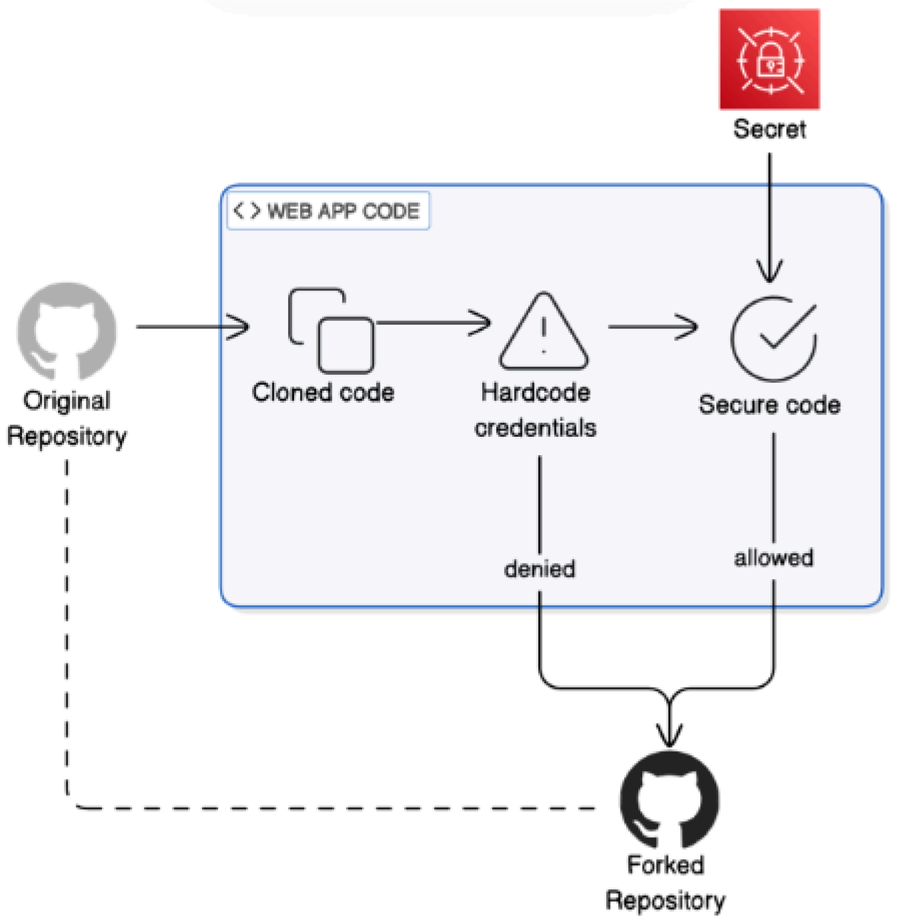
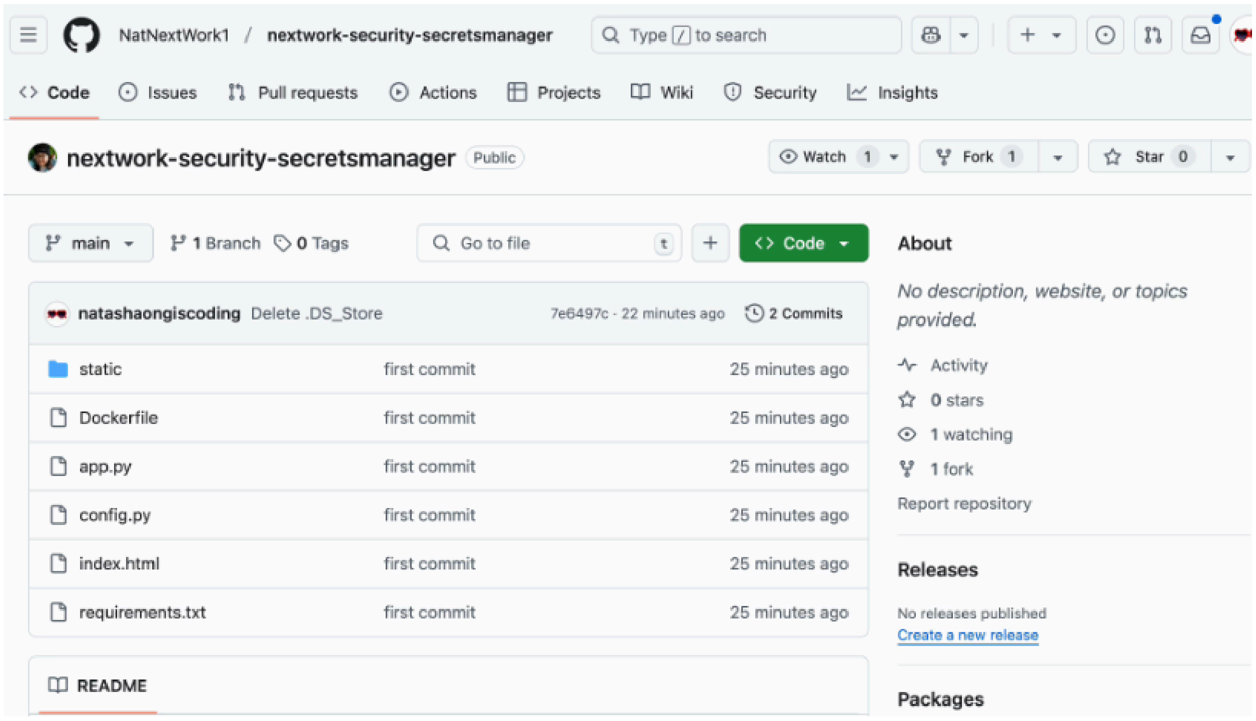
Access and Explore the Repository



- **Open** your web browser and head to [this GitHub repository.]()

- This GitHub repository contains the code for our insecure web app. This web app will display the names of S3 buckets in your AWS account.



nextwork-security-secretsmanager Public

natashaongiscoding Delete .DS_Store 7e6497c · 22 minutes ago 2 Commits

static	first commit	25 minutes ago
Dockerfile	first commit	25 minutes ago
app.py	first commit	25 minutes ago
config.py	first commit	25 minutes ago
index.html	first commit	25 minutes ago
requirements.txt	first commit	25 minutes ago

README

About

No description, website, or topics provided.

Activity
0 stars
1 watching
1 fork
Report repository

Releases

No releases published
Create a new release

Packages

Understand the Application Code

- In the repository's file list, select app.py

💡 **What is app.py?**
app.py is the main file containing the logic of our web app i.e. how the app will respond to user requests and interact with AWS services. In our case, app.py contains all the code that connects to your AWS account and lists the names of your S3 buckets

Let's explore app.py and understand how it works!

- Look at the very top of the app.py file.
- Find the lines that start with import. These lines are **import statements**

Next, in line 4, notice the line # Import your temporary, hard-coded credentials followed by import config.

Scroll down to see the read_index function. This function defines what happens when you visit the main page of the web app.

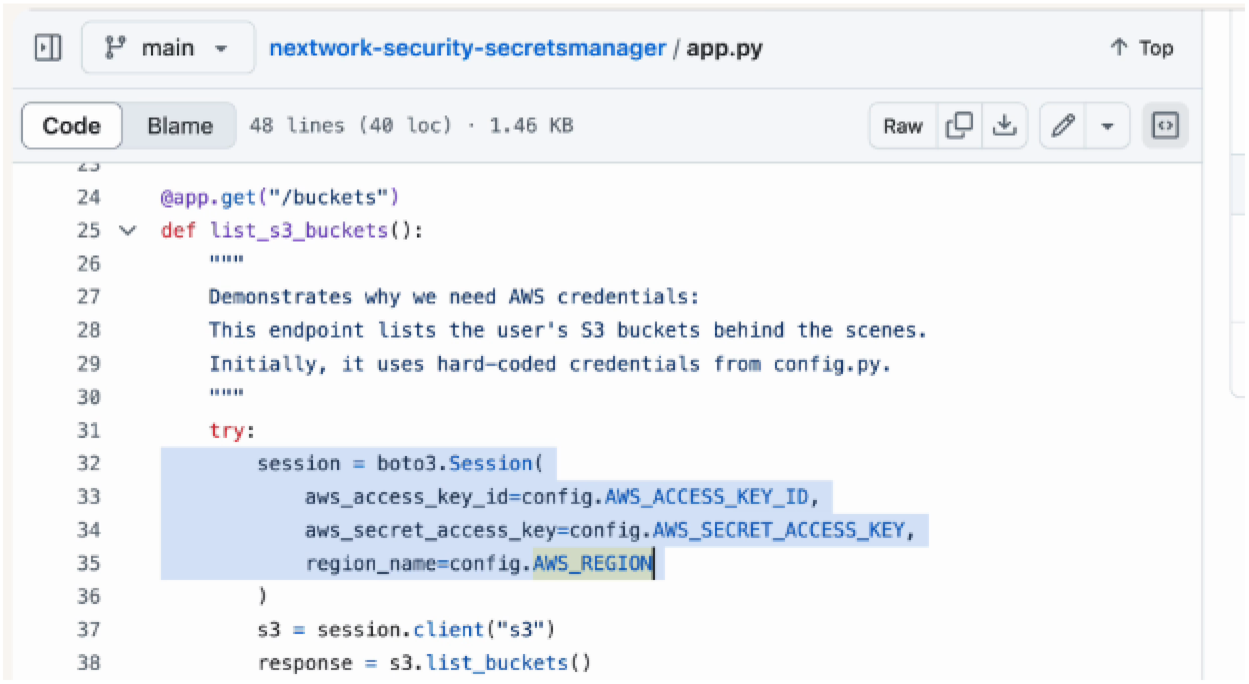
Scroll further down to find the list_s3_buckets function. This function is responsible for retrieving and displaying your S3 buckets.

Examine config.py

- In the file list on the left, let's select config.py next.

💡 **What is config.py?**
config.py is a file that contains the configuration settings for our web app. Think of it as your app's settings file.

-
- Let's see how app.py uses these credentials.
- Go back to app.py by selecting it in the file list.
- Find the list_s3_buckets function again. You can see that it uses config.AWS_ACCESS_KEY_ID, config.AWS_SECRET_ACCESS_KEY, and config.AWS_REGION to connect to AWS.


main
nextwork-security-secretsmanager / app.py
Top
Code
Blame
48 lines (40 loc) · 1.46 KB
Raw

```
@app.get("/buckets")
def list_s3_buckets():
    """
    Demonstrates why we need AWS credentials:
    This endpoint lists the user's S3 buckets behind the scenes.
    Initially, it uses hard-coded credentials from config.py.
    """
    try:
        session = boto3.Session(
            aws_access_key_id=config.AWS_ACCESS_KEY_ID,
            aws_secret_access_key=config.AWS_SECRET_ACCESS_KEY,
            region_name=config.AWS_REGION
        )
        s3 = session.client("s3")
        response = s3.list_buckets()
```

The Security Risks...

Notice how the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY are directly written in config.py. If this were a real, public application, and you shared the code on a GitHub repository, anyone could see these credentials! 😱

Exposing your AWS credentials publicly is a **major security risk** and is **extremely unsafe** for production applications. Once someone else gets access to your credentials, they can use them to access your AWS account, delete resources, steal data, and cause damage.

To make this web app more secure, we'll make a copy of it in our local computer, and then we'll play around with how config.py stores credentials in our copy.

Set Up Your Local Environment

We'll be using the terminal to make a copy of the GitHub repository's code:

From your Command Prompt, let's head to your **Documents** folder, a common place to store project files. We'll clone the repository there.

- In your Command Prompt, type cd Documents and press **Enter**.

Clone the GitHub Repository

Now we'll **clone** the GitHub repository to your local machine. **Cloning** copies all the files from the GitHub repository to a new folder in your computer.

- In your terminal, type the following command to clone the repository:
    - Git clone https://github.com/NatNextWork1/nextwork-security-secretsmanager.git
- After cloning, we need to go inside the newly created folder that contains the web app's code.
- In your terminal, type cd nextwork-security-secretsmanager and press **Enter**.
- Type dir and press **Enter** to list the files in your directory.
- You should see files like app.py, config.py, Dockerfile, and requirements.txt. Yup, these are the same files we saw in the GitHub repository!
- Open config.py in a text editor. You can use notepad config.
- Replace the placeholder values in config.py with these example AWS credentials:
    - AWS_ACCESS_KEY_ID = "AKIAW3MEFRAFTQM5FHKE"
    - AWS_SECRET_ACCESS_KEY = "F0b8s5m+pOZsttvBCirr1BOutuvCpqXMW2Y1qAxY"
    - AWS_REGION = "us-east-2"

Save the changes you've made in config.py

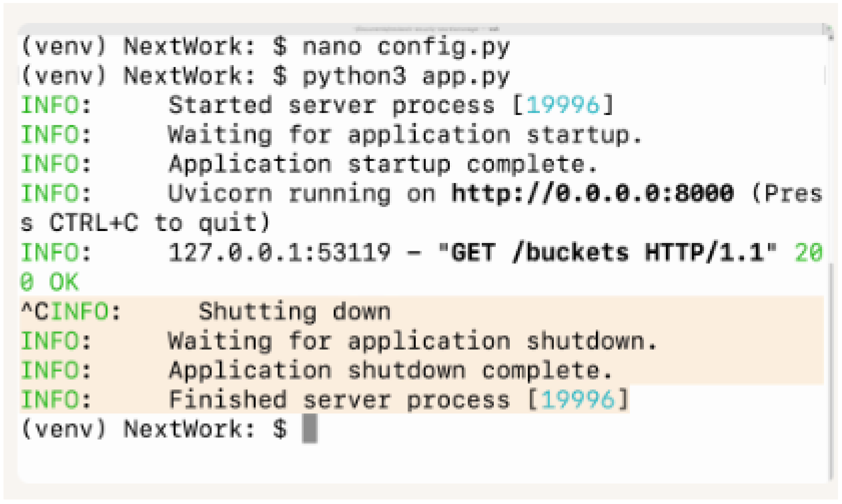
Fork the Web App Code on GitHub

Forking is like saving your own copy to your github with your own edits

- Go back to the original GitHub repository page in your browser: https://github.com/NatNextWork1/nextwork-security-secretsmanager.
- In the top right corner of the repository page, click the **Fork** button.
- After forking is complete, you should be redirected to your forked repository.
- The URL in your browser should now start with github.com/YOUR_GITHUB_USERNAME/nextwork-security-secretsmanager, where YOUR_GITHUB_USERNAME is your GitHub username. This is now your own copy of the web app code!

Push Your Code to GitHub

Nice! With your forked repository available, you can now push your version of the web app. Let's make it public! We'll get back to our terminal to connect our edited code to the forked repository.

- Head back to your terminal where the web app is running.
- Press Ctrl+C to stop the local web app.
- 

```
(venv) NextWork: $ nano config.py
(venv) NextWork: $ python3 app.py
INFO:     Started server process [19996]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Pres
s CTRL+C to quit)
INFO:     127.0.0.1:53119 - "GET /buckets HTTP/1.1" 20
0 OK
^CINFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [19996]
(venv) NextWork: $
```

In case you haven't already, let's initialize Git in your local project directory. This tells Git to track the changes you make to the code.

- In your terminal, type git init and press **Enter**. This command might show **Reinitialized existing Git repository** if it's already initialized, which is fine.
- Now, we need to connect your local repository to your forked repository on GitHub. To do this, we'll add a remote named "origin" that points to your fork's URL.
    - git remote add origin https://github.com/russgeis/nextwork-security-secretsmanager.git
- Go to your forked repository on GitHub in your browser.
- Click the green **Code** button.
- Make sure **HTTPS** is selected.
- Copy the **URL**.
- Paste this URL into the git remote add origin command in your terminal, replacing <your-forked-repo-url>.
- Let's change the existing remote "origin" to the new forked repository.
- In your terminal, run the following command:
- git remote set-url origin https://github.com/russgeis/nextwork-security-secretsmanager.git
- Let's verify that the remote "origin" was set up correctly.
- In your terminal, type git remote -v and press **Enter**. This command will show the remote repositories configured for your local repository.
- You should see output like this, showing that origin is set to your forked repository URL for both fetch (fetching code from GitHub) and push (pushing code to GitHub).
- Stage all the changes you've made in your local project by using git add .. This adds all files to the staging area, meaning Git is now tracking these files for changes.

- In your terminal, type git commit -m "Updated config.py" and press **Enter**. Committing in Git saves a snapshot of your changes.
- Finally, let's push your local commits to your forked repository on GitHub. This command uploads your code to the "main" branch of your "origin" remote.
- In your terminal, type git push -u origin main and press **Enter**.

GitHub's Secret Scanning Block

- Scroll up in the terminal output to see the detailed error message.

Uh oh! You probably see an error message saying **"Push cannot contain secrets"**. GitHub automatically scans your code for secrets and blocks the push if it detects any, like AWS credentials.

In our case, GitHub detected that you are trying to push code that contains an **AWS Access Key ID** and **Secret Access Key** in config.py. This is a great security feature from GitHub to prevent accidental exposure of secrets!

Man, GitHub is so smart 😮‍💨

- Running into this error highlights the exact problem we're trying to solve in this project: **how do you manage secrets securely** so they're not exposed in your code? In this case, it's even stopped us from sharing our code publicly!
- This is where **AWS Secrets Manager** comes in! In the next step, we'll learn how to use Secrets Manager to store your AWS credentials securely, plus retrieve them in your application without hardcoding them in config.py.

Create Secret in Secrets Manager

- So, GitHub rightly blocked us from pushing our code with hardcoded credentials. This highlights exactly why we need a secure way to manage secrets! The solution is to use a dedicated service for managing secrets securely - AWS Secrets Manager.
- In this step, we're going to use AWS Secrets Manager to store your AWS credentials securely. We'll create a new secret in Secrets Manager, and then update your `config.py` file to retrieve these credentials from Secrets Manager instead of hardcoding them.

  Head back to your [AWS Management Console]() in your browser.

- This time, let's head to the Secrets Manager console.

Now let's create a new secret to securely store our AWS credentials.

- In the left navigation pane of the Secrets Manager console, select **Secrets**.
- On the Secrets Manager dashboard, select **Store a new secret**.
- In the **Choose secret type** section, select the **Other type of secret** option.

- In the **Key** field, enter AWS_ACCESS_KEY_ID.
- Click **Add row**.
- In the new **Key** field that appears, enter AWS_SECRET_ACCESS_KEY
- Go back to your config.py file.
- Copy the **Access key ID**.
- Go back to the Secrets Manager console.
- In the **Value** field next to AWS_ACCESS_KEY_ID, paste the **Access key ID** you just copied.
- Then for AWS_SECRET_ACCESS_KEY, copy the **Secret access key** from config.py and paste it into the **Value** field.
- Scroll down to the **Encryption key** section below **Key/value pairs**. Secrets Manager encrypts your secrets to keep them secure. By default, it uses an AWS managed key.
- Click the **Next** button at the bottom right of the page.

Now, we need to give our secret a name and description.

- In the **Configure secret name and description** page, under **Secret name**, enter aws-access-key.
- Under **Description - optional**, you can enter a description like Created to replace hard-coded access key credentials in config.py.
- There are also optional settings sections that we'll skip for now: **Tags - optional**, **Resource permissions - optional**, and **Replicate secret - optional**.
- Click the **Next** button at the bottom right of the page again.
- We're now at the last setup page **Configure rotation - optional**
- **Select the Next button at the bottom right of the page.**
- Click the **Store** button at the bottom right of the page.

Update config.py

Alright, now comes the fun part! 🎨 We're going to transform our `config.py` file from a security risk into a secure, professional piece of code. Instead of having our AWS credentials just sitting there in plain text (yikes!), we'll use the sample code from Secrets Manager to fetch them securely.

Grab the Sample Code

- You should see a green banner at the top of the page saying **Successfully stored the secret**.
- Select **See sample code** - let's use the sample code from Secrets Manager to update our config.py file.
- We need the Python code snippet to integrate with our application. Let's copy the relevant portions.
- In the **Sample code** section that appears, click on the **Python 3** tab.

Select and copy the code starting from **line 6** (the first import statement) down to the end of the code block.

Let's delete the old config.py file and create a new one with our secure code! The instructions for this depends on your operating system

In the empty Notepad window, paste the Python code from Secrets Manager (don't forget to find this code in your Secrets Manager console).

Let’s get to knw the config.py file

- import boto3: This line imports the boto3 library, which is the AWS SDK for Python. We need boto3 to interact with AWS Secrets Manager.
- from botocore.exceptions import ClientError: This line imports the ClientError exception class from botocore, which is a dependency of boto3. We'll use this to handle potential errors when retrieving the secret.

The core of the pasted code is the get_secret() function:

This function is responsible for retrieving the secret from Secrets Manager. Let's break down what it does:

- secret_name = "aws-access-key": This line defines the name of the secret to retrieve, which is "aws-access-key", the name we gave to our secret in Secrets Manager.
- region_name = "us-east-2": This line defines the AWS region where Secrets Manager is located. **Make sure this matches the AWS region you are using (e.g., "us-east-2" for Ohio).**
- session = boto3.session.Session() and client = session.client(...): These lines create a boto3 client for interacting with the Secrets Manager service.
- The try...except block handles potential errors when retrieving the secret.

Next, we also need to add new lines to the code to retrieve the credentials from get_secret()!

Add this block to the end of the code:

return json.loads(secret)

# Retrieve credentials from Secrets Manager

credentials = get_secret()

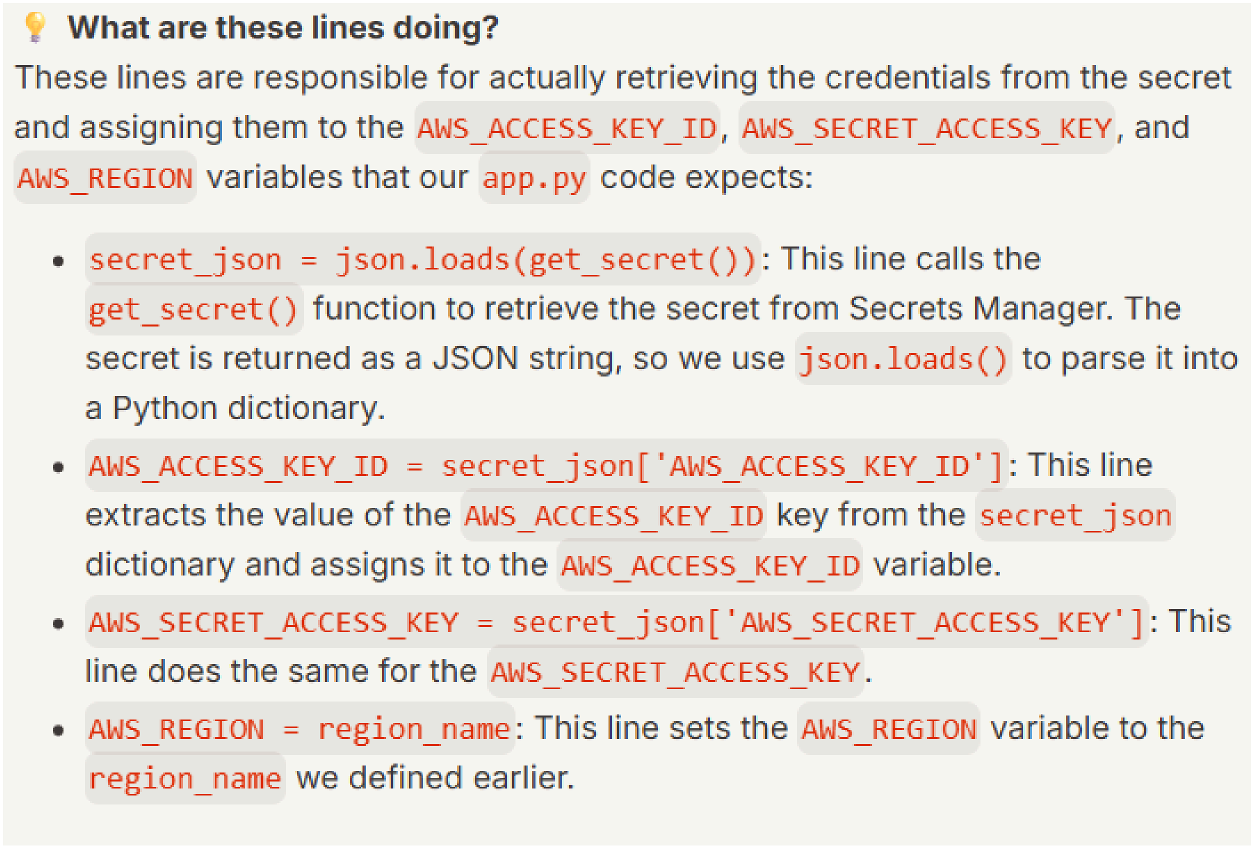
# Extract the values; if AWS_REGION isn't in the secret, use the region from the session

AWS_ACCESS_KEY_ID = credentials.get("AWS_ACCESS_KEY_ID")

AWS_SECRET_ACCESS_KEY = credentials.get("AWS_SECRET_ACCESS_KEY")

AWS_REGION = credentials.get("AWS_REGION", boto3.session.Session().region_name or "us-east-2")

- Add this to the import statement at the top since its a new json doc:
  - Import json



💡 **What are these lines doing?**

These lines are responsible for actually retrieving the credentials from the secret and assigning them to the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_REGION` variables that our `app.py` code expects:

- `secret_json = json.loads(get_secret())`: This line calls the `get_secret()` function to retrieve the secret from Secrets Manager. The secret is returned as a JSON string, so we use `json.loads()` to parse it into a Python dictionary.
- `AWS_ACCESS_KEY_ID = secret_json['AWS_ACCESS_KEY_ID']`: This line extracts the value of the `AWS_ACCESS_KEY_ID` key from the `secret_json` dictionary and assigns it to the `AWS_ACCESS_KEY_ID` variable.
- `AWS_SECRET_ACCESS_KEY = secret_json['AWS_SECRET_ACCESS_KEY']`: This line does the same for the `AWS_SECRET_ACCESS_KEY`.
- `AWS_REGION = region_name`: This line sets the `AWS_REGION` variable to the `region_name` we defined earlier.

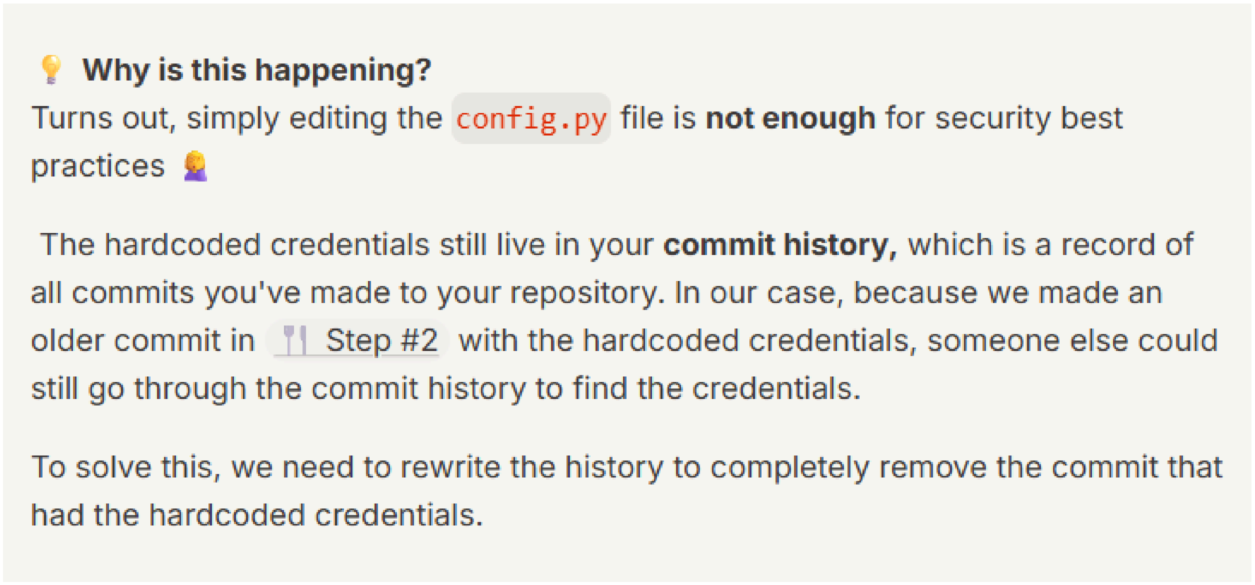
Push Changes

Let's share our newly secured code with the world! 🌎 Remember how GitHub's secret scanning blocked our push earlier? Now that we've properly moved our credentials to AWS Secrets Manager, we can confidently push our code without exposing any sensitive information.

Save, Commit and Push Your Changes

- Make sure you're back in your terminal.
- Stage all the changes you've made in your local project by using git add .. As a recap, this adds all files to the staging area, meaning Git is now tracking these files for changes.
- In your terminal, type git commit -m "Updated config.py with Secrets Manager credentials" and press **Enter**. Committing in Git saves a snapshot of your changes.
- Nice! You should see output showing you that files have been changed and committed.
- Finally, let's push your local commits to your forked repository on GitHub. Run git push -u origin main

- Oh no! We still get the same error in the terminal!



💡 **Why is this happening?**
Turns out, simply editing the `config.py` file is **not enough** for security best practices 🤦‍♀️

The hardcoded credentials still live in your **commit history,** which is a record of all commits you've made to your repository. In our case, because we made an older commit in 🍴 Step #2 with the hardcoded credentials, someone else could still go through the commit history to find the credentials.

To solve this, we need to rewrite the history to completely remove the commit that had the hardcoded credentials.

Remove Hardcoded Credentials from Commit History

We'll have to use use a special command called git rebase to 🪄 rewrite history 🪄 and remove the commit that introduced the credentials.

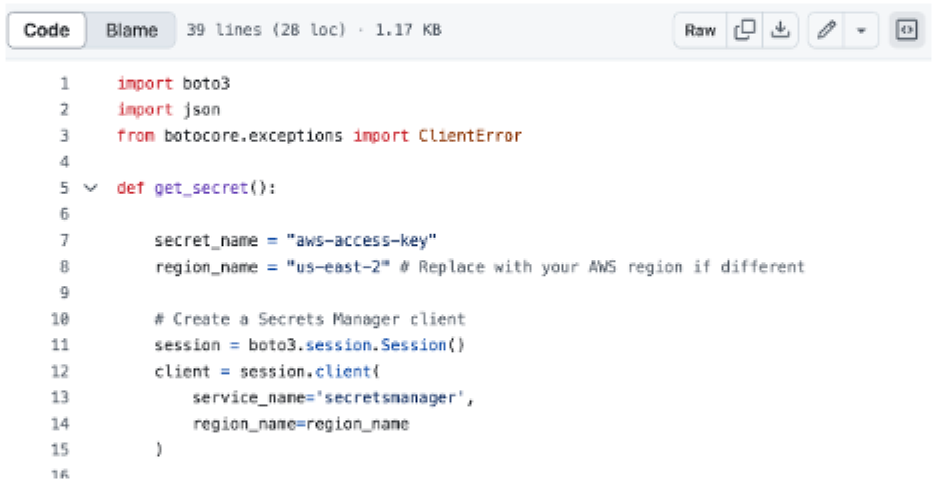
- To rewrite our commit history, we first need to identify the ID that we want to write out of history.
- Let's identify the ID of the commit where you added the AWS credentials - scroll up your terminal's history, and find the commit associated with exposed credentials.
- Once you've found the commit ID, take note of the **first 7 digits** of that ID. This is going to be important! You can even copy it and paste it in a random text file to reference it later.
- Once you've identified the commit, run the following command in your terminal:
    - git rebase -i --root
- This will open an interactive rebase editor in your default text editor. You'll see a list of your commits.
- Use your the up and down arrows on your keyboard to get to the commit that contains the AWS credentials. This is the 7-digit ID you identified earlier!
- Then, where it says pick at the beginning of that line, replace it with d and press down arrow to erase. This tells Git to **drop** i.e. remove this commit in the rebase.
- **Save** the file and **close** the editor. You can do this by typing :wq and pressing **Enter** on your keyboard.
- Git will now start the rebase, so it will remove the commit you marked for deletion from your commit history.
- Oh wait! Looks like the rebase is not so simple after all. You might run into **merge conflicts** during the rebase.
- Open the config.py file again

    - Use the arrow keys to navigate through the file.
    - Delete the entire section between <<<<<<< HEAD and ======= (this removes the hardcoded credentials).
    - Then, at the end of the file, delete the ======= line.
    - Delete the >>>>>>> feature-branch line.
    - Press Ctrl+O to save your changes.
    - Press Enter to confirm.
    - Press Ctrl+X to exit nano.
    -
- After resolving the conflicts, **save** and exit the file.
- In your terminal, run the following commands to stage and commit our changes:
    - git add config.py
    - git commit -m "Resolved merge conflicts"
    - git rebase --continue
    - Bash git push
- **Great job!** You've just removed some prettyyyy sensitive data from your Git commit history. Now, let's verify that the AWS credentials can't be found in your public repository!

Verify Your GitHub Repository

Let's check on GitHub that the commit history is updated and your config.py file doesn't contain any sensitive information.

- Head to your forked repository on GitHub.
- You should see that config.py is **clean** of any hardcoded credentials. It only has the code to retrieve credentials from AWS Secrets Manager!



```
import boto3
import json
from botocore.exceptions import ClientError

def get_secret():

    secret_name = "aws-access-key"
    region_name = "us-east-2" # Replace with your AWS region if different

    # Create a Secrets Manager client
    session = boto3.session.Session()
    client = session.client(
        service_name='secretsmanager',
        region_name=region_name
    )
```

-
-
-