MANAGING SECRETS SAFELY

AN AUTOMATION STORY

David Barroso Pardo <dbarrosop@dravetech.com>

https://github.com/dravetech/secrets-workshop

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

DISCLAIMER

This is not a security talk. Even though we will be discussing security topics the aim is on how to consume secrets.

Always be mindful of who you and your company are, the best practices for your industry and always follow the guidelines and instructions from your security team.

WHAT ARE SECRETS?

A secret is any piece of information that can be used for authenticating a user or system or to secure information.

Examples of secrets are:

- Passwords
- Tokens
- Private keys
- ~~Usernames~~
- ~~FQDNs~~
- ~~Server ports~~

USER SECRETS

```
localhost $ ssh user@remote.acme.com
user@remote.acme.com's password:
remote #
```

Easy to store and consume, i.e. store in a password manager and enter when requested

SERVICE SECRETS

```
from netmiko import ConnectHandler

params = {
    'device_type': 'cisco_ios',
    'host':   '10.10.10.10',
    'username': 'automator',
    'password': '???',
}
device = ConnectHandler(**params)
...
```

Not so easy to store and consume, needs a store that can be queried and that is secure. This store may require additional authentication.

> 💡 This applies not only to services but any piece of code that runs without user interaction.

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

DO’S AND DON’TS

Short tips about things you can do and you should avoid when dealing with secrets.

USE DIFFERENT SECRETS WITH TIGHT PERMISSIONS

- To reduce exposure if a secret is leaked
- To make it easier to rollout new secrets if needed

MINIMIZE PAPER TRAIL

Your application will always need some secret; certificates, tokens to access external services or even a secret to access your secrets store. Make sure you provide those secrets to your application with little to no paper trail:

```
$ export SOME_SECRET_TOKEN=`cat some_secret_token`
$ ./myapp --some-secret-token $SOME_SECRET_TOKEN
```

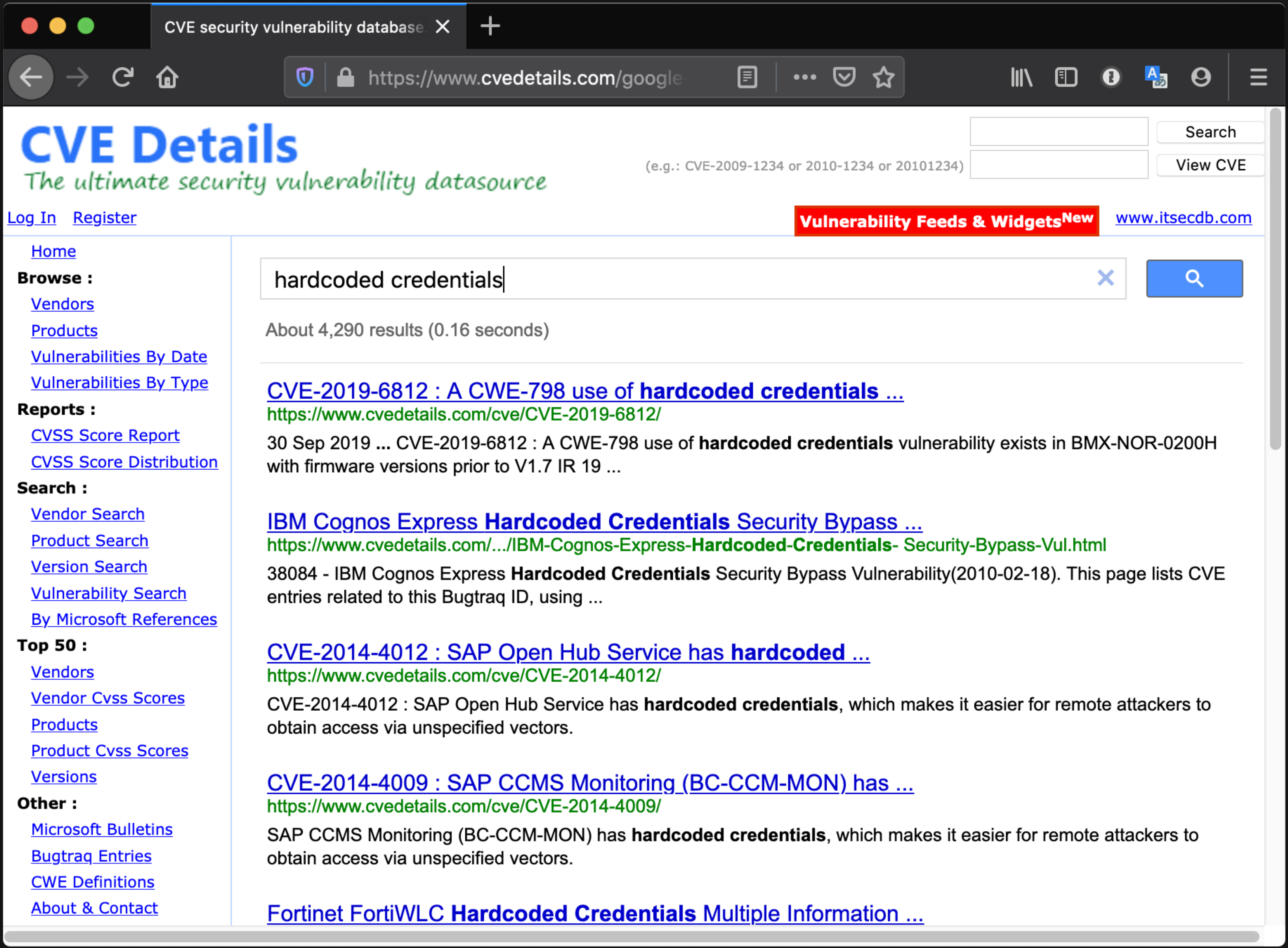
Careful with passing secrets in the command line as they will go to the history:

```
$ ./myapp --some-secret-token a-secret-token
history | tail -n 1
2476 ./myapp --some-secret-token a-secret-token
...
```

Orchestrators like `kubernetes`, `docker swarm`, etc. have mechanisms to inject secrets into environment variables and files for you, use those capabilities where available!

DO NOT HARDCODE SECRETS

Never hardcode secrets into your application, not even test/dev secrets.



CVE security vulnerability database
https://www.cvedetails.com/google
CVE Details
The ultimate security vulnerability datasource
Search
(e.g.: CVE-2009-1234 or 2010-1234 or 20101234)
View CVE
Log In
Register
Vulnerability Feeds & Widgets New
www.itsecdb.com
Home
Browse :
Vendors
Products
Vulnerabilities By Date
Vulnerabilities By Type
Reports :
CVSS Score Report
CVSS Score Distribution
Search :
Vendor Search
Product Search
Version Search
Vulnerability Search
By Microsoft References
Top 50 :
Vendors
Vendor Cvss Scores
Products
Product Cvss Scores
Versions
Other :
Microsoft Bulletins
Bugtraq Entries
CWE Definitions
About & Contact
hardcoded credentials
About 4,290 results (0.16 seconds)
CVE-2019-6812 : A CWE-798 use of hardcoded credentials ...
https://www.cvedetails.com/cve/CVE-2019-6812/
30 Sep 2019 ... CVE-2019-6812 : A CWE-798 use of hardcoded credentials vulnerability exists in BMX-NOR-0200H with firmware versions prior to V1.7 IR 19 ...
IBM Cognos Express Hardcoded Credentials Security Bypass ...
https://www.cvedetails.com/.../IBM-Cognos-Express-Hardcoded-Credentials- Security-Bypass-Vul.html
38084 - IBM Cognos Express Hardcoded Credentials Security Bypass Vulnerability(2010-02-18). This page lists CVE entries related to this Bugtraq ID, using ...
CVE-2014-4012 : SAP Open Hub Service has hardcoded ...
https://www.cvedetails.com/cve/CVE-2014-4012/
CVE-2014-4012 : SAP Open Hub Service has hardcoded credentials, which makes it easier for remote attackers to obtain access via unspecified vectors.
CVE-2014-4009 : SAP CCMS Monitoring (BC-CCM-MON) has ...
https://www.cvedetails.com/cve/CVE-2014-4009/
SAP CCMS Monitoring (BC-CCM-MON) has hardcoded credentials, which makes it easier for remote attackers to obtain access via unspecified vectors.
Fortinet FortiWLC Hardcoded Credentials Multiple Information ...

DO NOT COMMIT SECRETS TO GIT

If you store the secrets in files, add them to `.gitignore` or, even better, make sure they are outside of a git workspace.

#716292 JumpCloud API Key le

https://hackerone.com/reports/716292

SIGN IN | SIGN UP

hackerone

FOR BUSINESS FOR HACKERS HACKTIVITY COMPANY TRY HACKERONE

Vinoth Kumar (vinothkumar)

645	-	4.30	84th	19.75	91st
Reputation	Rank	Signal	Percentile	Impact	Percentile

249 #716292 **JumpCloud API Key leaked via Open Github Repository.**

Share:

State	Resolved (Closed)
Disclosed	**December 30, 2019 4:40pm +0100**
Reported To	Starbucks
Asset	Other non domain specific items (Other)
Weakness	Use of Hard-coded Credentials
Bounty	$4,000
Severity	Critical (9.7)
Participants	
Visibility	Disclosed (Full)

Collapse

SUMMARY BY STARBUCKS

vinothkumar discovered a publicly available Github repository containing a Starbucks JumpCloud API Key which provided access to internal system information.

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

INTRO TO GNUPG

> GnuPG is a complete and free implementation of the OpenPGP standard as defined by RFC4880 (also known as PGP). GnuPG allows you to encrypt and sign your data and communications; it features a versatile key management system, along with access modules for all kinds of public key directories. GnuPG, also known as GPG, is a command line tool with features for easy integration with other applications. A wealth of frontend applications and libraries are available. GnuPG also provides support for S/MIME and Secure Shell (ssh).

In this section we are going to see how to use it to encrypt a decrypt files with `GnuPG`

> 💡 Files for this section can be found in the folder `./code/gpg/`

ENCRYPTING FILES

File in clear-text:

```
$ cat unencrypted.txt
This is text we want to secure
```

Encrypting the file:

```
$ gpg --symmetric -o unencrypted.txt.gpg unencrypted.txt

$ file unencrypted.txt.gpg
unencrypted.txt.gpg: GPG symmetrically encrypted data (AES cipher)
```

DECRYPTING FILES

```
$ gpg --decrypt -o unencrypted.txt unencrypted.txt.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
```

> 💡 Password in the example file is `apassword`

MORE ABOUT GNUPG

- Smartcards
- Asymmetric encryption

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

ENCRYPTED ENVIRONMENT

In this example we are going to use an encrypted file to load environment variables that will only be available to our application.

> 💡 Example can be found under `./code/encrypted_environment/`

ENVIRONMENT FILE

The encrypted file with the secrets can be found in the same folder as `secrets.env.gpg`.

We can verify it's encrypted:

```
$ file secrets.env.gpg
secrets.env.gpg: GPG symmetrically encrypted data (AES cipher)
```

We can use the `gpg` tool to decrypt the file:

```
$ gpg --decrypt secrets.env.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
DEFAULT_PASSWORD=this-is-the-default-password
BMA_PASSWORD=this-is=the-password-for-bma
```

> 💡 Password for the encrypted file is `apassword`

As you probably noticed, we have two environment variables in the file:

- `DEFAULT_PASSWORD` - This is the password we will use unless specified otherwise
- `BMA_PASSWORD` - This is the password we will use for devices in the `bma` group

Simple python script to verify we can decrypt and load the environment correctly:

```
#!/usr/bin/env python
import os


default = os.getenv("DEFAULT_PASSWORD")
bma = os.getenv("BMA_PASSWORD")

print(f"DEFAULT_PASSWORD = {default}")
print(f"BMA_PASSWORD = {bma}")
```

> 💡 Full script can be found in the same folder as `script.py`

We can use the `env` command combined with the `gpg` tool to load the environment for our script:

```
$ env $(gpg --decrypt secrets.env.gpg) ./script.py
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
DEFAULT_PASSWORD = this-is-the-default-password
BMA_PASSWORD = this-is=the-password-for-bma
```

NORNIR EXAMPLE

As a more elaborate example we are going to see how we can apply what we have seen in this section to a nornir script

INVENTORY

> 💡 Nornir inventory can be found under `./code/nornir/`

Hosts:

```yaml
---
rtr00.bma:
    groups:
        - bma
rtr01.bma:
    groups:
        - bma
rtr00.cdg:
    groups:
        - cdg
rtr01.cdg:
    groups:
        - cdg
```

Groups:

```
---
bma: {}
cdg: {}
```

Defaults:

```
---
username: automator
```

SCRIPT

1. Create a simple task that will print each device's hostname, username and password
2. Add a short function to load the environment and assign the password to the inventory
3. Run the task over each device

> 💡 Full script can be found in the same folder as `nornir_script.py`

The task:

```
def print_credentials(task):
    print(
        f"{task.host.name} - {task.host.username}/{task.host.password}"
    )
```

The function to load the environment:

```python
def load_creds_from_env(nr):
    nr.inventory.defaults.password = os.getenv("DEFAULT_PASSWORD")

    for name, group in nr.inventory.groups.items():
        env_name = f"{name.upper()}_PASSWORD"
        group.password = os.getenv(env_name)
```

Running the task over all the hosts after loading the environment:

```
nr = InitNornir(...)

load_creds_from_env(nr)

nr.run(task=print_credentials)
```

Running the script:

```
$ env $(gpg --decrypt secrets.env.gpg) ./nornir_script.py
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
rtr00.bma - automator/this-is=the-password-for-bma
rtr01.bma - automator/this-is=the-password-for-bma
rtr00.cdg - automator/this-is-the-default-password
rtr01.cdg - automator/this-is-the-default-password
```

OTHER VARIANTS

1. Encrypt/Decrypt using asymmetric encryption
2. Decrypt directly in the python code with python-gnupg
3. Use pass

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

ENCRYPTED INVENTORY

In this example we are going to encrypt the data we want to consume with `gpg` and load it directly from the code.

To demonstrate it we are going to include the secrets in nornir’s inventory file, encrypt it and create a custom inventory plugin that will load the encrypted files directly after asking the user for the password.

THE INVENTORY

> 💡 Password for the encrypted files is `apassword`

Hosts:

```
$ gpg --decrypt hosts.yaml.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
---
rtr00.bma:
    groups:
        - bma
rtr01,bma:
    groups:
        - bma
rtr00.cdg:
    groups:
        - cdg
rtr01.cdg:
    groups:
        - cdg
```

Groups:

```
$ gpg --decrypt groups.yaml.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
---
bma:
    password: this-is-the-password-for-bma
cdg: {}
```

Defaults:

```
$ gpg --decrypt defaults.yaml.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
---
username: automator
password: this-is-the-default-password
```

THE INVENTORY PLUGIN

> 💡 Full script can be found in the same folder as `inv.py`

```python
def decrypt_and_load(filename, passphrase):
    # initialize gpg
    gpg = GPG(gnupghome=f"{os.environ['HOME']}/.gnupg")

    # open the encrypted file and decrypt it
    with open(filename, "rb") as f:
        data = gpg.decrypt_file(f, passphrase=passphrase)

    # check for errors
    if not data.ok:
        raise Exception(data.stderr)

    # load the contents as yaml
    yml = ruamel.yaml.YAML(typ="safe")
    return yml.load(data.data)
```

```python
class EncryptedInventory(Inventory):
    def __init__(
      self, host_file, group_file, defaults_file,
      passphrase, *args, **kwargs,
    ):

        hosts = decrypt_and_load(host_file, passphrase)
        groups = decrypt_and_load(group_file, passphrase)
        defaults = decrypt_and_load(defaults_file, passphrase)

        super().__init__(
          hosts=hosts, groups=groups, defaults=defaults,
          *args, **kwargs
        )
```

THE SCRIPT

```python
import getpass

nr = InitNornir(
    inventory={
        "plugin": "inv.EncryptedInventory",
        "options": {
            "host_file": "hosts.yaml.gpg",
            "group_file": "groups.yaml.gpg",
            "defaults_file": "defaults.yaml.gpg",
            "passphrase": getpass.getpass('Password:'),
        },
    },
)


nr.run(task=print_credentials)
```

💡 Full script can be found in the same folder as `nornir_script.py`

USAGE

```
$ ./nornir_script.py
Password:
rtr00.bma - automator/this-is-the-password-for-bma
rtr01.bma - automator/this-is-the-password-for-bma
rtr00.cdg - automator/this-is-the-default-password
rtr01.cdg - automator/this-is-the-default-password
```

Nornir specifics are not important, what’s important is to see you can leverage tools that have been vetted by experts in security and cryptography.

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. **Hashicorp Vault 101**
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

HASHICORP VAULT 101

Hashicorp's vault, often abbreviated as `hcv`, is a service that allows you to:

> secure, store and tightly control access to tokens, passwords, certificates, encryption keys [...] using a UI, CLI, or HTTP API.

Storing secrets:

```
$ vault kv put secret/bma/routers/ password=some-secret-password
Key                 Value
---                 -----
created_time        2020-01-06T17:32:33.690632732Z
deletion_time       n/a
destroyed           false
version             1

$ vault kv put secret/bma/switches password=some-secret-password-b
Key                 Value
---                 -----
created_time        2020-01-06T17:32:41.891355726Z
deletion_time       n/a
destroyed           false
version             1

$ vault kv put secret/cdg/routers/ password=some-secret-password-c
...
```

Listing secrets:

```
$ vault kv list secret
Keys
----
bma/
cdg/

$ vault kv list secret/bma
Keys
----
routers
switches
```

Reading secrets:

```
$ vault kv get secret/bma/routers
====== Metadata ======
Key                 Value
---                 -----
created_time        2020-01-06T17:32:33.690632732Z
deletion_time       n/a
destroyed           false
version             1

====== Data ======
Key         Value
---         -----
password    some-secret-password-a
```

Versioning support:

```
$ vault kv put secret/bma/routers password=a-new-password
Key                 Value
---                 -----
created_time        2020-01-06T17:36:52.977018765Z
deletion_time       n/a
destroyed           false
version             2
```

Retrieving latest version:

```
$ vault kv get secret/bma/routers
====== Metadata ======
Key                Value
---                -----
created_time       2020-01-06T17:36:52.977018765Z
deletion_time      n/a
destroyed          false
version            2

====== Data ======
Key           Value
---           -----
password      a-new-password
```

Retrieving a previous version:

```
$ vault kv get --version 1 secret/bma/routers
====== Metadata ======
Key                Value
---                -----
created_time       2020-01-06T17:32:33.690632732Z
deletion_time      n/a
destroyed          false
version            1

====== Data ======
Key           Value
---           -----
password      some-secret-password-a
```

PATHS

Paths allows you to create hierarchical structures to organize your passwords. For instance:

- `secret/$site` - secrets that apply to devices on a given site
- `secret/$site/$device` - secrets that apply to a given device
- `secret/credentials/$user` - credentials for a given user

Paths support multiple k/v pairs:

```
$ vault kv put secret/dbarroso favorite-color=$COLOR favorite-food=$FOOD
Key                Value
---                -----
created_time       2020-01-06T17:44:20.939288919Z
deletion_time      n/a
destroyed          false
version            1
```

SECRETS ENGINES

Secrets engines are plugins that allow you to generate, store or encrypt data. The default engine allows you to store arbitrary k/v pairs in memory pairs but other secret engines allows you to store them in cloud services, consul, etc., to integrate with Active Directory, Azure, etc., or to generate certificates or ssh keys on demand.

AUTHENTICATION

Vault support many mechanisms to authenticate users; LDAP, radius, JWT, Github, TLS certificates, Tokens, username/passwords…

POLICIES

Policies allow you to define which actions can be performed on a given path:

```
# a-sample-policy
path "secret/bma" {
  capabilities = ["create"]
}
path "secret/cdg/routers" {
  capabilities = ["read"]
}
```

Policies can be assigned to users to restrict their access:

```
$ vault write auth/userpass/users/dbarroso policies=admins
```

SEAL/UNSEAL

Vault data is stored encrypted. Unsealing is the act of telling vault how to decrypt the data while sealing is the act of telling vault to forget how to do it. Vault is quite powerful and it allows you to be able to split the keys amongst different users so a single actor can’t unseal the service.

DEV SERVER

If you want to play with vault you can:

1. Go to the folder `./code/hcv/` and start a dev server with the command `docker-compose up`
2. On a different shell execute `docker-compose exec vault-dev-server sh`, this should give you access to the docker container running vault
3. In the container execute `vault login` and enter the token `a-silly-token-for-dev`
4. Feel free to try the commands shown in the previous slides
5. Do not forget to stop the environment with `docker-compose down` once you are done.

Considerations:

1. The environment provided with this presentation is not ready for production purposes
2. Data is not persisted so stopping the environment will wipe out the secrets
3. There are no users and no policies and the root token is hardcoded

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

HCV: SECRETS

In this section we are going to:

1. store our passwords in hashicorp's vault
2. store the token to access the vault in an encrypted environment file
3. retrieve the passwords directly in our python code

> 💡 Example can be found under `./code/hcv_secrets/`

STARTING THE DEV SERVER

We can start the dev server automatically with `docker-compose up`:

```
$ docker-compose up
Creating network "hcv_secrets_default" with the default driver
Creating hcv_secrets_vault-dev-server_1 ... done
Attaching to hcv_secrets_vault-dev-server_1
vault-dev-server_1  | ==> Vault server configuration:
vault-dev-server_1  |
vault-dev-server_1  |              Api Address: http://0.0.0.0:8200
vault-dev-server_1  |                      Cgo: disabled
vault-dev-server_1  |          Cluster Address: https://0.0.0.0:8201
vault-dev-server_1  |               Listener 1: tcp (addr: "0.0.0.0:8200", cluster addres...
vault-dev-server_1  |                Log Level: info
vault-dev-server_1  |                    Mlock: supported: true, enabled: false
vault-dev-server_1  |            Recovery Mode: false
vault-dev-server_1  |                  Storage: inmem
vault-dev-server_1  |                  Version: Vault v1.3.1
...
```

Somewhere in the output you should see the following:

```
The unseal key and root token are displayed below in case you want to
seal/unseal the Vault or re-authenticate.

Unseal Key: WOLKhMovPzj30NsF2ZrAAuVaQ/957lezWWlEOS9/ID4=
Root Token: a-silly-token-for-dev

Development mode should NOT be used in production installations!
```

A reminder this is not suitable for production purposes. It also shows the root token you can use to interact with vault via its API.

Now we are going to prepopulate the server with some passwords using the script `init.sh`:

- `defaults` - Our default password if not other password is specified
- `bma` - Our default password for the group `bma`.
- `bma/rtr00` - A specific password for the device `rtr00.bma`

```
$ ./init.sh
Token (will be hidden):
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                a-silly-token-for-dev
token_accessor       h4FKkTlzYlQjrJgrvJGwTNRo
token_duration       ∞
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
...
```

TEST SCRIPT

To demonstrate we can read the passwords from vault we are going to use a test script.

> 💡 Full script can be found in the same folder as `test_script.py`

Next function will try to find the passwords for a given path or return `None` if none found:

```python
import hvac

def get_password_from_path(path):
    """
    Return the password if found or None if the path doesn't exist
    """
    client = hvac.Client()
    client.token = os.getenv("HCV_TOKEN")

    try:
        resp = client.secrets.kv.read_secret_version(path=path)
        return resp["data"]["data"].get("password")
    except InvalidPath:
        return None
```

Note that the token for vault is read from the environment. We will pass it via an encrypted file.

Now we can use the previous function to retrieve the known paths and print them on the screen:

```
print(f"default = {get_password_from_path('defaults')}")
print(f"bma = {get_password_from_path('bma')}")
print(f"bma/rtr00 = {get_password_from_path('bma/rtr00')}")
```

As seen previously, we are going to pass the secret via an encrypted environment file by combining the `env` and `gpg` tools :

```
$ gpg --decrypt secrets.env.gpg
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
HCV_TOKEN=a-silly-token-for-dev

$ env $(gpg --decrypt secrets.env.gpg) ./test_script.py
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
default = this-is-the-default-password
bma = this-is-the-password-for-bma
bma/rtr00 = this-is-rtr00-password
```

NORNIR SCRIPT

Now let’s combine everything we have seen so far to populate nornir’s passwords by reading them from vault.

Instead of `load_creds_from_env` we will use a new function that will use the previous function `get_password_from_path` to read the passwords from vault:

```
def load_creds_from_hcv(nr):
    nr.inventory.defaults.password = get_password_from_path("defaults")

    for name, group in nr.inventory.groups.items():
        group.password = get_password_from_path(name)
```

We need a transform function to populate the passwords for each host:

```python
def lookup_host_password(host):
    # a hostname can be rtr00.bma so we need to convert it to bma/rtr00
    path = f"{host.name.split('.')[1]}/{host.name.split('.')[0]}"
    host.password = get_password_from_path(path)
```

> 💡 A transform function is a function that receives a nornir Host and allows you to manipulate it. Once passed to `InitNornir`, nornir will make sure it's run on each host.

```python
nr = InitNornir(
    inventory={
        "options": {
            "host_file": "../nornir/hosts.yaml",
            "group_file": "../nornir/groups.yaml",
            "defaults_file": "../nornir/defaults.yaml",
        },
        "transform_function": lookup_host_password,
    },
)

load_creds_from_hcv(nr)

nr.run(task=print_credentials)
```

Running the script setting the environment:

```
$ env $(gpg --decrypt secrets.env.gpg) ./nornir_script.py
gpg: AES encrypted data
gpg: encrypted with 1 passphrase
rtr00.bma - automator/this-is-rtr00-password
rtr01.bma - automator/this-is-the-password-for-bma
rtr00.cdg - automator/this-is-the-default-password
rtr01.cdg - automator/this-is-the-default-password
```

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

PKI

In this section we are going to:

1. Enable a PKI in hashicorp's vault
2. Import a CA certificate
3. Use `vault`'s cli to interact with the PKI
4. Use a simple python script to generate certificates on-demand

> 💡 Example can be found under `./code/hcv_pki/`

STARTING VAULT

Like in previous examples, you can start vault with `docker-compose up`

```
$ docker-compose up
Starting hcv_pki_vault-dev-server_1 ... done
Attaching to hcv_pki_vault-dev-server_1
vault-dev-server_1  | ==> Vault server configuration:
vault-dev-server_1  |
vault-dev-server_1  |              Api Address: http://0.0.0.0:8200
vault-dev-server_1  |                      Cgo: disabled
vault-dev-server_1  |          Cluster Address: https://0.0.0.0:8201
vault-dev-server_1  |               Listener 1: tcp (addr: "0.0.0.0:8200", ...
vault-dev-server_1  |                Log Level: info
vault-dev-server_1  |                    Mlock: supported: true, enabled: false
vault-dev-server_1  |            Recovery Mode: false
vault-dev-server_1  |                  Storage: inmem
vault-dev-server_1  |                  Version: Vault v1.3.1
...
```

ENABLE PKI PLUGIN

```
# we create the following alias to invoke the vault tool inside the container
$ alias vault="docker-compose exec vault-dev-server vault"

# auth with vault, same password as previous sections
$ vault login

# Enable the pki plugin in the path switches_pki/...
$ vault secrets enable -path=switches_pki pki

# some config
$ vault secrets tune -max-lease-ttl=87600h switches_pki

$ vault write switches_pki/config/urls \
    issuing_certificates="http://127.0.0.1:8200/v1/switches_pki/ca" \
    crl_distribution_points="http://127.0.0.1:8200/v1/switches_pki/crl"
```

IMPORTING THE CA CERTIFICATE

In order to generate certificates we will need a CA certificate to sign them. We can either generate our own certificate and install them in our client machines or we can create an intermediate CA off another CA or intermediate CA.

To simplify things there is an already created self-signed CA in the folder `./ca.`. We can add it to vault as follows:

```
$ vault write switches_pki/config/ca pem_bundle=@/ca/root_bundle.pem
Success! Data written to: switches_pki/config/ca
```

PKI ROLE

Now we need to create one or more PKI roles. Roles define parameters like domains supported, maximum TTL for the certificates, etc.:

```
$ vault write switches_pki/roles/network-acme-com \
        allowed_domains="network.acme.com" \
        allow_subdomains=true \
        max_ttl="17520h" # 2 years
Success! Data written to: switches_pki/roles/network-acme-com
```

GENERATING CETIFICATES

```
$ vault write switches_pki/issue/network-acme-com common_name="rtr00.bma.network.acme.com"
Key                    Value
---                    -----
certificate            -----BEGIN CERTIFICATE-----
MIIEkjCCAnqgAwIBAgIUW/M4eFCs5LFA6smJrRlT2H8KqtUwDQYJKoZIhvcNAQEL
...

expiration             1581611003
issuing_ca             -----BEGIN CERTIFICATE-----
MIIFazCCA1OgAwIBAgIUA29vZkyJcXTOeWnmtD1FwVK44SQwDQYJKoZIhvcNAQEL
...

private_key            -----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAmtaw8eHgom1zB39PJj0Yx1NUnaDONbLSdFHlrvUJV2+fQ5Co
...

private_key_type       rsa
serial_number          15-32-65-89-7b-f9-29-86-57-13-6e-c4-c1-1e-9b-e6-5e-d4-d1-9f
```

LISTING CERTIFICATES

```
$ vault list switches_pki/certs
Keys
----
15-32-65-89-7b-f9-29-86-57-13-6e-c4-c1-1e-9b-e6-5e-d4-d1-9f
5b-f3-38-78-50-ac-e4-b1-40-ea-c9-89-ad-19-53-d8-7f-0a-aa-d5
```

DECODING CERTIFICATES

```
$ vault read switches_pki/cert/30-23-ce-9d-e2-a2-57-4a-2e-44-63-54-c9-15-3d-c5-a0-b0-68-ed \
    -format=json \
        | jq -r ".data.certificate" \
        | openssl x509 -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            30:23:ce:9d:e2:a2:57:4a:2e:44:63:54:c9:15:3d:c5:a0:b0:68:ed
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
        Validity
            Not Before: Jan 12 16:25:07 2020 GMT
            Not After : Feb 13 16:25:37 2020 GMT
        Subject: CN = rtr01.bma.network.acme.com
        Subject Public Key Info:
...
```

REVOKE CERTIFICATES

```
vault write switches_pki/revoke \
    serial_number=15-32-65-89-7b-f9-29-86-57-13-6e-c4-c1-1e-9b-e6-5e-d4-d1-9f
Key                              Value
---                              -----
revocation_time                  1578847049
revocation_time_rfc3339          2020-01-12T16:37:29.336790549Z
```

VERIFY CRL

```
$ curl -sS http://127.0.0.1:8200/v1/switches_pki/crl/pem \
    | openssl crl -inform PEM -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = AU, ST = Some-State, O = Internet Widgits Pty Ltd
        Last Update: Jan 12 17:22:05 2020 GMT
        Next Update: Jan 15 17:22:05 2020 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:48:8D:C9:56:33:AD:44:CD:51:55:D5:B2:27:F8:5B:15:62:CB:A6:6E

Revoked Certificates:
    Serial Number: 153265897BF9298657136EC4C11E9BE65ED4D19F
        Revocation Date: Jan 12 17:22:05 2020 GMT
    Signature Algorithm: sha256WithRSAEncryption
         36:51:1e:55:f1:91:e7:51:25:0c:4b:48:ea:ec:b7:e8:8c:d4:
...
```

GENERATING CETIFICATES PROGRAMMATICALLY

```python
def gen_cert(common_name):
    client = hvac.Client()
    client.token = os.getenv("HCV_TOKEN")

    resp = client.secrets.pki.generate_certificate(
        mount_point="switches_pki",    # path
        name="network-acme-com",       # role
        common_name=f"{common_name}.network.acme.com",
    )
    sn = resp.json()["data"]["serial_number"]
    key = resp.json()["data"]["private_key"]
    crt = resp.json()["data"]["certificate"]

    print(f"{common_name}.network.acme.com: {sn}")

    with open(f"certs/{common_name}.crt", "w+") as f:
        f.write(crt)  # save the cert
    with open(f"certs/{common_name}.key", "w+") as f:
        f.write(key)  # save the key
```

💡 Full example can be found in `cert_gen.py`

WHAT CAN I DO WITH THIS?

1. Integrate with ZTP to provision certificates
2. Use two CAs to provide mTLS authentication
3. Stop ignoring certificate warnings!

> 💡 Note that vault has a similar plugin to manage **ssh keys**

1. Introduction
2. Do's and Dont's
3. Intro to GnuPG
4. Encrypted Environment
5. Encrypted Inventory
6. Hashicorp Vault 101
7. Storing secrets in HCV
8. Building a PKI with HCV
9. Summary

SUMMARY

In this workshop we have seen:

1. What secrets are and why they are important
2. Some quick tips when dealing with secrets
3. Various examples on how to pass secrets to your application:
    a. Via encrypted environment
    b. Via encrypted files
4. A quick introduction to Hashicorp Vault
5. How to secure secrets with HCV
6. How to build a PKI with HCV

EOF

David Barroso Pardo <dbarrosop@dravetech.com>

https://github.com/dravetech/secrets-workshop