‘Bootstrapper Program’ of the target virtual machine. This Azure Key Vault is shared between the DEVCC and the Build System. The Azure Key Vault also contains the public key certificate required to verify the digital signature of artifacts.
   4. Azure identifier of the Azure Key Vault which contains the several keys, including:
      1. TLS master key set used to sign the public cert of ephemeral TLS keys generated at runtime (ECC384).
   5. Subscription Id, Application Id, Tenant ID and secret required to access the subscription that will be hosting the platform services.
   6. Subscription Id, Application Id, Tenant ID and secret required to access the subscription that will be hosting the SCN’s. This could be the same as (e).
   7. Name of Resource Group to use for Platform Services
   8. Identifier/path to the Azure blob storage which contains artifacts used to initialize virtual payloads.
4. Once the IV’s are uploaded, the bootstrapper program will create an ephemeral virtual encrypted disk and persist all of the IV’s within a specific folder on the disk.
5. Once the ephemeral encrypted disk is created, decrypt the binaries and persist them on the encrypted disk and then deploy them in an internal docker container. This will effectively deploy the DEVCC platform (i.e. all of the plugins will be started, the API portal will be started, etc…).
6. Each plugin can read the IV’s (which are basically a JSON blob on disk) in order to extract the data they need to initialize themselves and to work.
7. The Cryptography Plugin will access the proper Azure Key ring and start generating a bunch of cache ephemeral TLS key sets and associated public key certificates, with the public key certificates being signed by a master TLS key in the Azure Key ring. Those TLS key sets are consumed by other plugins whenever they need new TLS keys.
8. At this point, the DEVCC waits for transactions to come into the Api Portal or, waits for one of the CLI tools to be started and generate transactions towards the plugins.

Plugins  
SCN Provisioning  
This plugin is focused entirely on provisioning and initializing Secure Computational Nodes. In order to be able to accomplish this task, the plugin needs the following information:

1. Azure Subscription ID (provided in the IV’s)
2. Azure Application ID (provided in the IV’s)
3. Azure Tenant ID (provided in the IV’s)
4. Azure Secret (provided in the IV’s)
5. Path/Identifier of the Azure Blob Storage which contains the build artifacts (provided in the IV’s)
6. Type of virtual machine to deploy (provided in the request JSON)
7. Azure region where to spawn VM (provided in the request JSON)
8. Identifier of the research organization who is spawning the SCN (provided in the request JSON)
9. Identifier of the data owner organization who is providing the dataset (provided in the request JSON)
10. Identifier of the digital contract that will be used to enforce the behavior of the SCN (provided in the request JSON)
11. Identifier of the dataset that will be uploaded to the SCN (provided in the request JSON)
12. RSA public key for wrapping audit log key for researcher (provided in the request JSON)
13. RSA public key for wrapping audit log key for data owner (provided in the request JSON)
14. RSA public key for wrapping audit log key for auditor (provided in the request JSON)
15. RSA public key for wrapping audit log key for SAIL (provided in the request JSON)
16. AES data exchange encryption key (generated from Azure Key Vault)
17. Public root key certificate used to verify TLS keys(retrieved from Azure Key Vault)
18. Public root key certificate used to verify datasets (retrieved from Azure Key Vault)

The SCN Provisioning Plugin will need to generate the following randomly generated values:

1. Virtual Machine Identifier
2. Unique Name of Resource Group to create
3. Unique Name of Virtual Network to create.
   1. This includes creating a unique address space that is not used by any other SCN’s.
4. Unique Name of Network Security Group to create.
5. AES key used to encrypt compressed binaries package

*The unique names of ‘Resource Group’, ‘Virtual Network’, etc… all use a standard format which consists of:  
  
Virtual Machine Name = SAIL\_USWEST2\_SCN\_SCN\_{VMID}  
Resource Group = SAIL\_USWEST2\_SCN\_RG\_{VMID}  
Virtual Network = SAIL\_USWEST2\_SCN\_VNET\_{VMID}  
Network Security Group = SAIL\_USWEST2\_SCN\_NSG\_{VMID}*

Once this data is retrieved/acquired, the SCN plugin needs to package things up:

1. If it hasn’t already been done, retrieve the latest compressed binaries (for the SCN) from Azure Blob Storage (i.e. artifact store). This only needs to be done once, but the SCN Provisioning plugin should have a thread which continuously checks to see if there are newer versions of the binaries which have been dropped, and load those up instead.
2. Use an AES key stored in Azure Key Vault to decrypt the encrypted compressed binaries.
3. Use the AES key in (5) above to re-encrypt the compressed binaries. For now, this step can be skipped, but it will be implemented in the future.
4. Package up all of the information in one large JSON blob, ready to be uploaded to the bootloaded.

Once things are ready, the SCN plugin will:

1. Provision a new VM instance using all of the information above.
2. Wait for the provisioning to be done and the VM to mark as running in Azure.
3. Connect to the ‘Bootstrapped Program’ and:
   1. Execute a remote attestation check (not in scope right now). Currently, this will just be a “Hello” and corresponding “Hello Back” transaction. Eventually, this will be filled with a remote attestation challenge which is used by the plugin to make sure that the VM it is talking to is a confidential VM and that its measurement is correct.
   2. Execute a key exchange. This is basically:
      1. Plugin asks the ‘Bootstrapped Program’ for its RSA public key.
      2. The ‘Bootstrap Program’ returns a public RSA key component of an ephemeral RSA key pair that it has generated.
      3. The plugin will use the RSA public key to wrap the ephemeral AES key used encrypt the compressed binaries. All of that will be passed into the IV’s.
   3. Pass in the IV’s

That’s it. The IV is a JSON blob containing:  
Once an SCN Provisioning plugin is done, it needs to write a record to a database somewhere which effectively register this new SCN. Actually, the SCN plugin should write this record even before provisioning the SCN, and the update the status of the SCN based on what is going on (i.e. PROVISIONING, INITIALIZING, WAITING FOR DATA).

*{  
 “VirtualMachineIdentifier”:”56491337-faa0-4b3c-9534-984784e93f43”,  
 “ResearchOrganizationIdentifier”:”fdbf025a-f7bc-4602-86aa-f45765119c88”,  
 “DataOwnerOrganizationIdentifier”:”* *7dcf313c-2273-4b38-8ec2-d3ceeaf1e325”,  
 “DigitalContractIdentifier”:”624ad548-1371-43c1-9886-fb8f459a9072”,*

*“DatasetIdentifier”:”d4e3960c-5c47-4b2c-a19e-38f0d275ba38”,  
 “TlsKeysAndCert”:”<long base64 string>”,  
 “EncryptedAesKey”:”<long base64 string>”,  
 “EncryptedCompressedBinaries”:”<long base64 string>”,  
 “ResearchOrganizationRsaAuditPublicKey”:”<long base64 string>”,  
 “DataOwnerOrganizationRsaAuditPublicKey”:”<long base64 string>”,  
 “AuditorOrganizationRsaAuditPublicKey”:”<long base64 string>”,  
 “SailOrganizationRsaAuditPublicKey”:”<long base64 string>”,*

*“EncryptedAesDataExchangeKey”:”<long base64 string>”,  
 “IpAddressOfSailPlatformServices”:”123.45.34.34”,  
}*

Requests to this plugin are asynchronous, which means that an incoming request will return an acknowledgement right away, even before the transaction is completed. The response always contains the Virtual Machine Identifier, which is required so that the caller can then poll for the ongoing status of the SCN being provisioning/initialized.

## Platform Provisioning Plugin

The Platform Provisioning plugin is focused on the provisioning and initialization of virtual payloads that make up the SAIL platform back-end. This involves:

1. SAIL Platform Services API Portal.
2. SAIL Data Services API Portal.
3. SAIL WebUI Authenticated Customer Portal.
4. SAIL Remote Data Connector.

In order to be able to accomplish this task, the plugin needs the following information:

1. Azure Subscription ID (provided in the IV’s)
2. Azure Application ID (provided in the IV’s)
3. Azure Tenant ID (provided in the IV’s)
4. Azure Secret (provided in the IV’s)
5. Platform Services Resource Group Name
   1. This Azure Resource Group needs to be created if it does not exist already.
6. Platform Services Virtual Network Name
   1. This Azure Virtual Network needs to be created if it does not exist already.
7. Platform Services Network Security Group Name
   1. This Azure Network Security Group needs to be created if it does not exist already.
8. Path/Identifier of the Azure Blob Storage which contains the build artifacts (provided in the IV’s)
9. Public root key certificate used to verify TLS keys(retrieved from Azure Key Vault)
10. Public root key certificate used to verify datasets (retrieved from Azure Key Vault)

The SCN Provisioning Plugin will need to generate the following randomly generated values:

1. Virtual Machine Identifier
2. AES key used to encrypt compressed binaries package

*The unique names of ‘Resource Group’, ‘Virtual Network’, etc… all use a standard format which consists of:  
  
Virtual Machine Name = SAIL\_USWEST2\_SCN\_SCN\_{VMID}  
Resource Group = SAIL\_USWEST2\_SCN\_RG\_{VMID}  
Virtual Network = SAIL\_USWEST2\_SCN\_VNET\_{VMID}  
Network Security Group = SAIL\_USWEST2\_SCN\_NSG\_{VMID}*

Once this data is retrieved/acquired, the SCN plugin needs to package things up:

1. If it hasn’t already been done, retrieve the latest compressed binaries (for the SCN) from Azure Blob Storage (i.e. artifact store). This only needs to be done once, but the Platform Provisioning plugin should have a thread which continuously checks to see if there are newer versions of the binaries which have been dropped, and load those up instead.
2. Use an AES key stored in Azure Key Vault to decrypt the encrypted compressed binaries.
3. Use the AES key in (5) above to re-encrypt the compressed binaries. For now, this step can be skipped, but it will be implemented in the future.
4. Package up all of the information in one large JSON blob, ready to be uploaded to the bootloaded.

Once things are ready, the SCN plugin will:

1. Provision a new VM instance using all of the information above.
2. Wait for the provisioning to be done and the VM to mark as running in Azure.
3. Connect to the ‘Bootstrapped Program’ and:
   1. Execute a remote attestation check (not in scope right now). Currently, this will just be a “Hello” and corresponding “Hello Back” transaction. Eventually, this will be filled with a remote attestation challenge which is used by the plugin to make sure that the VM it is talking to is a confidential VM and that its measurement is correct.
   2. Execute a key exchange. This is basically:
      1. Plugin asks the ‘Bootstrapped Program’ for its RSA public key.
      2. The ‘Bootstrap Program’ returns a public RSA key component of an ephemeral RSA key pair that it has generated.
      3. The plugin will use the RSA public key to wrap the ephemeral AES key used encrypt the compressed binaries. All of that will be passed into the IV’s.
   3. Pass in the IV’s

That’s it. The IV is a JSON blob containing:  
Once an Platform Provisioning plugin is done, it needs to write a record to a database somewhere which effectively register this new payloads. Actually, the Platform Provisioning plugin should write this record even before provisioning the virtual payloads, and the update the status of the payloads based on what is going on (i.e. PROVISIONING, INITIALIZING, WAITING FOR DATA).

*{  
 “VirtualMachineIdentifier”:”56491337-faa0-4b3c-9534-984784e93f43”,*

*“TlsKeysAndCert”:”<long base64 string>”,  
 “EncryptedAesKey”:”<long base64 string>”,  
 “EncryptedCompressedBinaries”:”<long base64 string>”,*

*“IpAddressOf<Sail/Data>Services”:”123.45.34.34”  
}*

Requests to this plugin are blocking, which means that the transaction does not complete until the request has been fulfilled.

SCN Monitoring Plugin  
The SCN Monitoring plugin is focused on simply monitoring the subscription used to host SCN virtual payloads. Basically, the SCN plugin is running a thread which is continuously monitoring the Azure resources within a subscription and based on standard naming conventions, maintain the running status of all SCN payloads.

The plugin offers the following API calls:

1. Get VM status given a specific VM identifier.
2. Get all the list of VM which are associated with an organization.
3. Update the status of a VM given a specific VM identifier. This allows the SAIL Platform Services to notify the DEVCC that data has been successfully uploaded to an SCN and that the SCN is now in a ready state.

Requests to this plugin are blocking, which means that the transaction does not complete until the request has been fulfilled.

Platform Monitoring Plugin  
The Platform Monitoring plugin is focused on simply monitoring the subscription used to host platform virtual payloads. Basically, the Platform Monitoring plugin is running a thread which is continuously monitoring the Azure resources within a subscription and based on standard naming conventions, maintain the running status of all platform payloads.

The plugin offers the following API calls:

1. Get overall status of all platform payloads

Requests to this plugin are blocking, which means that the transaction does not complete until the request has been fulfilled.

Microsoft Azure Sentinel Monitoring Plugin  
The Microsoft Azure Monitoring plugin is focused on simply polling Microsoft Azure Sentinel audit events and displaying/analysing the audit events in real-time. The ultimate goal of this plugin is to generate warnings and notices automatically so that devops engineers can be warned of special or unusual events.

## All Other Plugins and Components

Since all of the other plugins and components are to be authored by SAIL, they will be omitted from this document.

That being said, HANU should be able to develop the previous plugins without any of the other components, since IPC between components consists of JSON blobs on disk. All of that can be ‘faked’ manually in order to test/develop plugins independently of one another.

# Topics of Discussion

There are some topics which are unresolved.

1. Authentication plugin. How is this to be properly implemented so that the various plugins can be properly authenticated to the Azure Key Vaults, Azure Blob Storage, and Microsoft Azure Sentinel for access?
2. Cryptographic plugin. As noted in 1, the Cryptographic plugin is used to access Azure Key Vaults and will require some form of authentication in order to do so. How can we achieve this?
3. Provisioning plugins. Both the SCN and Platform provisioning plugins require authenticated access to an Azure Blob Storage in order to retrieve build artifacts. How can we achieve this authentication?
4. Passing one-time-secrets to the virtual machine as a startup parameter. This can be used as an additional step in authenticating elements in the SAIL platform.