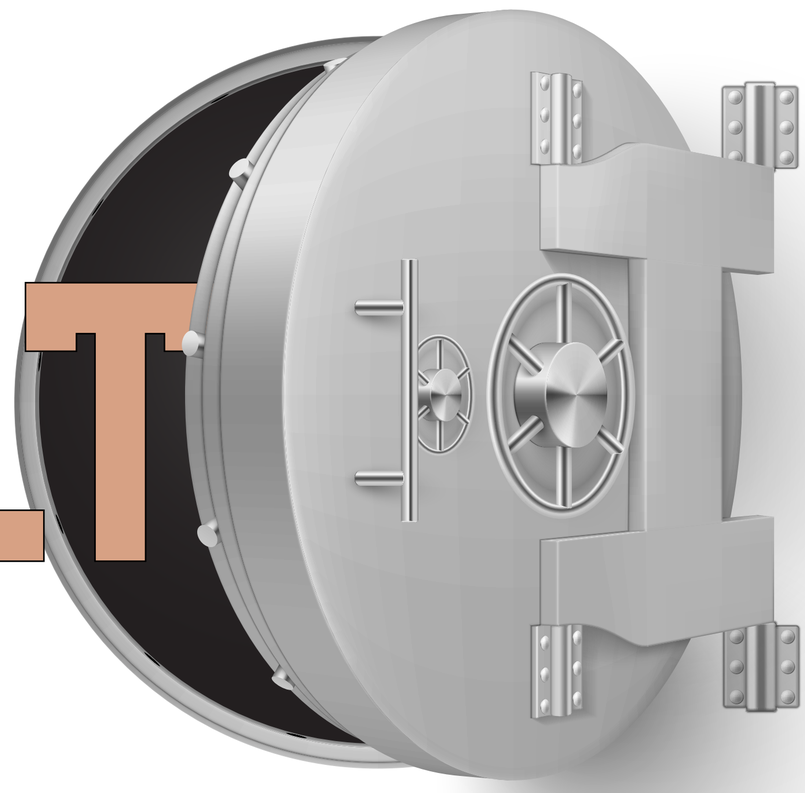
VAULT

BY DAVE COTTLEHUBER



Working from home is the new normal. But from a security perspective, things just got a lot more complicated. Gone are the secure offices with carefully manicured security perimeters and 24x7 physical security.

Security professionals talk about the categories of risks we care about as our "threat landscape", or "security posture".

That's a fancy way of saying that we can make decisions such as not caring about GCSB, KGB, CIA, Mossad, and other Government-funded attackers, but we do care about forgetting a laptop on the train, or having somebody's office broken into, to steal high value items. And we care about passwords and credentials. A lot.

We do care about forgetting a laptop on the train, or having somebody's office broken into, to steal high value items.

But Where To Store the Secrets?

Of particular interest, both to attackers, and to ourselves as sysadmins or developers, is managing, and rotating, secrets.

How many environments have you seen, where database credentials haven't changed in, well, forever? Where long-running batch jobs use the same password across multiple systems? Or where changing one of these can result in unexpected downtime, as a cascading butterfly effect ripples across the company, and ultimately your career?

Clearly sticky notes, and credentials checked into git repos, are neither scalable, nor secure. We need something that is reasonably generic, and yet supports a wide range of use cases.

So what's a sensible DevOps engineer supposed to do?

One solution is a typical password store, like BitWarden or 1Password, and to extend these tools across a team of people, and into various command-line tools. While these solve a part of the problem, they don't solve enough of it.

They are primarily user-facing tools, and are not easily wired up to a complex pipeline of git repos, puppet and ansible deployments, and ensuring that credentials are rotated regularly, and only accessible to appropriate users and systems.

Secrets Management Platforms

Sometimes called Key Management Systems, these are commonly found integrated into Cloud vendors. Azure, Amazon, Google, Oracle all offer tightly coupled and well-integrated tools for their platforms.

However if you're reading the FreeBSD journal, you are more likely to be viewing the idea of giving all your secrets to companies, with extreme trepidation. Ideally we'd like to manage our own secrets, without relying on a third party in a foreign country.

Trade Offs

These systems focus on enabling operations teams to distribute, and manage, key material, in a highly secure, and controlled fashion.

They aim to address the challenge of securely managing and orchestrating secrets in modern, complex environments where applications, systems, and users require access to very sensitive information.

For example, they may integrate changing a secret, to triggering a rollover of container using that secret, to the new version.

They may enable a batch job to decrypt briefly financial information, and then return it updated safely encrypted again.

Or it may provide a one-use token, that allows a newly provisioned service, to connect to a given database, in a way that we can be sure that token was neither hijacked in transit, nor able to be used by more than one instance.

When a new instance is provisioned or deployed, it will get a new one-time use token, unique to this instance, and this time.

This type of functionality is ideal for separating the deployment of systems and associated secrets, from direct access to those secrets. This type of smoke and mirrors is often called cubby-hole deployment — where a one-use token, tightly bound to a single deploy, is injected during deployment by an automation toolset. This token is used at instance boot, to fetch the runtime secret, dedicated and bound to this instance, perhaps by IP address or other caveats.

At startup a master key is needed to unlock all other keys.

Getting Started

This article devotes itself to introducing Hashicorp Vault, which has a feature-parity open fork called OpenBao, similar to the Terraform / OpenTofu licensing fork as well.

There are other tools, but Vault is ported to FreeBSD, and I'm sure OpenBao will land soon too.

Vault's Sweet Spot

Vault, and other KMS, are not great places to keep your Webstorm IDE license, or a scanned copy of your passport. It's not end-user friendly, and it's definitely not usable on a mobile device.

But if you're managing servers, databases, and networks, it's fantastic. It can be easily integrated with Terraform, Chef, Puppet, Ansible, and almost anything that has, or uses, a command-line or terminal interface.

Internals

Vault stores all keys and values encrypted on disk. At startup, therefore, a master key is needed to unlock all other keys. To avoid having a single lightweight key, Vault uses SSS, or Shamir's Secret Sharing, to split a large complex key into separate secrets, that can be

recombined to unlock the vault. Try the Shamir Demo in a modern WASM-capable web browser.

Cleverly, SSS allows a configurable degree of redundancy — say, 3 of 5 keys are needed to unlock the vault. Thus, your 3 main sysadmins can unlock it, but in the absence of any one, you can beg your lawyer or accountant to loan their key if needed, and reach your quorum of 3. Each admin submits their unlock key locally, and an API challenge is used to prevent any single admin from obtaining all segments of the master key.

Once the vault is unlocked, from a user perspective, it functions largely like any other HTTP-accessible key-value store. We can store small files like ssh private keys, or TLS certificates, the usual passwords, or even get vault to generate ephemeral passwords for a limited time, and limited purpose.

Getting Started

While vault supports complex deployments, with consensus protocols and multiple servers, I've found a small highly reliable physical server with a hot standby and a zfs replicated backup, to be sufficient. Of course, I rely on Tarsnap for a fully offline backup — an absolute essential requirement for something as critical as all of our secrets!

Install and Configure

The usual incantations must be performed as root:

```
# pkg install -r FreeBSD security/vault
# mkdir -p /var/{db,log}/vault /usr/local/etc/vault
# chown root:vault /var/{db,log}/vault /usr/local/etc/vault
# chmod 0750 /usr/local/etc/vault
# chmod 0770 /var/{db,log}/vault
```

The **rc.conf** settings, you can use **sysrc(8)** for this, or your preferred ops toolkit.

```
# /etc/rc.conf.d/vault or where-ever you prefer
vault_enable=YES
vault_config=/usr/local/etc/vault/vault.hcl
```

And vault's config file. There are of course many options, most of this is self-explanatory. For our test deployment, we will disable TLS and use the loopback IP.

```
# /usr/local/etc/vault/vault.hcl
default_lease_ttl = "72h"
max_lease_ttl = "168h"

ui = true
disable_mlock = false

listener "tcp" {
  address = "127.0.0.1:8200"
  tls_disable = 1
  tls_min_version = "tls12"
  tls_key_file = "/usr/local/etc/vault/vault.key"
  tls_cert_file = "/usr/local/etc/vault/vault.all"
```

```
}

storage "file" {
  path = "/var/db/vault"
}
```

Now run the daemon in the foreground:

```
$ vault server -config /usr/local/etc/vault/vault.hcl
==> Vault server configuration:

Administrative Namespace

             Api Address: http://127.0.0.1:8200
...
```

In a new terminal, let's check the status:

```
$ export VAULT_ADDR=http://localhost:8200/
$ vault status
vault status
Key                Value
---                -----
Seal Type          shamir
Initialized        false
Sealed             true
Total Shares       0
Threshold          0
Unseal Progress    0/0
Unseal Nonce       n/a
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Note that the vault is uninitialised, and still sealed. Let's fix that:

```
$ vault operator init --key-shares=3 --key-threshold=2
Unseal Key 1: jjcVgHTjWw3j4BsyDhugvS9we5t5qMAhJL8bSWzySjbG
Unseal Key 2: WfMeZPA7ixleQAMeeAqyey+gwrxDn9WNfSvdKzdLMaeA
Unseal Key 3: V9cd1eVBH6mstyoS2pbD6S80R7NJVz7jPvlPOcLOUVlw
Initial Root Token: hvs.RAeqzETRhOXOImMPw7xrXbAl

$ export VAULT_TOKEN=hvs.RAeqzETRhOXOImMPw7xrXbAl

$ vault status
```

```
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       3
Threshold          2
Unseal Progress    0/2
Unseal Nonce       n/a
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Note that the vault is now initialised, and also still sealed. So let's fix that next, using the newly generated key shards:

```
$ vault operator unseal
Unseal Key (will be hidden):
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       3
Threshold          2
Unseal Progress    1/2
Unseal Nonce       6ce4351d-012b-df3f-a176-34d266f00795
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Repeat the unsealing with a different key each time until **`sealed`** changes to **`false`**. The final step is to enable auditing, because security people love logs.

```
$ vault audit enable file path=/var/log/vault/audit.log
Success! Enabled the file audit device at: file/
```

Feel free to tail this, there are no secrets ever stored here, so it's only an audit log of requests.

Shamir Secret Ring

Now that you've unsealed the vault, distribute your secrets by encrypted avian carrier, to your chosen secret keepers. Some suitable ceremony is required, and also to ensure these secrets are adequately protected, both against incompetence or less, as well as Mossad and North Korean agents.

By now, you should be ready to store secrets.

Storing Secrets

Vault has a concept of engines - there's a simple key-value storage, also one for ssh certificates, AWS and Google Cloud integrations, RabbitMQ, PostgreSQL, and many more. Each one needs to be separately enabled.

```
$ vault secrets enable -version=2 kv
Success! Enabled the kv secrets engine at: kv/
```

From here on in, we need to specify both the engine type, and it's mount path. It's possible to retrieve data as JSON, or yaml as well, and even to store files directly.

```
$ vault kv put -mount=kv blackadder scarlet_pimpernel="we do not know"
=== Secret Path ===
kv/data/blackadder

======= Metadata =======
Key                Value
---                -----
created_time       2024-05-12T23:04:50.283028044Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

$ vault kv get -mount=kv -format=json blackadder
{
  "request_id": "48141452-8f8f-b497-9c53-1af71e24e2a5",
  "lease_id": "",
  "lease_duration": 0,
  "renewable": false,
  "data": {
    "data": {
      "scarlet_pimpernel": "we do not know"
    },
    "metadata": {
      "created_time": "2024-05-12T23:04:50.283028044Z",
      "custom_metadata": null,
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  },
  "warnings": null
}

$ vault kv put -mount=kv blackadder scarlet_pimpernel="comte de frou frou"
```

```
=== Secret Path ===
kv/data/blackadder

======= Metadata =======
Key                Value
---                -----
created_time       2024-05-12T23:08:22.369551931Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            2

$ vault kv get -mount=kv -format=yaml blackadder
data:
  data:
    scarlet_pimpernel: comte de frou frou
  metadata:
    created_time: "2024-05-12T23:08:22.369551931Z"
    custom_metadata: null
    deletion_time: ""
    destroyed: false
    version: 2
lease_duration: 0
lease_id: ""
renewable: false
request_id: 686965d9-811f-8689-d75f-a02f7dded9a7
warnings: null

$ vault kv put kv/blackadder scarlet_pimpernel=@/etc/motd.template
```

Role-based Access

Vault can be configured to require github authentication, and delegate roles and authentication to something other than LDAP. Many of you will rejoice at this news. With Github authentication, 2FA can be enforced for all users, so this represents a reasonable trade-off for small teams.

```
$ vault auth enable github
vault auth enable github
Success! Enabled github auth method at: github/

$ vault write auth/github/config organization=skunkwerks
Success! Data written to: auth/github/config

$ vault write auth/github/map/teams/admin value=admins
Success! Data written to: auth/github/map/teams/admin
```

Place this small policy file in **/usr/local/etc/vault/admins.hcl**

```
# grant members of github admins group all rights in the kv/ mount
path "kv/*" {
  capabilities = ["create", "read", "update", "delete", "list"]
}
```

And then enable it within vault:

```
$ vault policy write admins /usr/local/etc/vault/admins.hcl
Success! Uploaded policy: admins
```

You can of course make more restrictive policies, in rights, or in paths, or in selected mounts, for various groups, such as a deployment bot.

Finally, each user that wishes to use github auth, for vault, must go to https://github.com/settings/tokens and add a new personal token with privileges of **admin- read:org**.

This can be used now to generate your vault login token via

```
$ vault login  -method=github token=$GITHUB
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                    Value
---                    -----
token                  hvs....
token_accessor         ...
token_duration         72h
token_renewable        true
token_policies         ["admins" "default"]
identity_policies      []
policies               ["admins" "default"]
token_meta_org         skunkwerks
token_meta_username    dch

$ vault kv get -mount=kv -format=yaml blackadder
...
```

Vault in Automation

Automation tools such as Chef, Puppet, Ansible, and more can use vault to store secrets, for decryption at deployment time, or even in some circumstances, only to be decrypted at runtime.

Let's look at the first case, decrypting secrets at deploy time. Effectively, this extends the templating capabilities of automation tools, and relies on being able to trigger service re-starts, after having pushed new and updated secrets.

We can use vault in 4 ways:

- Ansible and similar tools can store secrets in vault, and only decrypt them at deploy time, using lookups

- `rc.d` framework scripts can use app roles to fetch their credentials at startup, letting the local root user have a delegated token that only permits issuing cubby-hole tokens. The daemon itself will only be able to retrieve its own credentials
- vault can issue time- and IP-bound dynamic credentials that are revoked on expiration, for daemons, cronjobs, & batch scripts that are time limited
- we can also template out files at runtime, using vault agent

Ansible

There are a number of plugins for ansible, and confusingly, there is an internal ansible "vault" module that is not compatible with Hashicorp Vault.

Install the plugin, and use the typical `lookup` functionality:

```
super_secret: "{{lookup('hashivault', 'kv', 'blackadder', version=2)}}"
```

rc.d App Roles

An AppRole is a built-in authentication method, specifically for machines and applications to authenticate to Vault, and then subsequently obtain a token that *only* then allows fetching relevant secrets. This is often called a cubby-hole credential, as it only permits unwrapping the outer layer, to get a key inside.

These can be restricted by time, a limited number of uses, and more. Our trusted root process generates this restricted secret id, and passes it and the role id to the daemon to fetch its own credentials. The generation of the secret id can be set up in such a way that only these credentials can be minted.

Once again, we enable the `approle` mount, before creating our app-specific credentials, as it is a form of authentication. For convenience, this approle will re-use the existing `admins` group policy used earlier, but it should have a more restrictive one, specifically for this daemon/service.

```
$ vault auth enable approle
Success! Enabled approle auth method at: approle/

$ vault write auth/approle/role/beastie \
        secret_id_ttl=60m \
        token_num_uses=10 \
        token_ttl=1h \
        token_max_ttl=4h \
        secret_id_num_uses=40 \
        policies="default,admins"
Success! Data written to: auth/approle/role/beastie

$ vault read auth/approle/role/beastie/role-id
Key        Value
---        -----
role_id    6caaeac3-d8fa-a0e3-83ba-7d37750603c2

$ vault write -f auth/approle/role/beastie/secret-id
```

```
Key                   Value
---                   -----
secret_id             8dd54c92-fe54-0d6d-bee6-e433e815aaa1
secret_id_accessor    cb9bc17c-c756-42b3-c391-b61ebde12bff
secret_id_num_uses    0
secret_id_ttl         0s
```

If we wanted to make these secrets usable within a hypothetical **beastie** daemon, these two parameters can be put in an **/etc/rc.conf.d/beastie** file, which can be secured to only be readable by root.

```
beastie_enable=YES
beastie_env="
  ROLE_ID=6caaeac3-d8fa-a0e3-83ba-7d37750603c2
  SECRET_ID=8dd54c92-fe54-0d6d-bee6-e433e815aaa1
  SECRET_PATH=kv/beastie
  VAULT_ADDR=http://localhost:8200/
"
```

The **/usr/local/etc/rc.d/beastie** script runs a pre-cmd that fetches the secret as root, and injects it into the child environment.

```
start_precmd=${name}_vault
beastie_vault() {
    # Authenticate with Vault using the approle
    VAULT_TOKEN=$(vault write auth/approle/login role_id="$ROLE_ID" \
      secret_id="$SECRET_ID" \
      -format=json | jq -r '.auth.client_token')

    # Retrieve the secret from Vault
    export BEASTIE_SECRET=$(vault kv get -field=data -format=json ${SECRET_PATH} | jq -r .)
}
```

Agents

Vault also provides an agent mode, which does a lot of the credential management for you, and supports templating of simple config files.

Sealing the Vault

Typically, a vault is left unsealed and running for months on end, barring patching and upgrades. In the event of a security incident, it suffices to halt the server that runs the vault daemon entirely, or optionally issue a **seal** command. This shuts vault, and unloads the master secret key.

```
$ vault operator seal
Success! Vault is sealed.
```

DAVE COTTLEHUBER has spent the last 2 decades trying to stay at least 1 step ahead of The Bad Actors on the internet, starting off with OpenBSD 2.8, and the last 9 years with FreeBSD since 9.3, where he has a ports commit bit, and a prediliction for using jails, and obscure functional programming languages that align with his enjoyment of distributed systems, and power tools with very sharp edges.

- Professional Yak Herder, shaving BSD-coloured yaks since ~ 2000
- FreeBSD ports@ committer
- Ansible DevOops master
- Elixir developer
- Building distributed systems with RabbitMQ and Apache CouchDB
- Enjoys telemark skiing, and playing celtic folk music on a variety of instruments