MONTANA STATE UNIVERSITY

2023.11.07

Malicious Code Analysis

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

Overview

01 Loader

02 Simulation Procedures

03 Solver Engine

04 Simulate State

05 Simulation Managers

Part One

01

Loader

Arch Information

```
import angr
import archinfo
proj = angr.Project('C:\\users\\defaultuser0.DESKTOP-931HL80\\Downloads\\project1-1\\Stardust.exe')

"""
arch information
"""


print(proj.arch)
print(proj.entry)
print(proj.filename)
print(proj.arch.bits)
```

Loader

```
“”””
loader
“”””
print(proj.loader)
print(proj.loader.shared_objects)
print(proj.loader.min_addr)
print(proj.loader.max_addr)
print(proj.loader.main_object.execstack) # sample query: does this binary have an executable stack?
print(proj.loader.main_object.pic)  #sample query: is this binary position-independent?

print(proj.loader.all_objects)
print(proj.loader.all_pe_objects)
```

Loader

```
“”””
Here’s the “externs object”, which we use to provide addresses for unresolved imports and angr internals
“”””
print(proj.loader.extern_object)

#This object is used to provide addresses for emulated syscalls
print(proj.loader.find_object_containing(0x400000))
print(proj.loader.main_object.entry)
print(proj.loader.main_object.min_addr)
print(proj.loader.main_object.max_addr)
```

Address

- **rebased_addr**
  - It is an actual address in the global address space.
- **linked_addr**
  - It is an address relative to the prelinked base of the binary (the address in the table).
- **relative_addr**
  - It is an address relative to the object base.

Loader

```
print(proj.loader.main_object.segments)
print(proj.loader.main_object.sections)
print(proj.loader.main_object.find_segment_containing(proj.loader.main_object.entry))
print(proj.loader.main_object.find_section_containing(proj.loader.main_object.entry))
print(proj.loader.main_object.imports['CloseHandle'])#Get the import address for a
symbol
print("rebased_addr", proj.loader.main_object.imports['CloseHandle'].rebased_addr)
CloseHandle = proj.loader.find_symbol('CloseHandle')
print(closeHandle)
print(closeHandle.name)
print(closeHandle.owner)
print(hex(closeHandle.rebased_addr))
print(hex(closeHandle.linked_addr))
print(hex(closeHandle.relative_addr))
print(hex(proj.loader.main_object.linked_base))
print(hex(proj.loader.main_object.mapped_base))
```

Loader

```
print(closeHandle.is_export)
print(closeHandle.is_import)
""""
On loader, the method is find_symbol because it performs a search operation to find the symbol.
On an individual object, the method is get_symbol because there can only be one symbol with a given name.
""""
main_symbols = proj.loader.main_object.symbols
print(main_symbols)
main_clHandle = proj.loader.main_object.imports["CloseHandle"]
print("main_clHandle is export?", main_clHandle.symbol.is_export)
print("main_clHandle is import?", main_clHandle.symbol.is_import)
print(main_clHandle.symbol.resolvedby)

print(proj.loader.main_object.imports)
print(proj.loader.main_object.relocs)
print(proj.loader.shared_objects['kernel32.dll'].imports)
```

Part Two

02

Simulation Procedures

Simulation Procedures

```
“”””
Simulation Procedures
“”””
stub_func = angr.SIM_PROCEDURES[‘stubs’][‘ReturnUnconstrained’] #this is a Class
proj.hook(0x10000, stub_func()) # hook with an instance of the class

print(proj.is_hooked(0x10000))
print(proj.hooked_by(0x10000))
print(proj.unhook(0x10000))

@proj.hook(0x20000, length=5)
def my_hook(state):
	state.regs.rax = 1
print(proj.is_hooked(0x20000))
```

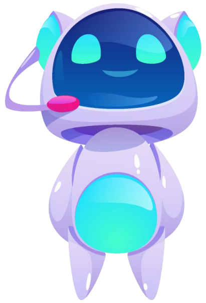
Part Three

03

Solver Engine

Solver Engine

```
“”””
Solver Engine
“”””
state = proj.factory.entry_state()
one =  state.solver.BVV(1, 64)
print(one)
one_hundred = state.solver.BVV(100, 64)
print(one_hundred)
weird_nine = state.solver.BVV(9, 27)
print(weird_nine)
print(one+one_hundred)
print(one_hundred + 0x100)
print(one_hundred - one*200)
weird_nine.zero_extend(64-27)
print(one + weird_nine.zero_extend(64-27))
```



Solver Engine

```
x = state.solver.BVS("x", 64)
y = state.solver.BVS("y", 64)
print(x, y)
print(x + one)
print((x+one)/2)
print(x - y)
tree = (x + 1)/ (y + 2)
print(tree)
print(tree.op)
print(tree.args)
print(tree.args[0].op)
print(tree.args[0].args)
print(tree.args[0].args[1].op)
print(tree.args[0].args[1].args)
```

Solver Engine

```
print(x == 1)
print(x==one)
print(x > 2)
print(x+y == one_hundred+5)
print(one_hundred > 5)
print(one_hundred > -5)
yes = one == 1
no = one == 2
maybe = x==y

print(state.solver.is_true(yes))
print(state.solver.is_false(yes))

print(state.solver.is_true(no))
print(state.solver.is_false(no))

print(state.solver.is_true(maybe))
print(state.solver.is_false(maybe))
```

Solver Engine

```
state.solver.add(x > y)
state.solver.add(y > 2)
state.solver.add(10 > x)
print(state.solver.eval(x))

#get a fresh state without constraints
state = proj.factory.entry_state()
input = state.solver.BVS('input', 64)
operation = (((input + 4)*3)>>1) + input
output = 200
state.solver.add(operation ==output)
print(state.solver.eval(input < 2**32))
print(state.satisfiable())
```

Solver Engine

```
state = proj.factory.entry_state()
state.solver.add(x-y>=4)
state.solver.add(y > 0)
print(state.solver.eval(x))
print(state.solver.eval(y))
print(state.solver.eval(x+y))
```

Solver Engine-floating point numbers

```
“”””
Floating point numbers
“”””
state = proj.factory.entry_state()
a = state.solver.FPV(3.2, state.solver.fp.FSORT_DOUBLE)
print(a)
b = state.solver.FPS(‘b’, state.solver.fp.FSORT_DOUBLE)
print(b)
print(a+b)
print(a+4.4)
print(b+2<0)
state.solver.add(b+2<0)
state.solver.add(b+2>-1)
print(state.solver.eval(b))
```

Solver Engine

```
print(a.raw_to_bv())
print(b.raw_to_bv())
print(state.solver.BVV(0, 64).raw_to_fp())
print(state.solver.BVS(‘x’, 64).raw_to_fp())
print(a.val_to_bv(12))
print(a.val_to_bv(12).val_to_fp(state.solver.fp.FSORT_FLOAT))
```

Part Four

04

Simulate State

SimState

```
"""
SimState
"""
state = proj.factory.entry_state()
print(state.regs.rip)
print(state.regs.eax)

#interpret the memory at the entry point as a C int
print(state.mem[proj.entry].int.resolved)

bv=state.solver.BVV(0x1234, 32)
print(bv)
ev = state.solver.eval(bv)
print(ev)
state.regs.esi = state.solver.BVV(3, 64)
print(state.regs.esi)
state.mem[0x1000].long = 4
print(state.mem[0x1000].long.resolved)
```

SimState

```
#copy rsp to rbp
state.regs.rbp = state.regs.rsp
#store rdx to memory at 0x1000
state.mem[0x1000].uint64_t = state.regs.rdx
#dereference rbp
state.regs.rbp = state.mem[state.regs.rbp].uint64_t.resolved
#add rax, qword ptr [rsp+8]
state.regs.rax+=state.mem[state.regs.rsp+8].uint64_t.resolved
```

SimState

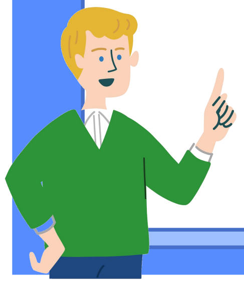
```
state = proj.factory.entry_state(stdin=angr.SimFile)
while True:
        succ = state.step()
        if len(succ.successors)==2:
                break
        state = succ.successors[0]

state1, state2 = succ.successors
print(state1)
print(state2)
input_data = state1.posix.stdin.load(0, state1.posix.stdin.size)
print(input_data, state1.posix.stdin.size)
print("input_data", state1.solver.eval(input_data, cast_to=bytes))
```

SimState

```
#store and load can also be used for registers
s = proj.factory.blank_state()
s.memory.store(0x4000, s.solver.BVV(0x0123456789abcdef0123456789abcdef, 128))
print(s.memory.load(0x4004, 6))
print(s.memory.load(0x4000, 4, endness=archinfo.Endness.LE))
```

SimState-practice

- Use load and store to read and write values in register rax, rbx, rcx, and rdx.
- Then use solver.eval to print out their values.



SimState

```
“”””
Example: enable lazy solves, an option that causes state satisfiability to be checked as infrequently as possible.
This change to the settings will be propagated to all successor states created from this state after this line.
“”””
s.options.add(angr.options.LAZY_SOLVES)

#create a new state with solves enabled
s = proj.factory.entry_state(add_options={angr.options.LAZY_SOLVES})

#Create a new state without simplification options enabled
s = proj.factory.entry_state(remove_options = angr.options.simplification)
```

SimState

```
#Create an angr state at the entry point
state = proj.factory.entry_state()
successors = state.step()
#Iterate over the state’s history and print each history node
for succ_state in successors:
	#check the length of the state’s history again
	print(len(succ_state.history))
	node = succ_state.history
	count = 0
	while node:
		print(f’History node :”)
		count +=1
		print(node)
		node = node.parent
	for addr in succ_state.history.bbl_addrs:
		print(addr)
	for kind in succ_state.history.jumpkinds:
		print(kind)
	for guard in succ_state.history.jump_guards:
		print(guard)
	print(count)
```

SimState

```
“”””

copy and merge
“”“‘

s = proj.factory.blank_state()
s1 = s.copy()
s2 = s.copy()
s1.mem[0x1000].uint32_t = 0x41414141
s2.mem[0x1000].uint32_t = 0x42424242
“”””

merge will return a tuple. the first element is  the merged state. the second element is a symbolic variable describing a state flag. the third element is a boolean describing whether any merging was done
(s_merged, m, anything_merged) = s1.merge(s2)
print(s_merged)
print(m)
print(anything_merged)
aaaa_or_bbbb = s_merged.mem[0x1000].uint32_t
print(“aaaa_or_bbbb”, aaaa_or_bbbb)
```

Part Five

05

Simulation Managers

Simulation Managers

```
""""
Simulation Managers
""""
simgr = proj.factory.simulation_manager(state)
print(simgr.active)
print(simgr.active[0])
simgr.step()
print(simgr.active)
print(simgr.active[0].regs.rip)
print(state.regs.rip)

state = proj.factory.entry_state()
simgr = proj.factory.simgr(state)
print(simgr.active)
while len(simgr.active)==1
        simgr.step()
print(simgr.active)
simgr.run()
print(simgr)
```

Stash types

Stash	Description
active	This stash contains the states that will be stepped by default, unless an alternate stash is specified.
deadended	A state goes to the deadended stash when it cannot continue the execution for some reason, including no more valid instructions, unsat state of all of its successors, or an invalid instruction pointer.
pruned	When using LAZY_SOLVES,states are not checked for satisfiability unless absolutely necessary. When a state is found to be unsat in the presence of LAZY_SOLVES, the state hierarchy is traversed to identify when, in its history, it initially became unsat. All states that are descendants of that point(which will also be unsat, since a state cannot become un-unsat) are pruned and put in this stash.
unconstrained	If the save_unconstrained option is provided to the SimulationManager constructor, states that are determined to be unconstrained (i.e.,with the instruction pointer controlled by user data or some other source of symbolic data) are placed here.
unsat	If the save_unsat option is provided to the SimulationManager constructor,states that are determined to be unsatisfiable(i.e.,they have constraints that are contradictory, like the input having to be both "AAAA" and "BBBB" at the same time) are placed here.

Exploration Techniques

- **DFS:** Depth first search, as mentioned earlier. Keeps only one state active at once, putting the rest in the deferred stash until it deadends or errors.

- **Explorer:** This technique implements the .explore() functionality, allowing you to search for and avoid addresses.

- **LengthLimiter:** Puts a cap on the maximum length of the path a state goes through.

Exploration Techniques

- **LoopSeer:** Uses a reasonable approximation of loop counting to discard states that appear to be going through a loop too many times, putting them in a spinning stash and pulling them out again if we run out of otherwise viable states.
- **ManualMergepoint:** Marks an address in the program as a merge point, so states that reach that address will be briefly held, and any other states that reach that same point within a timeout will be merged together.
- **MemoryWatcher**: Monitors how much memory is free/available on the system between simgr steps and stops exploration if it gets too low.

Exploration Techniques

- **Spiller:** When there are too many states active, this technique can dump some of them to disk in order to keep memory consumption low.

- **Threading:** Adds thread-level parallelism to the stepping process. This doesn't help much because of Python's global interpreter locks, but if you have a program whose analysis spends a lot of time in angr's native-code dependencies (unicorn, z3, libvex) you can seem some gains.

THE END

Fangtian Zhong
CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineerin
E-mail: fzhong@montana.edu

2023.11.07