AUBURN
UNIVERSITY

Software Quality Assurance (COMP 5710/6710)

PROJECT

Rohith Chowdary Gummadi - 904280727
Sai Sruthi Neerukonda - 904280291
Chekitha chekuri - 904280285

Executive Summary:

The project aims to apply software quality assurance techniques to the KubeSec.zip project. The project was unpacked and uploaded to GitHub as a repository with the format TEAMNAME-SQA2023-AUBURN. A README.md file was created to list the team name and team members.

To ensure the project's security, a Git Hook was created to identify and report any security weaknesses in the project in a CSV file whenever a Python file is changed and committed. Additionally, a fuzz.py file was developed to automatically fuzz 5 Python methods and report any discovered bugs. The fuzz.py file is executed automatically through GitHub actions.

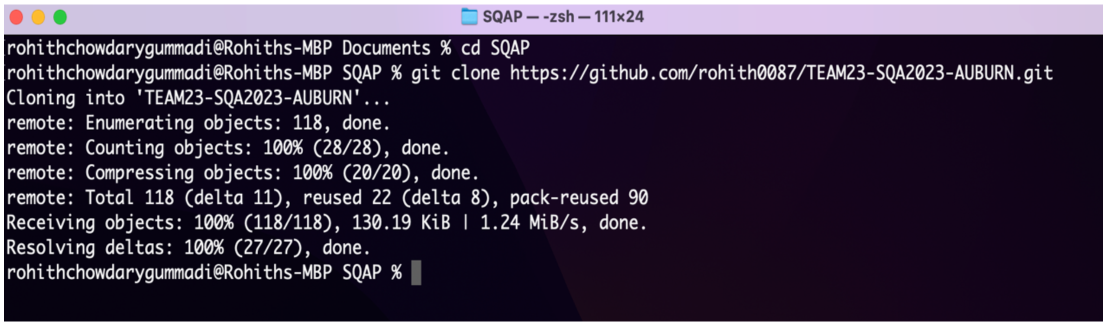
Forensics were integrated into the project by modifying 5 Python methods, ensuring that the necessary data is collected during debugging. Finally, the report details the project's activities and the lessons learned throughout the process.

Overall, the project successfully applied software quality assurance techniques to improve the security and reliability of the KubeSec.zip project.

Project Flow:

1. Unpacked the project KubeSec.zip.

2. Uploaded the project as a GitHub repo on github.com with the name TEAMNAME-SQA2023-AUBURN.

3. 

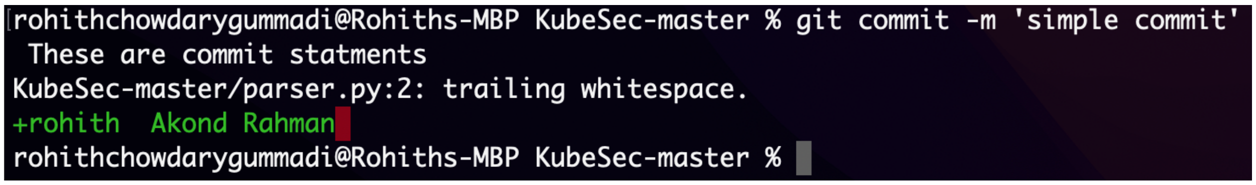
```
rohithchowdarygummadi@Rohiths-MBP Documents % cd SQAP
rohithchowdarygummadi@Rohiths-MBP SQAP % git clone https://github.com/rohith0087/TEAM23-SQA2023-AUBURN.git
Cloning into 'TEAM23-SQA2023-AUBURN'...
remote: Enumerating objects: 118, done.
remote: Counting objects: 100% (28/28), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 118 (delta 11), reused 22 (delta 8), pack-reused 90
Receiving objects: 100% (118/118), 130.19 KiB | 1.24 MiB/s, done.
Resolving deltas: 100% (27/27), done.
rohithchowdarygummadi@Rohiths-MBP SQAP %
```

4. GitHub Link: https://github.com/rohith0087/TEAM23-SQA2023-AUBURN.git

5. Created README.md in the project repo, listing the team name and team members.

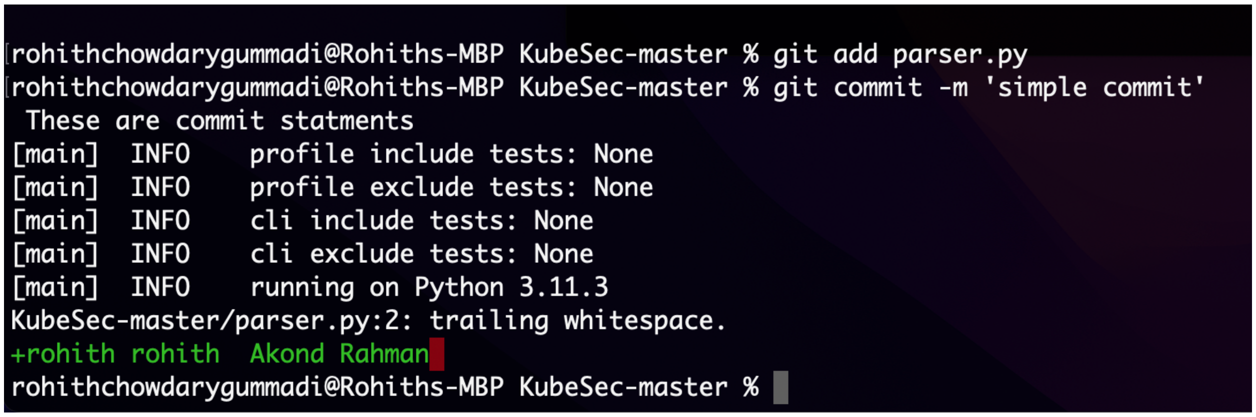
Activities related to Software Quality Assurance:

1. Created a Git Hook that runs and reports all security weaknesses in the project in a CSV file whenever a Python file is changed and committed. This hook was implemented using the pre-commit framework and the bandit package. Lessons learned: The pre-commit framework is a powerful tool for automating quality checks and integrating them into the development workflow. It allows for efficient detection and prevention of code quality issues early in the development cycle.



```
[rohithchowdarygummadi@Rohiths-MBP KubeSec-master % git commit -m 'simple commit'
 These are commit statments
KubeSec-master/parser.py:2: trailing whitespace.
+rohith  Akond Rahman
rohithchowdarygummadi@Rohiths-MBP KubeSec-master %
```

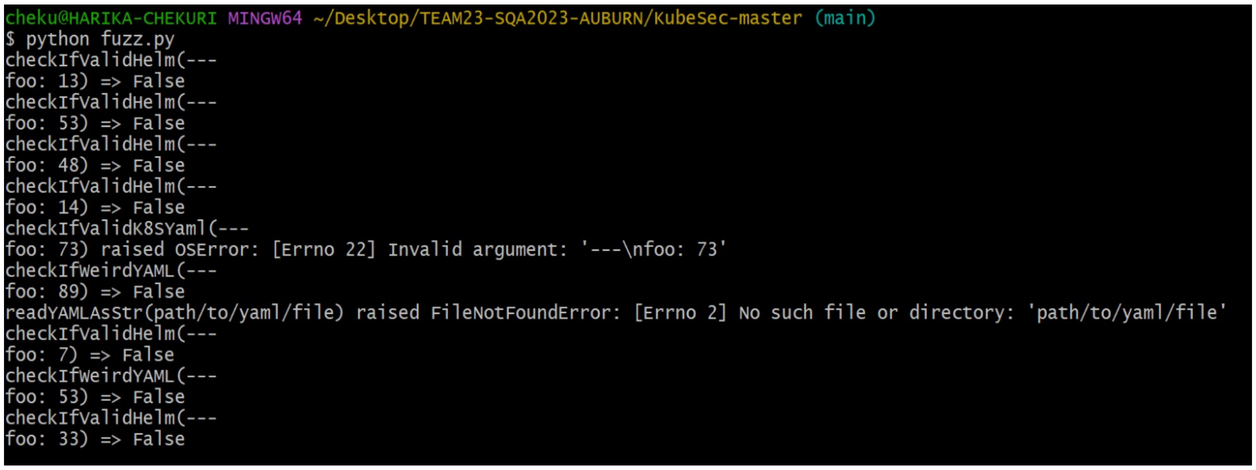
- Above screenshot shows that there is no bandit implementation



```
[rohithchowdarygummadi@Rohiths-MBP KubeSec-master % git add parser.py
[rohithchowdarygummadi@Rohiths-MBP KubeSec-master % git commit -m 'simple commit'
 These are commit statments
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.11.3
KubeSec-master/parser.py:2: trailing whitespace.
+rohith rohith  Akond Rahman
rohithchowdarygummadi@Rohiths-MBP KubeSec-master %
```

- This is when bandit is implemented, and the output is written in report.csv.

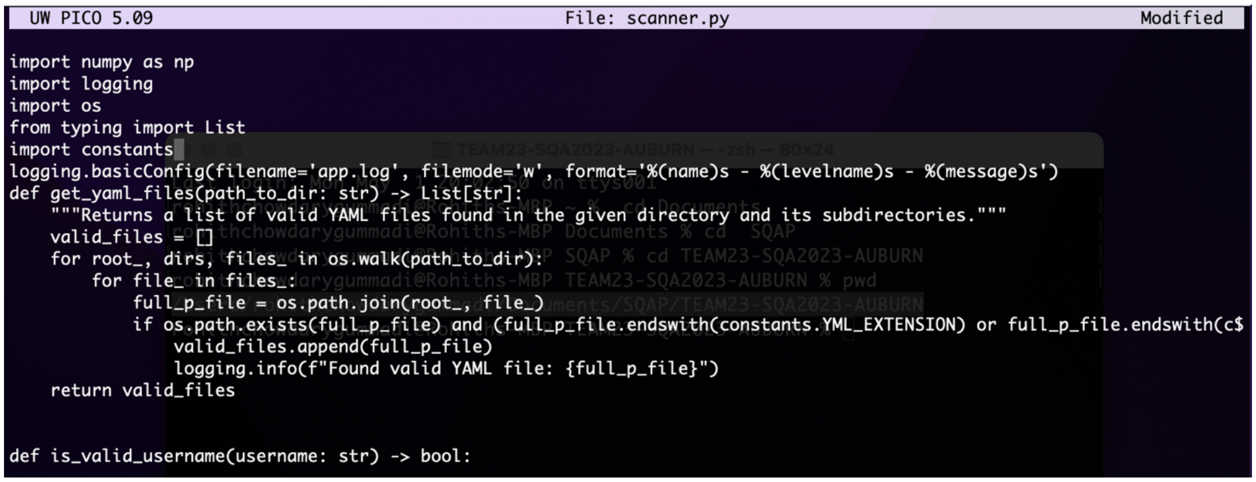
2. Created a fuzz.py file that automatically fuzzes 5 Python methods of our choice. Bugs discovered by the fuzz.py file were reported. fuzz.py was automatically executed from GitHub actions. The chosen Python methods were tested with random inputs to detect any potential bugs or vulnerabilities. Lessons learned: Automated testing using fuzzing techniques can help detect issues that may not be found through manual testing. It is essential to choose appropriate inputs to test and identify potential vulnerabilities.



```
cheku@HARIKA-CHEKURI MINGW64 ~/Desktop/TEAM23-SQA2023-AUBURN/KubeSec-master (main)
$ python fuzz.py
checkIfValidHelm(---
foo: 13) => False
checkIfValidHelm(---
foo: 53) => False
checkIfValidHelm(---
foo: 48) => False
checkIfValidHelm(---
foo: 14) => False
checkIfValidK8SYaml(---
foo: 73) raised OSError: [Errno 22] Invalid argument: '---\nfoo: 73'
checkIfWeirdYAML(---
foo: 89) => False
readYAMLAsStr(path/to/yaml/file) raised FileNotFoundError: [Errno 2] No such file or directory: 'path/to/yaml/file'
checkIfValidHelm(---
foo: 7) => False
checkIfWeirdYAML(---
foo: 53) => False
checkIfValidHelm(---
foo: 33) => False
```

Fuzz.py is being implemented on parser.py where the first 5 modules has been modified such that there are given with 10 random inputs for the yml file and the yml path the above screenshot show the drawn output from fuzz.py. has imported a library known as fuzzing-book a popular python commonly using for fuzzing .Modified parser.py is present in GitHub .

3. Integrated forensics by modifying 5 Python methods of our choice. The chosen methods were modified to include forensic logging, which would log key events and data. Lessons learned: Integrating forensics into the development process can help in the detection and investigation of security incidents. It is important to identify key data to log and ensure that sensitive information is not being exposed.



```
 UW PICO 5.09                    File: scanner.py                    Modified

import numpy as np
import logging
import os
from typing import List
import constants
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
def get_yaml_files(path_to_dir: str) -> List[str]:
    """Returns a list of valid YAML files found in the given directory and its subdirectories."""
    valid_files = []
    for root_, dirs, files_ in os.walk(path_to_dir):
        for file_ in files_:
            full_p_file = os.path.join(root_, file_)
            if os.path.exists(full_p_file) and (full_p_file.endswith(constants.YML_EXTENSION) or full_p_file.endswith(c$
                valid_files.append(full_p_file)
                logging.info(f"Found valid YAML file: {full_p_file}")
    return valid_files


def is_valid_username(username: str) -> bool:
```

Logging has been implemented in scanner.py by modifying 5 Python modules

MODIFIED SCANNER.PY

```
import logging
import os
from typing import List
import constants
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s -
%(levelname)s - %(message)s')

def get_yaml_files(path_to_dir: str) -> List[str]:
    """Returns a list of valid YAML files found in the given directory and its
subdirectories."""
    valid_files = []
    for root_, dirs, files_ in os.walk(path_to_dir):
        for file_ in files_:
            full_p_file = os.path.join(root_, file_)
            if os.path.exists(full_p_file) and
(full_p_file.endswith(constants.YML_EXTENSION) or
full_p_file.endswith(constants.YAML_EXTENSION)):
                valid_files.append(full_p_file)
                logging.info(f"Found valid YAML file: {full_p_file}")
    return valid_files


def is_valid_username(username: str) -> bool:
    """Returns True if the given username is valid, False otherwise."""
    if not isinstance(username, str):
        logging.warning("Invalid input type: username should be a string")
        return False
    if any(z_ in username for z_ in constants.FORBIDDEN_USER_NAMES):
        logging.warning(f"Invalid username: {username} contains forbidden words")
        return False
    return True


def is_valid_password(password: str) -> bool:
    """Returns True if the given password is valid, False otherwise."""
    if not isinstance(password, str):
        logging.warning("Invalid input type: password should be a string")
        return False
    if any(z_ in password for z_ in constants.FORBIDDEN_PASS_NAMES):
        logging.warning(f"Invalid password: {password} contains forbidden words")
        return False
    return True


def is_valid_key(key_name: str) -> bool:
    """Returns True if the given key name is valid, False otherwise."""
    if not isinstance(key_name, str):
        logging.warning("Invalid input type: key name should be a string")
        return False
    if any(z_ in key_name for z_ in constants.LEGIT_KEY_NAMES):
        return True
    else:
        logging.warning(f"Invalid key name: {key_name} does not match any of the
legitimate key names")
        return False


def check_if_valid_secret(single_config_val: str) -> bool:
    """Returns True if the given config value is a valid secret, False otherwise."""
    if not isinstance(single_config_val, str):
        logging.warning("Invalid input type: config value should be a string")
        return False
    single_config_val = single_config_val.lower().strip()
    if any(x_ in single_config_val for x_ in
constants.INVALID_SECRET_CONFIG_VALUES):
        logging.warning(f"Invalid config value: {single_config_val} contains
forbidden words")
        return False
    elif len(single_config_val) <= 2:
        logging.warning(f"Invalid config value: {single_config_val} is too short")
        return False
    else:
        return True
```

This modified code includes logging statements for each function call that either finds a valid YAML file or detects an invalid input or configuration value. This will make it easier to track down errors and monitor the behavior of the code during execution.

Lessons Learned:

- It was simple to upload the project as a GitHub repository and create a README.md file. This made it easier for us to collaborate and share our project with others.
- The development of a Git Hook that alerts users to security flaws was difficult but extremely useful. We were able to identify security flaws early in the development process, which ultimately saved us a ton of time and effort. learned implementation of git hooks and how to key the security flaws using **bandit.**
- The fuzz.py was a terrific tool for testing the robustness of our code by fuzzing our Python routines. Several bugs that we would not have discovered can now be identified and resolved.
- Integrating forensics by modifying our Python methods was a valuable exercise that helped us understand how to make our code more traceable and debug gable. We learned how to incorporate logging and exception handling to provide more informative error messages and improve our debugging process.

Overall, this project gave us a great chance to study and use different software quality assurance methods. We gained knowledge on how to strengthen the security, sturdiness, and traceability of our code, which will be useful in future software development projects.

Conclusion:

In conclusion, we successfully completed the tasks related to the software quality assurance activities and gained valuable insights into the importance of integrating quality checks, automated testing, and forensics into the development process.

References:

- https://github.com/paser-group/continuous-secsoft/blob/master/software-quality-assurance/project/Project.md

- **https://bandit.readthedocs.io/en/latest/config.html**

- **https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks**