Manage sensitive data with Docker secrets

- Windows support
- How Docker manages secrets
- Read more about docker secret commands
- Defining and using secrets in Compose files
- Examples
  - Simple example: Get started with secrets
  - Simple example: Use secrets in a Windows service
  - Intermediate example: Use secrets with a Nginx service
    - Generate the site certificate
    - Configure the Nginx container
  - Advanced example: Use secrets with a WordPress service
  - Example: Rotate a secret
- Build support for Docker Secrets into your images
- Use Secrets in Compose

In terms of Docker Swarm services, a *secret* is a blob of data, such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network or stored unencrypted in a Dockerfile or in your application's source code. You can use Docker *secrets* to centrally manage this data and securely transmit it to only those containers that need access to it. Secrets are encrypted during transit and at rest in a Docker swarm. A given secret is only accessible to those services which have been granted explicit access to it, and only while those service tasks are running.

You can use secrets to manage any sensitive data which a container needs at runtime but you don't want to store in the image or in source control, such as:

- Usernames and passwords
- TLS certificates and keys
- SSH keys

- Other important data such as the name of a database or internal server
- Generic strings or binary content (up to 500 kb in size)

> Docker secrets are only available to swarm services, not to standalone containers. To use this feature, consider adapting your container to run as a service. Stateful containers can typically run with a scale of 1 without changing the container code.

Another use case for using secrets is to provide a layer of abstraction between the container and a set of credentials. Consider a scenario where you have separate development, test, and production environments for your application. Each of these environments can have different credentials, stored in the development, test, and production swarms with the same secret name. Your containers only need to know the name of the secret to function in all three environments.

You can also use secrets to manage non-sensitive data, such as configuration files. However, Docker supports the use of configs for storing non-sensitive data. Configs are mounted into the container's filesystem directly, without the use of a RAM disk.

Windows support

Docker includes support for secrets on Windows containers. Where there are differences in the implementations, they are called out in the examples below. Keep the following notable differences in mind:

- Microsoft Windows has no built-in driver for managing RAM disks, so within running Windows containers, secrets are persisted in clear text to the container's root disk. However, the secrets are explicitly removed when a container stops. In addition, Windows does not support persisting a running container as an image using `docker commit` or similar commands.
- On Windows, we recommend enabling BitLocker on the volume containing the Docker root directory on the host machine to ensure that secrets for running containers are encrypted at rest.
- Secret files with custom targets are not directly bind-mounted into Windows containers, since Windows does not support non-directory file bind-mounts. Instead, secrets for a container are all mounted in `C:\ProgramData\Docker\internal\secrets` (an implementation detail which should not be relied upon by applications) within the container. Symbolic links are used to point from there to the desired target of the secret within the container. The default target is `C:\ProgramData\Docker\secrets`.
- When creating a service which uses Windows containers, the options to specify UID, GID, and mode are not supported for secrets. Secrets are currently only accessible by administrators and users with `system` access within the container.

How Docker manages secrets

When you add a secret to the swarm, Docker sends the secret to the swarm manager over a mutual TLS connection. The secret is stored in the Raft log, which is encrypted. The entire Raft log is replicated across the other managers, ensuring the same high availability guarantees for secrets as for the rest of the swarm management data.

When you grant a newly-created or running service access to a secret, the decrypted secret is mounted into the container in an in-memory filesystem. The location of the mount point within the container defaults to `/run/secrets/<secret_name>` in Linux containers, or `C:\ProgramData\Docker\secrets` in Windows containers. You can also specify a custom location.

You can update a service to grant it access to additional secrets or revoke its access to a given secret at any time.

A node only has access to (encrypted) secrets if the node is a swarm manager or if it is running service tasks which have been granted access to the secret. When a container task stops running, the decrypted secrets shared to it are unmounted from the in-memory filesystem for that container and flushed from the node's memory.

If a node loses connectivity to the swarm while it is running a task container with access to a secret, the task container still has access to its secrets, but cannot receive updates until the node reconnects to the swarm.

You can add or inspect an individual secret at any time, or list all secrets. You cannot remove a secret that a running service is using. See Rotate a secret for a way to remove a secret without disrupting running services.

To update or roll back secrets more easily, consider adding a version number or date to the secret name. This is made easier by the ability to control the mount point of the secret within a given container.

Read more about `docker secret` commands

Use these links to read about specific commands, or continue to the example about using secrets with a service.

- `docker secret create`

- `docker secret inspect`
- `docker secret ls`
- `docker secret rm`
- `--secret` flag for `docker service create`
- `--secret-add` and `--secret-rm` flags for `docker service update`

Defining and using secrets in Compose files

Using secrets in a Compose file alongside Docker Swarm is as simple as defining a secret using `docker secret create`.

```
$ printf "my-secret" | docker secret create some_secret -
```

In the `compose.yml` file you can declare the secret as `external` in the top-level `secrets` attribute.

```
services:
    api:
        image: my-web-app
        environment:
            MY_ENV: /run/secrets/some_secret
        secrets:
            - some_secret

secrets:
    some_secret:
        external: true
```

Both the `docker compose` and `docker stack` commands support defining secrets in a Compose file. See the Compose file reference for details.

Examples

This section includes three graduated examples which illustrate how to use Docker secrets. The images used in these examples have been updated to make it easier to use Docker secrets. To find out how to modify your own images in a similar way, see Build support for Docker Secrets into your

images.

> These examples use a single-Engine swarm and unscaled services for simplicity. The examples use Linux containers, but Windows containers also support secrets. See Windows support.

Simple example: Get started with secrets

This simple example shows how secrets work in just a few commands. For a real-world example, continue to Intermediate example: Use secrets with a Nginx service.

1. Add a secret to Docker. The `docker secret create` command reads standard input because the last argument, which represents the file to read the secret from, is set to `-`.

   ```
   $ printf "This is a secret" | docker secret create my_secret_data -
   ```

2. Create a `redis` service and grant it access to the secret. By default, the container can access the secret at `/run/secrets/<secret_name>`, but you can customize the file name on the container using the `target` option.

   ```
   $ docker service  create --name redis --secret my_secret_data redis:alpine
   ```

3. Verify that the task is running without issues using `docker service ps`. If everything is working, the output looks similar to this:

   ```
   $ docker service ps redis

   ID            NAME     IMAGE         NODE              DESIRED STATE  CURRENT
   STATE          ERROR  PORTS
   bkna6bpn8r1a  redis.1  redis:alpine  ip-172-31-46-109  Running        Running 8
   seconds ago
   ```

   If there were an error, and the task were failing and repeatedly restarting, you would see something like this:

   ```
   $ docker service ps redis

   NAME                    IMAGE         NODE  DESIRED STATE  CURRENT STATE
   ERROR                    PORTS
   redis.1.siftice35gla    redis:alpine  moby  Running        Running 4 seconds
   ```

```
ago
 \_ redis.1.whum5b7gu13e  redis:alpine  moby  Shutdown       Failed 20 seconds
ago      "task: non-zero exit (1)"
 \_ redis.1.2s6yorvd9zow  redis:alpine  moby  Shutdown       Failed 56 seconds
ago      "task: non-zero exit (1)"
 \_ redis.1.ulfzrcyaf6pg  redis:alpine  moby  Shutdown       Failed about a minute
ago  "task: non-zero exit (1)"
 \_ redis.1.wrny5v4xyps6  redis:alpine  moby  Shutdown       Failed 2 minutes
ago       "task: non-zero exit (1)"
```

4. Get the ID of the `redis` service task container using `docker ps` , so that you can use `docker container exec` to connect to the container and read the contents of the secret data file, which defaults to being readable by all and has the same name as the name of the secret. The first command below illustrates how to find the container ID, and the second and third commands use shell completion to do this automatically.

```
$ docker ps --filter name=redis -q

5cb1c2348a59

$ docker container exec $(docker ps --filter name=redis -q) ls -l /run/secrets

total 4
-r--r--r--    1 root     root            17 Dec 13 22:48 my_secret_data

$ docker container exec $(docker ps --filter name=redis -q) cat
/run/secrets/my_secret_data

This is a secret
```

5. Verify that the secret is not available if you commit the container.

```
$ docker commit $(docker ps --filter name=redis -q) committed_redis

$ docker run --rm -it committed_redis cat /run/secrets/my_secret_data

cat: can't open '/run/secrets/my_secret_data': No such file or directory
```

6. Try removing the secret. The removal fails because the `redis` service is running and has access to the secret.

```
$ docker secret ls

ID                          NAME                CREATED             UPDATED
wwwrxza8sxy025bas86593fqs   my_secret_data      4 hours ago         4 hours ago


$ docker secret rm my_secret_data

Error response from daemon: rpc error: code = 3 desc = secret
'my_secret_data' is in use by the following service: redis
```

7. Remove access to the secret from the running `redis` service by updating the service.

```
$ docker service update --secret-rm my_secret_data redis
```

8. Repeat steps 3 and 4 again, verifying that the service no longer has access to the secret. The container ID is different, because the `service update` command redeploys the service.

```
$ docker container exec -it $(docker ps --filter name=redis -q) cat
/run/secrets/my_secret_data


cat: can't open '/run/secrets/my_secret_data': No such file or directory
```

9. Stop and remove the service, and remove the secret from Docker.

```
$ docker service rm redis


$ docker secret rm my_secret_data
```

Simple example: Use secrets in a Windows service

This is a very simple example which shows how to use secrets with a Microsoft IIS service running on Docker for Windows running Windows containers on Microsoft Windows 10. It is a naive example that stores the webpage in a secret.

This example assumes that you have PowerShell installed.

1. Save the following into a new file `index.html`.

```
<html lang="en">
  <head><title>Hello Docker</title></head>
  <body>
    <p>Hello Docker! You have deployed an HTML page.</p>
  </body>
</html>
```

2. If you have not already done so, initialize or join the swarm.

```
> docker swarm init
```

3. Save the `index.html` file as a swarm secret named `homepage`.

```
> docker secret create homepage index.html
```

4. Create an IIS service and grant it access to the `homepage` secret.

```
> docker service create `
    --name my-iis `
    --publish published=8000,target=8000 `
    --secret src=homepage,target="\inetpub\wwwroot\index.html" `
    microsoft/iis:nanoserver
```

> There is technically no reason to use secrets for this example; configs are a better fit. This example is for illustration only.

5. Access the IIS service at `http://localhost:8000/`. It should serve the HTML content from the first step.
6. Remove the service and the secret.

```
> docker service rm my-iis
> docker secret rm homepage
> docker image remove secret-test
```

Intermediate example: Use secrets with a Nginx service

This example is divided into two parts. The first part is all about generating the site certificate and does not directly involve Docker secrets at all, but it sets up the second part, where you store and use the site certificate and Nginx configuration as secrets.

Generate the site certificate

Generate a root CA and TLS certificate and key for your site. For production sites, you may want to use a service such as `Let's Encrypt` to generate the TLS certificate and key, but this example uses command-line tools. This step is a little complicated, but is only a set-up step so that you have something to store as a Docker secret. If you want to skip these sub-steps, you can use Let's Encrypt to generate the site key and certificate, name the files `site.key` and `site.crt`, and skip to Configure the Nginx container.

1. Generate a root key.

   ```
   $ openssl genrsa -out "root-ca.key" 4096
   ```

2. Generate a CSR using the root key.

   ```
   $ openssl req \
             -new -key "root-ca.key" \
             -out "root-ca.csr" -sha256 \
             -subj '/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA'
   ```

3. Configure the root CA. Edit a new file called `root-ca.cnf` and paste the following contents into it. This constrains the root CA to signing leaf certificates and not intermediate CAs.

   ```
   [root_ca]
   basicConstraints = critical,CA:TRUE,pathlen:1
   keyUsage = critical, nonRepudiation, cRLSign, keyCertSign
   subjectKeyIdentifier=hash
   ```

4. Sign the certificate.

   ```
   $ openssl x509 -req  -days 3650  -in "root-ca.csr" \
                  -signkey "root-ca.key" -sha256 -out "root-ca.crt" \
                  -extfile "root-ca.cnf" -extensions \
                  root_ca
   ```

5. Generate the site key.

   ```
   $ openssl genrsa -out "site.key" 4096
   ```

6. Generate the site certificate and sign it with the site key.

   ```
   $ openssl req -new -key "site.key" -out "site.csr" -sha256 \
             -subj '/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost'
   ```

7. Configure the site certificate. Edit a new file called `site.cnf` and paste the following contents into it. This constrains the site certificate so that it can only be used to authenticate a server and can't be used to sign certificates.

```
[server]
authorityKeyIdentifier=keyid,issuer
basicConstraints = critical,CA:FALSE
extendedKeyUsage=serverAuth
keyUsage = critical, digitalSignature, keyEncipherment
subjectAltName = DNS:localhost, IP:127.0.0.1
subjectKeyIdentifier=hash
```

8. Sign the site certificate.

```
$ openssl x509 -req -days 750 -in "site.csr" -sha256 \
    -CA "root-ca.crt" -CAkey "root-ca.key"  -CAcreateserial \
    -out "site.crt" -extfile "site.cnf" -extensions server
```

9. The `site.csr` and `site.cnf` files are not needed by the Nginx service, but you need them if you want to generate a new site certificate. Protect the `root-ca.key` file.

Configure the Nginx container

1. Produce a very basic Nginx configuration that serves static files over HTTPS. The TLS certificate and key are stored as Docker secrets so that they can be rotated easily.
   In the current directory, create a new file called `site.conf` with the following contents:

```
server {
    listen                443 ssl;
    server_name           localhost;
    ssl_certificate       /run/secrets/site.crt;
    ssl_certificate_key   /run/secrets/site.key;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

2. Create three secrets, representing the key, the certificate, and the `site.conf`. You can store any file as a secret as long as it is smaller than 500 KB. This allows you to decouple the key, certificate, and configuration from the services that use them. In each of these commands, the last argument represents the path to the file to read the secret from on

the host machine's filesystem. In these examples, the secret name and the file name are the same.

```
$ docker secret create site.key site.key


$ docker secret create site.crt site.crt


$ docker secret create site.conf site.conf
```

```
$ docker secret ls

ID                          NAME                CREATED             UPDATED
2hvoi9mnnaof7olr3z5g3g7fp   site.key       58 seconds ago      58 seconds ago
aya1dh363719pkiuoldpter4b   site.crt       24 seconds ago      24 seconds ago
zoa5df26f7vpcoz42qf2csth8   site.conf      11 seconds ago      11 seconds ago
```

3. Create a service that runs Nginx and has access to the three secrets. The last part of the `docker service create` command creates a symbolic link from the location of the `site.conf` secret to `/etc/nginx.conf.d/`, where Nginx looks for extra configuration files. This step happens before Nginx actually starts, so you don't need to rebuild your image if you change the Nginx configuration.

   > Normally you would create a Dockerfile which copies the `site.conf` into place, build the image, and run a container using your custom image. This example does not require a custom image. It puts the `site.conf` into place and runs the container all in one step.

   Secrets are located within the `/run/secrets/` directory in the container by default, which may require extra steps in the container to make the secret available in a different path. The example below creates a symbolic link to the true location of the `site.conf` file so that Nginx can read it:

   ```
   $ docker service create \
        --name nginx \
        --secret site.key \
        --secret site.crt \
        --secret site.conf \
        --publish published=3000,target=443 \
        nginx:latest \
        sh -c "ln -s /run/secrets/site.conf /etc/nginx/conf.d/site.conf && exec nginx -g
   'daemon off;'"
   ```

   Instead of creating symlinks, secrets allow you to specify a custom location using the `target` option. The example below illustrates how the `site.conf` secret is made available

at `/etc/nginx/conf.d/site.conf` inside the container without the use of symbolic links:

```
$ docker service create \
     --name nginx \
     --secret site.key \
     --secret site.crt \
     --secret source=site.conf,target=/etc/nginx/conf.d/site.conf \
     --publish published=3000,target=443 \
     nginx:latest \
     sh -c "exec nginx -g 'daemon off;'"
```

The `site.key` and `site.crt` secrets use the short-hand syntax, without a custom `target` location set. The short syntax mounts the secrets in `/run/secrets/ with the same name as the secret. Within the running containers, the following three files now exist:

- `/run/secrets/site.key`
- `/run/secrets/site.crt`
- `/etc/nginx/conf.d/site.conf`

4. Verify that the Nginx service is running.

```
$ docker service ls

ID            NAME   MODE        REPLICAS  IMAGE
zeskcec62q24  nginx  replicated  1/1       nginx:latest

$ docker service ps nginx

NAME                  IMAGE         NODE  DESIRED STATE  CURRENT STATE
ERROR  PORTS
nginx.1.9ls3yo9ugcls  nginx:latest  moby  Running        Running 3 minutes ago
```

5. Verify that the service is operational: you can reach the Nginx server, and that the correct TLS certificate is being used.

```
$ curl --cacert root-ca.crt https://localhost:3000

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
```

```
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support. refer to
<a href="https://nginx.org">nginx.org</a>.<br/>
Commercial support is available at
<a href="https://www.nginx.com">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
$ openssl s_client -connect localhost:3000 -CAfile root-ca.crt

CONNECTED(00000003)
depth=1 /C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
verify return:1
depth=0 /C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
verify return:1
---
Certificate chain
 0 s:/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
   i:/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
---
Server certificate
-----BEGIN CERTIFICATE-----
…
-----END CERTIFICATE-----
subject=/C=US/ST=CA/L=San Francisco/O=Docker/CN=localhost
issuer=/C=US/ST=CA/L=San Francisco/O=Docker/CN=Swarm Secret Example CA
---
No client certificate CA names sent
```

```
---
SSL handshake has read 1663 bytes and written 712 bytes
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 4096 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : AES256-SHA
    Session-ID: A1A8BF35549C5715648A12FD7B7E3D861539316B03440187D9DA6C2E48822853
    Session-ID-ctx:
    Master-Key:
F39D1B12274BA16D3A906F390A61438221E381952E9E1E05D3DD784F0135FB81353DA38C6D5C021CB926E
844DFC49FC4
    Key-Arg   : None
    Start Time: 1481685096
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
```

6. To clean up after running this example, remove the `nginx` service and the stored secrets.

```
$ docker service rm nginx


$ docker secret rm site.crt site.key site.conf
```

Advanced example: Use secrets with a WordPress service

In this example, you create a single-node MySQL service with a custom root password, add the credentials as secrets, and create a single-node WordPress service which uses these credentials to connect to MySQL. The next example builds on this one and shows you how to rotate the MySQL password and update the services so that the WordPress service can still connect to MySQL.

This example illustrates some techniques to use Docker secrets to avoid saving sensitive credentials within your image or passing them directly on the command line.

> This example uses a single-Engine swarm for simplicity, and uses a single-node MySQL service because a single MySQL server instance cannot be scaled by simply using a replicated service, and setting up a MySQL cluster is beyond the scope of this example.
>
> Also, changing a MySQL root passphrase isn't as simple as changing a file on disk. You must use a query or a `mysqladmin` command to change the password in MySQL.

1. Generate a random alphanumeric password for MySQL and store it as a Docker secret with the name `mysql_password` using the `docker secret create` command. To make the password shorter or longer, adjust the last argument of the `openssl` command. This is just one way to create a relatively random password. You can use another command to generate the password if you choose.

   > After you create a secret, you cannot update it. You can only remove and re-create it, and you cannot remove a secret that a service is using. However, you can grant or revoke a running service's access to secrets using `docker service update`. If you need the ability to update a secret, consider adding a version component to the secret name, so that you can later add a new version, update the service to use it, then remove the old version.

   The last argument is set to `-`, which indicates that the input is read from standard input.

   ```
   $ openssl rand -base64 20 | docker secret create mysql_password -


   l1vinzevzhj4goakjap5ya409
   ```

   The value returned is not the password, but the ID of the secret. In the remainder of this tutorial, the ID output is omitted.
   Generate a second secret for the MySQL `root` user. This secret isn't shared with the WordPress service created later. It's only needed to bootstrap the `mysql` service.

   ```
   $ openssl rand -base64 20 | docker secret create mysql_root_password -
   ```

   List the secrets managed by Docker using `docker secret ls`:

   ```
   $ docker secret ls


   ID                          NAME                  CREATED             UPDATED
   l1vinzevzhj4goakjap5ya409   mysql_password        41 seconds ago      41 seconds ago
   yvsczlx9votfw3l0nz5rlidig   mysql_root_password   12 seconds ago      12 seconds ago
   ```

   The secrets are stored in the encrypted Raft logs for the swarm.
2. Create a user-defined overlay network which is used for communication between the MySQL and WordPress services. There is no need to expose the MySQL service to any external host or container.

```
$ docker network create -d overlay mysql_private
```

3. Create the MySQL service. The MySQL service has the following characteristics:
   - Because the scale is set to `1`, only a single MySQL task runs. Load-balancing MySQL is left as an exercise to the reader and involves more than just scaling the service.
   - Only reachable by other containers on the `mysql_private` network.
   - Uses the volume `mydata` to store the MySQL data, so that it persists across restarts to the `mysql` service.
   - The secrets are each mounted in a `tmpfs` filesystem at `/run/secrets/mysql_password` and `/run/secrets/mysql_root_password`. They are never exposed as environment variables, nor can they be committed to an image if the `docker commit` command is run. The `mysql_password` secret is the one used by the non-privileged WordPress container to connect to MySQL.
   - Sets the environment variables `MYSQL_PASSWORD_FILE` and `MYSQL_ROOT_PASSWORD_FILE` to point to the files `/run/secrets/mysql_password` and `/run/secrets/mysql_root_password`. The `mysql` image reads the password strings from those files when initializing the system database for the first time. Afterward, the passwords are stored in the MySQL system database itself.
   - Sets environment variables `MYSQL_USER` and `MYSQL_DATABASE`. A new database called `wordpress` is created when the container starts, and the `wordpress` user has full permissions for this database only. This user cannot create or drop databases or change the MySQL configuration.

     ```
     $ docker service create \
          --name mysql \
          --replicas 1 \
          --network mysql_private \
          --mount type=volume,source=mydata,destination=/var/lib/mysql \
          --secret source=mysql_root_password,target=mysql_root_password \
          --secret source=mysql_password,target=mysql_password \
          -e MYSQL_ROOT_PASSWORD_FILE="/run/secrets/mysql_root_password" \
          -e MYSQL_PASSWORD_FILE="/run/secrets/mysql_password" \
          -e MYSQL_USER="wordpress" \
          -e MYSQL_DATABASE="wordpress" \
          mysql:latest
     ```

4. Verify that the `mysql` container is running using the `docker service ls` command.

   ```
   $ docker service ls


   ID            NAME   MODE        REPLICAS  IMAGE
   wvnh0siktqr3  mysql  replicated  1/1       mysql:latest
   ```

5. Now that MySQL is set up, create a WordPress service that connects to the MySQL service. The WordPress service has the following characteristics:
   - Because the scale is set to `1`, only a single WordPress task runs. Load-balancing WordPress is left as an exercise to the reader, because of limitations with storing WordPress session data on the container filesystem.
   - Exposes WordPress on port 30000 of the host machine, so that you can access it from external hosts. You can expose port 80 instead if you do not have a web server running on port 80 of the host machine.
   - Connects to the `mysql_private` network so it can communicate with the `mysql` container, and also publishes port 80 to port 30000 on all swarm nodes.
   - Has access to the `mysql_password` secret, but specifies a different target file name within the container. The WordPress container uses the mount point `/run/secrets/wp_db_password`.
   - Sets the environment variable `WORDPRESS_DB_PASSWORD_FILE` to the file path where the secret is mounted. The WordPress service reads the MySQL password string from that file and add it to the `wp-config.php` configuration file.
   - Connects to the MySQL container using the username `wordpress` and the password in `/run/secrets/wp_db_password` and creates the `wordpress` database if it does not yet exist.
   - Stores its data, such as themes and plugins, in a volume called `wpdata` so these files persist when the service restarts.

```
$ docker service create \
     --name wordpress \
     --replicas 1 \
     --network mysql_private \
     --publish published=30000,target=80 \
     --mount type=volume,source=wpdata,destination=/var/www/html \
     --secret source=mysql_password,target=wp_db_password \
     -e WORDPRESS_DB_USER="wordpress" \
     -e WORDPRESS_DB_PASSWORD_FILE="/run/secrets/wp_db_password" \
     -e WORDPRESS_DB_HOST="mysql:3306" \
     -e WORDPRESS_DB_NAME="wordpress" \
     wordpress:latest
```

6. Verify the service is running using `docker service ls` and `docker service ps` commands.

```
$ docker service ls

ID            NAME       MODE        REPLICAS  IMAGE
wvnh0siktqr3  mysql      replicated  1/1       mysql:latest
nzt5xzae4n62  wordpress  replicated  1/1       wordpress:latest
```

```
$ docker service ps wordpress

ID            NAME         IMAGE             NODE  DESIRED STATE  CURRENT
STATE           ERROR  PORTS
aukx6hgs9gwc  wordpress.1  wordpress:latest  moby  Running        Running 52 seconds
ago
```

At this point, you could actually revoke the WordPress service's access to the `mysql_password` secret, because WordPress has copied the secret to its configuration file `wp-config.php`. Don't do that for now, because we use it later to facilitate rotating the MySQL password.

7. Access `http://localhost:30000/` from any swarm node and set up WordPress using the web-based wizard. All of these settings are stored in the MySQL `wordpress` database. WordPress automatically generates a password for your WordPress user, which is completely different from the password WordPress uses to access MySQL. Store this password securely, such as in a password manager. You need it to log into WordPress after rotating the secret.
   Go ahead and write a blog post or two and install a WordPress plugin or theme to verify that WordPress is fully operational and its state is saved across service restarts.
8. Do not clean up any services or secrets if you intend to proceed to the next example, which demonstrates how to rotate the MySQL root password.

Example: Rotate a secret

This example builds upon the previous one. In this scenario, you create a new secret with a new MySQL password, update the `mysql` and `wordpress` services to use it, then remove the old secret.

> Changing the password on a MySQL database involves running extra queries or commands, as opposed to just changing a single environment variable or a file, since the image only sets the MySQL password if the database doesn't already exist, and MySQL stores the password within a MySQL database by default. Rotating passwords or other secrets may involve additional steps outside of Docker.

1. Create the new password and store it as a secret named `mysql_password_v2`.
   ```
   $ openssl rand -base64 20 | docker secret create mysql_password_v2 -
   ```
2. Update the MySQL service to give it access to both the old and new secrets. Remember that you cannot update or rename a secret, but you can revoke a secret and grant access to it using a new target filename.

```
$ docker service update \
     --secret-rm mysql_password mysql


$ docker service update \
     --secret-add source=mysql_password,target=old_mysql_password \
     --secret-add source=mysql_password_v2,target=mysql_password \
     mysql
```

Updating a service causes it to restart, and when the MySQL service restarts the second time, it has access to the old secret under `/run/secrets/old_mysql_password` and the new secret under `/run/secrets/mysql_password`.
Even though the MySQL service has access to both the old and new secrets now, the MySQL password for the WordPress user has not yet been changed.

> This example does not rotate the MySQL `root` password.

3. Now, change the MySQL password for the `wordpress` user using the `mysqladmin` CLI. This command reads the old and new password from the files in `/run/secrets` but does not expose them on the command line or save them in the shell history.
   Do this quickly and move on to the next step, because WordPress loses the ability to connect to MySQL.
   First, find the ID of the `mysql` container task.

   ```
   $ docker ps --filter name=mysql -q


   c7705cf6176f
   ```

   Substitute the ID in the command below, or use the second variant which uses shell expansion to do it all in a single step.

   ```
   $ docker container exec <CONTAINER_ID> \
       bash -c 'mysqladmin --user=wordpress --password="$(<
   /run/secrets/old_mysql_password)" password "$(< /run/secrets/mysql_password)"'
   ```

   Or:

   ```
   $ docker container exec $(docker ps --filter name=mysql -q) \
       bash -c 'mysqladmin --user=wordpress --password="$(<
   /run/secrets/old_mysql_password)" password "$(< /run/secrets/mysql_password)"'
   ```

4. Update the `wordpress` service to use the new password, keeping the target path at `/run/secrets/wp_db_password`. This triggers a rolling restart of the WordPress service and the new secret is used.

```
$ docker service update \
     --secret-rm mysql_password \
     --secret-add source=mysql_password_v2,target=wp_db_password \
     wordpress
```

5. Verify that WordPress works by browsing to http://localhost:30000/ on any swarm node again. Use the WordPress username and password from when you ran through the WordPress wizard in the previous task.
   Verify that the blog post you wrote still exists, and if you changed any configuration values, verify that they are still changed.
6. Revoke access to the old secret from the MySQL service and remove the old secret from Docker.

   ```
   $ docker service update \
        --secret-rm mysql_password \
        mysql


   $ docker secret rm mysql_password
   ```

7. Run the following commands to remove the WordPress service, the MySQL container, the `mydata` and `wpdata` volumes, and the Docker secrets:

   ```
   $ docker service rm wordpress mysql


   $ docker volume rm mydata wpdata


   $ docker secret rm mysql_password_v2 mysql_root_password
   ```

Build support for Docker Secrets into your images

If you develop a container that can be deployed as a service and requires sensitive data, such as a credential, as an environment variable, consider adapting your image to take advantage of Docker secrets. One way to do this is to ensure that each parameter you pass to the image when creating the container can also be read from a file.

Many of the Docker Official Images in the Docker library, such as the wordpress image used in the above examples, have been updated in this way.

When you start a WordPress container, you provide it with the parameters it needs by setting them as environment variables. The WordPress image has been updated so that the environment variables which contain important data for WordPress, such as `WORDPRESS_DB_PASSWORD`, also have variants which can read their values from a file (`WORDPRESS_DB_PASSWORD_FILE`). This strategy ensures that backward compatibility is preserved, while allowing your container to read the information from a Docker-managed secret instead of being passed directly.

> Docker secrets do not set environment variables directly. This was a conscious decision, because environment variables can unintentionally be leaked between containers (for instance, if you use `--link`).

Use Secrets in Compose

```
services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_root_password
       - db_password

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
```

```
        WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
      secrets:
        - db_password


secrets:
   db_password:
     file: db_password.txt
   db_root_password:
     file: db_root_password.txt

volumes:
    db_data:
```

This example creates a simple WordPress site using two secrets in a Compose file.

The top-level element `secrets` defines two secrets `db_password` and `db_root_password`.

When deploying, Docker creates these two secrets and populates them with the content from the file specified in the Compose file.

The `db` service uses both secrets, and `wordpress` is using one.

When you deploy, Docker mounts a file under `/run/secrets/<secret_name>` in the services. These files are never persisted on disk, but are managed in memory.

Each service uses environment variables to specify where the service should look for that secret data.

More information on short and long syntax for secrets can be found in the Compose Specification.

□□□ #2
□ □□□ □□□ 2026-10-31 19:31:46 UTC
□ □□□ □□□ 2026-10-31 19:31:49 UTC