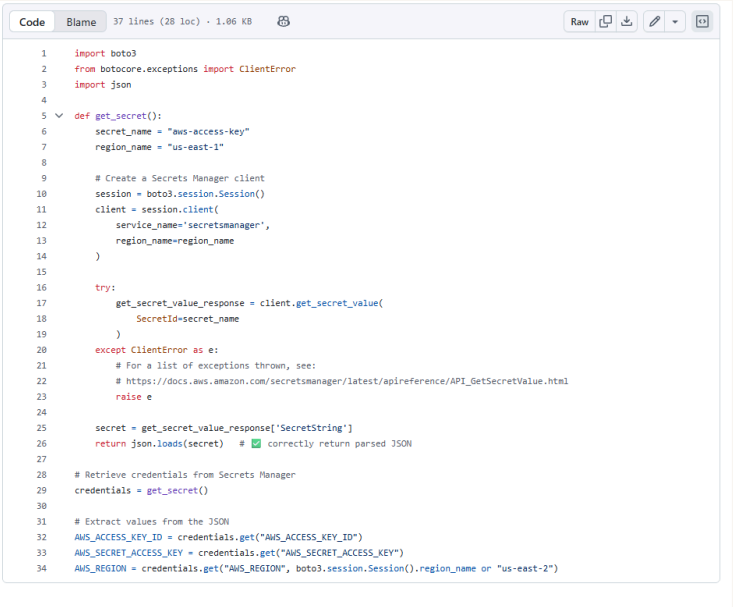
nextwork.org

Secure Secrets with Secrets Manager

Louis Moyo



L
Louis Moyo
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to first hardcode AWS credentials into a simple web app and then replace that insecure approach by securely storing and retrieving credentials with AWS Secrets Manager. I'm doing this project to learn why hardcoding credentials is risky, how Secrets Manager protects sensitive information, and how to update application code to safely use secrets without exposing them publicly.

Tools and concepts

Services I used were Git, GitHub, Notepad, PowerShell, and Python virtual environments. Key concepts I learnt include resolving merge conflicts, using git rebase --continue, committing securely without hardcoded credentials, and deleting a local repository once work is complete.

Project reflection

This project took me approximately a few hours. The most challenging part was resolving merge conflicts and pushing changes correctly. It was most rewarding to confirm that my repository was updated securely and no sensitive data was exposed.

placeholder

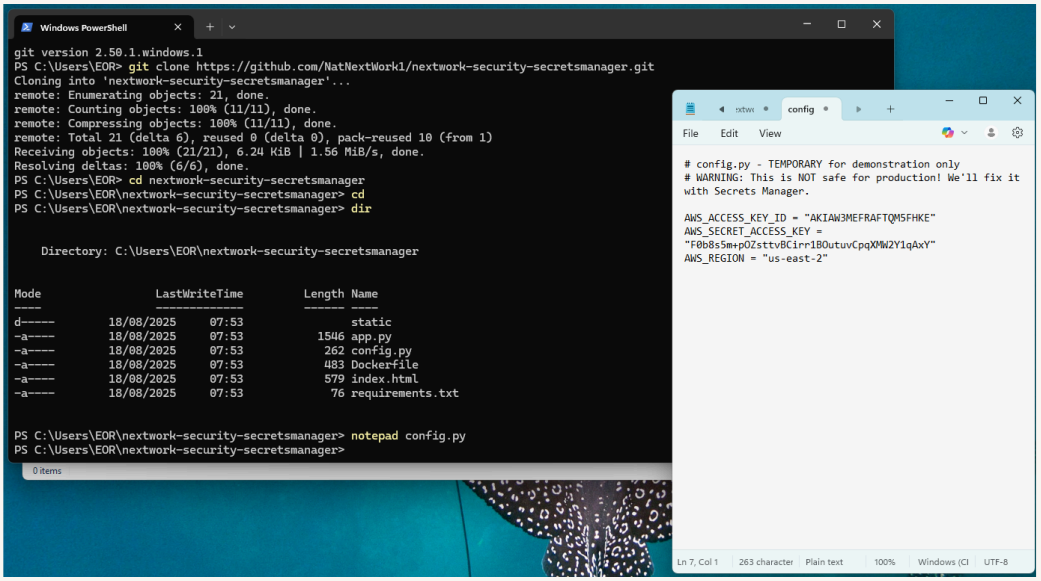
I did this project today to practise Git and GitHub workflows in a secure way. This project met my goals because I now feel more confident handling conflicts, pushing clean commits, and keeping my repository secure.

Louis Moyo
NextWork Student

Hardcoding credentials

In this project, a sample web app is exposing AWS credentials directly in its config.py file. It is unsafe to hardcode credentials because anyone who gains access to the code (for example, through a public GitHub repository) could see and misuse those credentials to access your AWS account, steal data, delete resources, or incur costs on your behalf. Storing secrets in code creates a major security vulnerability and violates best practices for securing sensitive information.

I've set up the initial configuration with an AWS Access Key ID, AWS Secret Access Key, and an AWS Region inside the config.py file. These credentials are just examples because the tutorial is first showing the insecure way of hardcoding secrets directly into the app. This helps demonstrate why storing credentials in code is a bad practice, and later we'll replace them with a secure approach using AWS Secrets Manager.

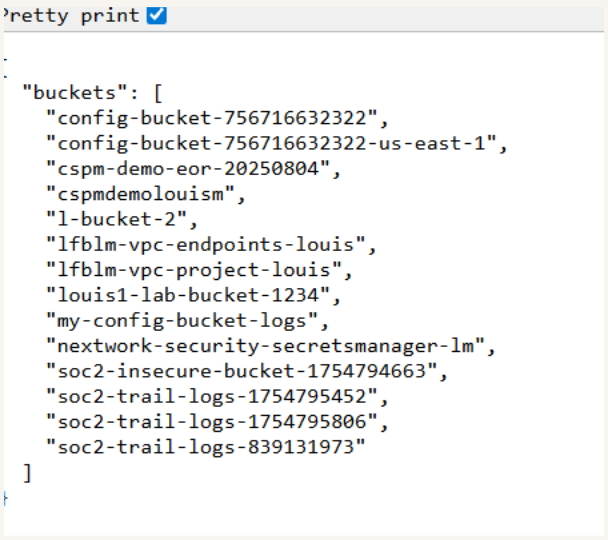

Windows PowerShell
git version 2.50.1.windows.1
PS C:\Users\EOR> git clone https://github.com/NatNextWork1/nextwork-security-secretsmanager.git
Cloning into 'nextwork-security-secretsmanager'...
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 21 (delta 6), reused 0 (delta 0), pack-reused 10 (from 1)
Receiving objects: 100% (21/21), 6.24 KiB | 1.56 MiB/s, done.
Resolving deltas: 100% (6/6), done.
PS C:\Users\EOR> cd nextwork-security-secretsmanager
PS C:\Users\EOR\nextwork-security-secretsmanager> cd
PS C:\Users\EOR\nextwork-security-secretsmanager> dir
Directory: C:\Users\EOR\nextwork-security-secretsmanager
Mode LastWriteTime Length Name
d----- 18/08/2025 07:53 static
-a---- 18/08/2025 07:53 1546 app.py
-a---- 18/08/2025 07:53 262 config.py
-a---- 18/08/2025 07:53 483 Dockerfile
-a---- 18/08/2025 07:53 579 index.html
-a---- 18/08/2025 07:53 76 requirements.txt
PS C:\Users\EOR\nextwork-security-secretsmanager> notepad config.py
PS C:\Users\EOR\nextwork-security-secretsmanager>
0 items
config
File Edit View
# config.py - TEMPORARY for demonstration only
# WARNING: This is NOT safe for production! We'll fix it with Secrets Manager.
AWS_ACCESS_KEY_ID = "AKIAW3MEFRAFTQM5FHKE"
AWS_SECRET_ACCESS_KEY = "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"
AWS_REGION = "us-east-2"
Ln 7, Col 1
263 character
Plain text
100%
Windows (CI
UTF-8

Using my own AWS credentials

As an extension for this project, I also decided to set up my virtual environment. To do this, I installed the packages listed in the requirements.txt file. These included: boto3 - lets Python applications talk to AWS services (needed to list S3 buckets and interact with AWS resources). fastapi - a modern web framework for building APIs quickly and easily. uvicorn - a lightweight ASGI server that runs the FastAPI app. These packages are essential because boto3 enables AWS interaction, FastAPI powers the web app's API layer, and Uvicorn serves the application so it can run locally in the browser.

When I first ran the app, I ran into an error because the AWS Access Key ID in my config.py file was just a placeholder and not a real credential. The app tried to call S3 but AWS rejected the request, returning {"error":"An error occurred (InvalidAccessKeyId)"}. This means the app was working correctly, but it couldn't access my AWS account since the credentials provided were invalid.

To resolve the 'InvalidAccessKeyId' error, I updated config.py with my real AWS credentials from IAM. I replaced the placeholder AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY with my actual keys, and set AWS_REGION to match the region of my S3 bucket. Now the file securely holds my access key, secret key, and correct region for the web app to connect.


Pretty print
{
"buckets": [
"config-bucket-756716632322",
"config-bucket-756716632322-us-east-1",
"cspm-demo-eor-20250804",
"cspmdemolouism",
"l-bucket-2",
"lfblm-vpc-endpoints-louis",
"lfblm-vpc-project-louis",
"louis1-lab-bucket-1234",
"my-config-bucket-logs",
"nextwork-security-secretsmanager-lm",
"soc2-insecure-bucket-1754794663",
"soc2-trail-logs-1754795452",
"soc2-trail-logs-1754795806",
"soc2-trail-logs-839131973"
]
}

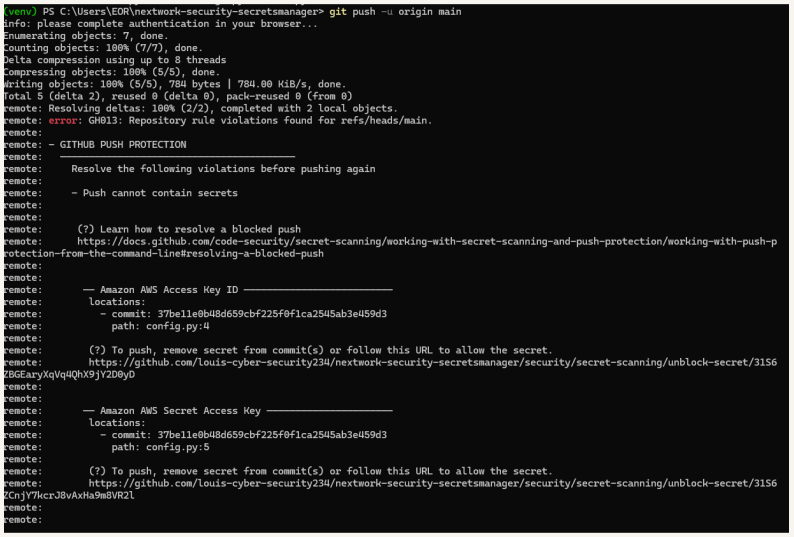
L
Louis Moyo
NextWork Student

Pushing Insecure Code to GitHub

Once I updated the web app code with credentials, I forked the repository because I wanted my own online copy in my GitHub account that I could edit and share. A fork is different from a clone since forking creates a public copy stored on GitHub, while cloning only makes a private local copy on my computer.

To connect my local repository to the forked repository, I set the remote URL with git remote set-url origin <my-forked-repo-url> and confirmed it using git remote -v. Then I used git add . to stage my changes and git commit -m "Updated config.py with hardcoded credentials" to save a snapshot of those changes. Finally, git push -u origin main uploads my commits to the main branch of my forked repository on GitHub, and also sets it as the default upstream branch for future pushes.

GitHub blocked my push because I accidentally tried to upload sensitive data (hardcoded credentials) into the repository. This is a good security feature because it prevents secrets, passwords, or API keys from being exposed publicly where attackers could misuse them. It forces me to use safer practices, like storing credentials in environment variables or using secret managers, instead of committing them directly into the code.

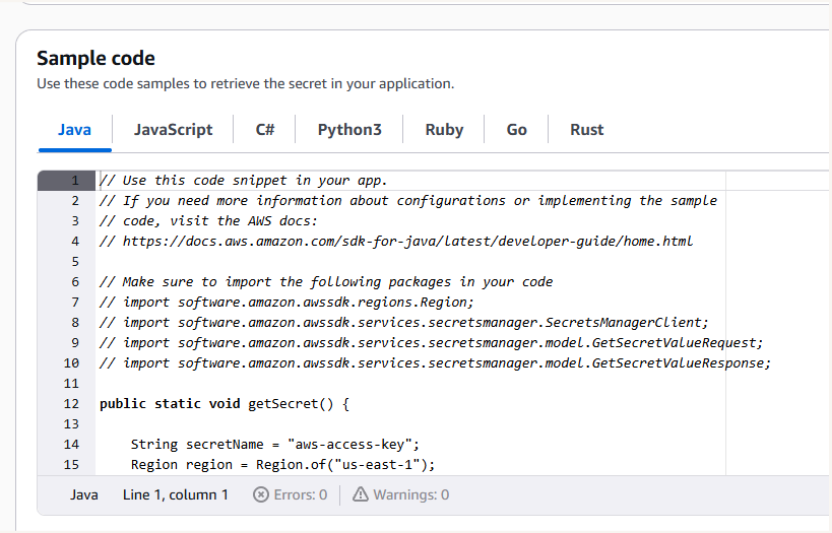

(venv) PS C:\Users\EOR\nextwork-security-secretsmanager> git push -u origin main
info: please complete authentication in your browser...
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 784 bytes | 784.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote:
remote: - GITHUB PUSH PROTECTION
remote: ——————————————————————————
remote: Resolve the following violations before pushing again
remote:
remote: - Push cannot contain secrets
remote:
remote:
remote: (?) Learn how to resolve a blocked push
remote: https://docs.github.com/code-security/secret-scanning/working-with-secret-scanning-and-push-protection/working-with-push-protection-from-the-command-line#resolving-a-blocked-push
remote:
remote:
remote: — Amazon AWS Access Key ID ————————————————
remote: locations:
remote: - commit: 37be11e0b48d659cbf225f0f1ca2545ab3e459d3
remote: path: config.py:4
remote:
remote: (?) To push, remove secret from commit(s) or follow this URL to allow the secret.
remote: https://github.com/louis-cyber-security234/nextwork-security-secretsmanager/security/secret-scanning/unblock-secret/31S6ZBGEaryXqVq4QhX9jY2D0yD
remote:
remote:
remote: — Amazon AWS Secret Access Key ————————————
remote: locations:
remote: - commit: 37be11e0b48d659cbf225f0f1ca2545ab3e459d3
remote: path: config.py:5
remote:
remote: (?) To push, remove secret from commit(s) or follow this URL to allow the secret.
remote: https://github.com/louis-cyber-security234/nextwork-security-secretsmanager/security/secret-scanning/unblock-secret/31S6ZCnjv7kcrJ8vAxHa9m8VR2l
remote:
remote:

Secrets Manager

Secrets Manager is an AWS service that securely stores and manages sensitive information like API keys, database passwords, and access tokens. I'm using it to store my AWS Access Key ID and Secret Access Key instead of hardcoding them in my code. Other common use cases include managing database credentials, rotating secrets automatically, and providing applications with secure access to sensitive data at runtime.

Another feature in Secrets Manager is secret rotation, which means automatically updating and replacing secrets on a schedule. It's useful in situations where credentials like database passwords or API keys are high-risk - if they're compromised, the attacker only has a short window before the secret changes. This reduces the impact of leaks and strengthens security for critical systems.

Secrets Manager provides sample code in multiple languages like Python, Java, JavaScript, and Go to show developers how to retrieve secrets securely. This is helpful because instead of hardcoding sensitive values like API keys or database passwords into the application, developers can fetch them at runtime directly from Secrets Manager. The AWS SDKs handle authentication and decryption automatically, so the app only gets access if it's authorised. This reduces the risk of accidental leaks (e.g., pushing code to GitHub), enforces encryption, and supports compliance requirements.


Sample code
Use these code samples to retrieve the secret in your application.
Java
JavaScript
C#
Python3
Ruby
Go
Rust
1 // Use this code snippet in your app.
2 // If you need more information about configurations or implementing the sample
3 // code, visit the AWS docs:
4 // https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/home.html
5
6 // Make sure to import the following packages in your code
7 // import software.amazon.awssdk.regions.Region;
8 // import software.amazon.awssdk.services.secretsmanager.SecretsManagerClient;
9 // import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
10 // import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;
11
12 public static void getSecret() {
13
14 String secretName = "aws-access-key";
15 Region region = Region.of("us-east-1");
Java
Line 1, column 1
Errors: 0
Warnings: 0

Updating the web app code

I updated the config.py file to retrieve AWS credentials securely from Secrets Manager instead of hardcoding them. The get_secret() function will create a Secrets Manager client using boto3, fetch the secret value by its name, and return the decrypted credentials. This ensures that the AWS Access Key ID and Secret Access Key are encrypted at rest and only retrieved securely at runtime, reducing the risk of accidental exposure in code or version control.

I also added code to config.py to extract the individual values from the secret returned by AWS Secrets Manager, such as AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_REGION. This is important because instead of hardcoding credentials directly into the file, the application now dynamically retrieves them from a secure location at runtime. That way, the credentials remain hidden from source code, reducing the risk of accidental exposure in GitHub or logs, and ensuring best practices for secure, maintainable cloud applications.

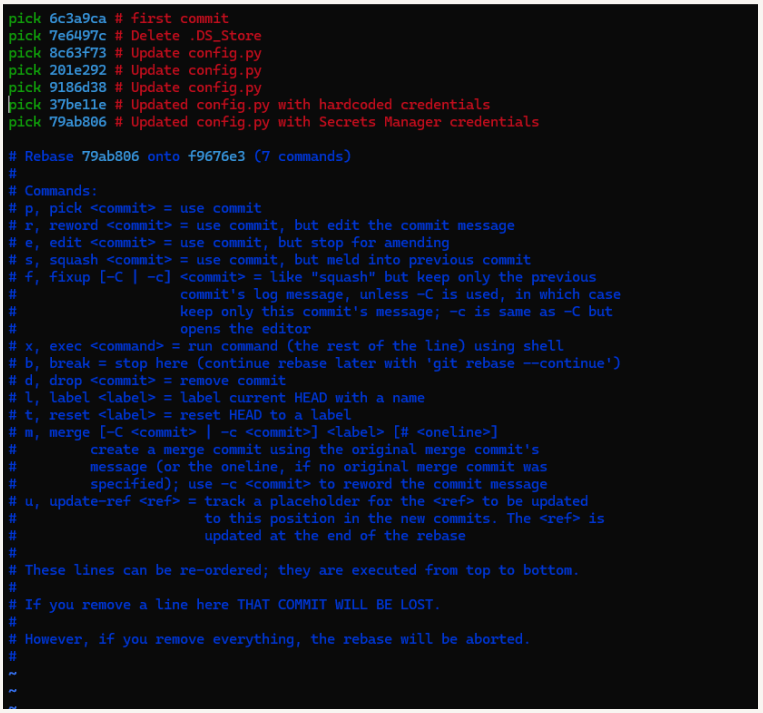
```
import boto3
from botocore.exceptions import ClientError


def get_secret():

    secret_name = "aws-access-key"
    region_name = "us-east-1"

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

    try:
        get_secret_value_response = client.get_secret_value(
            SecretId=secret_name
        )
    except ClientError as e:
        # For a list of exceptions thrown, see
        # https://docs.aws.amazon.com/secretsmanager/latest/apireference/API_GetSecretValue.html
        raise e

    secret = get_secret_value_response['SecretString']

    # Your code goes here.
```

Rebasing the repository

Git rebasing is the process of rewriting or reorganising commit history so that your branch history is linear and clean. I used it to drop the commit where I had added hardcoded credentials to config.py. This was necessary because even though I updated the file later, the old commit still contained the sensitive keys in the repository's history. By rebasing and dropping that commit, I removed the exposed credentials completely, ensuring they cannot be accessed by anyone reviewing the Git history.

A merge conflict occurred during rebasing because both commits modified the config.py file in the same section - one version contained hard-coded AWS credentials while the other introduced the new Secrets Manager integration. Git couldn't automatically decide which code to keep, so it flagged the conflict. I resolved the merge conflict by opening the file, removing the conflict markers, and keeping only the new Secrets Manager implementation. This ensured the final version had no exposed credentials and securely retrieved secrets.

Once the merge conflict was resolved, I verified that my hardcoded credentials were out of sight in the repository by opening the config.py file, checking that the lines containing AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY were removed, and ensuring only the Secrets Manager integration code remained.


pick 6c3a9ca # first commit
pick 7e6497c # Delete .DS_Store
pick 8c63f73 # Update config.py
pick 201e292 # Update config.py
pick 9186d38 # Update config.py
pick 37be11e # Updated config.py with hardcoded credentials
pick 79ab806 # Updated config.py with Secrets Manager credentials
Rebase 79ab806 onto f9676e3 (7 commands)
Commands:
p, pick <commit> = use commit
r, reword <commit> = use commit, but edit the commit message
e, edit <commit> = use commit, but stop for amending
s, squash <commit> = use commit, but meld into previous commit
f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
commit's log message, unless -C is used, in which case
keep only this commit's message; -c is same as -C but
opens the editor
x, exec <command> = run command (the rest of the line) using shell
b, break = stop here (continue rebase later with 'git rebase --continue')
d, drop <commit> = remove commit
l, label <label> = label current HEAD with a name
t, reset <label> = reset HEAD to a label
m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
create a merge commit using the original merge commit's
message (or the oneline, if no original merge commit was
specified); use -c <commit> to reword the commit message
u, update-ref <ref> = track a placeholder for the <ref> to be updated
to this position in the new commits. The <ref> is
updated at the end of the rebase
#
These lines can be re-ordered; they are executed from top to bottom.
#
If you remove a line here THAT COMMIT WILL BE LOST.
#
However, if you remove everything, the rebase will be aborted.
#
~
~

The place to learn & showcase your skills

Check out nextwork.org for more projects