**HashiCorp Boundary**

The complexity of managing traditional models for private resource access is well known. Firewalls, VPNs, SSH basions / jump hosts: getting credentials for all of these through a private network introduces a few problems:

1. IP-based management doesn't scale when dealing with dynamic infrastructure or a larger infrastructure footprint.
2. These traditional controls often lack API-driven controls and require manual configuration.
3. Access policies don't change often in this model because they're so difficult to manage. This leads to standing privileges that increase the threat of compromise.
4. Traditional models limit how precisely privileges can be defined on a per-identity or per-user basis.
5. Failure to effectively manage this matrix of controls, resources, and users in a traditional model can disrupt end-user productivity.

In this whiteboard presentation, HashiCorp co-founder and CTO Armon Dadgar will present the challenges and goals for secure sessions management that led HashiCorp to create the open-source project: [Boundary](https://boundaryproject.io/).

Boundary gives you ease of configuration with a fully API-driven service and the security enhancements from just-in-time authorization of each session. It enables authenticated and authorized TCP sessions to applications with role-based access controls (RBAC). Users can automate access management to dynamic targets with the [Boundary Terraform provider](https://github.com/hashicorp/terraform-provider-boundary), the API, or SDK. Boundary also supports monitoring and logging of session metadata.

We designed the architecture of Boundary to be easy to understand, highly scalable, and fault tolerant. Users can interact with Boundary through the CLI, API, or a web interface. It can run on-premises, in the cloud, or in secure enclaves, and it does not require you to install an agent on target hosts.

Learn more in this whiteboard presentation and transcript below.

**Transcript**

I want to spend a little bit of time today talking about [HashiCorp Boundary](https://boundaryproject.io/), but before we get right into what the product is and how the product works, I want to set a little bit of context about how we access private resources.

**Traditional Privileged Access Management (PAM)**

When we talk about how we would traditionally access a private resource, let's say a database that's on a private network, there's a handful of different systems that are typically in between the user and the target system. As an example, let's say we have our private network. This could be a private datacenter on-premises. This could be a cloud VPC. It doesn't really matter. This is a network that we intend to be private.

Our user typically isn't going to be on the network all the time. This could be, for example, a developer who needs access to a private system that is running. It could be a database administrator who needs access to a database, but in either case, the user is not on the private network, they're working in their office or they're working in their home and they need access to it.

**VPNs and SSH Bastion Hosts.**

Let's suppose in this case, it's a database that we're trying to access. Typically, what you would see is that we'd run some form of a gateway on the edge of our network. This gateway could be a VPN. It could be an SSH bastion host. But we're going to run something like that so that the user can connect to it and effectively bridge themselves onto the private network.

In the VPN's case, they might get a literal IP address provided by the VPN host that puts them on the network. In the case of SSH, they're forwarding all their traffic through this master and host, and the bastion host sits on the private network. The problem is: we don't necessarily want this user to be able to connect to any system. So, we might have some systems, this case, let's say it's our super-secret system that nobody should have access to.

**Firewalls**

We don't want you to be able to connect to the VPN and then just willy nilly connect to everything inside of this private network. So, in practice, what we probably also do is have some form of a firewall running that's going to restrict where traffic that's originating from the VPN or originating from the SSH bastion host can go.

In this case, we're going to white-label traffic that says, great, it can go from our SSH or from our VPN to this database. And so, once the user connects all the way through to the database, they also need a database username and password to connect to it.

If we consider this the typical type of workflow, there's a bunch of information the user needed. First, the user needed to have VPN or SSH credentials. Second, they needed to know the IP or the host name to connect to. Then third, they needed the database credentials to get all the way through to that system.

**Challenges with the Traditional Approach**

There are several challenges in this type of approach. One is the onboarding and off-boarding of users. I don't necessarily want to have to distribute VPN and SSH keys to everyone who starts or everyone who leaves the company. This gets to be a burden, especially if we consider best practices like [key rotation](https://www.hashicorp.com/blog/why-we-need-dynamic-secrets/) on a regular interval.

Second, these IPN hosts may not be static. So, in the case of a database, it's more likely to be static. But if I'm connecting to a web service deployed on top of Kubernetes, that's likely to be a dynamic address. And so, this gets challenging if we consider things like a [firewall-based or static-IP based control](https://www.hashicorp.com/resources/why-should-we-use-identity-based-security-as-we-ado).

Lastly, if I must expose database credentials directly to my user, what happens if those get exposed or compromised somehow? Ideally, I'm not giving those credentials directly to an end user to begin with. So, there are several challenges in this space.

**A Zero Trust Model for Secure Sessions Management**

We really looked at this problem when we were thinking about Boundary and said, "What's a different way of solving this?" Knowing that what's changing in these environments is: increasingly we want to move towards a [zero trust security model](https://www.hashicorp.com/resources/how-zero-trust-networking), meaning I don't want to trust my private network. And I don't want to give people access to a private network at all. We're also moving to more dynamic, more ephemeral environments.

This model worked okay when we didn't deploy very often and IPs tended to be static, but as we adopted dynamic infrastructure that [auto-scales up and down](https://www.nomadproject.io/docs/autoscaling), we moved to ephemeral platforms like Kubernetes, where if a node dies, applications might be migrated to a different node. We're adopting more serverless things where you really don't think about a host or a node or an IP anymore. It's really about this logical service that you're trying to route to. With all of those, we're really moving away from relatively static infrastructure to much more dynamic, much more ephemeral infrastructure. So, if we're trying to manage a static set of IPs or hosts that you're allowed to connect to and enforce that with a firewall, that gets to be very brittle.

One of the things we focused with [HashiCorp Vault](https://www.vaultproject.io/) is this notion of [dynamic secrets](https://www.hashicorp.com/blog/why-we-need-dynamic-secrets/). I don't want to have a static, long-lived credential because invariably that gets leaked. Someone stores it on their laptop, they put it in a Wiki, they paste it into Slack. It gets saved into a log file somewhere, etc. There are many different avenues for a static credential to get leaked. So how do we move away from that and instead create a credential on-demand that's short-lived and we revoke it when we don't need it any longer.? How do we bring that approach, that methodology to this [human to machine use case](https://www.hashicorp.com/resources/what-is-difference-between-iam-pam) as well?

**Boundary Architecture**

All of those are the problems Boundary looks at, so architecturally it's going to look very similar, but we're going to do all of this with us slight shift now. I'm going to have my private datacenter. I'm going to have my database that I want access to.

**Pick an IdP**

In this case, there will also be a gateway. When the user wants to connect, they don't have to present a unique SSH or VPN credential, but they do have to perform a single sign-on. We want is a strong assertion of the user's identity. We're going to have a three-legged connection here with our identity provider (IdP) of choice. This could be ADFS. It could be normal Active Directory. It could be Okta, Ping, pick your favorite IdP. It doesn't really matter. We want to assert that you can prove that you're a valid user who's authenticated using whatever IdP of choice. So, this is a pluggable by design. We don't care—bring whatever your favorite existing solution is.

**Configure a Set of Policy**

Once the user has authenticated, then there's going to be an authorization based on that set of policy. So, we're going to have a policy and that policy is really going to drive role-based access control. We might say in this case, our database administrators, they get access to these databases. But the granularity that we're going to specify this policy at is not necessarily single host, single IP, it's going to be at the logical service. What we really care about is that notion of a logical service. This is important because the logical service lets us elevate from the details that are dynamic.

If what we do is manage a control that says this user or this group can talk to this IP, what happens when that IP changes, or we auto-scale up and down, or a node fails, and the workload gets migrated? It means that the rule, the control itself, must be changed to be kept in lockstep. Versus, if we can abstract it and say, the database administrators can talk to the database, the set of databases can be dynamic. The IPs can come and go. They can change. We can scale up and down that rule. The control stays the same.

**Request a Session**

We're going to specify this policy in this way, and now what's going to happen is, when the user wants to connect, they're going to initiate a session. The way this might work is, if they're using, let's say the CLI for example, they're going to use the boundary command and they're going to run something like boundary connect. And what that's going to do is, on their local machine, we're going to spin up a little agent behind the scenes for them. This agent is going to do the authorization flow with the gateway.

If the user is authorized and has access to the endpoint system, the gateway will directly establish that connection to the target system. The user is directly interacting with their normal CLI tools, so in this case, if they're using connecting to a database, maybe they execute PSQL and PSQL talks to that local host agent.

The advantage of this approach is, any tool we might want to use, whether we're using PSQL or SSH, etc. We don't break any of the existing tools. Any tool that the developer or operator is familiar with and comfortable with, great, you use it. You always just talk to your localhost agent. That agent is basically tunneling this connection back, very similar to what we might do with an SSH port forward. This agent is going to work with the gateway, the gateway is forwarding the traffic all the way through to this endpoint system, and great, now we have access to it.

**Advantages of the Boundary Approach**

One of the key advantages of this approach is you'll notice where it's specifically not bridging the client onto the private network. If there was now, the same secret service that we shouldn't have access to, if the user doesn't have a policy that grants them access to it, they can't compel the gateway. They cannot ask the gateway to initiate this connection. This is impossible because the user isn't on the private network, the user is on the public internet, in this case, they're talking to the gateway and the gateway is directly connecting proxy traffic to only the target system.

And so, when we talk about the advantages of this approach, it's a few different things. One is when we talk about on-boarding and off-boarding users, it is just about adding or removing them from the IdP. There's not an additional step of distributing VPN or SSH credentials.

Two, we talk about managing the set of policy, we can focus on high-level logical rules, which set of users should have access to which set of services. We don't need to worry about managing low-level IP-based controls, which are brittle. By virtue of never actually giving users access to the private network.

* We stay in line with sort of a zero-trust network philosophy where we're not bridging users onto this private network.
* We don't have to worry about them being on the private network implying access.

The access is only applied by these strict policies. The last one is because the gateway is directly interfacing with the target system. We don't have to give the user credentials necessarily. You can imagine that this gateway would interact with a system like Vault. It would be able to fetch the database credential and authenticate that session to the database without ever having to expose the actual database credential to the user. In the cases where we *do* need to expose the credential to the user, possibly because the gateway isn't protocol aware, when the gateway doesn't necessarily understand every possible application protocol, we can still leverage Vault support for dynamic secrets.

**Just-in-Time Credentials from Vault**

We can fetch a secret that's dynamically generated just-in-time where Vault supports almost every common relational database, NoSQL system, cloud API, message queue, etc. We can get that dynamic credential and provide it to the user. The user is using that maybe as part of their PSQL command where they need to provide their username and their password. But that the credential is now dynamic and short-lived. So, after some fixed period, an hour, 24 hours a week, etc, Vault will dynamically destroy that credential.

In the best-case scenario, we never actually have to give the credential to the user. The authentication is fully invisible and happening between the gateway and the target system. In the worst case, we can move to our model where it's still a dynamic credential. It is provided to the user, but it's short-lived and we're able to shred it and quickly rotate those credentials. This gives us this advantage of how we manage credentials to minimize exposure.

**Boundary Components**

This is Boundary at a very, very high level. There are a few essential moving pieces to this. I simplified here to just describing it as a singular gateway. In practice, there are a few moving pieces.

**Controllers**

The controllers provide the API to the system. They're sort of the brains and they manage the state. They manage all the API calls, so if you make an API call to the system, it's a controller that's answering that. Controllers are themselves stateless but interact with a shared database behind the scenes. And we might be running multiple controllers, either for high availability or for sharding of traffic if we have a lot of activity.

**Workers**

Then there's what we call workers, and workers are responsible for proxying traffic. Again, you might run one or more of these for the same reasons, either for HA or just because you're sharding work. The workers only communicate with the controllers to get metadata and to authenticate connections and things like that. So, when a user session comes in, API calls are being made to the actual controller to authenticate, to authorize, to request a session, get created. But then any actual connection is flowing on the client to a worker and from a worker all the way through to whatever the target endpoint is.

**Built to Scale**

This architecture allows these different pieces to scale independently. You can have a small number of controllers, but you can have workers that run across different sites. So, for example, for different networks, you want to have one in AWS, one in GCP, one in an on-premises environment. You can run these workers in multiple different places and share a set of controllers. It also lets you scale these things and have them decoupled. If you're just starting out and it's a simplistic installation, you don't need all that complexity. You can run both the controller and worker as part of a single agent, a single binary on one node. And then over time as you have the need for more availability, more scale, more throughput, etc, the architecture is designed to support that.

**Interacting with Boundary**

For clients, (obviously we don't want them to have to deal with a raw API, that would be painful) there's a few different pieces that enable clients to make this a lot easier. For our power users, we expect them to be largely comfortable using the CLI. If you're day-to-day trying to connect to targets instances, manage the system, interact with it. It's probably the most efficient way and that's what we expect power users to be using. At the same time. If you're new to the system, you're trying to understand how it is configured, you just want to play around with it, it does have a web UI as well. You can use that to configure the system, view how things work, and get a mental understanding of the model.

There's also going to be a desktop app. The desktop app is designed for clients to simplify the process of connecting to these target systems instead of necessarily having to interact with the CLI and be comfortable with that. You can open the desktop app, browse the catalog of services and hosts you have access to, and simply connect through that. Then all the workflow around actually spinning up the agents, proxying launching, whatever your app is, all that gets hidden for you and automated as part of this.

When we talk about managing the system, doing these configurations, making it dynamic, a big piece of this is the [Boundary Terraform provider](https://github.com/hashicorp/terraform-provider-boundary). The [Terraform](https://www.terraform.io/) provider lets you manage Boundary configuration as code. Taking an [infrastructure as code](https://www.hashicorp.com/resources/what-is-infrastructure-as-code) approach to how we manage all these policies, all these controls, who has access to what, etc.

**Service Naming**

Over time, like I said, one of the key goals is to make sure that we're not specifying these policies on an IP level, but rather a logical service level. So how does that work? How do we know what the set of databases are? What the set of web services are as an example. This is where there'll be integrations and dynamic host sets.

A dynamic host set is sort of what it sounds like. It's a set of hosts that is in fact dynamic. These will get sourced by integration with various systems. An example of this might be AWS, where you might query a tag and say, "Okay, if you're tagged billing, then we know you're a billing service. If you're tagged web, we know you're a web service".

It could be [HashiCorp Consul](https://www.consul.io/). Consul provides a strong notion of a [service registry](https://www.consul.io/use-cases/service-discovery-and-health-checking) where you have discreet services. An integration with Consul would allow us to automatically source what services belong, what are their IP addresses and health status are, etc. You can imagine a tight integration here as well with Kubernetes, where we integrate with the Kubernetes service catalog query where you select your labels and use those selector labels to populate a dynamic host set.

As we talk about the management of the system to the degree that we have a static configuration, how do we manage that as code through something like Terraform? Of course, it's all API driven, so you could use a different tool as well to the degree that information is dynamic. It's an evolving set of hosts. It's IPs that are coming and going or auto-scaling. How do we directly integrate with those different systems, so that catalog is kept up to date automatically and doesn't require us to constantly be modifying different controls.

**Next Steps**

This is Boundary at a very, very high level, looking at how we move towards this identity-centric, [policy-driven](https://docs.hashicorp.com/sentinel/concepts/policy-as-code/) approach to security, moving towards zero trust and getting rid of users having direct access to credentials or direct access to the private network. I hope this was helpful. For more information, please check out the [Boundary website](https://boundaryproject.io/). Thanks.