Junior Student Security Engineer Challenges

TECHNISCHE UNIVERSITÄT DARMSTADT

Max Pengrin
November 29, 2020

Contents

1 Challenge 1 (Static Analysis)

1.1 Explanation of my approach

The code is basically divided in three main parts:

- Commandline parameter parsing
- Parsing of the input File
- Traversing the callgraph

The first part is quite trivial. The second and third part is more interesting.
In the second part I construct a directed graph while parsing the file.
In the third part I traverse the graph using a depth first approach.
The code can be seen in [1]. The used Libraries are mentioned in the next section.

1.2 Used Libraries

argparse	-	to parse commandline parameters
re	-	for regular expressions

Table 1: Used Libraries with simple explanation

2 Challenge 2 (Dynamic Analysis)

2.1 Explanation of my approach

The code is reused from challenge 1[1.1]. This time the code is divided in two main parts:

- Commandline parameter parsing

- Input, parse and output

The first part is basically the same like in challenge 1.
In the second part, I write to the output file while using the same parser from challenge 1. We don't need to build a callgraph this time though. But the state change is useful to trigger to write to the file.
The code can be seen in 2. The used Libraries are mentioned in the next section.

2.2 Used Libraries

argparse	-	to parse commandline parameters
re	-	for regular expressions

Table 2: Used Libraries with simple explanation

3 Code

Listing 1: Solution Challenge 1: call-graph.py

```
#!/usr/bin/env python3
# coding: utf-8

import argparse

parser = argparse.ArgumentParser()
parser.add_argument('input', nargs=1, help='Input file name.')
args = parser.parse_args()

filename = args.input[0]

import re
# func main() {
re_func=re.compile(r"^func (.*)\(.*\) *\{$")
# bar()
re_call=re.compile(r"^\s*([^\.]*)\(.*\)")

def parse(line,re):
    m = re.search(line)
    if m is None:
        return None
    else:
        return m.group(1)


dict_CallGraph={}
DEBUG =0

with open(filename) as input:
  re=re_func
  for line in input:
    p=parse(line,re)

    if DEBUG > 0:
      print("## looking for ",end='')
      if re is re_func:
        print("func")
      if re is re_call:
        print("call")
```

```
      print(line,end='')

    if re == re_func and p != None:
      #cg=CallGraph(p, [])
      func = p
      calls = []


      if DEBUG >0: print(f"## we found the func: {p}")
      if DEBUG >1: print(f"#### cg: {cg}")

      re=re_call #next look for calls

    elif re == re_call and line.strip() == "}":
      #dict_CallGraph[cg.func]=cg.calls
      dict_CallGraph[func]=calls

      if DEBUG >0: print(f"## func {cg.func} has ended")
      if DEBUG >1: print(f"#### cg: {cg}")

      re=re_func  #look for func again

    elif re == re_call and p != None:
      #cg.calls.append(p)
      calls.append(p)

      if DEBUG >0: print(f"## we found a func call: {p}()")
      if DEBUG >1: print(f"#### cg: {cg}")

    if DEBUG >0: print()

DEBUG =0
maxFuncName=5
def traverse (graph,source,visited):
    #print (("{}".format(source)).ljust(maxFuncName), end = '')
    if DEBUG>0: print (' <{}|{}|{}>'.format(len(graph),len(visited),visited), end = '')
    if source in visited:
        if DEBUG>0: print ("cycle!", end = '')
        for i in visited:
                print("{}-->".format(i),end='')
        print (("{}".format(source)).ljust(maxFuncName))
        #print()
        #visited.remove(source)
        return
    visited.append (source)
    if DEBUG >2: print ("(:{})".format(graph[source]), end = '')
    i=0
    for node in graph[source]:
        if len(graph[node]) > 0:
            traverse (graph, node, visited)
        else:
            if DEBUG>0: print (' <{}|{}|{}>'.format(len(graph),len(visited),visited), end = '')
            for i in visited:
                print("{}-->".format(i),end='')
            print ("{} ".format(node))
            return
        i+=1
    visited.remove(source)
    return
```

```
traverse(dict_CallGraph, 'main', list())
```

Listing 2: Solution Challenge 2: instrument-function-names.py

```
#!/usr/bin/env python3
# coding: utf-8

import argparse

parser = argparse.ArgumentParser()
parser.add_argument('input', nargs=1, help='Input file name.')
parser.add_argument('output', nargs=1, help='Outpu file name.')
args = parser.parse_args()

in_filename = args.input[0]
out_filename = args.output[0]

import re
# func main() {
re_func=re.compile(r"^func (.*)\(.*\) *\{$")
# bar()
re_call=re.compile(r"^\s*([^\.]*)\(.*\)")

def parse(line,re):
    m = re.search(line)
    if m is None:
        return None
    else:
        return m.group(1)


DEBUG =0
#DEBUG=1
DEBUG=2

with open(out_filename,'w') as output, open(in_filename) as input :
  re=re_func
  for line in input:
    output.write(line)
    p=parse(line,re)

    if DEBUG > 0:
      print("## looking for ",end='')
      if re is re_func:
        print("func")
      if re is re_call:
        print("call")

      print(line,end='')

    if re == re_func and p != None:
      #cg=CallGraph(p, [])
      func = p
      calls = []

      output.write(f"\tfmt.Println(\"{p}\")\n")

      if DEBUG >0: print(f"## we found the func: {p}")
      #if DEBUG >1: print(f"#### cg: {cg}")
      re=re_call #next look for calls
```

```
elif re == re_call and line.strip() == "}":
  #dict_CallGraph[cg.func]=cg.calls

  if DEBUG >0: print(f"## func {func} has ended")
  #if DEBUG >1: print(f"#### cg: {cg}")

  re=re_func  #look for func again

elif re == re_call and p != None:
  calls.append(p)

  if DEBUG >0: print(f"## we found a func call: {p}()")
  #if DEBUG >1: print(f"#### cg: {cg}")

if DEBUG >0: print()
```