Vault: Cubbyhole Authentication Principles

NOV 04 2015 JEFF MITCHELL

PLEASE NOTE: This post is kept for historical purposes only and to avoid breaking inbound links. The paradigms discussed here have been extended, formalized, and standardized into a built-in Vault feature called *Response Wrapping*. Check out the Response Wrapping concept page for more information.

Last Updated: 2016-04-27

In the Vault 0.3 release post, the `cubbyhole` backend was introduced, along with an example workflow showing how it can be used for secure authentication to Vault. Here at HashiCorp, we believe that Cubbyhole-based authentication is the best approach for authenticating to Vault in a wide variety of use-cases. In this post we will explain why Cubbyhole may be the right authentication model for you, and present multiple considerations around Cubbyhole authentication to help fit a number of real-world deployment scenarios.

This post will first explain the motivation behind developing the Cubbyhole authentication model, then describe the model itself, and finally present some considerations for designing deployment scenarios.

Please note: Cubbyhole authentication provides a set of useful security primitives. The information in this post is intended to be useful, but it is not exhaustive and should not be taken as a recommendation of one approach or another. It is up to the implementor to ensure that any given authentication workflow meets their organization's policy.

Why Cubbyhole?

In Vault, there are two main types of authentication backends available:

1. User-oriented authentication backends: These generally rely on knowledge of a shared secret, such as a password for `userpass` and `ldap` or a GitHub API token for `github`. This shared knowledge is distributed out-of-band.
2. Machine-oriented authentication backends: These rely on some intrinsic property of the client (such as a machine MAC address for `app-id`) or a secret distributed out-of-band (such as a TLS private key/certificate for `cert`) in order to facilitate automatic authentication of machines and applications.

User-oriented backends are fairly straightforward and well understood, as they map to authentication paradigms used by many other systems.

However, machine-oriented authentication is more difficult in any circumstance, and Vault is no exception:

- Many intrinsic properties of a machine are discoverable by other machines. For instance, the `app-id` backend relies on a machine knowing some unique property of itself as one of the shared secrets. The given suggestion is to use the MAC address of a machine as a unique machine property, or a hash of it with some salt. Without a salted hash, the MAC address is discoverable to any other machine on the subnet. If you are using a salted hash, you need a secure way to distribute the shared salt value to the machine, placing you into an out-of-band secret distribution scenario (see below). Choosing a value that is not discoverable to other machines often means that it is not easily discoverable to the process populating `app-id` either, again resulting in an out-of-band secret distribution problem.
- Out-of-band distribution relies on having a secure channel to a machine. If such a secure channel exists, for instance to distribute a private key for a certificate to allow a client to use `cert` authentication, it may as well be used to distribute a Vault token. There are additional complications when you want to use Vault itself to establish the secure channel, such as using the `pki` backend to issue client/server TLS certificates to enable further secure communications.
- Both kinds of secrets are vulnerable to operator snooping. For instance, using `app-id` as an example once again:
    - The MAC address of a machine (an intrinsic property) is almost certainly known to operators.
    - The other half of `app-id` authentication is a unique UUID for the machine (an out-of-band distributed secret), suggested to be stored in configuration management. However, storing this value in configuration management often means that it is stored in plaintext on-disk (likely in many places if using a DVCS to version configuration management information, as is very common). Even if it is encrypted on-disk (as most configuration management utilities are able to do), any operator or service that is used to deploy to a machine must know the decryption key, which itself is a secret that must now be protected and securely distributed.

There is no perfect answer to automatic secret distribution, which in turn means that there is no perfect answer to automatic machine authentication. However, risk and exposure can be minimized, and the difficulty of successfully using a stolen secret can be heightened. Using the `cubbyhole` backend along with some of Vault's advanced token properties enables an authentication model that is in many cases significantly more secure than other methods both within Vault and when compared to other systems. This in turns enhances Vault's utility as an access control system to secrets stored both within Vault itself and other systems (such as MySQL or PostgreSQL or Cassandra), or even for directly accessing other systems (for instance, via SSH or TLS certificates).

The Cubbyhole Authentication Model

As a quick refresher, the `cubbyhole` backend is a simple filesystem abstraction similar to the `generic` backend (which is mounted by default at `secret/`) with one important twist: the entire filesystem is scoped to a single token and is completely inaccessible to any other token.

Since many users are using Vault secrets to establish secure channels to other machines, we wanted to develop an authentication model that itself does not rely on a secure channel (but of course can be enhanced by a secure channel).

Instead, the model takes advantage of some of the security primitives available with Vault tokens and the Cubbyhole backend:

- Limited uses: tokens can be valid for only a certain number of operations
- Limited duration: tokens can be revoked after a configurable duration
- Limited access: only a single token can be used to set or retrieve values in its cubbyhole

Using these primitives, and with input and feedback from security experts both in the field and at several commercial companies, we constructed a new authentication model. Like all authentication models, it cannot guarantee perfect security, but it sports a number of desirable properties.

The model works as follows, supposing that we are trying to get a Vault token to an application (which could be a machine or container or VM instead):

1. A process responsible for creating Vault authentication tokens creates two tokens: a permanent (`perm`) token and a temporary (`temp`) token. The `perm` token contains the final set of policies desired for the application in the container. The `temp` token has a short lease duration (e.g. 15 seconds) and a maximum use count of 2.
2. The `temp` token is used to write the `perm` token into the cubbyhole storage specific to `temp`. This requires a single write operation, reducing the remaining uses of `temp` to one.
3. The invoked process gives the `temp` token to the application management engine (for instance, Docker), which starts the application and injects the `temp` token value into the application environment.
4. The application reads the `temp` token from the environment and uses it to fetch the `perm` token from the cubbyhole. This read operation exhausts the `temp` token's use limit, the `temp` token is revoked, and its cubbyhole is destroyed.

Let's take a look at some of the properties of this method:

- Even if the value of the `temp` token is passed in cleartext to a process, the value of the `perm` token will be (should be!) covered by TLS.
- The value of the `temp` token is useless after it has either expired or been revoked, and outside of the target application's memory, the value of the `perm` token is lost forever at that point because the `temp` token's cubbyhole is destroyed. Accordingly:
    - A `temp` token written to disk in order to pass it to an application presents no long-term security threat.
    - If the value is logged (for instance, because it was handed to a container in its environment variable settings), the value in the log presents no long-term security threat.
- Accesses by bad actors are detectable, because the application will be unable to use the `temp` token to fetch the `perm` token, at which point it can raise an alert.
    - Because accesses to the cubbyhole are logged in Vault's audit log, an operator can then use the audit log to discover whether an application took too long to start (only one instance of the cubbyhole being accessed will appear in the audit logs before revocation) or whether another process used the token (two instances will appear in the audit logs) to steal the `perm` token.

Of course, depending on the particular setup, there is still the potential for malfeasance, but there are also mitigations:

- If the disclosure of the `temp` token to the application happens over an insecure channel, it may be subject to sniffing.
    - Some of the considerations detailed in the next section provide defense against this.
    - If the sniffer actually fetches the value of the `perm` token, this is detectable.

- If an operator or another machine is able to intercept the `temp` token value while it is still valid (for instance, by reading an application's environment immediately after startup, or if logs containing the environment are available within the `temp` token's time-to-live), they can use this to retrieve the value of the `perm` token.
    - This is detectable as well.

No authentication mechanism is perfect. However, limited uses, limited duration, and limited access together form a powerful set of security primitives. The ability to detect accesses by bad actors is also an extremely important property. By combining them, the Cubbyhole authentication model provides a way to authenticate to Vault that likely meets the needs of even the most stringent security departments.

Cubbyhole Authentication Considerations

A natural question at this point is "who or what is responsible for creating the `temp` and `perm` tokens?" Because this authentication workflow is simply a model, there isn't a single structured way to use it. The flip side, however, is that there is a lot of flexibility to easily craft the solution that meets your needs.

This section will present a few considerations targeted to different deployment and operation models. They are not exhaustive, but should provide ideas for implementing Cubbyhole authentication in your own Vault deployment.

Token Distribution Mechanisms

Pushing

In a normal push model, tokens are generated and pushed into an application, its container, or its virtual machine, usually upon startup. A common example of this would be generating the `temp` and `perm` tokens and placing the `temp` token into a container's environment.

This is a convenient approach, but has some drawbacks:

- Environment information is often logged at application or container startup, and may be accessible by others within a short time frame.
    - However, as with all Cubbyhole models, malfeasance is detectable; if a bad actor gains access to the environment and manages to consume the `temp` token before the intended target, the target can log a security alert that its given token was invalid.
- If an application fails and it (or its container) is restarted, it may see an out-of-date `temp` token in its environment.

Additionally, it may be difficult or impossible to configure the provisioning and/or orchestration system to perform this functionality, especially without writing a framework, executor, or other large chunk of code.

As a result, this is a conceptually easy approach but can be difficult to implement. Often, a hybrid approach with a push model as well as a pull or coprocess model will be the right approach.

Pulling

In a pull model, the application, upon startup, reaches out to a token-providing service to fetch a `temp` token. Alternately, rather than coding this logic into each application or container, a small bootstrapping application could perform this task, then start the final application and pass the value of the `perm` token in. The same bootstrapping application could be used across machines or containers.

The pull model provides some benefits:

- It requires constructing a simple token providing service rather than potentially modifying an application scheduling system.
- If an application fails and it (or its container) is restarted, it can pull another token in rather than seeing an out-of-date `temp` token in its environment.

However, there is a major drawback to the pull model as well: it requires a token providing service to implement some logic and heuristics to determine whether a request is valid. For instance, upon receiving a request from a specific IP and with a specified application, it could check with the application or container scheduler to determine if, in fact, that application was just spun up on that node. This may not work well if the application has a runtime manager that restarts it locally if it fails.

Coprocesses

A third approach uses a coprocess model: applications implement a known HTTP endpoint, then an agent, usually on the local machine, pushes a token into the application.

At first, this approach might seem like it combines the complexity of implementing both a push and a pull model. However, its advantages are enough to warrant serious consideration (the following points assume that the token-providing service is on each local machine, although it does not have to be):

- Unlike the pull model, the coprocess does not need to trust that the calling application is what it says it is -- it sends a token to the known good location of the application. If that application did not request a new token, it can send an alert to trigger an investigation.
- If the application is running in a container, a local coprocess could write the token to a directory on the host file system created specifically for that container and bind-mount it into the container upon startup, rather than pushing it in via environment variables that are more likely to be logged.
- If the application is running in a chroot, a local coprocess could write the token into the chroot before application startup.
- The coprocess can be responsible for generating tokens at predictable intervals and storing them into the appropriate file system location, whether in a Docker container, chroot, or simply a directory with appropriate access permissions. Then the application only needs to implement a simple watch on the file to see if it has been updated, and to fetch its new Vault token when that happens.

An example of a coprocess-based approach would be a coprocess that watches the local Docker event API to determine when new containers have started, then generates a `temp`/`perm` token pair and sends the `temp` token to the well-known endpoint of the application in the new container. The coprocess itself would need a token; this could also use Cubbyhole authentication, with the `temp` token injected via a push approach to the host, perhaps from a central coprocess that manages host authentication.

Overall, a coprocess-based approach requires little support from the provisioning and orchestration systems and minimal changes to applications, which can usually be implemented with a common library. As a result, it generally offers the best combination of flexibility and operational robustness.

Token-based Authentication Pros and Cons

Finally, some remarks on the differences between using Cubbyhole authentication, which is *token-based*, versus a normal authentication backend, which is *credential-based*.

Tokens carry with them some restrictions:

- When the TTL of a token expires or is otherwise revoked, any child tokens it created will be revoked as well.
- Child tokens must contain a subset of their parent's policies.

Compare this to an authentication backend, which can create tokens up to a lifetime determined by the mount properties or system configuration, and with any set of policies as configured by a backend administrator.

Of course, whether these restrictions are pros or cons depends on your own needs. However, looking at the restrictions another way:

- A parent token's TTL expiring forces all tokens to eventually rotate, preventing accidental long-lived tokens (at least, longer than the token used to create the `perm` token)
- A backend cannot accidentally or maliciously be configured to associate more (or more permissive) policies with a token, because the set of available policies is constrained by the parent. In fact, a token-providing service could have multiple tokens each associated with a limited set of policies, and use the correct token to create the correct child token depending on the needs of the application.

If you want the benefits of the restricted set of policies, but not tie child tokens to the lifetime of the parent token, Vault 0.4 will contain a `auth/token/create-orphan` endpoint which will allow tokens with policies granting them access to this endpoint to create orphan tokens without requiring `root` or `sudo` access to `auth/token/create`.

Share the Knowledge

We think that the Cubbyhole authentication model will be extremely useful to many organizations. We hope that as our users deploy their authentication solutions based on Cubbyhole, they will provide feedback, tips, and useful code to the community. Please be sure to spread the knowledge around on the vault-tool mailing list!