Secure Cloud Service Credentials

Overview

Services deployed to the edge often require access to cloud services, which means the service needs credentials to authenticate to the cloud service. The existing support for this capability is not very secure. It requires the service developer to create a service variable (in the service definition) to hold the credentials, and the service deployer to provide the credentials by setting that variable in the deployment policy (or pattern) that will deploy the service. This leaves the credentials exposed because deployment policies are readable by any user in the org. Service variables are also logged. This design will enhance OpenHorizon to provide a secure mechanism which allows credentials to be transported and managed without exposing them within OH metadata (e.g. service defs and policies).

Design

<Describe how the problem is fixed. Include all affected components. Include diagrams for clarity. This is the longest section in the document. Use the subsections below to call out specifics related to each topic, and refer back to this section for context. Provide links to any relevant external information.>

Secrets

A secret is a userid/pw, certificate, RSA key, or any other credential that grants access to a protected resource which an edge application needs in order to perform it's function. Secrets are stored in a vault, in the management hub. In the vault, a secret has a name, which is used to identify the secret, but which provides no information about the details of the secret itself. The vault and secrets in it are administered by a vault administrator, using the vault's UI or CLI.

When a service developer needs a secret, their code is written such that it can refer to the secret by a name they chose. The service deployer attaches a secret from the vault to the deployment of the service, by associating the name of the secret used by the developer with the name of the secret in the vault. This is called a **secret association**. For example; suppose a developer needs to access a cloud service via

basic auth. The OpenHorizon service definition is updated to include a secret called myCloudServiceCreds. The service deployer sees that the service requires a secret in order to deploy it, and is aware of a secret in the vault named cloudServiceXYZSecret. The service deployer updates the deployment policy (or pattern) to indicate that the service's secret named myCloudServiceCreds should contain the credentials from the vault secret named cloudServiceXYZSecret.

The service definition schema is updated to include **secrets** as a top level key, using the following JSON pseudo schema:

```
...
"secrets": {
    "<secret-name>": {
        "type": "<credential type hint>",
        "description": "<some long description>"
    }
},
```

where;

- **<secret-name>** is the name of the secret as chosen by the service developer.
- **type** is a hint from the service developer to the service deployer indicating what kind of credential should be provided. The value of this field is not used or validated by OH, it can be any string. This field is optional.
- **description** is a long description of what this secret is used for. It is used to provide more information to the service deployer if necessary. This field is optional.

Question:
UX-7 - Does Vault expose secret type? is secret type enforceable or is it softer than that, more like a hint to the deployer?

The pattern and deployment policy schema is updated to include **secret associations**, using the following JSON snippet:

```
"secretBinding": [
    {
        "serviceOrgid": "<service-org-id>",
        "serviceUrl": "<service-name-or-URL>",
        "serviceArch": "<service-hardware-architecture>",
```

```
        "serviceVersionRange": "x.y.z",
        "serviceContainer": "<container-name>",
        "bindings": {
          "<service-secret-name>": {
            "vaultSecret": "<vault-secret-name>"
          }
        }
    },
]
```

where;

- **"serviceOrgid", "serviceUrl", "serviceArch",** and **"serviceVersionRange"** are used in the same way as the similarly named fields in the "userInput" section. These fields are required.
- **"serviceContainer"** is the name of a container as defined in the deployment configuration of a service. Within the deployment configuration is a map called "services", where the key for each element in the map is the container name. This field is optional when there is only 1 container defined in a service's deployment configuration. This field does not contain a container image name.
- **<service-secret-name>** is the name of the secret as chosen by the developer. This name must match a secret name in the service identified by "serviceOrgid", "serviceUrl", "serviceArch", and "serviceVersionRange". This field is required.
- **<vault-secret-name>** is the name of a secret in the vault that the service will use at runtime.

In a running service container, the secret(s) will be mounted to the container in a folder named /open-horizon-secrets, and the file name (for each secret) will match the secret name provided by the developer. This allows the developer to access the secret by their chosen name, regardless of the secret's name in the vault.

The deployer can provide secret associations for any service in the deployed service's dependency tree. This allows reusable services to be configured with different secrets for different usages and consumers of those services.

Secret Management

The agent and agbot MUST NOT log or display secret details, but MUST log secret names and **secret associations**.

The secret details are kept on the node in root protected storage managed by the agent, and mounted ONLY to the service containers that need them. Secrets are overwritten with garbage and then deleted from the edge node when the owning service terminates. Secrets are kept in plain text when at rest. If encryption at rest is desired, encrypt the underlying disk using the tools provided by the node's operating system. Since the secret needs to be in plain text for the application (because it is impossible to provide a decryption key that is not exposed somewhere) it has to be in plain text in the agent's host filesystem. Only root users on the host will be able to view the secrets.

??? What if we allowed the service author to provide a public key for encryption of the secret at rest (when they publish the service). The service author would package the private key inside the container, and would be responsible for decrypting the secret. The problem is how to roll the decryption key which would require integration with the vault which creates a chicken and egg problem. ??? BP: having the private key in the docker image is no more secure that having the secret in a root-accessible-only volume.

??? What if we gave each service container the URL to the vault in the hub. The service deployer could provide a secret that allowed the app to authenticate to the vault directly and the app could do its own secret management. This completely removes OH from the path of secret management. It requires that the vault be able to support 10s of thousands of nodes. ??? BP: wouldn't we still have to provide the vault creds to the service? We can't make them put that in the deployment policy userInput section.

Secret details are delivered from the hub to an agent through the agreement protocol. The agbot will retrieve the details of all secret associations for all services involved in the agreement and copy those details into the agreement proposal that is sent to the agent. All agreement protocol messages are encrypted such that only the target agent can decrypt them, providing message protection for the secret details while they are being transmitted over the network. The agent or service NEVER interacts with the Vault API directly.

When the agbot is evaluating a node for an agreement, and the service (or one of it's dependencies) is associated with a secret that does not exist in the vault, then the agbot will log the failure and refuse to make an agreement. If the associated secret appears in the vault at some point in the future, an agreement will be attempted once again.

The agbot periodically queries the vault, looking for updates to secrets or secrets that might have been previously missing and takes action to make new agreements or to update existing agreements.

Updating secrets

Secrets can be updated in the vault, by the vault administrator. Updated secrets are transported to the appropriate node via the agreement protocol and made available to services via a new REST API hosted by the agent. The REST API allows the service implementation to query for new secrets and to receive them when it is ready to do so. The agent already hosts the MMS APIs, so the new secret related APIs will be handled by a different http handler, but listening on the same network endpoint as the MMS API.

REST API details:
To query for updated secrets, the URL path is:
/secret

GET /secrets
200: Returns a JSON encoded array of secret names that have been updated since the service was deployed.
For example, the returned array looks like this snippet:
["<secret-name>","<secret-name>",...]

where <secret-name> is a secret name used by the developer in the service definition.

404: No secrets have been updated

GET /secrets/<secret-name>
200: Returns a JSON encoded secret, as follows:

```
{"<secret-name>":{
     "details":"<base64-encoded-secret-details>"
   },
}
```

where <secret-name> is a secret name used by the developer in the service definition.

400: <secret-name> is not defined for the service calling the API

POST /secrets/<secret-name>?received=true

201: Acknowledges that secret <secret-name> has been received by the service and no longer needs to be returned on a subsequent GET /secrets API call. The GET /secrets API will continue to return an updated secret until it is acknowledged using this API.

where <secret-name> is a secret name used by the developer in the service definition.

400: <secret-name> is not defined for the service calling the API

If a service container restarts AFTER receiving an updated secret from the GET /secrets/<secret-name> API, the updated secret will be provided within the /open-horizon-secrets mount point inside the service container. The service will receive HTTP status code 404 on GET /secrets following a restart.

API Authentication

Every service is provided with an authentication id and token which is used to access the ESS API. The same id and token are used to authenticate to this API.

??? Perhaps we should deprecate the ESS env vars given to the service for auth'cn purposes and replace them with more generic sounding names, since we now have a precendent for 2 features using the "service to agent" callback API. ???

Detecting Secret Updates

When the agbot detects an updated secret in the vault, it will employ a new extension to the agreement protocol that enables the secret to be updated within an existing agreement. The protocol extension will be implemented such that it can be used to update an existing agreement for any purpose. The extension is not specific to updating secrets. An agreement update can be rejected by an agent, which does not imply that the agreement should be cancelled. Of course the agent or agbot can always terminate an agreement for any reason. The new protocol extension message will include an ACK as usual. The updated secret details are sent encrypted to the affected agents. Downlevel agents ignore protocol messages they don't understand, so the agbot has to be prepared to handle that case by retrying a few times before giving up on the update. An un-ACKed protocol extension does not cause the agreement to be cancelled.

??? Should there be something in the config of an agent (or it's org) that causes the agent to reject some or all agreement updates? For example, what if the customer wants to allow secret updates, but disallow some other update we havent devised yet. Same question for the agbot, should there be some config that tells the agbot to cancel and agreement if an update is rejected in some cases but not others ???

If accepted, the secret details are updated in the agent's storage (in case the target service restarts). The API described above will return the updated secret when called by the application. The application can store the updated secret anywhere inside the container. If the service restarts after receiving an updated secret, the secret details mounted to the restarted container will be the updated secret details. The application does not need to invoke the API in order to retrieve the recently updated secret.

Hashicorp Vault

Hashicorp Vault is running in the management hub, or somewhere else in the network.
The Vault admin uses the vault's CLI and UI to perform UX-8, 9, 10, and 12 below.

???Do we need to write an auth'cn plugin for the vault that delegates auth'cn to the exchange, I think no right now.??? BP: if we don't, how does the agbot get the vault creds? Pass them in via a kube secret, which is not secret at all?Yes, the same way the agbot gets creds to talk to the exchange.

The agbot is the only component of the hub that will invoke Vault APIs. It is desirable to protect the vault from attack, thus only the OH agbot will have credentials to access the vault. Further, allowing agents to access the vault would require the vault to support 1000s of concurrent requests, which it is not really designed to support. Therefore, access to the vault is always through the agbot.

Each OH management hub will optionally contain a Hashicorp vault instance. The agbot must gracefully handle the absence of the vault. Vault absence could be temporary, such as when the vault component is being updated, or permanent when the vault is not configured in the management hub. Either way, the agbot will refuse to make agreements when a deployment (policy or pattern) requires a secret, but the vault is absent to provide the secret. The hzn deploycheck command is used to discover that a secret is missing or unavailable.

Describe how vault federation/proxy works.....

Edge Clusters

This design does not apply to cluster nodes because the problem is already solved by k8s. For edge clusters, OH deploys an operator which can refer to k8s secrets that have been setup by the k8s admin (possibly using Vault). It is possible to configure a vault in k8s such that:
1) there is a vault in the edge cluster that handles secret management
2) there is a vault in the edge cluster that delegates all secret management to another vault instance
3) there is no vault in the edge cluster, and the cluster is configured to use the vault in the management hub, or somewhere else

Service Testing with the mock Agent

Testing a service implementation (hzn dev service start/stop) does not have the benefit of a management hub to support it. hzn dev service start/stop will allow the user to provide upstream creds directly, essentially providing the resolution of secret names to secret details without the need for a vault. It's a testing environment after all.

See the "Command Line Interface" for details of the changes to the hzn dev commands.

Validation of Secrets

The CLI commands for publishing services, deployment policy and patterns, as well as the deploycheck command need some level of verification that:
a) secrets are supported in the management hub, and
b) that required secrets are defined in the vault.

Since the Agbot is the only management hub component that interacts with the vault, it will be enhanced to provide APIs that the CLI can use to validate secrets. The CLI commands need to know the URL of the agbot in order to call an API that can validate secrets and secret bindings. To support this, a GET API is added to the exchange that returns OH management hub URLs. See the "APIs" section for details on this API.

User Experience

<Describe which user roles are related to the problem AND the solution, e.g. admin, deployer, node owner, etc. If you need to define a new role in your design, make that very clear. Remember this is about what a user is thinking when interacting with the system before and after this design change. This is not about a UI, it's more abstract. This should explain the aspects of the change that will surface to users.>

UX-1: As a service developer, I want to specify the need for a secret used to authenticate to another service or API.

UX-2: As a service developer, I want to specify the type of secret (e.g. basic auth, certificate, etc) that should be associated with a secret defined in my service definition.

UX-3: As a service developer, when my service starts I want my service to be given the secrets it needs to authenticate with upstream services.

UX-4: As a service developer, I want to test my service with hzn dev commands such that my service can invoke upstream cloud services which require authentication.

UX-5: As a service developer, I want my service to be given an updated secret if the secret expires or is revoked and has been subsequently replaced with a valid secret of the same type.

UX-6: As a service deployer, I want to assign a secret from the vault to a service without exposing the details of the secret.

UX-7: As a service deployer, when I am associating vault secrets with services, I want to be sure that I am associating a secret with the correct type.

UX-8: As a secret administrator, I want to manage (create, update, and delete) secrets that represent credentials used by edge services to authenticate with upstream services.

UX-9: As a secret administrator, I want to revoke a secret (which is never replaced) resulting in services that depend on that secret to be undeployed.

UX-10: As a secret administrator, I want to replace a secret which causes existing deployed services to be updated with the new secret, without having to involve the service developer or service deployer.

UX-11: As a management hub admin, I want the Vault automatically installed and connected to the rest of the management hub.

UX-12: As a management hub admin, I want to config Vault in the management hub to integrate with another Vault deployment.

UX-13: As a service developer or deployer, i want problems surfaced to me that cause the service to not be published or deployed to a node.

User Interface

<Describe any proposed changes to any part of the UI. This can be abstract initially because the UX design team will engage in the details.>

Not applicable to OpenHorizon.

Command Line Interface

<Describe any changes to the CLI, including before and after command examples for clarity. Include which users will use the changed CLI.>

There are no OpenHorizon CLI commands for interacting with the vault. Interactions with the vault should be done using the vault's CLI and UI.

BP: i assume this means **hzn** needs to call the vault API or CLI and therefore we need new **HZN_VAULT_URL** and **HZN_VAULT_AUTH** environment variables and corresponding values in **/etc/default/horizon** and **hzn.json** ? **DAB:** No, I dont think so. The CLI will get the agbot API from the new exchange /hubapis resource, and the agbot will support (limited) API access to the vault for the CLI.

hzn dev service start/stop supports secret resolution by the user. The user can provide credentials on the hzn dev service command that are setup by the mock agent on the service container(s), so that the application code within the container can authenticate to a cloud service.

hzn dev service start --secret mysecret.txt

A new flag --secret allows the user to specify a file name containing the secret details needed for accessing an upstream service. This flag can be repeated once for each secret that the service needs. The file will be mounted to the service container at the same place it will be mounted by the real agent, so that the service implementation can be tested as if it were running in the real agent.

hzn deploycheck is enhanced to ensure that secrets are resolvable by a deployment policy or pattern. If a secret association refers to a vault secret that doesnt exist, then deploycheck will return an appropriate error. Further, deploycheck will also ensure that all of the deployed service's dependencies also have valid secrets configured. This same check is performed by the hzn exchange deployment addpolicy and hzn exchange pattern publish commands, so the code should be built so that it can be used in all 3 commands. This command will use the new /hubapis resource in the exchange to obtain the agbot API URL. This is accomplished by calling the agbot API GET /secrets/<vault-secret-name> to validate the secret. The hzn CLI does not have the URL to the vault.

hzn exchange deployment addpolicy and **hzn exchange pattern publish** are updated to verify that the service (and all of it's dependencies) being deployed has all of it's secrets associated with valid vault secrets. These commands will use the new /hubapis resource in the exchange to obtain the agbot API URL. The agbot API GET /secrets/<vault-secret-name> is used to verify that a vault secret exists. The hzn CLI does not have the URL to the vault.

hzn exchange service publish will verify that the use of secrets are not supported for services with only an edge cluster deployment configuration, an error is returned to the user in this case. A warning is given if secrets are defined for a service that has both cluster and non-cluster deployment config, indicating that the secrets are only supported for edge devices.

External Components

<Describe any new or changed interactions with components that are not the agent or the management hub.>

The Hashicorp Vault is a new optional component of the management hub. If configured, it needs to be started, stopped and health checked with the rest of the management hub.

The agbot needs to be configured such that it can invoke Vault APIs. The agbot needs authority to list secrets and extract secret details but does not need authority to modify or delete secrets.

If the vault becomes locked such that it cannot be accessed by the agbot, the management hub admin needs to be told because service deployment of services requiring secrets is going to halt if this happens.

Affected Components

<List all of the internal components in which this feature needs to be implemented. Include a link to the github epic for this feature (and the epic should contain the github issues for each component).>

Agent, agbot, exchange, e2edev, all in one management hub, CLI

See the rest of this document for details.

Security

<Describe any related security aspects of the solution. Think about security of components interacting with each other, users interacting with the system, components interacting with external systems, permissions of users or components>

The agbot needs to authenticate to the Vault as a trusted user. The agbot config will need credentials and a URL to the vault API. The agbot is the only hub component that interacts with the Vault directly.

The Agent will store secret details on the host file system, in the clear, but in root protected storage. As a result, readonly permission is not granted to any user or group outside of root.

BP: And the hzn command needs to access the Vault, right? **DAB**: No, see elsewhere for clarifications.

The application code running within a service container has readonly access to secret details configured for that service, but no others. Conversely application code running in a service container (running as root or non-root) is unable to read the secret details configured to any other service. In addition, non-root code running outside of a container is also unable to read the secret details configured to any other service.

APIs

<Describe and new/changed/deprecated APIs, including before and after snippets for clarity. Include which components are users will use the APIs.>

Exchange
The following exchange resources are being updated with new schemas as per the design above:
- service def
- deployment policy
- pattern

Add a new resource called /API:
GET returns a JSON encoded list of all management hub URLs.
200: CSS, Agbot and the UI URL, and its own exchange URL (why not?)

BP: why isn't the vault URL included in this?**DAB**: Because I think we need to protect the Vault API by disallowing components outside the hub from interacting with it. All access to it will be through the Agbot API.

{

```
  "exchange": "<exchange-url>",
  "mms": "<css-url>",
  "management-console": "<ui-url>",
  "agreement-bot": "<agbot-secure-api-url>"
}
```

URLs are given to the exchange via it's config file.

Agbot
Add a new resource called /secrets/<vault-secret-name>
GET returns a JSON encoded object indicating the existence or non-existence of the named secret.
200: Secret exists

```
{
  "exists":<boolean>
}
```

404: Secret does not exist
503: There is no vault component in the management hub. The caller should retry this API call a small number of times with a short delay between calls to ensure that the vault is really not there.

The agbot's non-remotable API /health is extended to report it's connection with the vault. A new field is added to the HealthTimestamps struct to indicate the last time the agbot successfully invoked the Vault API:

"lastVaultInteraction":<linux-epoch-timestamp>

Build, Install, Packaging

<Describe any changes to the way any component of the system is built (e.g. agent packages, containers, etc), installed (operators, manual install, batch install, SDO), configured, and deployed (consider the hub and edge nodes).>

The all in one management hub needs to be updated to (optionally) include the Hashicorp vault. By default the vault will be started, an option is provided to turn it off.

Documentation Notes

<Describe the aspects of product documentation and open source documentation that will be new/changed/updated. Be sure to indicate if this is new or changed doc, the impacted artifacts (e.g. KC, open-source, internal, etc) and links to the KC and the related doc issue(s) in github.>

Documentation is needed for:
1) The new secrets section in service defs, deployment policy and pattern.
2) The programming model for how the app receives updated secrets.
3) Tasks performed by the developer to enable usage of secrets in the app.
4) Tasks performed by the deployer to make secret associations.
5) Configuring the vault in the hub

Test

<Summarize new tests that need to be added as a result of this feature, and describe any special test requirements that you can foresee.>

1. Automated tests in e2edev include services that consume and verify secrets. Would suggest this is done using the usehello workload since it already invokes the ESS API.
2. Automated tests in e2edev for the CLI commands, extending the existing tests on the updated CLI commands.
3. Exchange automated tests for the updated resources.
4. Test with federated Vaults.
5. Edge cluster tests are not needed because edge clusters are out of scope.

Work Breakdown

- Add Hashicorp Vault to e2edev - only started when HZN_VAULT env var is set to something
    - vault config file is sourced in anax repo and templated

    https://github.com/open-horizon/anax/issues/2404
- Add Hashicorp Vault to all in one mgmt hub - only started when HZN_VAULT env var is set to something
    - vault config file is sourced in devops repo and templated

    https://github.com/open-horizon/devops/issues/46
- Update resources in the Exchange, tests for resource updates

    https://github.com/open-horizon/exchange-api/issues/491
- Add new API resource for management hub URLs in the exchange, tests

    https://github.com/open-horizon/exchange-api/issues/492

- Agbot update config for vault
- Add secure API for verifying a secret exists, add health API update
- Update hzn exchange service publish, document service def
- Update hzn exchange deployment addpolicy, document deployment policy updates
- Update hzn exchange pattern publish, document pattern updates
- Agbot add secrets to the agreement protocol
- Agbot extract secrets and send in proposal
- Agent receive secrets from protocol msg, store locally, cleanup when agreement terminates
- Agent mount secrets to service container, document application's use of this mount point
- Agent add /secret et. al. APIs, document APIs
- Agbot and agent - add agreement protocol extension to update existing agreement
- Agbot monitor secrets in the vault and send update agreement protocol extension messages
- Update hzn dev service new/start/stop to support secrets