Code Analysis Report

Analyzed Code:

```
#
# Copyright (C) 2013-2018 Freedom of the Press Foundation & al
# Copyright (C) 2018 Loic Dachary
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see .
#

import argparse
import logging
import os
import shutil
import subprocess
import sys
from typing import Iterator, List

sdlog = logging.getLogger(__name__)

DIR = os.path.dirname(os.path.realpath(__file__))
VENV_DIR = os.path.join(DIR, ".venv3")

# Space-separated list of apt dependencies
APT_DEPENDENCIES_STR = "python3-virtualenv \
python3-yaml \
python3-pip \
```

```
virtualenv \
libffi-dev \
libssl-dev \
libpython3-dev \
sq-keyring-linter"

def setup_logger(verbose: bool = False) -> None:
"""Configure logging handler"""
# Set default level on parent
sdlog.setLevel(logging.DEBUG)
level = logging.DEBUG if verbose else logging.INFO

stdout = logging.StreamHandler(sys.stdout)
stdout.setFormatter(logging.Formatter("%(levelname)s: %(message)s"))
stdout.setLevel(level)
sdlog.addHandler(stdout)

def run_command(command: List[str]) -> Iterator[bytes]:
"""
Wrapper function to display stdout for running command,
similar to how shelling out in a Bash script displays rolling output.

Yields a list of the stdout from the `command`, and raises a
CalledProcessError if `command` returns non-zero.
"""
popen = subprocess.Popen(command, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT)
if popen.stdout is None:
raise OSError("Could not run command: None stdout")
yield from iter(popen.stdout.readline, b"")
popen.stdout.close()
return_code = popen.wait()
if return_code:
raise subprocess.CalledProcessError(return_code, command)

def is_tails() -> bool:
with open("/etc/os-release") as f:
return "TAILS_PRODUCT_NAME" in f.read()

def clean_up_old_tails_venv(virtualenv_dir: str = VENV_DIR) -> None:
"""
```

```
When upgrading major Tails versions, we need to rebuild the virtualenv
against the correct Python version. We can detect if the Tails
version matches the correct Python version - if not, delete the
venv, so it'll get recreated.
"""
if is_tails():
with open("/etc/os-release") as f:
os_release = f.readlines()
for line in os_release:
if line.startswith("TAILS_VERSION_ID="):
version = line.split("=")[1].strip().strip('"')
if version.startswith("5."):
# Tails 5 is based on Python 3.9
python_lib_path = os.path.join(virtualenv_dir, "lib/python3.7")
if os.path.exists(python_lib_path):
sdlog.info("Tails 4 virtualenv detected. Removing it.")
shutil.rmtree(virtualenv_dir)
sdlog.info("Tails 4 virtualenv deleted.")
break

def checkenv(args: argparse.Namespace) -> None:
clean_up_old_tails_venv(VENV_DIR)
if not os.path.exists(os.path.join(VENV_DIR, "bin/activate")):
sdlog.error('Please run "securedrop-admin setup".')
sys.exit(1)

def is_missing_dependency() -> bool:
"""
Check if there are any missing apt dependencies.
This applies to existing Tails systems where `securedrop-setup` may not have
been
run recently.
"""
# apt-cache -q0 policy $dependency1 $dependency2 $dependency3 | grep
"Installed: (none)"
apt_query = f"apt-cache -q0 policy {APT_DEPENDENCIES_STR}".split(" ")
grep_command = ["grep", "Installed: (none)"]

try:
```

```
sdlog.info("Checking apt dependencies are installed")
apt_process = subprocess.Popen(apt_query, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
grep_process = subprocess.Popen(
grep_command, stdin=apt_process.stdout, stdout=subprocess.PIPE,
stderr=subprocess.PIPE
)
# Wait for the process to complete before checking the returncode
grep_process.communicate()
returncode = grep_process.returncode
# If the above command returns 0, then one or more packages are not
installed.
return returncode == 0
except subprocess.CalledProcessError as e:
sdlog.error("Error checking apt dependencies")
sdlog.debug(e.output)
raise

def maybe_torify() -> List[str]:
if is_tails():
return ["torify"]
else:
return []

def install_apt_dependencies(args: argparse.Namespace) -> None:
"""
Install apt dependencies in Tails. In order to install Ansible in
a virtualenv, first there are a number of Python prerequisites.
"""
sdlog.info("Installing SecureDrop Admin dependencies")
sdlog.info(
"You'll be prompted for the temporary Tails admin password,"
" which was set on Tails login screen"
)
apt_command = [
"sudo",
"su",
```

```
"-c",
f"apt-get update && \
apt-get -q -o=Dpkg::Use-Pty=0 install -y {APT_DEPENDENCIES_STR}",
]
try:
# Print command results in real-time, to keep Admin apprised
# of progress during long-running command.
for output_line in run_command(apt_command):
print(output_line.decode("utf-8").rstrip())
except subprocess.CalledProcessError:
# Tails supports apt persistence, which was used by SecureDrop
# under Tails 2.x. If updates are being applied, don't try to pile
# on with more apt requests.
sdlog.error(
"Failed to install apt dependencies. Check network" " connection and try again."
)
raise

def envsetup(args: argparse.Namespace, virtualenv_dir: str = VENV_DIR) -> None:
"""Installs Admin tooling required for managing SecureDrop. Specifically:
* updates apt-cache
* installs apt packages for Python virtualenv
* creates virtualenv
* installs pip packages inside virtualenv
The virtualenv is created within the Persistence volume in Tails, so that
Ansible is available to the Admin on subsequent boots without requiring
installation of packages again.
"""
# clean up old Tails venv on major upgrades
clean_up_old_tails_venv(virtualenv_dir)
# Check apt dependencies and ensure all are present.
if is_missing_dependency():
install_apt_dependencies(args)
# virtualenv doesnt exist? Install dependencies and create
```

```
if not os.path.exists(virtualenv_dir):

# Technically you can create a virtualenv from within python
# but pip can only be run over Tor on Tails, and debugging that
# along with instaling a third-party dependency is not worth
# the effort here.
sdlog.info("Setting up virtualenv")
try:
sdlog.debug(
subprocess.check_output(
maybe_torify() + ["virtualenv", "--python=python3", virtualenv_dir],
stderr=subprocess.STDOUT,
)
)
except subprocess.CalledProcessError as e:
sdlog.debug(e.output)
sdlog.error("Unable to create virtualenv. Check network settings" " and try
again.")
sdlog.debug("Cleaning up virtualenv")
if os.path.exists(virtualenv_dir):
shutil.rmtree(virtualenv_dir)
raise
else:
sdlog.info("Virtualenv already exists, not creating")

if args.t:
install_pip_dependencies(
args,
requirements_file="requirements-testinfra.txt",
desc="dependencies with verification support",
)
else:
install_pip_dependencies(args)

if os.path.exists(os.path.join(DIR, "setup.py")):
install_pip_self(args)

sdlog.info("Finished installing SecureDrop dependencies")


def install_pip_self(args: argparse.Namespace) -> None:
```

```
pip_install_cmd = [os.path.join(VENV_DIR, "bin", "pip3"), "install", "-e",
DIR]
try:
subprocess.check_output(maybe_torify() + pip_install_cmd,
stderr=subprocess.STDOUT)
except subprocess.CalledProcessError as e:
sdlog.debug(e.output)
sdlog.error("Unable to install self, run with -v for more information")
raise

def install_pip_dependencies(
args: argparse.Namespace,
requirements_file: str = "requirements.txt",
desc: str = "Python dependencies",
) -> None:
"""
Install Python dependencies via pip into virtualenv.
"""

# Ansible version 2.9.* cannot be directly upgraded and must be removed
# before attempting to install a later version - so let's check for it
# and uninstall it if we find it!

ansible_vercheck_cmd = [
os.path.join(VENV_DIR, "bin", "python3"),
"-c",
"from importlib.metadata import version as v; print(v('ansible'))",
]

ansible_uninstall_cmd = [
os.path.join(VENV_DIR, "bin", "pip3"),
"uninstall",
"-y",
"ansible",
]

ansible_ver = subprocess.run(
maybe_torify() + ansible_vercheck_cmd, text=True, capture_output=True
)
if ansible_ver.stdout.startswith("2.9"):
sdlog.info("Ansible is out-of-date, removing it.")
```

```
delete_result = subprocess.run(
maybe_torify() + ansible_uninstall_cmd, capture_output=True, text=True
)
if delete_result.returncode != 0:
sdlog.error(
"Failed to remove old ansible version:\n"
f" return num: {delete_result.returncode}\n"
f" error text: {delete_result.stderr}\n"
"Attempting to continue."
)

pip_install_cmd = [
os.path.join(VENV_DIR, "bin", "pip3"),
"install",
"--no-deps",
"-r",
os.path.join(DIR, requirements_file),
"--require-hashes",
"-U",
"--upgrade-strategy",
"only-if-needed",
]

sdlog.info(f"Checking {desc} for securedrop-admin")
try:
pip_output = subprocess.check_output(
maybe_torify() + pip_install_cmd, stderr=subprocess.STDOUT
)
except subprocess.CalledProcessError as e:
sdlog.debug(e.output)
sdlog.error(f"Failed to install {desc}. Check network" " connection and try
again.")
raise

sdlog.debug(pip_output)
if "Successfully installed" in str(pip_output):
sdlog.info(f"{desc} for securedrop-admin upgraded")
else:
sdlog.info(f"{desc} for securedrop-admin are up-to-date")
```

```
def parse_argv(argv: List[str]) -> argparse.Namespace:
parser = argparse.ArgumentParser()
parser.add_argument(
"-v", action="store_true", default=False, help="Increase verbosity on output"
)
parser.add_argument(
"-t", action="store_true", default=False, help="Install additional test dependencies"
)
parser.set_defaults(func=envsetup)

subparsers = parser.add_subparsers()

envsetup_parser = subparsers.add_parser("envsetup", help="Set up the admin virtualenv.")
envsetup_parser.set_defaults(func=envsetup)

checkenv_parser = subparsers.add_parser(
"checkenv", help="Check that the admin virtualenv is properly set up."
)
checkenv_parser.set_defaults(func=checkenv)

return parser.parse_args(argv)


if __name__ == "__main__":
args = parse_argv(sys.argv[1:])
setup_logger(args.v)

try:
args.func(args)
except Exception:
sys.exit(1)
else:
sys.exit(0)
```

Static Code Metrics:

Metric	Value
Lines of Code (LOC)	293
Comment Density	0.14901960784313725

Cyclomatic Complexity	36
Maintainability Index	54.47987841395785
Halstead Metrics	
Unique Operators	6
Unique Operands	24
Total Operators	13
Total Operands	24
Volume	181.5549520375152
Effort	544.6648561125455
Depth of Inheritance Tree (DIT)	0
Coupling Between Object classes (CBO)	0
Lack of Cohesion of Methods (LCOM)	0
Fan-in	21
Fan-out	21
Number of Methods (NOM)	0

McCall's Intermediate Quality Metrics:

Metric	Value
Modifiability	0.32
Testability	0.41
Reliability	0.41
Understandability	0.45
Self-Descriptiveness	0.22
Reusability	1.0
Portability	0.58
Efficiency	0.44

GPT Analysis Report:

Code Analysis and Improvement Suggestions

1. **Comment Density and Documentation:**

- **Issue:** The comment density is relatively low (approximately 14.9%). While there are function docstrings, they are not extensive and could improve in detail.

- **Suggestion:** Enhance the clarity of what each function does, especially complex parts of the code. Each function can benefit from additional comments at critical points explaining why certain decisions were made, particularly in functions like `envsetup` and `install_apt_dependencies`.

2. **Function Complexity and Size:**

- **Issue:** The cyclomatic complexity is relatively high (36), indicating that several functions contain complex control flows that make them hard to follow and test.

- **Suggestion:** Refactor long functions into smaller, more cohesive ones. For instance, the `envsetup` function could be split into:

- A function to check dependencies and install them.

- A function to create a virtual environment.

- A function for pip self-installation.

This will significantly improve readability and testability.

3. **Error Handling:**

- **Issue:** The error handling is primarily done at a high level, which could obscure the source of the error.

- **Suggestion:** Implement more granular error handling within functions. Instead of just logging an error and raising a generic exception, catch specific exceptions where they occur, log relevant information, and handle each case appropriately. Additionally, ensure that important context is included in the logs.

4. **Use of Constants:**

- **Issue:** The list of APT dependencies (`APT_DEPENDENCIES_STR`) is hard-coded as a string with newlines, which makes maintenance harder if changes are needed.

- **Suggestion:** Store this information as a list of strings instead of a single string that gets split. For example:

```
python
APT_DEPENDENCIES = [
"python3-virtualenv",
"python3-yaml",
"python3-pip",
"virtualenv",
"libffi-dev",
"libssl-dev",
"libpython3-dev",
"sq-keyring-linter",
]
```

This would make it easier to add/remove dependencies and improves readability.

5. **Avoid Hard-Coding Paths:**

- **Issue:** The program uses hardcoded paths like `"/etc/os-release"` and `"/bin/pip3"`. This makes testing and portability difficult.

- **Suggestion:** Consider parameterizing paths or using environment variables for configurable items. This allows for better adaptability in different environments.

6. **Improve the Use of Subprocess:**

- **Issue:** The use of subprocesses is somewhat convoluted, particularly in `is_missing_dependency` and `install_apt_dependencies`.

- **Suggestion:** Use the `check_output` method with `text=True` for commands that you want to capture output from instead of managing multiple processes. This reduces complexity and potential errors.

- For example, using a single call:

```
python
output = subprocess.check_output(apt_query, text=True)
```

7. **Streamline Logger Usage:**

- **Issue:** Multiple places in code have repeated logging.

- **Suggestion:** Create utility functions that encapsulate recurring logging patterns. For example, a function could be created for error logging that handles the formatting and context automatically.

8. **Function Return Types:**

- **Issue:** Some functions, especially those that perform checks, don't specify return types properly.

- **Suggestion:** Update the function signatures to include explicit return types. This makes it easier for the reader to understand the expected behavior at a glance and aids any type-checking tools.

9. **Consider Using Libraries for Package Management:**

- **Issue:** Frequent calls to command line utilities like `apt-get` and `pip` introduce complexity.

- **Suggestion:** Use Python libraries like `apt` for package management and `pip` programmatically through its API. This would promote better integration and error handling.

10. **Testing Considerations:**

- **Issue:** The current structure makes it challenging to conduct isolated testing of smaller functionalities.

- **Suggestion:** Reorganize the code into a more modular structure (e.g., classes or utility modules) that can then be tested more easily in isolation. Consider adopting a testing framework like `pytest` for systematic testing.

Conclusion

The provided code is functioning but has several areas that could be improved in terms of complexity, readability, error handling, and maintainability. Focusing on these improvements can lead to a more robust, less error-prone application that is easier for future developers (or yourself) to work on. By adopting best practices and restructuring portions of the code as suggested, you will enhance the overall quality of the software.