Ghidra guide: Static binary program analysis and hands-on with Ghidra

* Access /Tools/ghidra
* To start ghidra run the script ghidraRun ./ghidraRun
* Create a new project, name s3
* Drag and drop the file (dnstracer) under your active project and press the ghidra logo on the upper left side.
* At the task bar on the top you can press the green play button to open the script manager.
* There you can find your script and by pressing the green play button again on the top right you can run it.
* At the task bar you can press the help and open the Ghidra API Help which is very useful for your scripting. With the search bar on the top right, you can have information for anything you want to do with your script.
* The skeleton of the script you will use is available in the Scripts folder call-graph-back-slice.py
* The complete script is the following:

# For Ghidra Scripting tutorial purpose.

# Copy this file into ghidra\_scripts directory (the one which is configured during installation of ghidra, where ghidra can import it.)

#@author Sanjay Rawat

#@category \_NEW\_

#@keybinding

#@menupath

#@toolbar

#GHIDRA IMPORTS ###############################################################

#from ghidra.program.model.block import \*

#from ghidra.program.model.listing import \*

import ghidra.program.model.block as BL

from ghidra.program.model.lang.OperandType import SCALAR, REGISTER

import generic.continues as CNT

import ghidra.app.util.bin.ByteProvider

import ghidra.app.util.bin as UBIN

import ghidra.app.util.bin.format.elf as ELF

import java.io as IO

import ghidra.program.util as PUTIL

####################################################################################

from collections import deque

import struct

import shutil

import os

import sys

import copy

###########################################################################################

######## Ghidra global definitions ##############

sbm=BL.SimpleBlockModel(currentProgram, False)

ref\_manager=currentProgram.getReferenceManager()

func\_manager=currentProgram.getFunctionManager()

#################################################

def get\_function\_information():

file\_content\_dict = {}

function\_list = []

# Get the current program's function names

function = getFirstFunction()

while function is not None:

monitor.checkCanceled()

addr\_view=function.getBody()

func\_min=addr\_view.getMinAddress().getOffset()

func\_max=addr\_view.getMaxAddress().getOffset()

func\_name=function.getName()

#print "name %s, entry: 0x%x, Min: 0x%x, Max: 0x%x "%(function.getName(),function.getEntryPoint().getOffset(),func\_min, func\_max)

function=getFunctionAfter(function)

def get\_xref\_information(function):

#print("-----------------------------------------get\_xref\_information-------------------------------------------------")

callers = set() #set of the callers of the function

print "[\*] Function: %s"%(function.getName(),)

addr\_view=function.getBody()

# func\_min=addr\_view.getMinAddress().getOffset()

#func\_max=addr\_view.getMaxAddress().getOffset()

#print "name %s, entry: 0x%x, Min: 0x%x, Max: 0x%x "%(function.getName(),function.getEntryPoint().getOffset(),func\_min, func\_max)

current\_function\_address = function.getEntryPoint()

#location=PUTIL.FunctionSignatureFieldLocation(function.getProgram(),current\_function\_address)

ref\_to\_current=ref\_manager.getReferencesTo(current\_function\_address)

while(ref\_to\_current.hasNext()):

ref=ref\_to\_current.next()

if ref.getReferenceType().isCall()==True:

callsite\_addr=ref.getFromAddress()

caller\_func=func\_manager.getFunctionContaining(callsite\_addr)

if caller\_func==None:

continue

callers.add(caller\_func)

caller\_BB=sbm.getCodeBlocksContaining(callsite\_addr, monitor)

#print "\t callsite: 0x%x, caller BB len %d"%(callsite\_addr.getOffset(),len(caller\_BB))

#print "\t caller: %s (0x%x), BB: 0x%x, callsite: 0x%x"%(caller\_func.getName(), caller\_func.getEntryPoint().getOffset(), caller\_BB[0].getFirstStartAddress().getOffset(),callsite\_addr.getOffset())

print "Done. Found %d callers"%(len(callers),)

return callers.copy()

def get\_call\_graph():

print("===== Compute Call Graph Slice =====")

# Using the get-xref-information() function, start with a source function and

# get it's callers. Iterate over this list of callers and

# repeat the process. This will give you a backward slice starting from

# the 1st function until main()

# in this example, we will look for strcpy function and then start a backward callgraph slice.

func\_iter=func\_manager.getFunctions(True)

while(func\_iter.hasNext() and monitor.isCancelled() != True):

function=func\_iter.next()

fname=function.getName()

if 'strcpy' in fname:

print "[\*] Found strcpy.."

break

#lets get back all the callers of the function

caller=get\_xref\_information(function)

print "[\*] callers of strcpy:"

for cfun in caller:

print '{0:s} - 0x{1:x}'.format(cfun.getName(),cfun.getEntryPoint().getOffset())

if \_\_name\_\_ == '\_\_main\_\_':

#get\_func\_attribute(functionAttrPath,file\_name)

exe\_format=currentProgram.getExecutableFormat()

#if idc.GetLongPrm(idc.INF\_FILETYPE) == idc.FT\_ELF:

if 'ELF' in exe\_format:

print("Linux ELF File")

# config all the paths

#file\_path=currentProgram.getExecutablePath()

#base,file\_name=os.path.split(file\_path)

#disassembledFileName = file\_name

print("-------- Starting --------------------")

# generate xref file, ok

#get\_xref\_information()

# generate call graph file, ok

get\_call\_graph()

# generate function information

#get\_function\_information()

print("Analysis finished")

else:

print("not suitable file type")

The analysis has finished and the expected output of the vulnerability can be seen in the console – scripting. You can see the function strcpy with the printable name 0x1032e0 and if you click on that you will be able to see the vulnerability point in the code.