Managing Secrets at Scale

•••

Alex Schoof, Fugue - Velocity EU 2015

Secrets are EVERYWHERE

Mishandling secrets can be **very** bad

We’re really bad at managing secrets

We tend to manage secrets the same way we tell people *not* to manage software

Secret management should be core infrastructure

Lots of new secret management systems have been released in 2015

There is no one-size-fits-all solution

You have to understand your Threat Model and Trust Boundaries

What are we going to talk about?

- Core principles of modern secret management
- How a modern secret management system works
- Some existing open source secret management systems

Core Principles

Principle

Minimal Privilege

The set of principals (actors) able to perform an action on a secret should be the smallest set required.

Implications

- We have authentication and authorization on usage of secrets
- We have enforceable access control policy for operations on secrets
- We have an audit trail to make sure we are doing it right

Principle

Secrets Expire

Secrets expire for a variety of reasons: built-in expiration in protocols, third party policy, internal policy, compromise, etc.

Implications

- Deployment of any given secret is not a one-time event
- We must be able to easily rotate secrets
- A deployment artifact should not be the authoritative source of a secret

Secrets should *never* be in source control

Corollary: Changing a secret should not require a full deployment

Principle

It should be easier to do the right thing than the wrong thing

If operations on secrets require human intervention, long waits, or complex ceremony, people will do other things, like hardcode secrets

Implications

- We need to provide easy-to-use programmatic operations on secrets
    - CRUD operations (Create, Read, Update, Delete)
    - List
    - Audit

Principle

The Weakest Link

The security of a secret is only as strong as the weakest access control mechanism on the storage, transmission, or use of that secret

Implications

- We need strong controls on every secret operation
- Secrets should **NEVER** be persisted in plain text
- Proper root secret storage is very important
  - Operator laptops are now part of your trust boundary for secrets they can access

Principle

Secrets must be highly available

For many applications, access to secrets is a requirement for basic functionality

Implications

- Tier-0 service
- Need good operational and deployment practices
- Be aware of circular dependencies and bootstrapping problems

Principles of Secret Management

- Minimal Privilege
- Secrets Expire
- It should be easier to do the right thing than the wrong thing
- Weakest Link
- Secrets must be highly available

So, we need a system that:

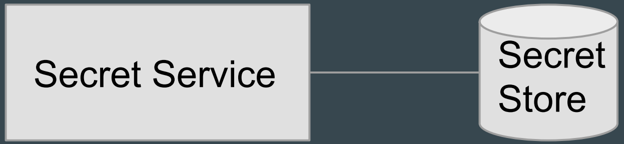
- Supports easy-to-use programmatic CRUD operations on secrets
- Provides an audit trail
- Protects secrets at every stage of their lifecycle
- Permits access control to be defined by policy
- Is highly available

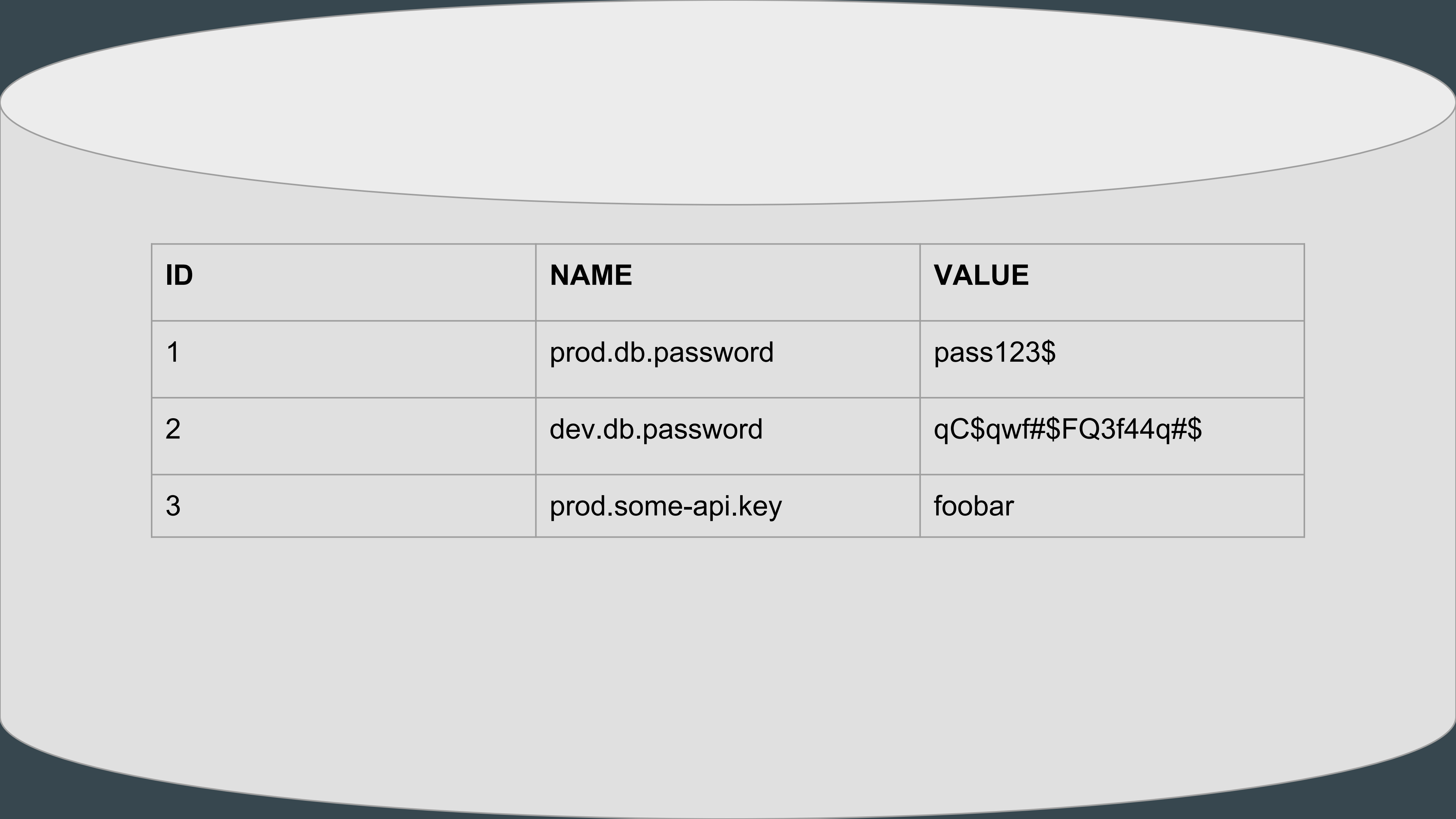
Building a Secret Management System

Quick Caveat

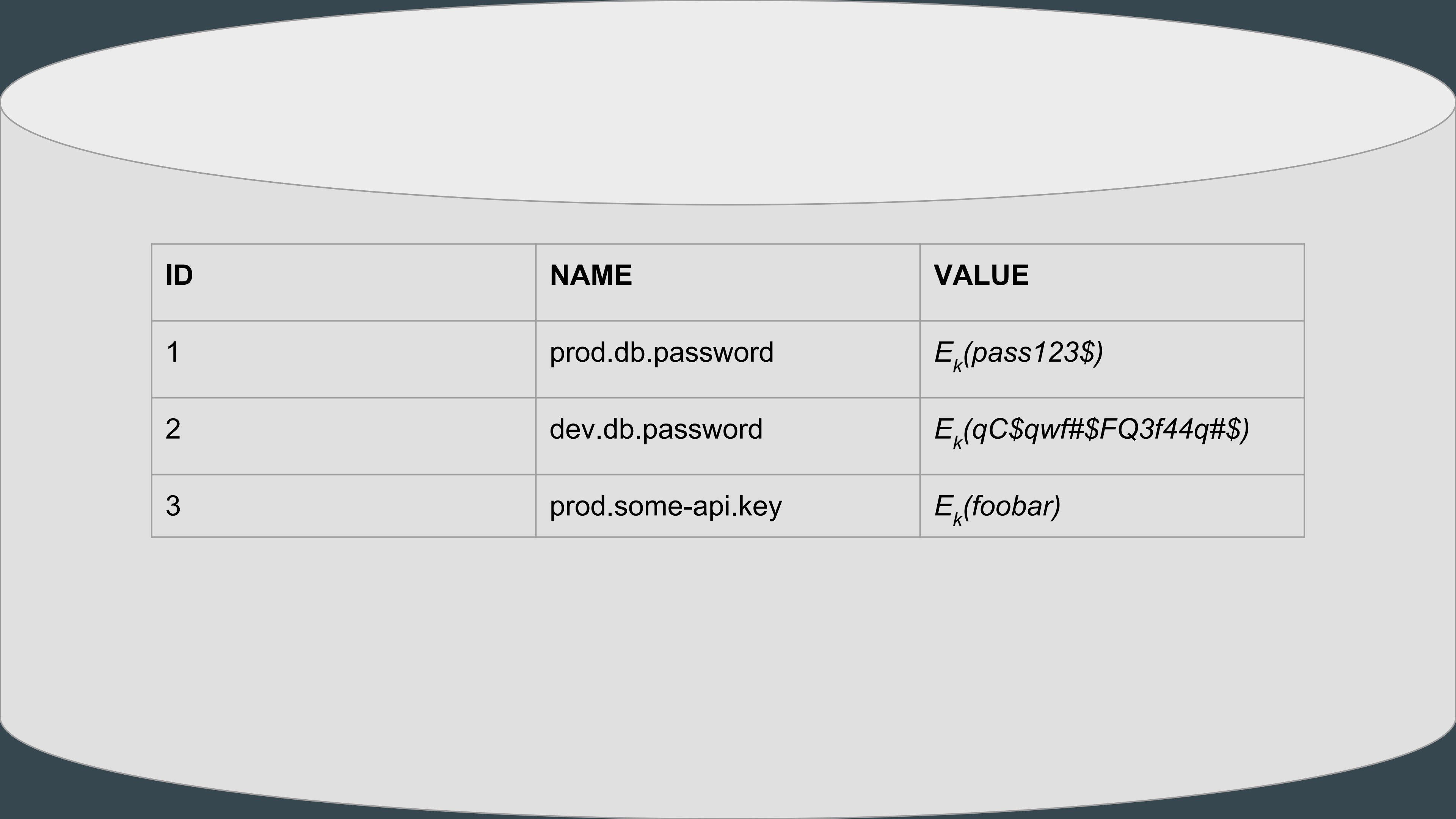
- **DON'T** invent your own crypto
- **DON'T** implement someone else's crypto
- Use standard libraries and battle-tested algorithms

Secret Service


Secret Service
Secret Store



ID	NAME	VALUE
1	prod.db.password	pass123$
2	dev.db.password	qC$qwf#$FQ3f44q#$
3	prod.some-api.key	foobar



ID	NAME	VALUE
1	prod.db.password	E_k*(pass123$)*
2	dev.db.password	E_k*(qC$qwf#$FQ3f44q#$)*
3	prod.some-api.key	E_k*(foobar)*

How do we store the key?

Secure key storage

Hardware

- Hardware Security Module (HSM)
- Trusted Platform Module
- Smartcard

Provide hardware key storage and usually have hardware-accelerated cryptography

All use PKCS#11-style interface

Can generate or import keys, can use keys, cannot export keys

Generally requires a certificate or password to use

Key Management Services

- Presents an HSM-like interface, typically over the network
- available in public cloud environments (i.e. AWS KMS)
- much less expensive than dedicated HSMs

Still need some credential to access the KMS

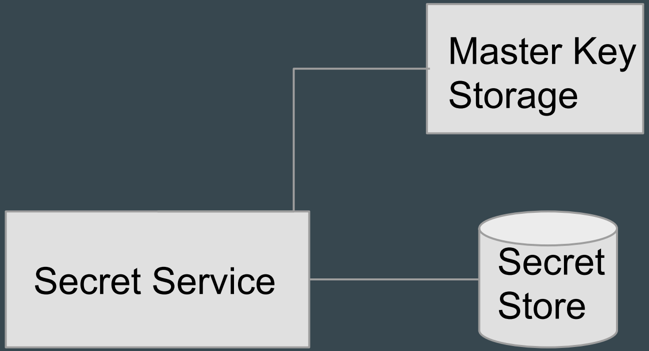
Or, we could not store the key...

Quorum Systems

- Shamir’s Secret Sharing
  - Make a *k-1*-degree polynomial with a y-intercept of the secret value
  - Take *n* points from the polynomial
  - Any *k* of the *n* points can recreate the original polynomial via interpolation
- Can keep shares on smartcards, in safes, etc.
- Introduces a collusion requirement

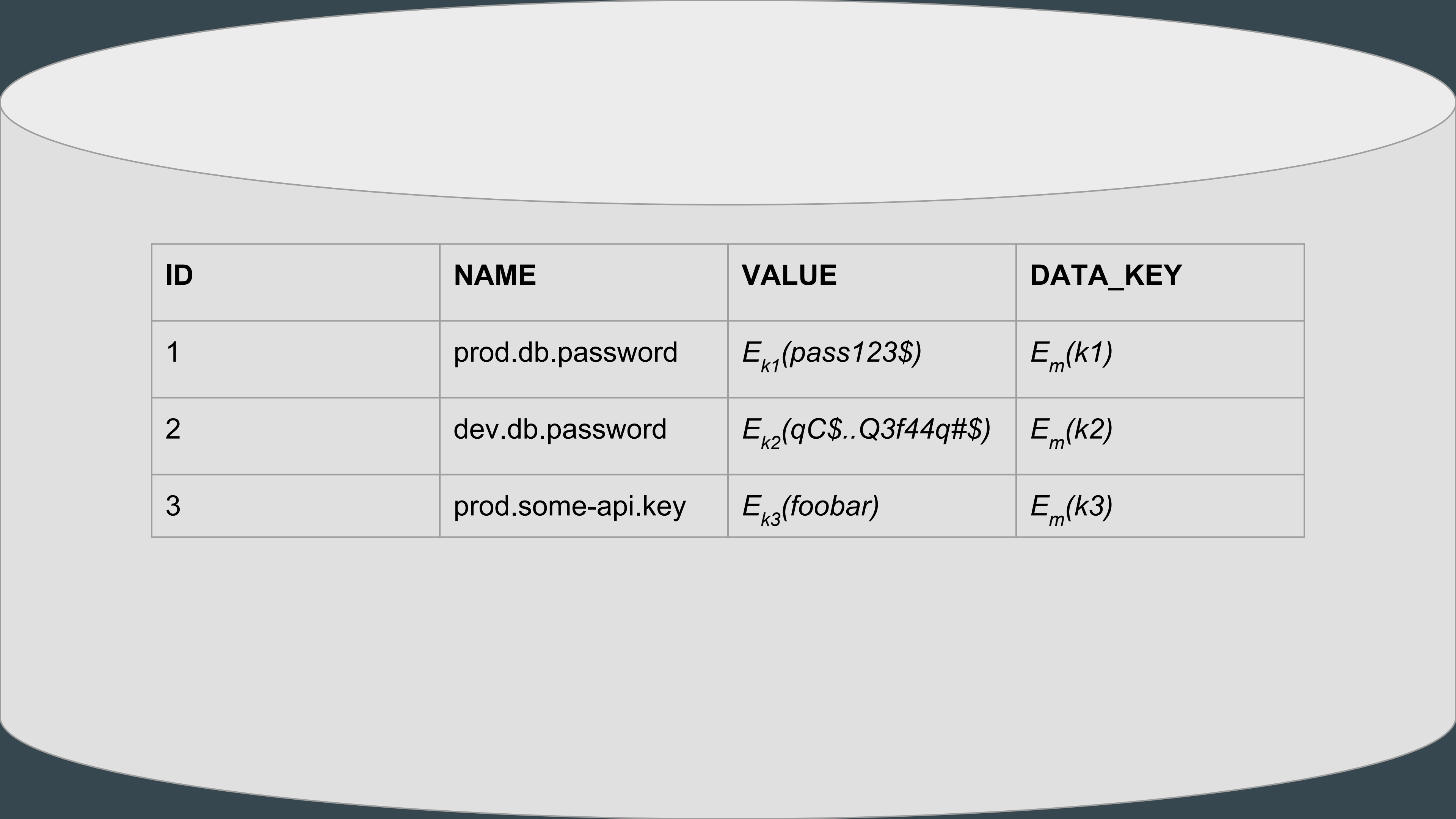
Peer-to-peer induction of secret servers

- Other secret servers that are up verify that the new server is trusted, and then transfer the secret to it, directly or via secret sharing
- Need secure boot/remote attestation to do safely
- Completely hands-off, but very hard to do right

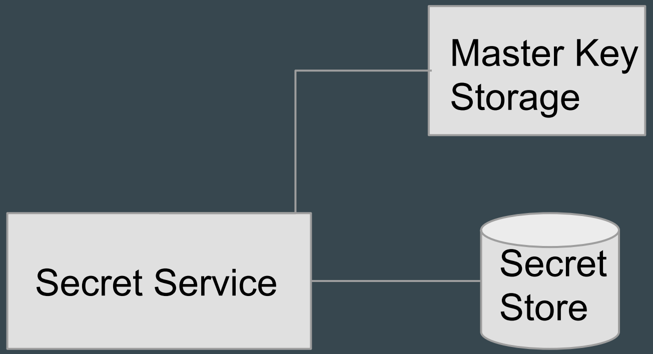

Master Key Storage
Secret Service
Secret Store

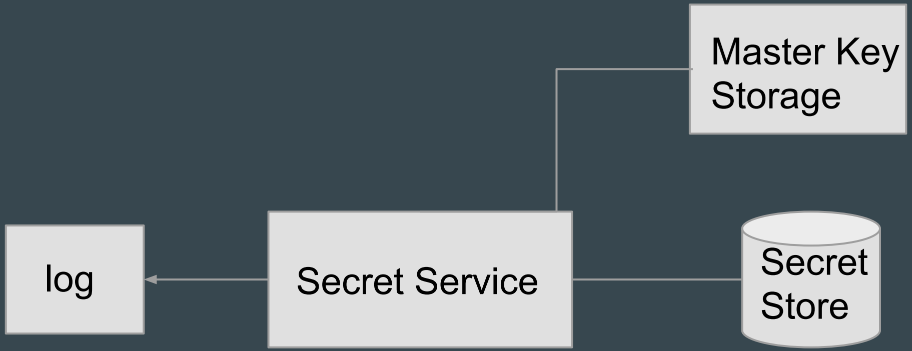
Key Wrapping

- Secure key storage is expensive
- We might want arbitrary-sized secrets
- Have one (or more) master key(s) on an HSM, or in a KMS
- For each secret in the secret-store, generate a unique data-encryption key
- encrypt the secret with the data encryption key
- encrypt the data-encryption key with the master key
- store the encrypted (wrapped) data-encryption key next to the encrypted data
- Add message authentication code (MAC) or use AES-GCM to prevent tampering and chosen-ciphertext attacks



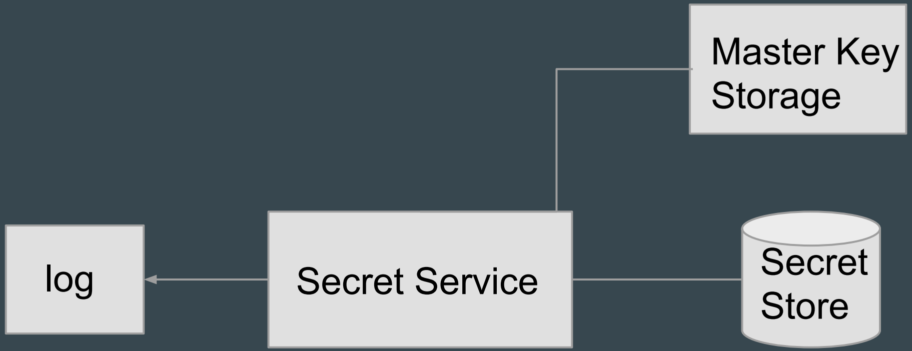
ID	NAME	VALUE	DATA_KEY
1	prod.db.password	E_{k1}*(pass123$)*	$E_m(k1)$
2	dev.db.password	E_{k2}*(qC$..Q3f44q#$)*	$E_m(k2)$
3	prod.some-api.key	$E_{k3}(foobar)$	$E_m(k3)$

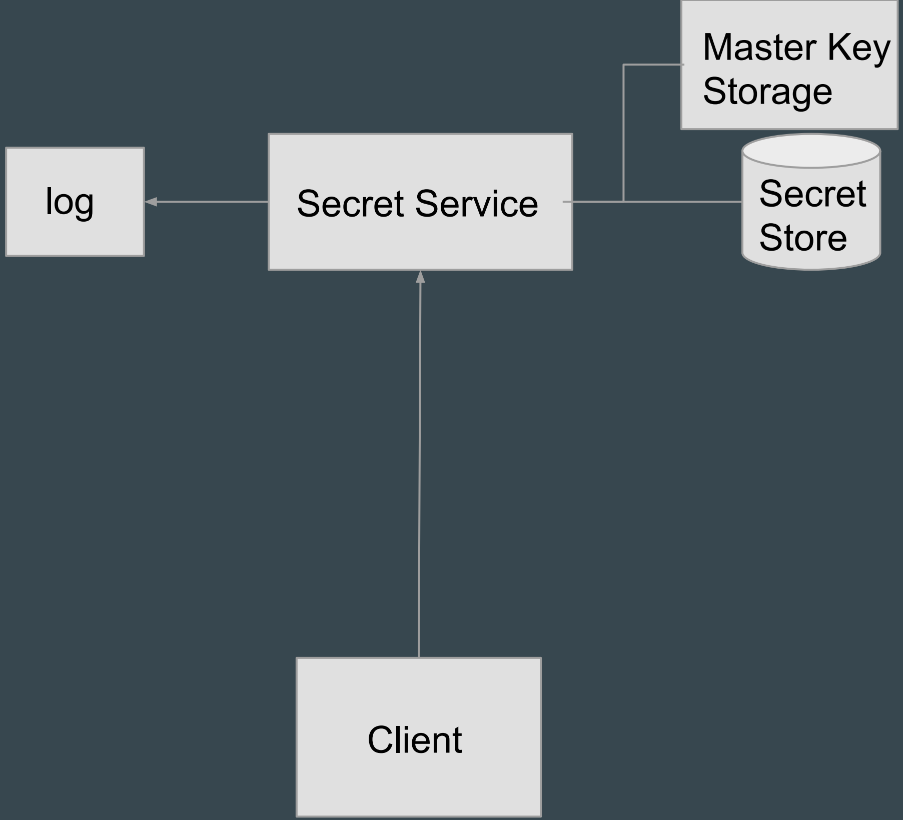

Master Key Storage
Secret Service
Secret Store


Master Key Storage
log
Secret Service
Secret Store

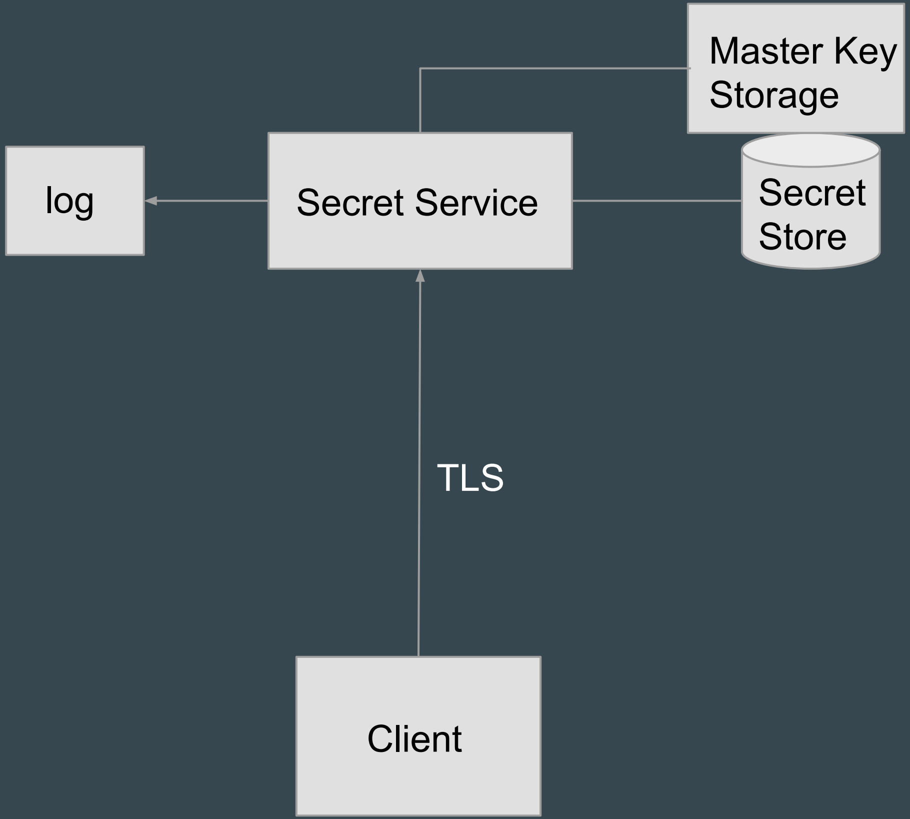
Secure logging

- Append-only
- Signed logs to detect tampering
  - Signed logs are *really, REALLY* hard to get right
  - Use an existing secure logging implementation


Master Key Storage
log
Secret Service
Secret Store


Master Key Storage
log
Secret Service
Secret Store
Client

Authenticating the service


Master Key Storage
log
Secret Service
Secret Store
TLS
Client

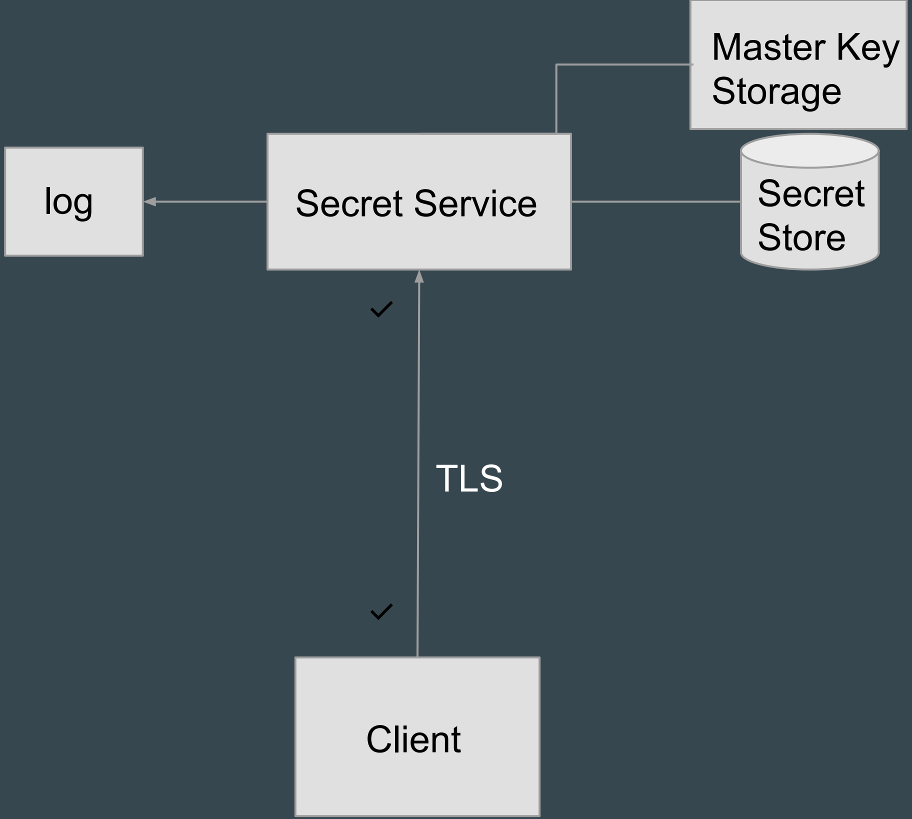
Authenticating the Client

Client Authentication

- Either API key to sign requests to the secret service, or SSL client certificate
- Created at provision or launch of the instance
    - Can use this token as a root of trust for giving other tokens to the client
- Should try to use some out-of-band mechanism to verify the host
- Examples:
    - IAM Roles for EC2
    - Chef client certificates
    - Generate CSR at boot and send to CA that can list running instances for verification
    - Keys generated in hardware

Protecting the authentication token

- Agent-based systems
  - Can let you do cool things like FUSE drivers that use POSIX auth
- Control access to certificate or API keys using standard access control mechanisms
  - Filesystem permissions
  - Mandatory Access Control systems (SELinux, AppArmor, etc)
  - Kernel Key Retention Service


Master Key Storage
log
Secret Service
Secret Store
✓
TLS
✓
Client

Basic Operations

CREATE(name, value)

- Access control check
- Log the request
- Wrap *value* and store in secret store
- Set any initial access control policy (i.e. set owner)

READ(name)

- Perform access control check for *name*
- Log the request
- Unwrap value for *name* and send it back

UPDATE(name, value)

- Access control check
- Log the request
- Wrap *value*, overwrite old *value*

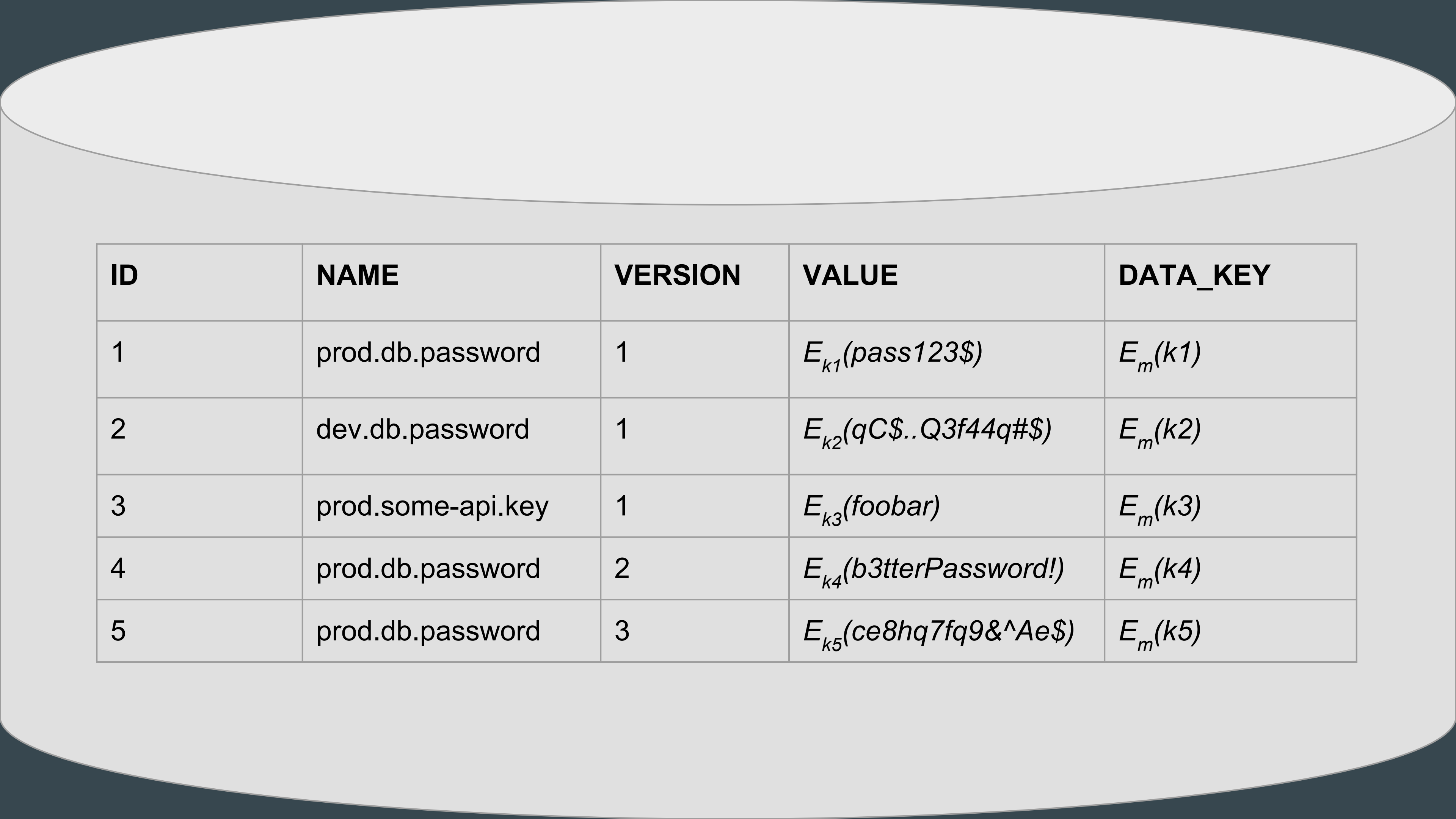
UPDATE(name, value)

- Access control check
- Log the request
- ~~Wrap *value*, overwrite old *value*~~
- Generate new version of *value*

In general, secrets should be versioned and immutable

Operations with versions

- CREATE(name, value) = UPDATE(name, value, 1)
- READ(name, version=`latest`)
- UPDATE(name, value, version)



ID	NAME	VERSION	VALUE	DATA_KEY
1	prod.db.password	1	E_{k1}*(pass123$)*	E_{m}*(k1)*
2	dev.db.password	1	E_{k2}*(qC$..Q3f44q#$)*	E_{m}*(k2)*
3	prod.some-api.key	1	E_{k3}*(foobar)*	E_{m}*(k3)*
4	prod.db.password	2	E_{k4}*(b3tterPassword!)*	E_{m}*(k4)*
5	prod.db.password	3	E_{k5}*(ce8hq7fq9&^Ae$)*	E_{m}*(k5)*

DELETE(name, value, version)

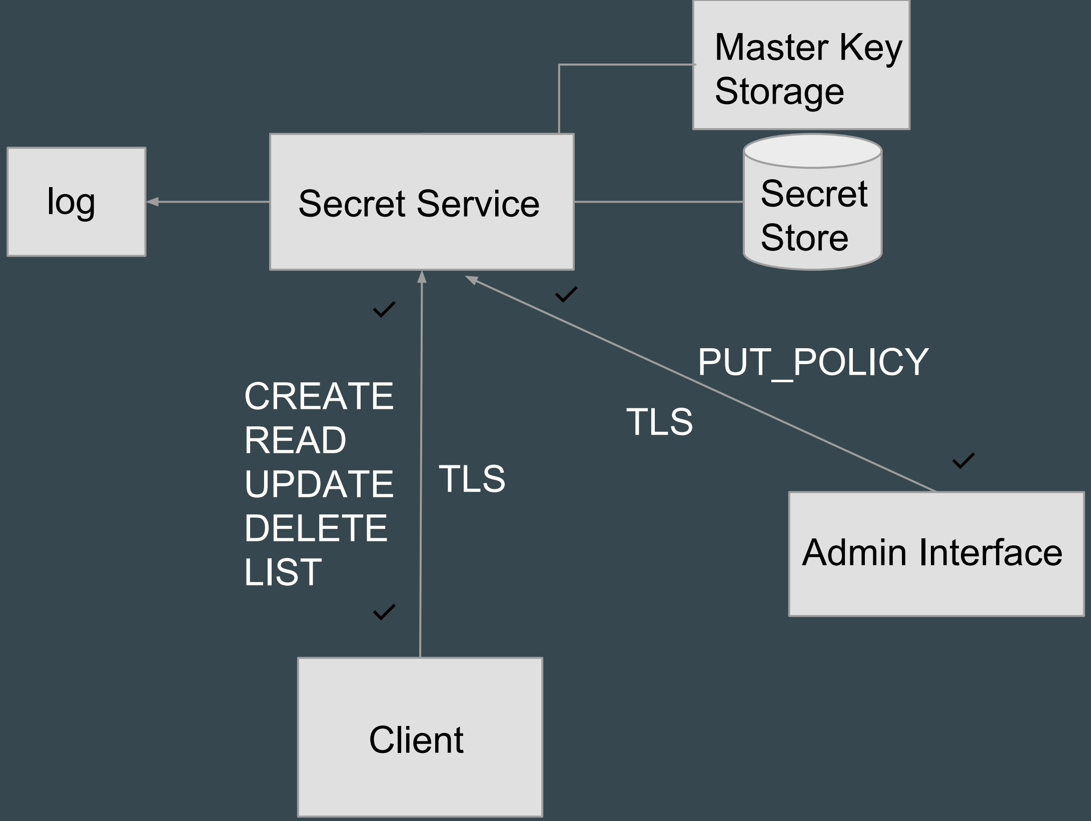
- Access control check
- Log the request
- Distinguish between actual delete and logical delete
    - Will you need to decrypt old data encrypted with a deleted key?
- Revoke asynchronous public keys

LIST(name?, version?)

- Access control check
- Log the request
- Do a scan or prefix search

PUT_POLICY(name, value, policy)

- This is a privilege escalation avenue
- Need to require higher-level interface
- Makes sense to use separate channel or interface
- Policy can range from complex documents to a set of flags
    - IAM policy
    - POSIX-style ACL
    - Booleans for each user


Master Key Storage
log
Secret Service
Secret Store
CREATE
READ
UPDATE
DELETE
LIST
TLS
PUT_POLICY
TLS
Admin Interface
Client

Considerations for Interfaces and Libraries

- Make sure you don’t leak secrets outside your trust boundary
  - Logs
  - Core dumps
  - Swap
- Easy-to-use verbs help usability
  - `get_secret(“prod-password”, version=3)`

Availability

High-Availability Secret Services

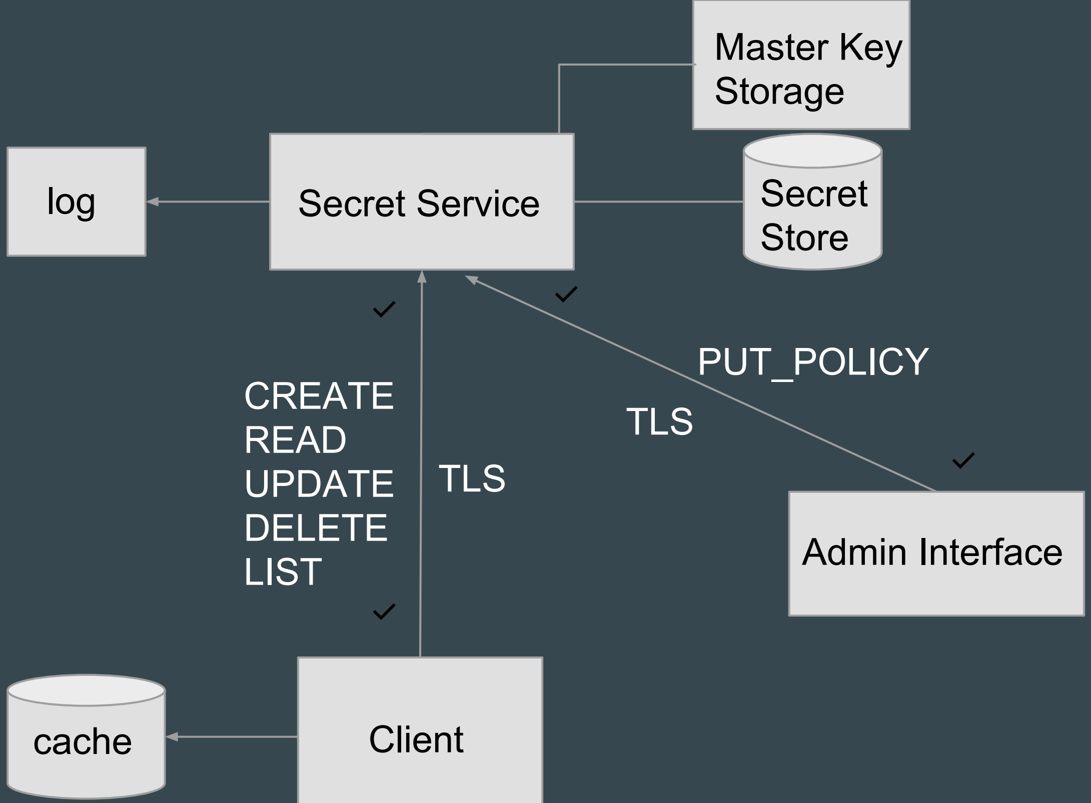
- Run in multiple Availability Zones or Data Centers
- Want secret service to be close to clients (physically and topologically)
- Data store needs to be highly available
- Need to be able to boot/induct new instances of the secret service
- Spend time hardening your secret servers!

Secret management is Tier-0

- Operational metrics
- Canaries
- Integration tests
- Code reviews
- ACL on commits/deployment

Client-side caching

- Publish/Subscribe can be a useful semantic here
- Helps with network partition from the secret service
- Can also reduce read-load on the secret service
- Make sure the cache never writes plaintext to disk
  - Or swap, or core dump, etc.
- May need to perform access control check on the client
  - Unless security boundary ~= instance boundary


Master Key Storage
log
Secret Service
Secret Store
PUT_POLICY
TLS
CREATE
READ
UPDATE
DELETE
LIST
TLS
Admin Interface
cache
Client

Some existing tools

Vault

- Client/Server system for secret management
- Pluggable backend for secret storage (disk or Consul)
- Uses secret splitting (quorum system) to protect master key(s)
- Symmetric tokens or TLS certs for authentication
- HCL (Hashicorp Config Language -- JSON-like language) used for writing authorization policy
- Secret management operations available via CLI or HTTP API
- Nifty “dynamic secret” system

https://www.vaultproject.io/

Keywhiz

- Client/Server secret management system
- Uses relational DB for secret storage
- Derivation key(s) stored in memory or an HSM, data keys derived from derivation keys using HKDF (HMAC-based key derivation -- see RFC5869)
- Client TLS certificates used to authenticate for CRUD operations
- LDAP or passwords used to authenticate for admin operations (like ACL management)
- Group-based ACLs
- CLI, API, or sweet FUSE driver

https://square.github.io/keywhiz/

CredStash

- Lightweight Python utility for managing secrets on AWS
- Uses DynamoDB for secret store
- Uses KMS to hold master key(s)
- Uses IAM policies for access control
- Uses IAM Roles for EC2 for instance identity
- Secret management operations available via CLI, Python library, or Puppet/Ansible modules

https://github.com/fugue/credstash

Sneaker

- Golang utility for managing secrets on AWS
- Uses S3 for secret store
- Uses KMS to hold master key(s)
- IAM and S3 ACLs for access control policy
- Operates on files, rather than values
- Secret management operations available via CLI

https://github.com/codahale/sneaker

Most of this is very new

You have to understand your Threat Model and Trust Boundaries

Secret management should be core infrastructure

Q & A

Slides: http://bit.ly/1kXSTaa

Further reading

- Threat modeling: https://msdn.microsoft.com/en-us/library/ff648644.aspx
- Remote attestation: https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf
- KMS: https://docs.aws.amazon.com/kms/latest/developerguide/overview.html
- Signed syslog: https://tools.ietf.org/html/rfc5848
- Kernel Key Retention service: https://www.kernel.org/doc/Documentation/security/keys.txt
- Shamir’s Secret Sharing: https://en.wikipedia.org/wiki/Shamir's_Secret_Sharing

- PKCS#11: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html