```
# Sality Extractor
# Copyright (C) 2017 quangnh89, develbranch.com
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU Affero 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 Affero General Public License for more details.
#
# You should have received a copy of the GNU Affero General Public License
# along with this program. If not, see .
#
# blog: https://develbranch.com
# email: contact[at]develbranch.com

import pefile
import struct
import re
import argparse
from unicorn import *
from unicorn.x86_const import *
from capstone import *
from keystone import *
from datetime import datetime


class SalityExtractor():
def __init__(self, sample_file=None, output_file=None):
self.md = Cs(CS_ARCH_X86, CS_MODE_32)
self.md.detail = True
self.sample = sample_file
self.output = output_file
self.detected = False
self.control_server = []

# utility methods
@staticmethod
# display log message
def log(msg):
print str(datetime.now()), msg

# dump all mapped memory to file
def dump_to_file(self, mu, pe, filename, new_ep_rva=None, runable=True):
memory_mapped_image = bytearray(mu.mem_read(pe.OPTIONAL_HEADER.ImageBase, pe.OPTIONAL_HEADER.SizeOfImage))
for section in pe.sections:
va_adj = pe.adjust_SectionAlignment(section.VirtualAddress, pe.OPTIONAL_HEADER.SectionAlignment,
pe.OPTIONAL_HEADER.FileAlignment)
if section.Misc_VirtualSize == 0 or section.SizeOfRawData == 0:
continue
if section.SizeOfRawData > len(memory_mapped_image):
continue
if pe.adjust_FileAlignment(section.PointerToRawData, pe.OPTIONAL_HEADER.FileAlignment) > len(
memory_mapped_image):
continue
pe.set_bytes_at_rva(va_adj, bytes(memory_mapped_image[va_adj: va_adj + section.SizeOfRawData]))
```

```
pe.write(filename)
# set new entrypoint
if new_ep_rva is not None:
self.log("New entry point %08x" % new_ep_rva)
f = open(filename, 'r+b')
f.seek(pe.DOS_HEADER.e_lfanew + 4 + pe.FILE_HEADER.sizeof() + 0x10)
f.write(struct.pack(' if not runable:
f.seek(0)
f.write('mz')
f.close()
print ('[+] Save to file {}'.format(filename))

@staticmethod
def assembler(address, assembly):
ks = Ks(KS_ARCH_X86, KS_MODE_32)
encoding, _ = ks.asm(assembly, address)
return ''.join(chr(e) for e in encoding)

# callback for tracing invalid memory access (READ or WRITE)
# noinspection PyUnusedLocal
@staticmethod
def hook_mem_invalid(uc, access, address, size, value, user_data):
# return False to indicate we want to stop emulation
return False

# callback for tracing fake-IAT interrupt
# noinspection PyUnusedLocal
def hook_intr(self, uc, intno, user_data):
# only handle fake-IAT interrupt
if intno != 0xff:
print ("got interrupt %x ???" % intno)
uc.emu_stop()
return
eax = uc.reg_read(UC_X86_REG_EAX)
dll_name, address, name, _ = self.import_addrs[eax]
if 'kernel32' in dll_name.lower():
if name == 'LoadLibraryA':
uc.reg_write(UC_X86_REG_EAX, 0xabababab)
elif name == 'GetProcAddress':
uc.reg_write(UC_X86_REG_EAX, 0xbcbcbcbc)
elif name == 'VirtualProtect':
uc.reg_write(UC_X86_REG_EAX, 0x1)

# noinspection PyBroadException
# noinspection PyUnresolvedReferences
def emulate_sality_dll(self, memory):
try:
pe = pefile.PE(data=memory, fast_load=True)
except:
return None

self.log("[+] Parse Sality DLL")
pe.parse_data_directories()
self.import_addrs = []
for entry in pe.DIRECTORY_ENTRY_IMPORT:
for imp in entry.imports:
nparam = 1
if entry.dll.lower() in 'kernel32.dll':
if imp.name == 'LoadLibraryA':
nparam = 1
elif imp.name == 'GetProcAddress':
nparam = 2
elif imp.name == 'VirtualProtect':
nparam = 4
```

```
self.import_addrs.append((entry.dll, imp.address, imp.name, nparam))

self.log('[+] Analyze UPX stub code')
entry_point_code = str(pe.get_memory_mapped_image())[pe.OPTIONAL_HEADER.AddressOfEntryPoint:]
begin_addr = pe.OPTIONAL_HEADER.ImageBase + pe.OPTIONAL_HEADER.AddressOfEntryPoint
end_addr = begin_addr
for i in self.md.disasm(str(entry_point_code), begin_addr):
if i.mnemonic.lower() in ['popad', 'popal', 'popa']:
end_addr = i.address + 1
break
self.log("[+] Initialize emulator in X86-32bit mode")
mu = Uc(UC_ARCH_X86, UC_MODE_32)
# map memory for this emulation
mu.mem_map(pe.OPTIONAL_HEADER.ImageBase, pe.OPTIONAL_HEADER.SizeOfImage)
# stack
stack_addr = 0x1000
stack_size = 0x4000
mu.mem_map(stack_addr, stack_size)
# write machine code to be emulated to memory
mu.mem_write(pe.OPTIONAL_HEADER.ImageBase, pe.get_memory_mapped_image())
# initialize machine registers
mu.reg_write(UC_X86_REG_ESP, stack_addr + stack_size / 2)
# intercept invalid memory events
mu.hook_add(UC_HOOK_MEM_READ_UNMAPPED | UC_HOOK_MEM_WRITE_UNMAPPED, self.hook_mem_invalid)
# build IAT table
iat_addr = 0x10000
e = self.assembler(iat_addr, 'mov eax, 1;int 0xff;ret 0xffff')
iat_size_adj = pe.adjust_SectionAlignment(len(self.import_addrs) * len(e) + pe.OPTIONAL_HEADER.SectionAlignment,
pe.OPTIONAL_HEADER.SectionAlignment, pe.OPTIONAL_HEADER.FileAlignment)
mu.mem_map(iat_addr, iat_size_adj)
for i in range(len(self.import_addrs)):
_, iat_entry, _, nparam = self.import_addrs[i]
func_addr = iat_addr + i * len(e)
if nparam > 1:
c = self.assembler(func_addr, 'mov eax, %x;int 0xff;ret %x' % (i, nparam))
else:
c = self.assembler(func_addr, 'mov eax, %x;int 0xff;ret' % i)
mu.mem_write(func_addr, c)
mu.mem_write(iat_entry, struct.pack(' # handle interrupt ourselves
mu.hook_add(UC_HOOK_INTR, self.hook_intr)
self.log('[+] Emulate machine code')
mu.emu_start(begin_addr, end_addr)
decoded_memory = mu.mem_read(pe.OPTIONAL_HEADER.ImageBase, pe.OPTIONAL_HEADER.SizeOfImage)
return decoded_memory

@staticmethod
def check_sality(code):
signature = [(0,
'\xE8\x00\x00\x00\x00\x5D\x8B\xC5\x81\xED\x05\x10\x40\x00\x8A\x9D\x73\x27\x40\x00\x84\xDB\x74\x13\x81\xC4'),
(0x23,
'\x89\x85\x54\x12\x40\x00\xEB\x19\xC7\x85\x4D\x14\x40\x00\x22\x22\x22\x22\xC7\x85\x3A\x14\x40\x00\x33\x33\x33\x33\xE9\x82\x00\x00\x00\x3
for offset, s in signature:
if s != code[offset:offset + len(s)]:
return False
return True

# callback for tracing instructions
# noinspection PyUnusedLocal
def hook_code(self, uc, address, size, user_data):
# I expect 'retn'
if size != 1:
return
if uc.mem_read(address, size) != '\xc3':
return
```

```
esp = uc.reg_read(UC_X86_REG_ESP)
sality_entrypoint = struct.unpack(' code = uc.mem_read(sality_entrypoint, 0x100)
if not self.check_sality(code):
return
self.detected = True
uc.emu_stop()

# noinspection PyBroadException
def extract(self):
if self.sample is None:
return
self.log("[+] Parse PE File")
try:
self.sample.seek(0)
except:
pass
pe = pefile.PE(data=self.sample.read(), fast_load=True)
self.log("[+] Initialize emulator in X86-32bit mode")
mu = Uc(UC_ARCH_X86, UC_MODE_32)
# map memory for this emulation
mu.mem_map(pe.OPTIONAL_HEADER.ImageBase, pe.OPTIONAL_HEADER.SizeOfImage)
# stack
stack_addr = 0x1000
stack_size = 0x4000
mu.mem_map(stack_addr, stack_size)
# write machine code to be emulated to memory
mu.mem_write(pe.OPTIONAL_HEADER.ImageBase, pe.get_memory_mapped_image())
# initialize machine registers
mu.reg_write(UC_X86_REG_ESP, stack_addr + stack_size / 2)
# tracing all instructions with customized callback
mu.hook_add(UC_HOOK_CODE, self.hook_code)
# intercept invalid memory events
mu.hook_add(UC_HOOK_MEM_READ_UNMAPPED | UC_HOOK_MEM_WRITE_UNMAPPED, self.hook_mem_invalid)
self.log('[+] Emulate machine code')
begin_addr = pe.OPTIONAL_HEADER.ImageBase + pe.OPTIONAL_HEADER.AddressOfEntryPoint
end_addr = pe.OPTIONAL_HEADER.ImageBase + pe.OPTIONAL_HEADER.SizeOfImage
try:
mu.emu_start(begin_addr, end_addr)
except Exception as e:
self.log('[-] Emulator error: %s' % e)
return
if not self.detected:
self.log('[-] Sality not found')
return
self.log("[+] Find Sality section")
sality_section_addr = None
eip_rva = mu.reg_read(UC_X86_REG_EIP) - pe.OPTIONAL_HEADER.ImageBase
for section in pe.sections:
va_adj = pe.adjust_SectionAlignment(section.VirtualAddress, pe.OPTIONAL_HEADER.SectionAlignment,
pe.OPTIONAL_HEADER.FileAlignment)
if va_adj <= eip_rva < va_adj + section.Misc_VirtualSize:
sality_section_addr = va_adj
break
if sality_section_addr is None:
self.log("[-] Sality section not found")
return
mapped_memory = str(mu.mem_read(pe.OPTIONAL_HEADER.ImageBase + sality_section_addr,
pe.OPTIONAL_HEADER.SizeOfImage - sality_section_addr))
self.detect_control_server(mapped_memory)
for m in re.finditer('MZ', mapped_memory):
sality_dll = mapped_memory[m.start():]
decoded_sality_dll = self.emulate_sality_dll(sality_dll)
if decoded_sality_dll is None:
continue
```

```
self.detect_control_server(decoded_sality_dll)
if self.output is not None:
self.output.write(decoded_sality_dll)
self.log("[+] Write Sality DLL to file successfully")
self.log("[+] Analyze Sality DLL successfully")

def detect_control_server(self, memory):
# detect URL
urls = re.findall('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+', memory)
for _ in urls:
self.control_server.append(str(_))


def get_args():
"""This function parses and return arguments passed in"""
# Assign description to the help doc
parser = argparse.ArgumentParser(description='Script extracts URLs of Win32-Sality variants from a given file.')
# Add arguments
parser.add_argument('-z', '--zip', action='store_true')
parser.add_argument('-p', '--password', type=str, help='Password to open zip file', required=False,
default=None)
parser.add_argument('-n', '--name', type=str, help='File name in zip file', required=False, default=None)
parser.add_argument('-d', '--dump', type=str, help='Dump sality DLL to file', required=False, default=None)
parser.add_argument('file', nargs='?')
# Array for all arguments passed to script
args = parser.parse_args()
file_name = None
if args.file is not None and len(args.file) > 0:
file_name = args.file
# Return all variable values
return file_name, args.zip, args.password, args.name, args.dump


def main():
# Match return values from get_args()
# and assign to their respective variables
z = None
file_name, is_zip, password, name, dump = get_args()
if file_name is None:
print "Enter file name"
return
if is_zip:
from zipfile import ZipFile

z = ZipFile(file_name)
f = z.open(name, 'r', password)
else:
f = open(file_name, 'rb')
if dump is not None:
d = open(dump, 'wb')
else:
d = None
sd = SalityExtractor(f, d)
sd.extract()
if len(sd.control_server) > 0:
print sd.control_server
else:
print 'Found nothing'
if f is not None:
f.close()
if z is not None:
z.close()
if d is not None:
d.close()
```

```
if __name__ == '__main__':
main()
```