**Title:**

Managing and Rotating Secrets with Azure KeyVault, Managed Services, and some automation – Part 1

**Teaser:**

Secret rotation is not a new problem. In cloud-based environments many services implement secret-based authentication schemes. For many organizations, these secrets must be rotated on a regular schedule. In addition to the actual problem of rotating the access keys, there exists a problem of how these newly rotated credentials are propagated to all the applications and systems that utilize them. In this multi-part blog series, I will discuss a solution that, through automation, addresses both the scheduled rotation and dependency notification/update requirements.

**Body:**

Many organizations have long standing security mandates to rotate application secrets. These secrets can range from specific identify passwords to service access keys. As more enterprises move their workloads to the cloud, the need for adherence to these policies becomes more apparent. More and more workloads take advantage of a multitude of PaaS services, many of which utilize different access control schemes. As the number of such services and custom applications that rely on them grows, IT administrators are often left to their own devices to figure out how to manage periodic rotation of the multitude of secrets as well as how to update their dependent applications when such change is made.

Azure offers some automation to help solve a portion of these problems, specifically [automated storage account rotation by Key Vault](https://docs.microsoft.com/en-us/azure/key-vault/secrets/overview-storage-keys) and general guidance on how to use [automation to solve these types of problems](https://docs.microsoft.com/en-us/azure/key-vault/secrets/tutorial-rotation-dual) for other services. Unfortunately, this is often not enough to ease the tasks associated with managing this problem space. Not only are these solutions insufficient in addressing a multitude of services organizations use in the cloud, they also do not address the “update dependent applications” part of the problem at all.

Having worked with multiple Azure customers who have faced these challenges, I decided to address this whitespace by building a solution to tackle this specific area.

I am not that creative when it comes to naming things, so after a short deliberation period I went with my go to approach for naming projects – Greek / Roman Mythologies. And that is how project “[Harpocrates](https://en.wikipedia.org/wiki/Harpocrates)” was born. I have many ideas on how this solution could be distributed in the future, but for now, its source code is available on [GitHub](https://github.com/gkli/harpocrates). At the time of this writing, the codebase is in a rather early stage. The solution works and can rotate secrets for several different services. The code, however, can stand some refactoring and additional features, which I plan on working on over time. So for all intensive purposes, this should be treated as work in progress.

At the root of the problem Harpocrates addresses is a relatively simple business process depicted below.



Figure Secret Rotation Business Process

Incorporating this business process with the [guidance given by Azure](https://docs.microsoft.com/en-us/azure/key-vault/secrets/tutorial-rotation-dual) one can utilize the following high level flow



Figure Harpocrates Logical Flow

Using the two diagrams depicted as the basic premise for Harpocartes we have an application that can monitor events raised out of Key Vault. At the time of writing, Harpocrates can act on 3 specific events: “Secret Expired”, “Secret Expiring”, and “Secret Created”. In future, there is a desire to add management of KV Keys & Certificate objects as well. Harpocrates is specifically built to automate the following logical flow.



Figure Harpocrates Business Process

To allow for extensibility, specifically around solution’s ability to manage a multitude of downstream services, including those hosted in other clouds as well as custom applications, Harpocrates takes a factory-based approach to interactions with these services’ management endpoints. Thera are concrete implementations for management of Azure Storage Accounts, Azure CosmosDB, Azure Application Registration (Service Principals), etc. These constructs can be easily extended to add “Secret Management Providers” for AWS S3 buckets or your custom application. The only assumption Harpocrates makes is that the downstream services management endpoints are available for its consumption. This means the management endpoint needs to be addressable on the network and Harpocrates will have permissions to invoke the relevant functionality. For the purpose of this solution, the term “management endpoint” is being used loosely to represent an entity the application can interact with to manage the service secrets. This can be a restful endpoint, a custom API, or simply a file path where config files are stored, thus allowing for a high degree of extensibility.

Harpocrates is also being designed to support deployments into various compute platforms such as AKS, App Service, VM, or even ACI or Azure Functions, as well as support a number of different storage technologies as to not place a significant burden on deploying the solution into your own environment. Depicted below are high level deployment diagrams of Harpocrates in AKS and App Service. These are just examples as the solution can be deployed into serverless environments as well. I plan on having an article in this blog series dedicated to specific deployment scenarios.



Figure Harpocrates Azure Kubernetes Service Deployment Diagram



Figure Harpocrates Azure AppService Deployment Diagram

If you’re still reading this and are interested in learning more about Harpocrates and its journey to addressing this problem space, please take a look at other articles in this blog series and as always, feel free to browse the codebase and provide feedback.

Post # 2

In the previous post I discussed the bigger picture for Harpocrates’ existence and goals. In this post, I will begin by diving into the constructs defined by Harpocrates and how they can be leveraged to achieve the desired configuration for your environment.

As discussed before, Harpocrates addresses the need for a periodic rotation of secrets. And as such, we need a primitive to describe this “period”. To allow for support of other governing constructs in the future, Harpocrates utilizes a concept of a “Policy”. This primitive is meant to enable the system administrator to define a set of rotation intervals that can then be applied to other constructs in the system. At the time of writing of this post, the Policy is only used to define a rotation interval, though it could be extended to add additional governing properties.

Policies are applied to specific services. This means a specific storage account can have a “15-day” rotation policy, while another one can have a “90-day” policy applied to it. This enables the administrator to describe how frequently a particular service needs to have its credentials rotated. This brings us to the next primitive: “Service”. A “Service” primitive is used to describe a specific service you want Harpocrates to manage. For example, you may want to enable Harpocrates to rotate keys for “Storage Account A”, and as such, you would create a “Service” definition for such service. “Service” definition would also provide additional metadata about the given service, such as the type of the service this is (i.e. storage account, cosmosdb account, Redis cache, etc); and lastly, a “Service” will be configured w/ a “connection string” that would describe how this particular service could be managed by the secret management provider. In most cases, this connection string would simply contain a resource group and the service uri. However, it is possible to add additional tokens to this piece of metadata, as long as the syntax used to define it is understood by the respective secret management provided. For example, for Azure Storage Account secret management provider, the connection string looks like this: “AccountEndpoint=https://storageaccounta.core.windows.net;ResourceGroup=myrg;”. If you had extended the secret management provider by adding your own custom provider, the connection string used to configure such service would simply need to match the syntax your customer provider knows how to interpret.

Harpocrates uses Azure Key Vault as the secure store for service secrets as well as the eventing mechanism to signal when a specific secret is to be rotated. In Key Vault, secrets are identified by a unique Uri. Here’s an example of a Uri for KV Secret named “my-secret”: [https://mykevault.vault.azure.net/secrets/my-secret/{version id}](https://mykevault.vault.azure.net/secrets/my-secret/%7bversion%20id%7d). Since each KV secret has a unique, well defined uri, we can use that Uri to correlate secrets stored in KV to the services we defined in Harpocrates. To do this, we need a third primitive: “Secret”. As you have probably guessed, “Secret” will provide “mapping information”, allowing Harpocrates to “translate” a KV uri to a service we’re trying to manage. In addition to this important piece of information, a “Secret” also contains information about the “type” of secret it is, as well as the format in which this secret is to be stored in KV (more on this later). There are currently two types of secrets supported by Harpocrates: Attached & Dependency.

“Attached” secrets represent the entries in KV that store raw values of the downstream service secret. For example, when creating management metadata for “StorageAccountA”, one would create a Key Vault secret named “my-storage-account-a-master-key”. This KV Secret would be associated with a Harpocrates “Secret” “StorageAccountASecret” that would in turn be associated with the “StorageAccountA” service definition and be of type “Attached”. Since “StorageAccountA” service definition is associated with a particular “Policy”, that policies will be used to set the expiration date of the key vault secret named “my-storage-account-a-master-key”, and as such, would cause this secret to expire, firing a KV event that Harpocrates can respond to by rotating the storage account key & updating the associated KV secret. The above described flow is depicted below



“Dependency” secrets represent entries in Key Vault that depend on the value of another secret. This type of a secret would typically be used to define a secret consumed by an application that needs credentials to access one of the services managed by Harpocrates. Most application do not store secrets in plain form but rather use a specific format, such as a connection string. In a typical application, one would likely see a configuration entry in a form of a database connection string, rather than multiple entries each representing the: server name, username, user password, etc. As such, “Dependency” secrets support a concept of a Format Expression. When setting up a “Dependency” secret, a system admin would provide Harpocrates with an expression that matches the expectations of the consuming application and may look something like this: “Server={{server-secret-id}};User={{user-secret-id}};Password={{password-secret-id}}”. The values specified in the “{{“ “}}” are system generated ids of the secrets managed by Harpocrates that can be resolved at time of dependency update, thus allowing the actual KV secret value to be: “Server=myDbServer;User=MyDbUserName;Password=P@ssword@1”.

Now that we have the language to describe a Policy, Service, and Secrete, we need one last construct to simplify the dependency secret resolution: Secret Dependency. This is not as much of a primitive as just a relationship that is defined in the system. When a secret has dependencies and that secret has its associated service keys rotated, the system can schedule updates to all other secrets that depend on it. Since dependencies do not technically require that they depend on “attached” secrets, one could design a pretty extensive cascading secret rotation strategy to be executed by Harpocrates. The above described flow is depicted in the diagram below.



Now that we have the vocabulary required to describe the behavior we want Harpocrates to automate, we can discuss the runtime requirements and assumptions made by Harpocrates. We will do this in the next installment of this blog series. As always, your feedback is welcomed and greatly appreciated.

Blog 3: runtime requirements and assumptions

Blog X: AKS Deployment

Blog X+1: App Service Deployment

Blog X+2: Azure Functions Deployment

Blog X+3: ACI Deployment

Other Topics To Cover: