169 lines - 3 Removals

```
"""
    juliet_access: Enumerate and access testcases f
rom the Juliet suite.
"""
import pathlib
import re
import typing

from . import filters, flow_variants


def enumerate_flow_variants() -> list[dict]:
    """Enumerate all flow variants."""
    return list(flow_variants.FLOW_VARIANT_DESCRIPT
IONS.values())


def enumerate_testcases(cwe: dict):
    """Enumerate all testcases associated with the
 given CWE."""
    cwe_path: pathlib.Path = cwe["Path"]
```

246 lines + 82 Additions

```
"""
    juliet_access: Enumerate and access testcases f
rom the Juliet suite.
"""
import pathlib
import re
import typing

from . import filters, flow_variants


def enumerate_flow_variants() -> list[dict]:
    """Enumerate all flow variants."""
    return list(flow_variants.FLOW_VARIANT_DESCRIPT
IONS.values())


def enumerate_testcases_malware(cwe: dict):

    cwe_path: pathlib.Path = cwe["Path"]

    # Collect object files
    testcase_file_paths = []
    for testcase_file_path in cwe_path.iterdir():
        testcase_file_paths += [testcase_file_path]

    testcase_files = {}
    count = 1
    for testcase_file_path in sorted(testcase_file_
paths):
        functional_variant = "MalVar-" + str(count)
        count = count + 1
        flow_variant = 1

        key = (functional_variant, flow_variant)
        if key not in testcase_files.keys():
            testcase_files[key] = []

        subfile_identifier = testcase_file_path.__s
tr__().rsplit("_", 1)[-1]
        testcase_files[key] += [
            {"SubfileIdentifier": subfile_identifie
r, "Path": testcase_file_path}
        ]

    testcases = [
        {
            "Weakness": cwe,
            "FunctionalVariant": functional_varian
t,
            "FlowVariant": flow_variant,
            "Files": v,
        }
        for (functional_variant, flow_variant), v i
n testcase_files.items()
    ]

    return testcases


def enumerate_testcases(cwe: dict):
    """Enumerate all testcases associated with the
 given CWE."""
    cwe_path: pathlib.Path = cwe["Path"]
```

```
    # Collect object files
    testcase_file_regex = r"^CWE[0-9]+_.+__(.+)_([0-9][0-9])(.+)?\..+$"
    testcase_file_paths = []
    if (cwe_path / "s01").exists():
        for split_path in cwe_path.iterdir():
            for testcase_file_path in split_path.iterdir():
                if re.match(testcase_file_regex, testcase_file_path.name) is not None:
                    testcase_file_paths += [testcase_file_path]
                # else:
                #    print(f'Skipping {testcase_file_path}')
    else:
        for testcase_file_path in cwe_path.iterdir():
            if re.match(testcase_file_regex, testcase_file_path.name) is not None:
                testcase_file_paths += [testcase_file_path]
            # else:
            #    print(f'Skipping {testcase_file_path}')

    # Parse filenames (see https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf)
    testcase_files = {}
    for testcase_file_path in sorted(testcase_file_paths):
        matches = re.match(testcase_file_regex, testcase_file_path.name)
        functional_variant = matches[1]
        flow_variant = int(matches[2])

        key = (functional_variant, flow_variant)
        if key not in testcase_files.keys():
            testcase_files[key] = []

        subfile_identifier = matches[3]
        testcase_files[key] += [
            {"SubfileIdentifier": subfile_identifier, "Path": testcase_file_path}
        ]

    testcases = [
        {
            "Weakness": cwe,
            "FunctionalVariant": functional_variant,
            "FlowVariant": flow_variant,
            "Files": v,
        }
        for (functional_variant, flow_variant), v in testcase_files.items()
    ]

    return testcases


def _get_juliet_base_dir() -> pathlib.Path:
```

```
    # Collect object files
    testcase_file_regex = r"^CWE[0-9]+_.+__(.+)_([0-9][0-9])(.+)?\..+$"
    testcase_file_paths = []
    if (cwe_path / "s01").exists():
        for split_path in cwe_path.iterdir():
            for testcase_file_path in split_path.iterdir():
                if re.match(testcase_file_regex, testcase_file_path.name) is not None:
                    testcase_file_paths += [testcase_file_path]
                # else:
                #    print(f'Skipping {testcase_file_path}')
    else:
        for testcase_file_path in cwe_path.iterdir():
            if re.match(testcase_file_regex, testcase_file_path.name) is not None:
                testcase_file_paths += [testcase_file_path]
            # else:
            #    print(f'Skipping {testcase_file_path}')

    # Parse filenames (see https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf)
    testcase_files = {}
    for testcase_file_path in sorted(testcase_file_paths):
        matches = re.match(testcase_file_regex, testcase_file_path.name)
        functional_variant = matches[1]
        flow_variant = int(matches[2])

        key = (functional_variant, flow_variant)
        if key not in testcase_files.keys():
            testcase_files[key] = []

        subfile_identifier = matches[3]
        testcase_files[key] += [
            {"SubfileIdentifier": subfile_identifier, "Path": testcase_file_path}
        ]

    testcases = [
        {
            "Weakness": cwe,
            "FunctionalVariant": functional_variant,
            "FlowVariant": flow_variant,
            "Files": v,
        }
        for (functional_variant, flow_variant), v in testcase_files.items()
    ]

    return testcases


def delete_linked_files(juliet_base_dir_env):
    import os
    try:
```

```
                                                                    for filename in os.listdir(juliet_base_dir_env):
                                                                        if filename.startswith("linked-"):
                                                                            file_path = os.path.join(juliet_base_dir_env, filename)
                                                                            os.remove(file_path)
                                                                            print(f"Deleted: {file_path}")
                                                                except FileNotFoundError:
                                                                    print(f"Directory '{juliet_base_dir_env}' not found.")
                                                                except PermissionError:
                                                                    print(f"Permission denied while trying to delete files in '{juliet_base_dir_env}'.")
                                                                except Exception as e:
                                                                    print(f"An error occurred: {e}")


                                                            def _get_juliet_base_dir(is_malware: bool) -> pathlib.Path:
    import os                                                   import os

    try:                                                        try:
        juliet_base_dir_env = os.environ["JULIET_BASE_DIR"]         if is_malware:

                                                                        juliet_base_dir_env = os.environ["MALWARE_BASE_DIR"]
                                                                        delete_linked_files(juliet_base_dir_env+"/dataset")
                                                                    else:
                                                                        juliet_base_dir_env = os.environ["JULIET_BASE_DIR"]
        juliet_base_dir = pathlib.Path(juliet_base_dir_env)         juliet_base_dir = pathlib.Path(juliet_base_dir_env)

    except KeyError:                                            except KeyError:
        # Environment not set, try local share folder               # Environment not set, try local share folder
        local_share_folder = pathlib.Path.home() / ".local" / "share"       local_share_folder = pathlib.Path.home() / ".local" / "share"
        if local_share_folder.exists():                             if local_share_folder.exists():
            # See if we have to make the ROMEO folder                   # See if we have to make the ROMEO folder
            local_share_romeo_folder = local_share_folder / "romeo"         local_share_romeo_folder = local_share_folder / "romeo"
            local_share_romeo_folder.mkdir(exist_ok=True)                   local_share_romeo_folder.mkdir(exist_ok=True)

            # Inside, there should be a "C" folder with the testcases       # Inside, there should be a "C" folder with the testcases
            juliet_base_dir = local_share_romeo_folder / "C"                juliet_base_dir = local_share_romeo_folder / "C"
            if not juliet_base_dir.exists():                                if not juliet_base_dir.exists():
                # If not, attempt to extract the object files                   # If not, attempt to extract the object files
                tar_file = (                                                    tar_file = (
                    pathlib.Path(__file__).parent.parent.parent                     pathlib.Path(__file__).parent.parent.parent
                    / "object_files"                                                / "object_files"
                    / "romeo_object_files.tar.gz"                                   / "romeo_object_files.tar.gz"
                )                                                               )
                if not tar_file.exists():                                       if not tar_file.exists():
                    raise ValueError(                                               raise ValueError(
                        f"Attempted to extract object files from ROMEO distribution, but could not locate them at {str(tar_file)}"                 f"Attempted to extract object files from ROMEO distribution, but could not locate them at {str(tar_file)}"
                    )                                                               )
```

```
                # Extract the file
                import tarfile

                tar_obj = tarfile.open(tar_file)
                tar_obj.extractall(local_share_romeo_folder)
                tar_obj.close()
            pass
        else:
            raise ValueError(
                "JULIET_BASE_DIR environment variable not set and ~/.local/share does not exist"
            )

    return juliet_base_dir


def enumerate_cwes() -> list[dict]:
    """Enumerate all CWEs in the Juliet test suite, which can later be used to access testcases."""
    juliet_base_dir = _get_juliet_base_dir()

    assert str(juliet_base_dir.name).lower() == "c"
    testcases = juliet_base_dir / "testcases"

    cwe_regex = r"^CWE([0-9]+)_(.+)$"
    cwes = []

    for cwe_path in testcases.iterdir():
        cwe_path_basedir = cwe_path.name
        assert (matches := re.match(cwe_regex, cwe_path_basedir)) is not None

        cwe = {}
        cwe["WeaknessID"] = int(matches[1])
        cwe["WeaknessShortenedName"] = str(matches[2]).replace("_", " ")
        cwe["Path"] = cwe_path

        cwes += [cwe]

    return list(sorted(cwes, key=lambda cwe: cwe["WeaknessID"]))


def filter_testcases(
    cwes: list[dict],
```

```
                # Extract the file
                import tarfile

                tar_obj = tarfile.open(tar_file)
                tar_obj.extractall(local_share_romeo_folder)
                tar_obj.close()
            pass
        else:
            raise ValueError(
                "JULIET_BASE_DIR environment variable not set and ~/.local/share does not exist"
            )

    return juliet_base_dir


def enumerate_cwes_malware(is_malware: bool) -> list[dict]:
    """Enumerate all CWEs in the Juliet test suite, which can later be used to access testcases."""

    testcases = _get_juliet_base_dir(is_malware)

    malwares = []

    for cwe_path in testcases.iterdir():
        cwe_path_basedir = cwe_path.name

        malware = {"WeaknessID": 1998, "WeaknessShortenedName": "Mal-007", "Path": cwe_path}

        malwares += [malware]

    return list(sorted(malwares, key=lambda cwe: cwe["WeaknessID"]))


def enumerate_cwes(is_malware: bool) -> list[dict]:
    """Enumerate all CWEs in the Juliet test suite, which can later be used to access testcases."""
    juliet_base_dir = _get_juliet_base_dir(is_malware)

    assert str(juliet_base_dir.name).lower() == "c"
    testcases = juliet_base_dir / "testcases"

    cwe_regex = r"^CWE([0-9]+)_(.+)$"
    cwes = []

    for cwe_path in testcases.iterdir():
        cwe_path_basedir = cwe_path.name
        assert (matches := re.match(cwe_regex, cwe_path_basedir)) is not None

        cwe = {}
        cwe["WeaknessID"] = int(matches[1])
        cwe["WeaknessShortenedName"] = str(matches[2]).replace("_", " ")
        cwe["Path"] = cwe_path

        cwes += [cwe]

    return list(sorted(cwes, key=lambda cwe: cwe["WeaknessID"]))


def filter_testcases(
    cwes: list[dict],
```

```
    testcases: list[dict],
    testcase_allowed: typing.Callable[[dict], bool],
) -> list[dict]:
    """Apply a given filter function to the testcases."""
    allowed_testcases: list[dict] = []
    for testcase in testcases:
        if testcase_allowed(testcase):
            allowed_testcases += [testcase]

    return allowed_testcases


def filter_cwes(
    cwes: list[dict],
    testcases: list[dict],
    cwe_allowed: typing.Callable[[dict, dict], bool],
) -> list[dict]:
    """Apply a given filter function to the cwes."""
    allowed_cwes: list[dict] = []
    for cwe in cwes:
        if cwe_allowed(cwe, testcases):
            allowed_cwes += [cwe]

    return allowed_cwes


__all__ = [
    "filters",
    "filter_cwes",
    "filter_testcases",
    "enumerate_cwes",
    "enumerate_testcases",
]
```

```
    testcases: list[dict],
    testcase_allowed: typing.Callable[[dict], bool],
) -> list[dict]:
    """Apply a given filter function to the testcases."""
    allowed_testcases: list[dict] = []
    for testcase in testcases:
        if testcase_allowed(testcase):
            allowed_testcases += [testcase]

    return allowed_testcases


def filter_cwes(
    cwes: list[dict],
    testcases: list[dict],
    cwe_allowed: typing.Callable[[dict, dict], bool],
) -> list[dict]:
    """Apply a given filter function to the cwes."""
    allowed_cwes: list[dict] = []
    for cwe in cwes:
        if cwe_allowed(cwe, testcases):
            allowed_cwes += [cwe]

    return allowed_cwes


__all__ = [
    "filters",
    "filter_cwes",
    "filter_testcases",
    "enumerate_cwes",
    "enumerate_testcases",
    "enumerate_cwes_malware",
    "enumerate_testcases_malware",
]
```