Distributing Secrets

Securely?

Presented by
Simo Sorce
Red Hat, Inc.

Flock 2015
Historically

Monolithic applications on single servers potentially hooked to a central authentication system.
Distributing Secrets?

Containers…

it's all their fault! :-)

Not really, most distributed systems need credentials to access resources like databases or 3rd party APIs.
Clouds, microservices

Distributed applications multi-tier services, 3\textsuperscript{rd} party services, clouds...
What happens today?

- Provisioning systems like puppet and ansible are used to distribute data, storing credentials in the clear somewhere and pushing them around to various hosts.

- Some people even bake credentials directly in container images or keep them in some version control system directly accessible by images ...
I DO THAT TOO!

ARE YOU SAYING .. I AM DOING IT WRONG ?
Secrets != configuration

- Configuration in many cases can be public information and it is rarely an issue if it gets disclosed.
  
  [security through obscurity?]

- Secrets are never public, and should be treated differently from the rest.
WHAT IF I TOLD YOU WE CAN DO BETTER
Let's define the problem space

- What is that we really need to do with secrets and passwords?
  - And what options we have?
  - 5 things we care for: PUPPA
    - Provide, Update, Preserve, Protect, Audit

- On the following slide:
  - **P** = provisioning copies of secrets in files
  - **A** = Use of an API to retrieve secrets
How do I get a secret for a specific service to a specific application?

P) Push secret into application config files from some place that stores them

A) Make the application (or helper) pull the secrets as needed
How do I update a secret in all applications when needed?

P) Push secrets again and/or restart application/container

A) 1. Notify application
2. Application pulls the new secret
Preserve

- How do I preserve correct credentials when a container image is rebuilt?

P) Keep side volume with credentials stored there “in the clear” or inject at every startup

A) Let applications pull their secrets
How do I limit access to these credentials exclusively to what needs them?

P) Trust the provisioning system and all the people involved to get it right.

A) Store secrets encrypted, use Access Controls to limit who gets what.
Audit

- How do I know who got secrets?

  P) Add auditing capabilities throughout the whole infrastructure.
  (LAUGHS)

  A) Store secrets encrypted, audit who access what at retrieval time.
But wait ...

... how do I authenticate to an API if I do not have credentials?
That's a GREAT question!
The host itself is trusted by the applications it runs, containers and VMs included.

Conversely, applications running on the host are (ideally) not trusted.

The host is critical to address bootstrapping issues, and will have to be provisioned accordingly.

- Give hosts an identity (a x509 cert, a keytab, a password) and a role when provisioning.
Example use case

- Trying to get FreeIPA domain controllers installed in an automated way.
- Bootstrapping the installation is problematic, as there is a need to transfer passwords, keys, certificates from one server to another.
- Traditionally done manually by preparing and transferring an encrypted install file.
  - Does not scale well in a dynamic environment.
What's hard about it?

- Some of these keys change over time, others are created over time, so we cannot just keep a copy “somewhere”

- We needed:
  - A way to fetch a set of credentials from an existing server over the network
  - A secure method to transfer those keys over the wire
  - A way to authorize access to those keys
What do I need?

- An API and service to distribute secrets
- Can encrypt information at rest and in transit
- Provides a useful REST API to access/store data
- Modular design allow to configure authentication/authorization and storage methods including proxying requests to other services.
Custodia

- Built in Python
- Simple HTTP server, can listen on a unix socket and served via a proxy (Apache)
- Uses jwcrypto to implement the JOSE standard for web encryption (uses python-cryptography for crypto ops)
- Very easy to extend (see ipakeys)
Works as a Pipeline

- The path used to fetch a secret can be munged and composed by intermediaries.
Transferring secrets ...

- Simple type (use over TLS, please!):
  - Retrieve a secret:
    → GET /secrets/test/mypassword
    ← 200 OK
    {type: simple, value: "secret" }
  - Store a secret:
    → PUT /secrets/test/mypassword
    {type: simple, value: "secret" }
    ← 201 OK
Retrieve using Key Exchange Message:

→ GET /secrets/three/levels/down/mysecret?
type=kem&value=aaaaaa.bbbbbb.cccccc

aaaaaa.bbbbbb.cccccc == Encoded {
  "protected": {
    "kid": "<public-key-identifier>",
    "alg": "<valid alg name>" },
  "claims": {
    "sub": "mysecret",
    "exp": "<expiration time>",
    "value": "<arbitrary>" },
  "signature": "ABCDEFGHIJKLMNOPQRSTUVWXYZ"}
... and sealing ...

- Retrieve using Key Exchange Message:

  ← 200 OK

  aaaaaa.bbbbbb.cccccc.ddddddd.eeeee == {
  "protected": {
    "kid": <public-key-identifier>,
    "alg": <valid alg name>,
    "enc": <valid enc type> },

  "encrypted_key": <JWE Encrypted Key>,
  "iv": <Initialization Vector>,
  "ciphertext": <Encrypted JWS token>,
  "tag": <Authentication Tag>
  }
Accessing Custodia

- Authentication is normally done via tokens in headers, fully pluggable, determined by configuration statements

- Example Authentication directive:

  ```
  [auth:header]
  handler = custodia.httpd.authenticators.SimpleHeaderAuth
  name = REMOTE_USER
  ```

- Example Authorization directive:

  ```
  [authz:kemkeys]
  handler = ipakeys.kem.IPAKEMKey
  paths = /keys
  store = ipa
  server_keys = /etc/ipa/custodia/server.keys
  ```
Within FreeIPA

1. Admin stores new server's Public Keys in LDAP [Provisioning step]
2. New replica sends signed request for keys
3. Custodia fetches Public Keys and Authorizes based on data in LDAP
4. Custodia sends back signed and encrypted reply with keys
Questions?

Contact: simo@redhat.com