**HashiCorp Vault**

Today I want to spend some time talking about [HashiCorp Vault](https://www.hashicorp.com/products/vault).

When we talk about Vault, the problem we're talking about solving is the secrets management problem. When we start talking about secrets management, the first question that naturally comes up is, "What is a secret?".

When we talk about secret management what we're really talking about is managing a set of different credentials. What we mean when we talk about these credentials is anything that might grant you authentication to a system or authorization to a system. Some examples of this might be usernames and passwords, it might be things like database credentials, it might be things like API tokens, or it might be things like TLS Certificates.

The point is, any of these things, we can use these to either log into a system and authenticate, such as a username and password, or we're using it to prove our identity like a TLS certificate, and so we're using it authorize access, potentially.

All these things fall into the realm of secrets, and these are things we want to manage carefully. We want to understand who has access to them, we want to understand who's been using these things, and in the case of most of these, we want some story around how we can periodically rotate these.

When we look at the state of the world of how these things get managed, in practice, what we see is secret sprawl. What we mean by secret sprawl is that these end up everywhere. They're in plain text inside of our source code, so maybe it's hardcoded in a header what the username and password is. It ends up inside of things like configuration management.

Again, this is living in plain text, and Chef, or Puppet, Ansible, so anyone can log in and see what these credentials are. Ultimately, all this typically ends up living in a version control system like GitHub, or GitLab, or Bitbucket. These things end up strewn about or sprawled all around our infrastructure.

What are the challenges with this world? Well, we don't know who has access to all these things, so we don't know, "Does anyone at our organization with access to GitHub, can they log in and see the source code and thus see what our database credentials are?".

Even if they could do it, we don't know if they have done it. We have no audit trail that says just because I, Armon, could have seen that secret, did he go and access it? So, we have no fine-grain ability to manage who has access or even to audit who's done what with it.

Worse yet, how do we rotate any of these things? If we realize we do need to change our database credential, there's been a compromise, or we're doing a periodic rotation, it becomes very, very difficult if we're in a place where it's hardcoded in our source code or it's strewn about in so many different systems. It's difficult to know how to do this rotation effectively.

This state of the world is what we refer to as secret sprawl. One of our first goals when we started working on Vault was to really look at this problem and say, "How can we improve it?".

This is really where Vault came from. Vault really starts by looking at the secret sprawl problem and saying, "We can only solve it by centralizing". Instead of having things live everywhere, we move all these secrets to a central location, and Vault promises that we're going to encrypt everything both at rest inside of Vault as well as in-transit between Vault and any of the clients that want to access it.

This gives us a few properties. One, unlike these systems where we were storing this stuff in plain text, at least now, if you could see where the secret is stored at rest, it's encrypted. So, you don't get implicit access to just be able to see the secret.

The next thing is Vault lets us overlay fine-grain access control on top of all this. Instead of being anyone in our organization who has access to GitHub and can see the source code, now we can go much more fine grain and say, "The web server needs access to the database credential, the API server needs the API tokens, but everyone shouldn't have access to everything".

Then, on top of this, we have an audit trail. Now we can see what credentials did the web server access, what credentials did Armon access from the system. We have much more visibility and control over how these things are all being managed.

This is the level-one challenge with Vault, is at least moving from a world of sprawl, where things are everywhere, to a world of centrality, where we have strong guarantees that it's encrypted, strong guarantees around who has access, and strong visibility into this.

This becomes our first level thing. The next-level challenge becomes realizing who we're giving these credentials to. So, great, we've stored all these credentials safely in Vault, and now we're going to thread these out and provide it to an application.

The challenge is applications do a terrible job keeping secrets. Inevitably the application will log its credentials out to a logging system so that it might write it out to standard out, this gets shipped off to Splunk, and is now in a central log that anyone can see.

It shows up in any sort of diagnostic output, so maybe our application has an exception, and it shows the username and password and the traceback or inside of an error report. It might be shipping it out to external monitoring systems when there is an error.

In general, what we find is applications do a poor job keeping things secret. So even if we do a great job centralizing it, and strongly controlling it, and encrypting it on the way to the application, the app isn't trusted.

One of the second-level capabilities Vault introduces is what we call dynamic secrets. The idea behind a dynamic secret is, instead of providing a long-lived credential to the application which it inevitably leaks, we provide short-lived ephemeral credentials. These things are dynamically created, but they're ephemeral. We might only give a credential to an application that's valid for, say, 30 days.

The value of this is a few folds. Now, even if the application leaks this credential out, it's only valid for a founding period of time. It might write it to a logging system, and that becomes visible, but we create a moving target for an attacker by constantly revoking and issuing new certificates.

The other thing that's valuable is now each credential is unique to each client. Previously, if I had 50 web servers, all of them would come in and read a static database credential. This means if there is a compromise and that database credential gets out, it's very hard to pinpoint where the point of compromise was. Fifty servers are all sharing the same credential, versus in a dynamic secret world, each of those 50 web servers had a unique credential. So, we know very precisely that web machine 42 was the point of compromise.

The last thing that this lets us do is to have a much better revocation story. Now if we know web machine 42 was our point of compromise, we can revoke the username and password for just web machine 42 and isolate that leak. But if all 50 machines were sharing the same username and password, the moment we try to revoke it, we'd cause the entire service to have an outage. So, the blast radius of a revocation is much larger when you have a shared secret versus a dynamic secret.

The third challenge we found was that applications are often storing data ultimately. The challenge becomes, how do the applications protect their own data at rest? Because we're not going to be able to store all information within Vault; Vault is meant just to manage secrets, not anything that might be confidential.

What we often see is that one, Vault is being used as a centralized secret management store, people are storing encryption keys. We might put an encryption key inside of Vault and then distribute that key back out to the application; the application is doing cryptography to protect data at rest.

What we find, though, is applications generally don't implement cryptography correctly. There are lots of subtle nuances and it's easy to get wrong. With these kinds of mistakes, often it compromises the whole cryptography when those mistakes are made.

One of the challenges we often look at is, how do we get away from Vault just storing an encryption key and handing it to the application and assuming the app will do cryptography right?

This has evolved into a capability the Vault calls "encrypt as a service". The idea here is, instead of expecting that we're just going to deliver a key to a developer and the developer implements cryptography correctly, Vault will do a few things. One is it will let you create a set of named keys. I might create a key that I call "credit card information," and a second one I call "social security number," and one for "PII".

These are just names. I'm going to just name this key, and I'm not going to give this value out. But then what we expose is a set of high-level APIs to do cryptography. These APIs would be the classic operations you expect. Things like to encrypt, or decrypt, or sign, or verify.

Now, as a developer, what I'm doing is calling Vault with an API and saying, "I want to do an HMAC using my credit card key and some piece of data". What Vault is shielding is, Vault is providing the implementation, so we don't have to trust the developers implemented these high- level operations correctly. And the key management is also being provided by Vault. The developer never actually sees the underlying key.

This lets us do a few things. One, it ensures that the cryptography's correctly implemented because we're using a vetted implementation by Vault. This implementation's vetted both by us, by the open-source community, and by external auditors that we use.

It also lets us offload key management. If we think cryptography is hard, key management's even harder. In practice, when you ask how many applications properly implement key versioning, key rotation, key decommissioning, and the full lifecycle of key management, the answer is very few because it's challenging. But, by offloading this to Vault, we can use high-level APIs to do all of this. We get the full key lifecycle as well provided by Vault.

So, in practice, these end up being the three major challenges that we're trying to help developers with. How do we move these credentials out of plain text and sprawled across many different systems into a scenario where they're centrally managed with tight access control and clear visibility? Then, how do we go even further and protect against applications that aren't necessarily to be trusted in keeping secrets?

We do this by being ephemeral. We create this moving target where what we're managing is that the web server should have access to the database and that credential that enables it is a dynamic one instead of static. Then, lastly, how do we go further in helping the application protect its own data at rest? That's done through a series of key management and high-level cryptographic offload. These three are the core principles of Vault.

Now maybe we'll zoom in quickly and talk a bit about high-level architecture of, how does this get implemented? When we talk about Vault's architecture, there are a few important things to realize. One is that Vault is highly pluggable. It has many different plugin mechanisms.

When we talk about Vault, it has the central core, which has many responsibilities, including the lifecycle management ensuring requests are processed correctly, and then there are many different extension points that allow us to fit it into our environment.

The first one that's extremely important is the authentication backends. These are what allow Vault to allow clients to authenticate from different systems. For example, if we're booting an EC2VM, this EC2VM might authenticate using our AWS authentication plugin. This plugin allows us to tie back into Amazon's notion of identity to prove that the caller is, for example, a web server.

But if we have a human user, they might be coming in and using something like LDAP or Active Directory to prove their identity. If we're using a high-level platform, maybe something like Kubernetes, we might be using our Kubernetes authentication provider.

The goal of these authentication providers is to take some system we trust, whether it's Kubernetes, LDAP or AWS, and use this to provide application or human identity. That's what we're getting out of this: a notion of the identity of the caller.

This is great, and then we use that to connect to an auditing backend, which allows us to connect and stream out request/response auditing to an external system that gives us a trail of who's done what. This might be Splunk, as an example, where we're going to send all our different audit logs. Vault will allow us to have multiple different audit logs. We can also send to Splunk as well as a system like Syslog, as an example.

The next-level challenge is, where does Vault store its own data at rest? If we're going to read and write secrets to Vault, it needs to be able to store these things somewhere. These are what we call storage backends.

Storage backends are responsible for storing data at rest. This can be a couple of different things. It could be a standard RDBMS in a MySQL Postgres. It could be a system like Consol, and it could be a cloud-managed database like Google Spanner. But the goal of these backend systems is to provide durable storage in a way that's highly available so we can tolerate the loss of one of these backend systems.

The last bit is, how does Vault provide access to different secrets? These are the secret backends, themselves. These come in a few different forms. The biggest use of these is to enable the dynamic secret capability we talked about before.

One form of secret backend is a simple one. It's just key value. I might just store a static username and password in there, and I'm giving it a username and a password, and these things are static. This is just a key value store that's encrypted at rest. However, as we get more sophisticated, we might want to use the dynamic secret capability we talked about. That is where these different plugins start coming in.

We have different database plugins; database plugin will allow us to dynamically manage MySQL, and Postgres, and Oracle, et cetera credentials. We have things like RabbitMQ, so maybe we're doing dynamic credentialing for our message queues.

This goes on. You can even apply the same principle to something like AWS. We might have applications that need to read and write from S3, but we don't want to give them long-lived access to IM. Instead, we define a role in our AWS backend, and we'll go and dynamically generate short-lived credentials as needed. This extends that dynamic secret paradigm.

This is an extension point that allows Vault to apply this same principle to many different things. One common use of this is PKI. In practice, certificate management tends to be a nightmare, and what we often see is very long-lived certificates; maybe five- to 10-year-lived certificates because we don't want to go through the process of generating them. Versus with Vault, we can find them and programmatically generate it. In practice, people will use very short-lived certificates. Maybe as short as 72, 24 hours. This way, you're constantly moving, creating a moving target.

This list goes on and includes things like SSH as an example. We can broker access to SSH as well, so you don't have a single PIN to rule them across a large state of machines.

At its core, this is what makes Vault so flexible. It allows Vault to manage clients that are authenticating against a different set of identity providers. We can audit against a variety of different trusted sources of log management, we can store data in almost any durable system, and then we can extend the surface area of what types of secrets can be either statically or dynamically managed by adding new secret backends.

This becomes Vault in a single-instance nutshell. As we talk about running a Vault instance, each instance of it is one of these. Then, in a broader deployment, what this will look like is we run multiple Vault instances to provide high availability. At the highest level, we'd have a shared backend. For example, this might be Consol, which internally is three different servers, as an example, providing us HA. Then we will run multiple Vaults in front.

What Vault does is it will coordinate with the shared backend to perform a leader role action. One of these might be elected our current leader, and so as a client, when we're making requests, we're talking to the leader. Even when we speak to a non-leader, we'll be transparently forwarded to the active leader.

In this way, if any node dies, power loss, process crashes, maybe network connectivity is an issue, we will detect this and promote a new one to leader automatically, and this instance takes over an active operation, and our other secondaries will begin to promote.

This is what Vault looks like at a high level. It operates as a shared-network service, and we're talking to it just as an API client over the network. What Vault typically exposes is a restful JSON API. So, it's JSON over HTTP, making it relatively easy to integrate with our applications.

I hope this was useful as a high-level introduction to Vault. Please check out our other resources to learn more. Thank you.