File Name: Exercise 3 Report

Student Name: Wuli Zuo

Student ID: a1785343

Vulnerabilities Assigned: CVE-2018-11087，CVE-2018-11797，CVE-2017-5647

GitHub account: DoriaSummer

Link to the repository of Exercise 3: https://github.com/DoriaSummer/sse_exercise3

In Exercise 3, Python and Git commands are used to identify VCCs. The structure of the Python scripts is described in Section 1 of this report, and detailed implementation of Python codes and Git commands are explained in Section 2. Only example output is showed in this section, so the key output of the three cases is included in Section 4. For the complete output, please refer to the file `Output_of_identify`. The test results of different parameters in `git.blame` are analysed in Section 3. The analyses of the VCCs are the same as in Exercise 2, this report will not go in details with the implementation, and just puts the analysis results of the VCCs into `Output_of _analyse`.

1. Structure of the automated scripts

   The automated scripts are written with Python. Functions from `GitPython` library are used to get necessary information and identify target commits.

   In the file `Identify.py`, the function `git_identify()` is used to identify the latest commit that modified each deleted line and added line scope, and to identify the most frequently identified commit as the VCC. Each question of Task 3 is answered by this file. In the file `TestBlame.py`, the function `git_test_blame()` is used to identify VCCs using different parameters as options of the `git.blame` command. This is to answer the question of Task 5.

   The file `Analyse.py` is to analyse the VCCs. It is reused of the same one as in Exercise 2.

   In the source code files, `git.blame()` is used for identifying the latest commit information of a certain commit, or a certain file/a certain line or line scope based on a

commit. Other methods from `GitPython`, including `git.clone()`, `git.reset()`, and `git.show()`are used similarly as in Exercise 2.

Specific comments are included in the source code files. It is most recommended to read the code in Github to get details.

2. Task 3 - Explanations of the Git commands and Python codes

This part aims to explain Git commands used to answer questions of task 3, and corresponding Python codes in `Identify.py` are showed as well.

The `main()` function calls `git_identify()` for three cases one by one:

Code pieces 1: main()

```
# main
class Progress(RemoteProgress):
    def update(self, op_code, cur_count, max_count=None, message=''):
        print(self._cur_line)

# Case 1
remote_link = "https://github.com/spring-projects/spring-amqp"
local_link = "../spring-amqp"
if not os.path.isdir(local_link):
    Repo.clone_from(remote_link, local_link, progress=Progress())
fixing_commit = "444b74e95bb299af5e23ebf006fbb45d574fb95"
print("\nOperation of repo: %s\ncommit: %s" % (remote_link, fixing_commit))
git_identify(local_link, fixing_commit)

# Case 2
remote_link = "https://github.com/apache/pdfbox"
local_link = "../pdfbox"
if not os.path.isdir(local_link):
    Repo.clone_from(remote_link, local_link, progress=Progress())
fixing_commit = "4fa98533358c106522cd1bfe4cd9be2532af852"
print("\nOperation of repo: %s\ncommit: %s" % (remote_link, fixing_commit))
git_identify(local_link, fixing_commit)

# Case 3
remote_link = "https://github.com/apache/tomcat80"
local_link = "../tomcat80"
if not os.path.isdir(local_link):
    Repo.clone_from(remote_link, local_link, progress=Progress())
fixing_commit = "ec10b8c785d1db91fe58946436f854dde04410fd"
print("\nOperation of repo: %s\ncommit: %s" % (remote_link, fixing_commit))
git_identify(local_link, fixing_commit)
```

The function `git_identify()` first resets to the given commit, then operate all the affected files one by one to deal with each affected line, in order to find the latest commit that modified each deleted lines and added line scope, with the frequency of each commit

calculated during this process, and last select the most frequently identified commit as VCC for this vulnerability.

Code pieces 2: structure of git_identify()

```
for file in repo.git.show('--name-only', '--format=').splitlines(): # Operate all the affected files one by one
    if file == "webapps/docs/changelog.xml": # Ignore log file
        continue
    else:
        print("\nFile: %s" % file)
        lines = repo.git.show(file).splitlines()
        line_count = 0
        start_line = 0

        # Find target commit of deleted lines
        print("\n(a). Find latest modifying commit for each deleted lines")
        for line in lines:...

        # Find target commit of added lines' smallest scope
        file_blame_info = repo.git.blame("-l", fixing_commit+"^", file).splitlines()
        line_count = 0
        start_line = 0

        # Find smallest scope for each added scope in current commit
        print("\n(b)-1. Find smallest scope for each added lines")
        # Find all added lines
        # added_lines = set()
        added_lines_num = set()
        scopes = set()
        for line in lines:...
        # Find smallest scope for each added scope in current commit
        for line_num in added_lines_num:...

        # Find target commit for each line of each identified scope in previous commit
        print("\n(b)-2. Find target scope for each scope")
        for scope in scopes:...

# Find most frequently identified commit as the VCC
print("\n(c). Select VCC")
print("All commits: %s" % commits)
print("Count of commits: %s" % commits_count)

vcc = commits[commits_count.index(max(commits_count))]
print("VCC is: %s" % vcc)
```

a. If a line was deleted, identify the latest commit that modified that line and record that commit.

First, identify the line number for a deleted line by finding the line start with `'@@'` to get the line number of the first line of code scope in the log and counting from this line. Then use the line number as a parameter for `git.blame()` to get the blame information of this line and read the commit ID. If the commit ID does not exist in the list `commits`, add it and add a new element as `1` to the list `commits_count`. If it already exists in the list `commits`, plus 1 to the corresponding element in the list `commits_count`. In this way, the frequency of every identified commit is calculated.

For the method `git.blame()`, the parameter **"-L"** is used to limit the output scope of line/lines in `(m, n)`, and here we use `(current line number, +1)` to print only current line; the parameter **"-l"** is used to print long commit ID; the parameter **"--line-porcelain"** is used to print the blame information in separated lines to make is readable by a computer; the parameter `fixing_commit^` is used here to locate to the previous commit because we need to find the latest modification for this line before it was deleted, and `file` is to specify which file is to blame.

Code pieces 3: identify commits of deleted lines (Question a)

```
# Find target commit of deleted lines
print("\n(a). Find latest modifying commit for each deleted lines")
for line in lines:
    if line.startswith("@@"):
        temp_start = line.split()
        temp_start = temp_start[1].split("-")
        temp_start = temp_start[1].split(",")
        start_line = int(temp_start[0])
        print("start line: %d" % start_line)
        line_count = 0
    else:
        if start_line == 0:
            continue
        else:
            if line.startswith("+"):
                continue
            if line.startswith("-"):
                line_num = int(start_line) + int(line_count)
                print("Deleted line: %d" % line_num)
                # For the deleted line, blame the previous commit
                blame_info = repo.git.blame("-L", str(line_num)+",+1", "--line-porcelain",
                                            fixing_commit+"^", file).splitlines()
                target_commit = blame_info[0]
                target_commit = target_commit.split()[0]
                print("Target commit for Line %d: %s" % (line_num,target_commit))
                if not target_commit in commits:
                    commits.append(target_commit)
                    commits_count.append(1)
                else:
                    index = commits.index(target_commit)
                    commits_count[index] += 1
                line_count += 1
            else:
                line_count += 1
```

Example output 1: identify commits of deleted lines (one affected file from Case 1)

```
Operation of repo: https://github.com/spring-projects/spring-amqp
commit: 444b74e95bb299af5e23ebf006fbb45d574fb95

File: build.gradle

(a). Find latest modifying commit for each deleted lines
start line: 82
Deleted line: 85
Target commit for Line 85: ebd10bbd9d6955d687af67c72dabc6c8ab351cfd
```

b. If a line is added, first identify the smallest enclosing scope (e.g., class/interface, function, for, while, if, else, switch, try, catch, finally). In case there are also other relevant pieces of code that you think can also lead to the current added line, please specify and explain them in details. Then, identify the latest commit that modified the selected lines and record that commit.

   Step 1, use a similar way to get line number of each added line, find the smallest code scope if this line by searching for the first non-paired `'{'` above the added line and the first non-paired `'}'` below the added line. Then use the code of these two lines as start line and end line to match the previous commit, so that the scope of the added line can be identified, and store the scope as `(begin line number, end line number)`into the set `scopes`.

   Step 2, use a similar way to identify the latest commit for each line in each scope in the set, and calculate the frequency during the process.

   No other relevant pieces of code also lead to the added lines.

Code pieces 4: identify the smallest scope of added lines (Step 1 of Question b)

```
# Find target commit of added lines' smallest scope

line_count = 0
start_line = 0

# Find smallest scope for each added scope in current commit
print("\n(b)-1. Find smallest scope for each added lines")
# Find all added lines
# added_lines = set()
added_lines_num = set()
scopes = set()
for line in lines:
    if line.startswith("@@"):
        temp_start = line.split()
        temp_start = temp_start[2].split("+")
        temp_start = temp_start[1].split(",")
        start_line = int(temp_start[0])
        print("start line: %d" % start_line)
        line_count = 0
    else:
        if start_line == 0:
            continue
        else:
            if line.startswith("-"):
                continue
            if line.startswith("+"):
                line_num = int(start_line) + int(line_count)
                print("Added line: %d" % line_num)
                # added_lines.add(line)
                added_lines_num.add(line_num)
                line_count += 1
            else:
                line_count += 1
```

```
# Find smallest scope for each added scope in current commit
file_blame_info = repo.git.blame(fixing_commit+"^", file).splitlines()
begin_lines = set()
end_lines = set()
for line_num in added_lines_num:
    scope_begin_line = ""
    scope_end_line = ""
    check = 0 # check if a line's scope is in a new added scope
    # Find scope begin line
    for i in range (1, line_num-1):
        scope_begin_signs = 0
        scope_end_signs = 0
        blame_info = repo.git.blame("-L", str(line_num-i)+",+1", "--line-porcelain",
                                    fixing_commit, file).splitlines()
        # print("blame_info: %s" % blame_info)
        code = blame_info[len(blame_info)-1]
        # print("Line %d, Code content: %s" % (line_num-i,code))
        if '{' in code:
            scope_begin_signs +=1
        if '}' in code:
            scope_end_signs +=1
        if i == line_num-1:
            break
        else:
            if scope_begin_signs > scope_end_signs:
                if line_num-i not in added_lines_num and line_num-i not in begin_lines:
                    scope_begin_line = code
                    check = 1
                    begin_lines.add(i)
                    # print("scope_begin_line of line %d: %s" % (line_num,scope_begin_line))
                break
```

```
        # Find scope end line
        # file_blame_info = repo.git.blame(fixing_commit, file).splitlines()
        # print("test: %s" % file_blame_info[0])
        for i in range (line_num+1, len(file_blame_info)):
            scope_begin_signs = 0
            scope_end_signs = 0
            blame_info = repo.git.blame("-L", str(i)+",+1", "--line-porcelain",
                                        fixing_commit, file).splitlines()
            # print("blame_info: %s" % blame_info)
            code = blame_info[len(blame_info)-1]
            # print("Line %d, Code content: %s" % (i,code))
            if '}' in code:
                scope_end_signs +=1
            if '{' in code:
                scope_begin_signs +=1
            if i == len(file_blame_info):
                break
            else:
                if scope_end_signs > scope_begin_signs:
                    if i not in added_lines_num:
                        scope_end_line = code
                        check = 1
                        # print("scope_end_line of line %d: %s" % (line_num,scope_end_line))
                    break
```

```
        if check == 1:
            # Find scope begin and end line number in the previous commit
            blame_info_pre = repo.git.blame(fixing_commit+"^", file).splitlines()
            scope_begin_line_num = 1
            scope_end_line_num = len(blame_info_pre)-1

            for i in range(1, len(blame_info_pre)):
                line = repo.git.blame("-L", str(i)+",+1", "--line-porcelain",
                                      fixing_commit+"^", file).splitlines()
                code = line[len(line) - 1]
                # print("%s, %s" % (code, scope_begin_line))
                if scope_begin_line_num == 1:
                    if code == scope_begin_line:
                        # print("%s, %s" % (code, scope_begin_line))
                        scope_begin_line_num = i
                        continue
                else:
                    if code == scope_end_line:
                        scope_end_line_num = i
                        break
            scope = str(scope_begin_line_num) + "," + str(scope_end_line_num)
            print("scope for line %d in the previous commit: %s" % (line_num, scope) )
            scopes.add(scope)
```

Code pieces 5: identify commits of lines in scopes (Step 2 of Question b)

```
    # Find target commit for each line of each identified scope in previous commit
    print("\n(b)-2. Find target commit for each scope")
    for scope in scopes:
        blame_info_pre = repo.git.blame("-L", scope, fixing_commit+"^", file).splitlines()
        # print("blame info: %s" % blame_info)
        for line in blame_info_pre:
            # print("blame_line: %s" % line)
            target_commit = line.split()[0]
            print("Target commit for a line in scope %s: %s " % (scope,target_commit))
            if not target_commit in commits:
                commits.append(target_commit)
                commits_count.append(1)
            else:
                index = commits.index(target_commit)
                commits_count[index] += 1
```

Example output 2: identify commits of added lines scopes (one affected file from Case 1)

```
(b)-1. Find smallest scope for each added lines
start line: 82
Added line: 85
scope for line 85 in the previous commit: 72,91

(b)-2. Find target scope for each scope
Target commit for a line in scope 72,91: 5c719a46461178fce345d7208a848f56d41233bb
Target commit for a line in scope 72,91: f678078e293fd0d23278febfcf315bd2e663baad
Target commit for a line in scope 72,91: f678078e293fd0d23278febfcf315bd2e663baad
Target commit for a line in scope 72,91: 8bb92e516bee00262dc7b8d62866538c95394ed6
Target commit for a line in scope 72,91: f25947350f3613e1686d06661cc9e029cfc51942
Target commit for a line in scope 72,91: d5494f4ed4544c2e472415ddc9b0a46842d90be6
Target commit for a line in scope 72,91: d316dbf6c8eb66b8e525bc0486b7334712ac11b5
Target commit for a line in scope 72,91: d73f4ffc0a2d866ea4f0349a5f48d9763f433a5f
Target commit for a line in scope 72,91: 80848fd65810e34092116d4642a53c7a3d6c9f5b
Target commit for a line in scope 72,91: 80848fd65810e34092116d4642a53c7a3d6c9f5b
Target commit for a line in scope 72,91: d316dbf6c8eb66b8e525bc0486b7334712ac11b5
Target commit for a line in scope 72,91: 6b219ae30174393b94c156acd94af2595275ebe4
Target commit for a line in scope 72,91: 80848fd65810e34092116d4642a53c7a3d6c9f5b
Target commit for a line in scope 72,91: ebd10bbd9d6955d687af67c72dabc6c8ab351cfd
Target commit for a line in scope 72,91: 8bb92e516bee00262dc7b8d62866538c95394ed6
Target commit for a line in scope 72,91: 5c719a46461178fce345d7208a848f56d41233bb
Target commit for a line in scope 72,91: 8df8260421eef3b30f98dfce42627d0be18b827e
Target commit for a line in scope 72,91: d5494f4ed4544c2e472415ddc9b0a46842d90be6
Target commit for a line in scope 72,91: d547a865b45fe9a1d4a03760448dda6423b9807c
Target commit for a line in scope 72,91: 5c719a46461178fce345d7208a848f56d41233bb
```

c. Then, select the most frequently identified commit as the VCC for the current vulnerability. In case there is more than one such commit with the same frequency, select all of them.

   Because each identified commit and its frequency are stored in a list respectively, here the only thing we need to do is to find the max value in the list `commits_count`, and the element in the list `commits` with same index is the most frequency identified commit and is selected as VCC.

Code pieces 6: identify VCC (Question c)

```
# Find most frequently identified commit as the VCC
vcc = commits[commits_count.index(max(commits_count))]
print("VCC is: %s" % vcc)
```

Example output 3: identify VCC (Case 1)

```
(c). Select VCC
VCC: 127d6aabc
```

3. Task 5 – compare results of different parameter in `git.blame()`

As `TestBlame.py` is kind of repetition process of `Identify.py`, and the only difference is that different parameters of `['-w', '-wM', '-wC', '-wCC']` are used in the method `git.blame()`, the implementation of this file will not been explained in details.

Code pieces 7: test different parameters in git.blame() (Task 5)

```python
# Function of git commands to identify VCC
def git_test_blame(local_link, fixing_commit):

    # Create repo object
    repo = Repo(local_link)

    # Reset to given commit
    repo.git.reset('--hard', fixing_commit)

    # Task 5: Compare results of different git.blame parameters
    # '-wCCC' is ignored because of the error: fatal: bad revision '-wCCC'

    parameters = ['-w', '-wM', '-wC', '-wCC']

    for parameter in parameters:

        # Frequencies of commits are calculated during Question a and b
        # Question a: identify the latest commit that modified each deleted line
        commits = []
        commits_count = []
```

Example output 4: VCCs with different parameters (All cases)

```
/usr/local/bin/python3.7 /Users/zuowuli/Workspace/sse/TestBlame.py

Operation of repo: https://github.com/spring-projects/spring-amqp
VCC with Parameter -w is: 127d6aabc
VCC with Parameter -wM is: 127d6aabc
VCC with Parameter -wC is: 127d6aabc
VCC with Parameter -wCC is: 127d6aabc

Operation of repo: https://github.com/apache/pdfbox
VCC with Parameter -w is: 0043363995
VCC with Parameter -wM is: 0043363995
VCC with Parameter -wC is: 0043363995
VCC with Parameter -wCC is: 0043363995

Operation of repo: https://github.com/apache/tomcat80
VCC with Parameter -w is: 85a546bbf3
VCC with Parameter -wM is: 85a546bbf3
VCC with Parameter -wC is: 85a546bbf3
VCC with Parameter -wCC is: 85a546bbf3
```

There is no difference among different parameters, and the identified VCCs are all the same as with the VCCs got from Section 2 using `git.blame()` with no parameter of `'w'`, `'M'` or `'C'`, perhaps because there is no cross file modification occurred on these codes.

One thing needs to be noted is that the parameter `'-wCCC'` is not included, because it was found that there is an error of 'bad revision' reported both in IDE and bash terminal.

Error message 1: IDE error report of '-wCCC'

```
git.exc.GitCommandError: Cmd('git') failed due to: exit code(128)
  cmdline: git blame -wCCC -L 85,+1 -l --line-porcelain 444b74e95bb299af5e23ebf006fbb45d574fb95 build.gradle
  stderr: 'fatal: bad revision '-wCCC''
```

Error message 2: bash terminal error report of '-wCCC'

```
(base) zuowulideMBP:spring-amqp zuowuli$ git blame -wCCC build.gradle
fatal: bad revision '-wCCC'
```

4. Key Output

4.1 VCC of Case 1:

- CVE-ID: CVE-2018-11087

- Link: https://github.com/spring-projects/spring-amqp
- Fixing Commit: f8e7732ce69e5f3e591700bebbf00682ce7ab231 (the identified commit)

```
Key output of case 1:
```

```
Operation of repo: https://github.com/spring-projects/spring-amqp

commit: 444b74e95bb299af5e23ebf006fbb45d574fb95

VCC is:  127d6aabc
```

4.2 VCC of Case 2

- CVE-ID: CVE-2018-11797
- Link: https://github.com/apache/pdfbox
- Fixing Commit: 4fa98533358c106522cd1bfe4cd9be2532af852

```
Key output of case 2:
```

```
Operation of repo: https://github.com/apache/pdfbox

commit: 4fa98533358c106522cd1bfe4cd9be2532af852

VCC is: 004336399
```

4.3 VCC of Case 3

- CVE-ID: CVE-2017-5647
- Link: https://github.com/apache/tomcat80
- Fixing Commit: ec10b8c785d1db91fe58946436f854dde04410fd

```
Key output of case 3:
```

```
Operation of repo: https://github.com/apache/tomcat80

commit: ec10b8c785d1db91fe58946436f854dde04410fd

VCC is: 85a546bbf3
```