Intro to angr and concolic analysis

Foo-Manroot

https://foo-manroot.github.io/

Agenda

- Theory
  - What is symbolic execution?
  - Concrete + Symbolic = Concolic
  - angr: platform and use cases
- (BREAK !)
- Hands-on
  - Sample CTF binaries

What is symbolic execution?

What is symbolic execution?

Different approaches to analyze a program: concrete values

```
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

- Goal: find inputs so that `whatever()` returns "1000" or above
- Approach 1: random (*concrete*) testing.
  - Execute the function with different values and check the output.
  - We can't be 100% sure that we covered all paths.
    - We missed the `if (a == 43)` branch.

Provided value	Output	Return value
`whatever (0);`	"PANIK!	*(fail)*
`whatever (12);`	"kalm"	9
`whatever (-1);`	"PANIK!"	*(fail)*

What is symbolic execution?

Different approaches to analyze a program: symbolic values

```c
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

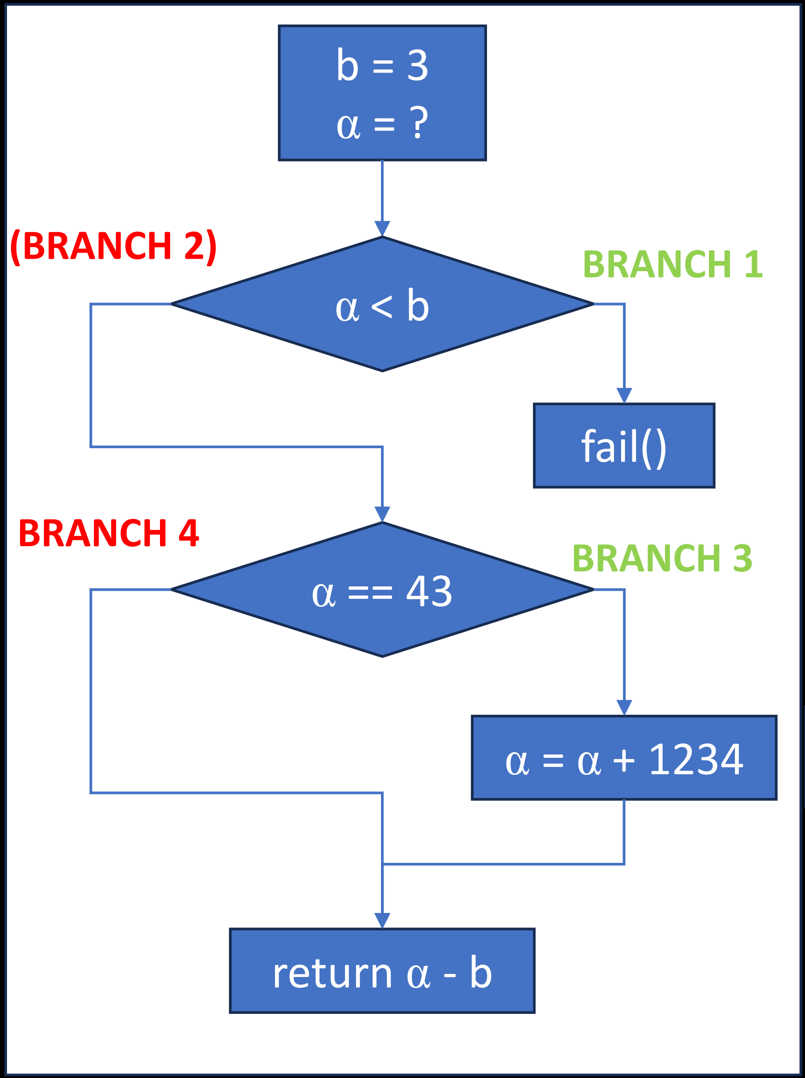
- Goal: find inputs so that `whatever()` returns "1000" or above
- Approach 2: *symbolic* testing.
  1. Replace unknown values (`int a`) by an arbitrary *symbol* (α).
  2. Gather all constraints for each branch.
  3. Solve the equations to obtain possible input values to reach that specific branch.

(or, at least, that's the *very* simplified explanation)

What is symbolic execution?

Get constraints for each branch

```
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```



BR 1

$\alpha < 3$

BR 3

$\alpha >= 3$

$\alpha == 43$

return = $\alpha + 1234 - 3$

BR 4

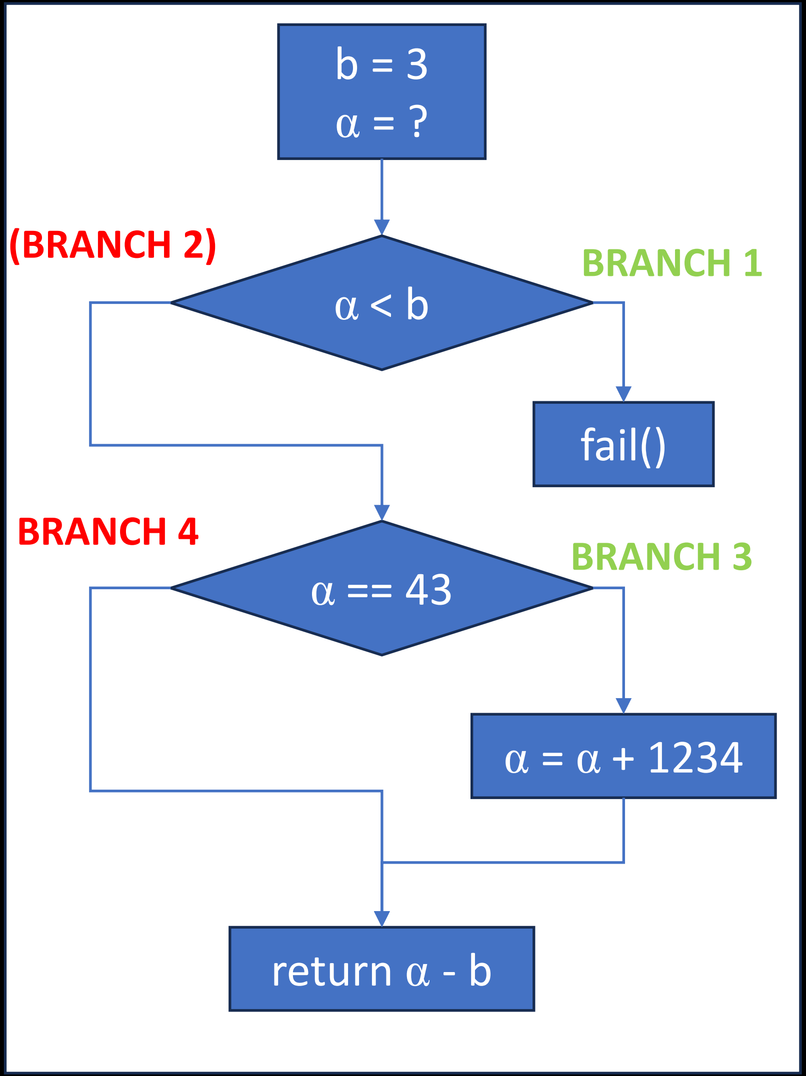
$\alpha > 3$

$\alpha != 43$

return = $\alpha - 3$

What is symbolic execution?

Solve the constraints (SMT solver)



- Example constraints: execute Branch 3

$$\begin{cases} \alpha >= 3 \\ \alpha == 43 \end{cases}$$

  - Solving this equations gives us a concretized value for α : **43**

- Execute Branch 4

$$\begin{cases} \alpha >= 3 \\ \alpha \mathrel{!=} 43 \end{cases}$$

  - Concrete value for α : **17**
  - There are multiple solutions, but we only need one

What is symbolic execution?

Limitations

- Solving the equations (**SMT**) is an **NP-hard** problem.
  - Domain-specific reasoning can be combined with Boolean solvers (SAT) to speed-up execution. Some problems are **still insolvable** (e.g.: ***hashing***)

- Taking all branches means an **explosion in possible states** to keep track of.
  - Search **can be optimised**:
    - Avoid certain paths (e.g.: “finish exploration when reaching `fail()`”)
    - Depth-First search
    - ...

- Modelling **environment**
  - Normal programs open **sockets**, read **files**, allocate **memory**, ...

What is symbolic execution?

Advantages

- **Full coverage** of all branches.
  - We’re exploring all possible branches.
- **Platform-independent.**
  - An ELF can be analysed from a Windows machine.
- **Traceability.**
  - There’s no difficulty reproducing the state: we know the constraints required to trigger it (e.g.: `random_func()` must return `1337` to crash).

Concrete + Symbolic = Concolic

Concolic execution

Overview

- Using concrete and symbolic execution to support each other
  - Concrete execution **avoids** the need to model the **environment** (API calls, file operations, ...)
  - **Symbolic** execution helps to **find concrete values** that will cover all paths
- Advantages:
  - **No more paths explosions** (drawback from symbolic exec.)
  - **Better code coverage** (drawback from concrete exec.)

Concolic execution

Practical example: branch 1

```
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

Concrete values	Symbolic values	Path conditions
a = 0	a = α	-

Concolic execution

Practical example: branch 1

```c
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

Concrete values	Symbolic values	Path conditions
a = 0	$a = \alpha$	-
a = 0	$a = \alpha$	$\alpha < b$

Concolic execution

Practical example

- The concrete execution found a branch and reached its end: `fail()`
- Now, the concolic engine:
    1. Negates one of the path conditions:
        1. $\neg(\alpha < b) \rightarrow (\alpha >= b)$
    2. Asks the SMT solver to found new concrete values
    3. Executes the function again, to follow the other branch

Concolic execution

Practical example: branch 2

```c
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

Concrete values	Symbolic values	Path conditions
a = 123	$a = \alpha$	-

Concolic execution

Practical example: branch 2

```
int whatever (int a)
{
    int b = 3;

    if (a < b)
    {
        printf ("PANIK!");
        fail ();
    }
    else if (a == 43)
    {
        a += 1234;
    }

    printf ("kalm");

    return a - b;
}
```

Concrete values	Symbolic values	Path conditions
a = 123	a = α	-
a = 123	a = α	$\neg(a < b)$

Concolic execution

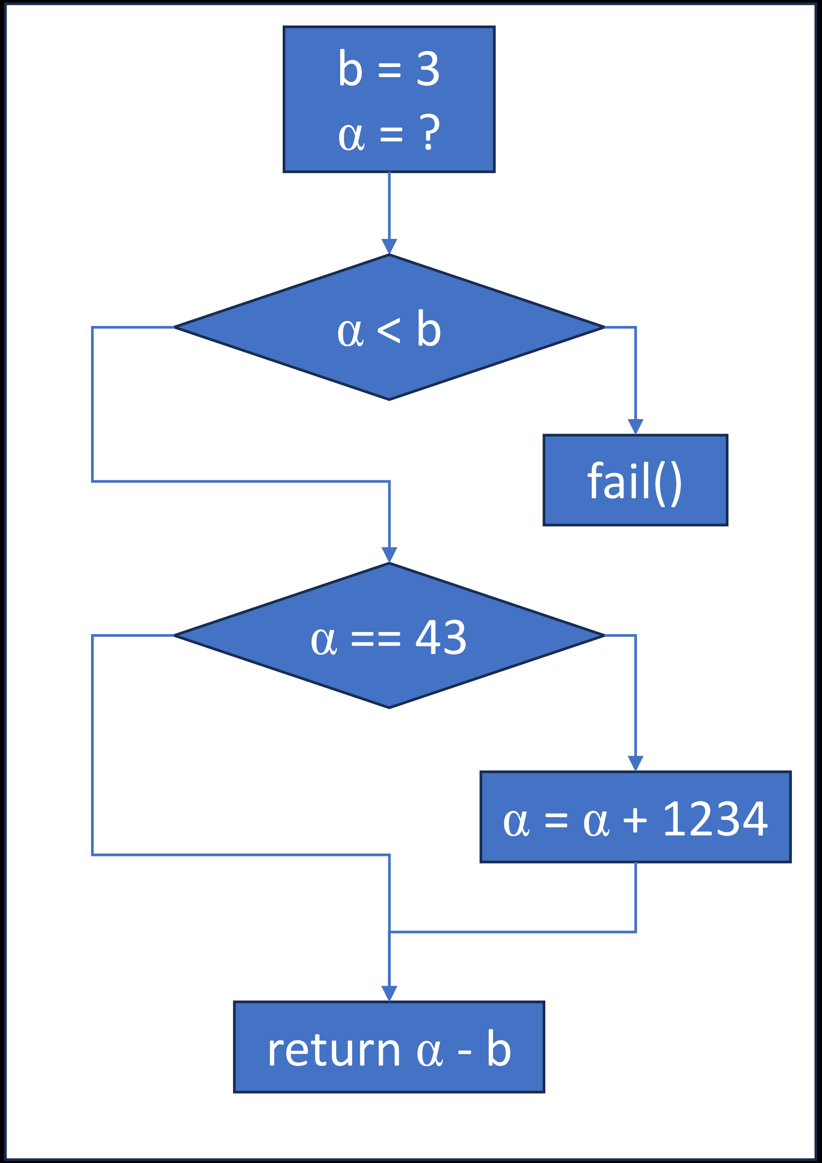
Practical example: branch 2

```c
int whatever (int a)
{
	int b = 3;

	if (a < b)
	{
		printf ("PANIK!");
		fail ();
	}
	else if (a == 43)
	{
		a += 1234;
	}

	printf ("kalm");

	return a - b;
}
```

Concrete values	Symbolic values	Path conditions
a = 123	$a = \alpha$	-
a = 123	$a = \alpha$	$\neg(a < b)$
a = 123	$a = \alpha$	$\neg(a < b) \wedge (a \neq 43)$

Concolic execution

Multiple executions lead to full coverage



Execution 1
Execution 2
Execution 3

angr: platform and use cases

About angr

Scope

- Binary **concolic** analysis platform
- **Emulation** of multiple architectures (arm64, x86, mips, JVM bytecode, ...) and executable formats (PE, ELF, Mach-O, ...)
- **Multi-platform** (all the Python fanboys raise your hand)
- Applications:
  - CFG generation (IDA-like) // https://github.com/angr/angr-management
  - Search for vulnerabilities // https://github.com/angr/angr
  - Exploit generation // http://github.com/shellphish/rex
  - ...

About angr

Quick Start

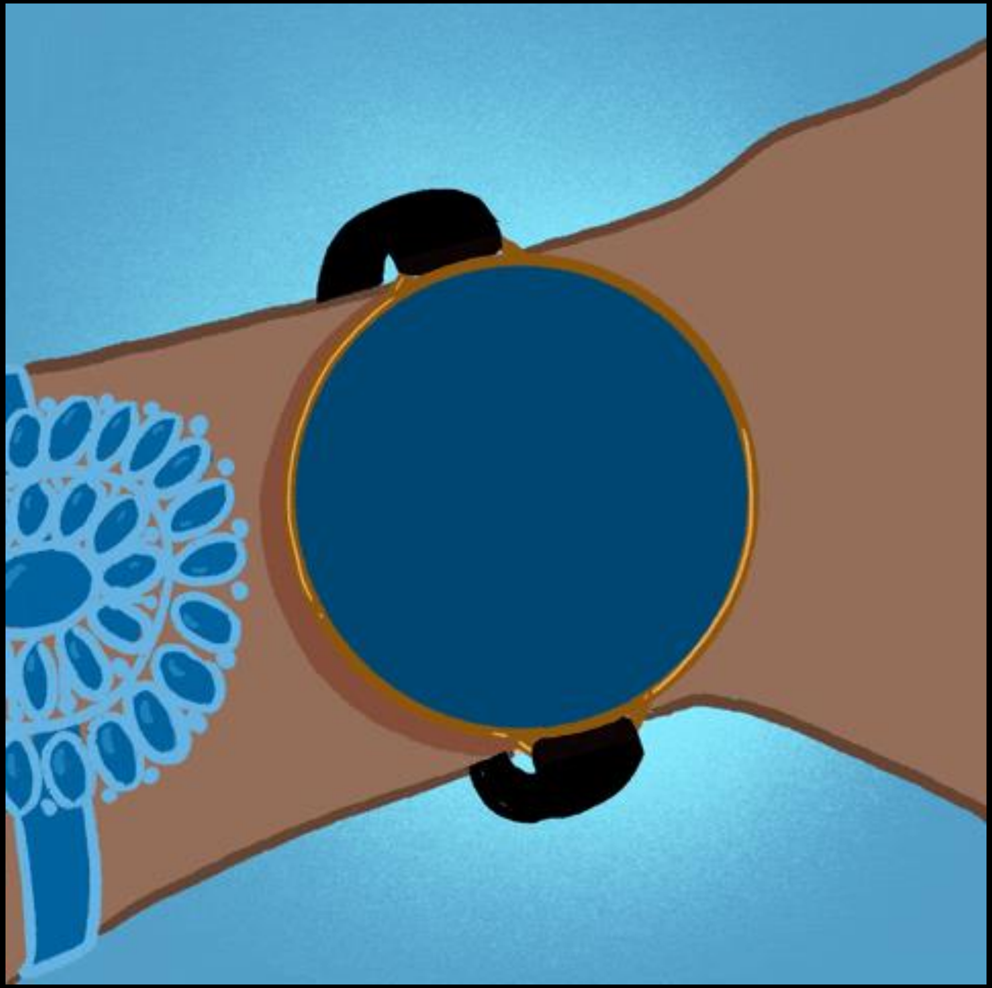
```
C:\ > pip install angr
(...)
C:\ > python
>>> import angr
>>> proj = angr.Project ("fauxware")
>>> proj.arch
<Arch AMD64 (LE)>
>>> proj.loader.main_object
<ELF Object fauxware, maps [0x400000:0x60105f]>


>>> # Create a specific state starting at the entrypoint
>>> state = proj.factory.entry_state ()
>>> state.regs.rip
<BV64 0x400580>
```

```
>> # Constraint solver
>>> x = state.solver.BVS ("x", 64)
>>> y = state.solver.BVS ("y", 64)
>>> state.solver.add (x - y >= 4)
[<Bool x_3_64 - y_4_64 >= 0x4>]
>>> state.solver.add (y > 0)
[<Bool y_4_64 > 0x0>]
>>> state.solver.eval (x) # Concretize a value
3
>>> state.solver.eval (y)
18446744073709551615


>>> # Simulations are controlled by the Simulation Manager
>>> simgr = proj.factory.simulation_manager (state)
>>> simgr.step () #
<SimulationManager with 1 active>
>>> simgr.active[0].regs.rip
<BV64 0x400540>
```

https://github.com/angr/angr-examples/blob/master/examples/fauxware/fauxware

15’ Break

(+ get **angr**, IDA/Ghidra/whatever and angr_ctf ready)



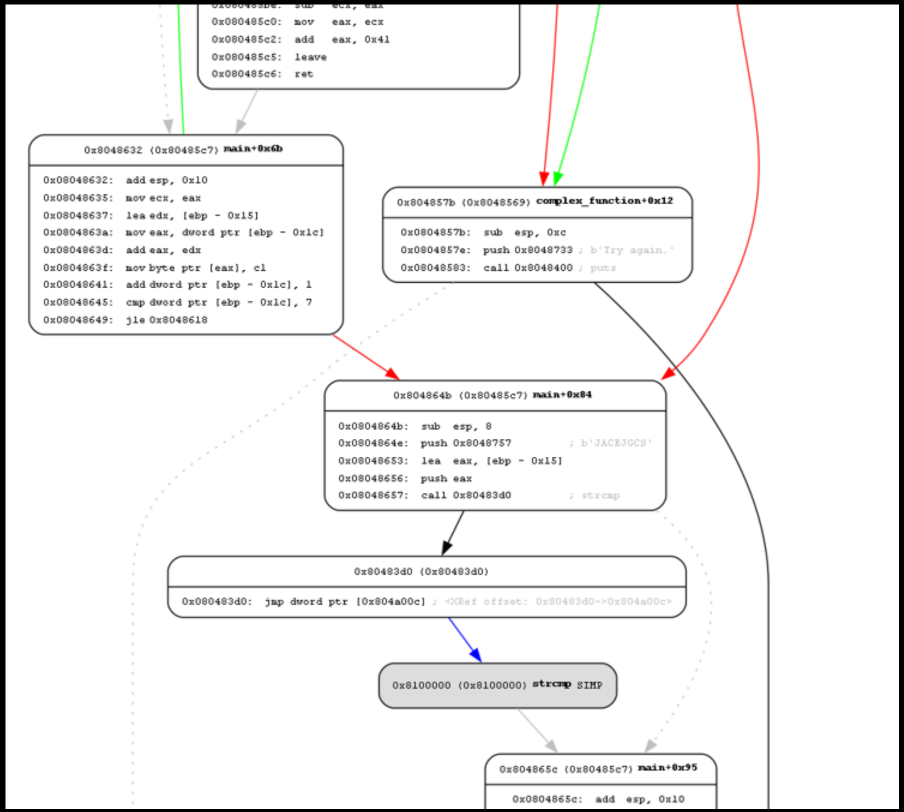
Learning on-the-go with CTFs

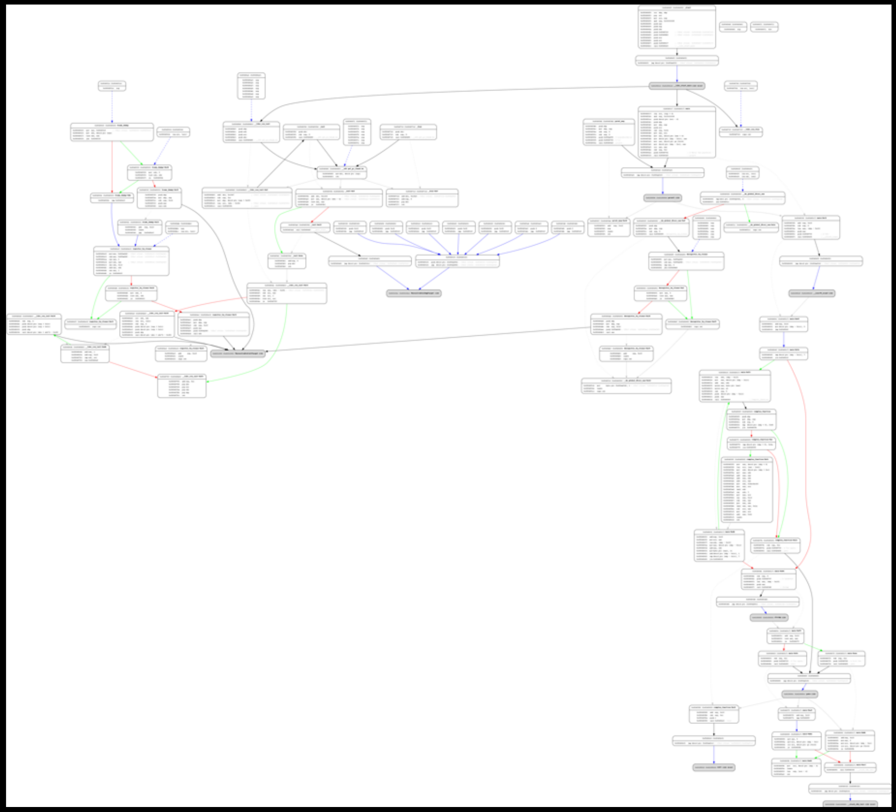
angr CTF

00_angr_find // Option 1: Manually create CFG

```
C:\ > pip install angr-utils
C:\ > python
>>> import angr
>>> from angrutils import *
>>> proj = angr.Project (".\\00_angr_find", load_options={'auto_load_libs': False})
>>> cfg = proj.analyses.CFG ()
>>> plot_cfg (cfg, "ctf_00-CFG", asminst = True)
```

Requires
https://graphviz.org/download/
to paint the CFG

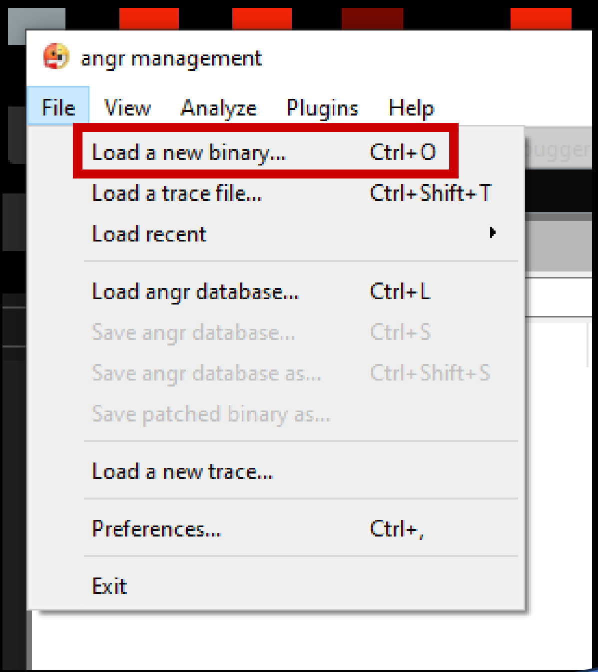




angr CTF

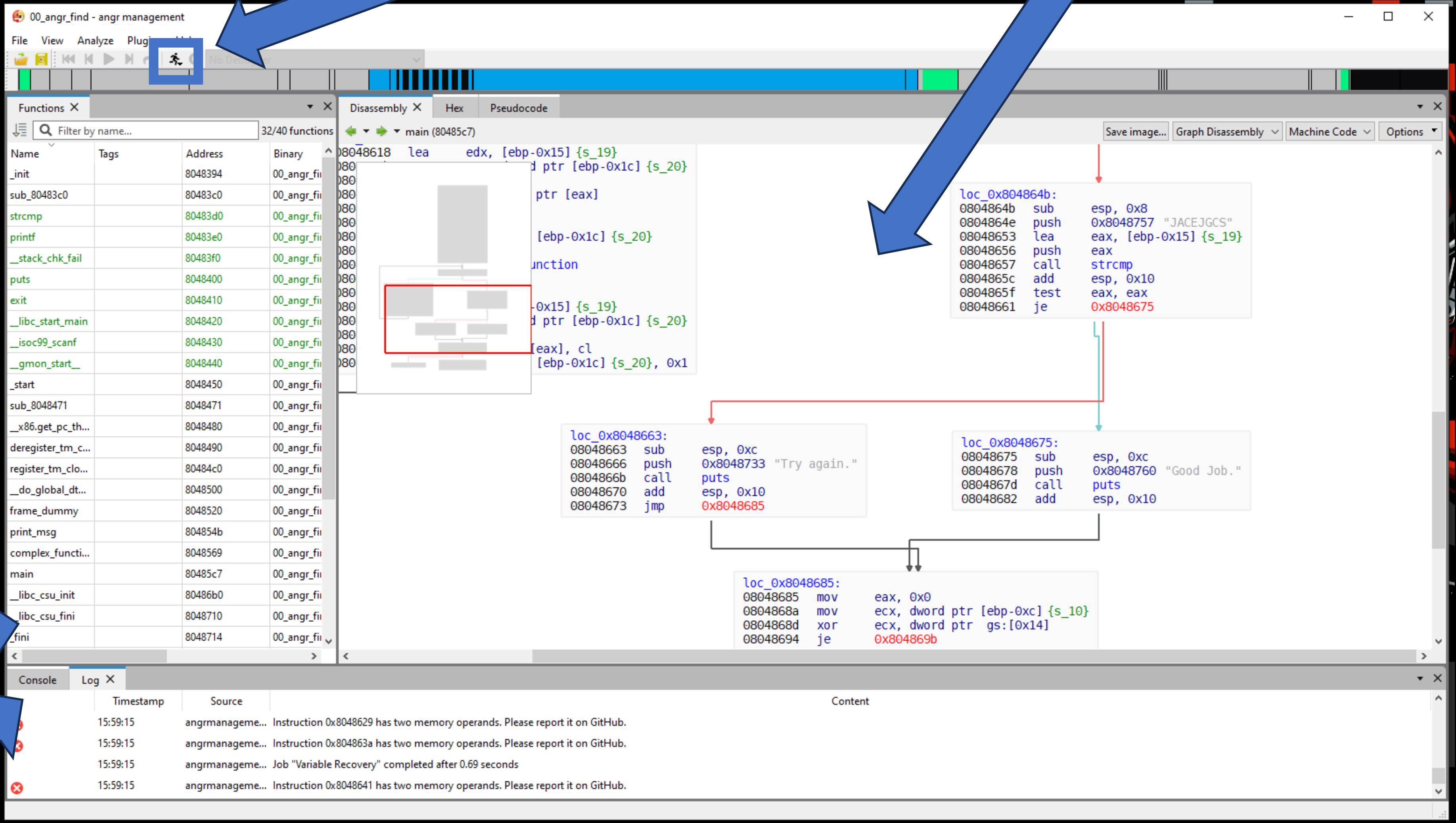
00_angr_find // Option 2: Use angr-management

C:\ > pip install angr-management
C:\ > angr-management



New simulation

CFG

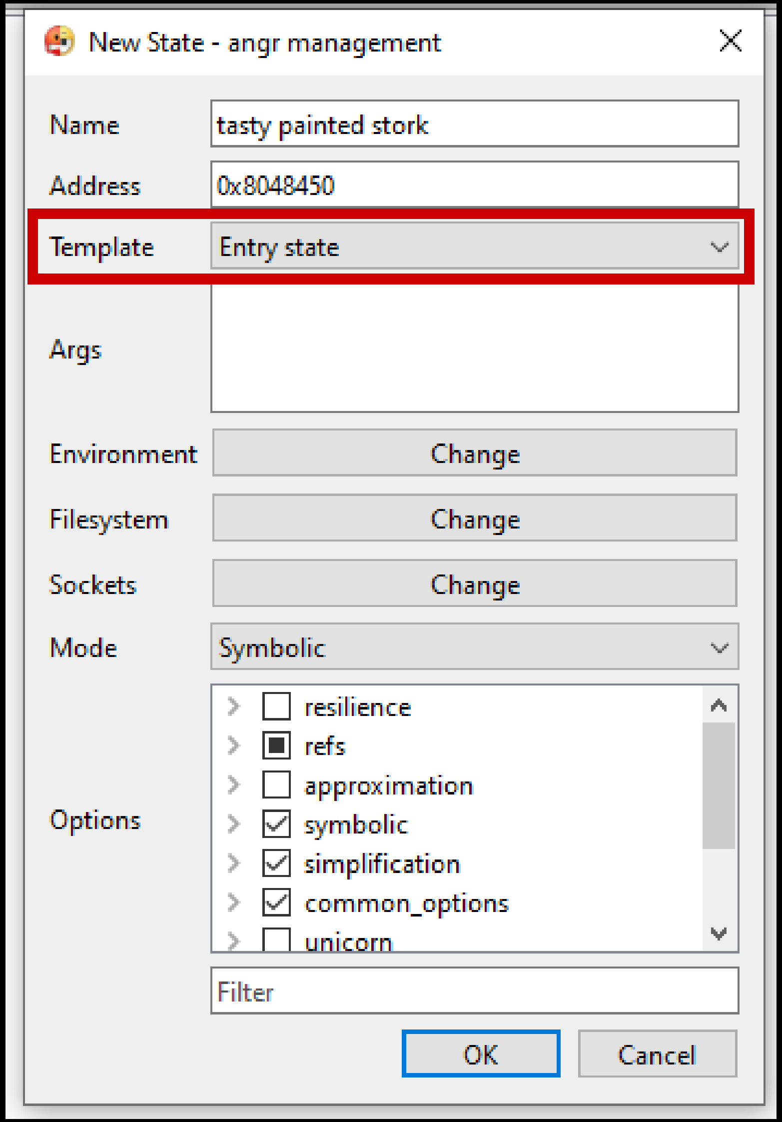


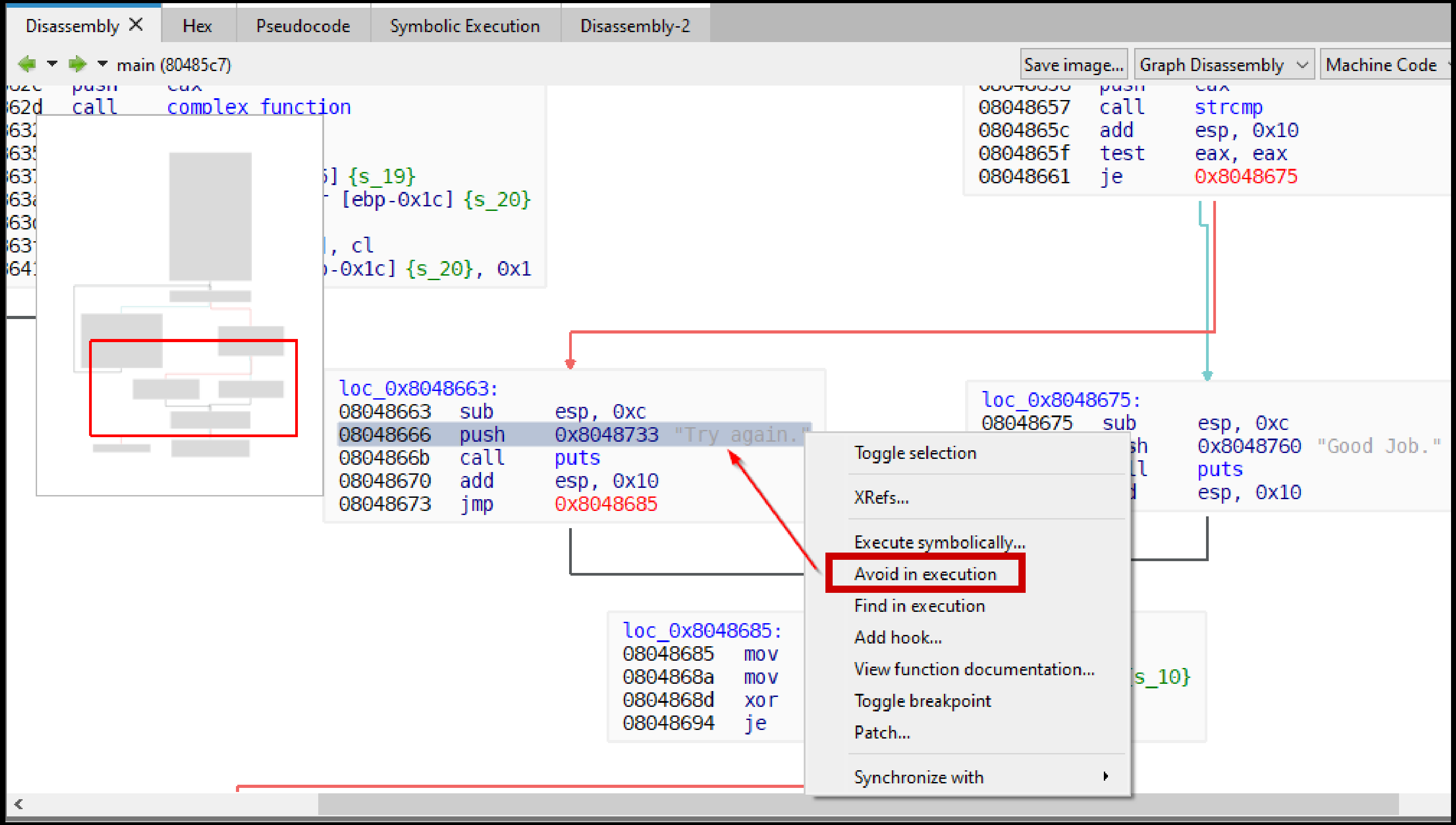
Functions

Console

angr CTF

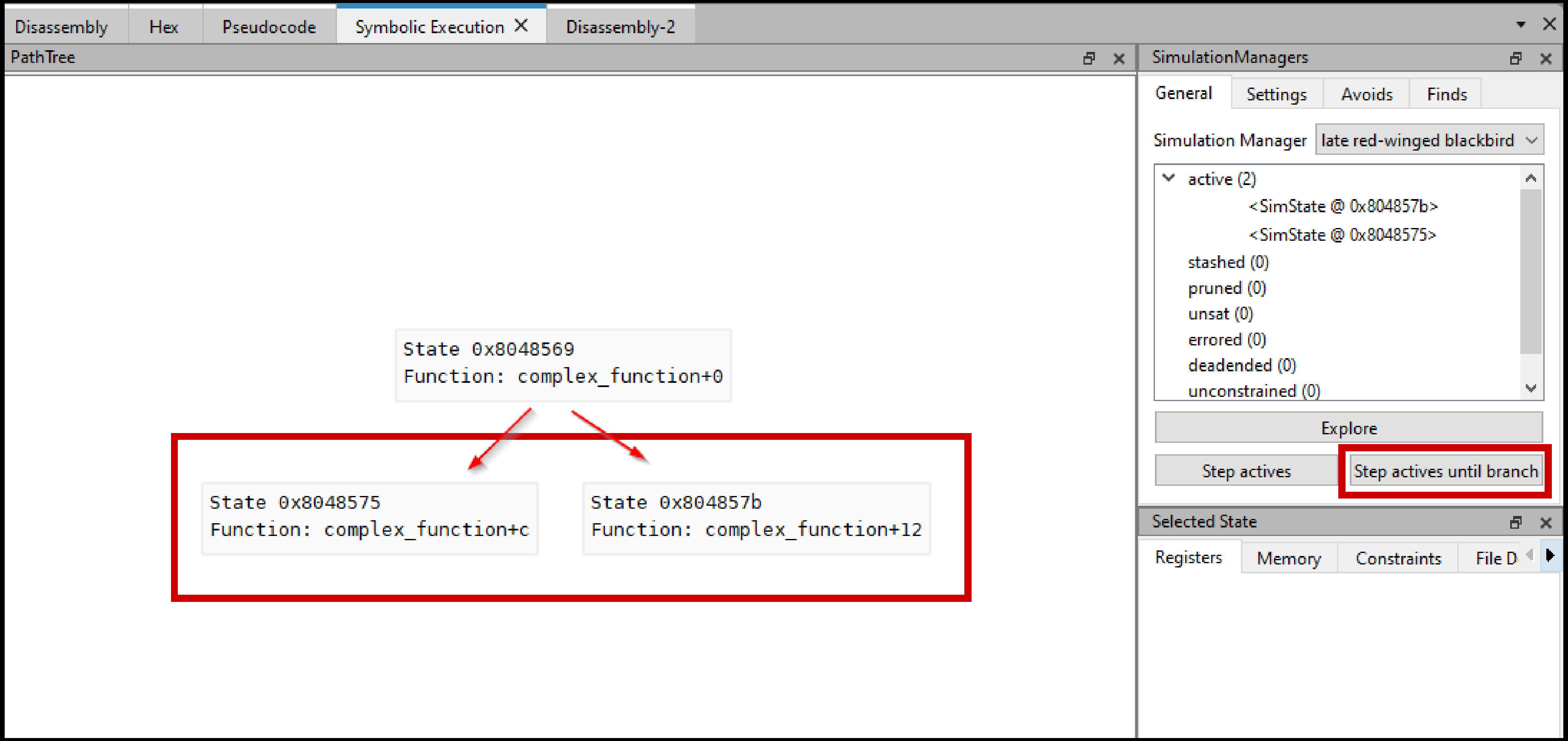
00_angr_find // Option 2: Use angr-management // Create simulation





angr CTF

00_angr_find // Option 2: Use angr-management // Manage simulation

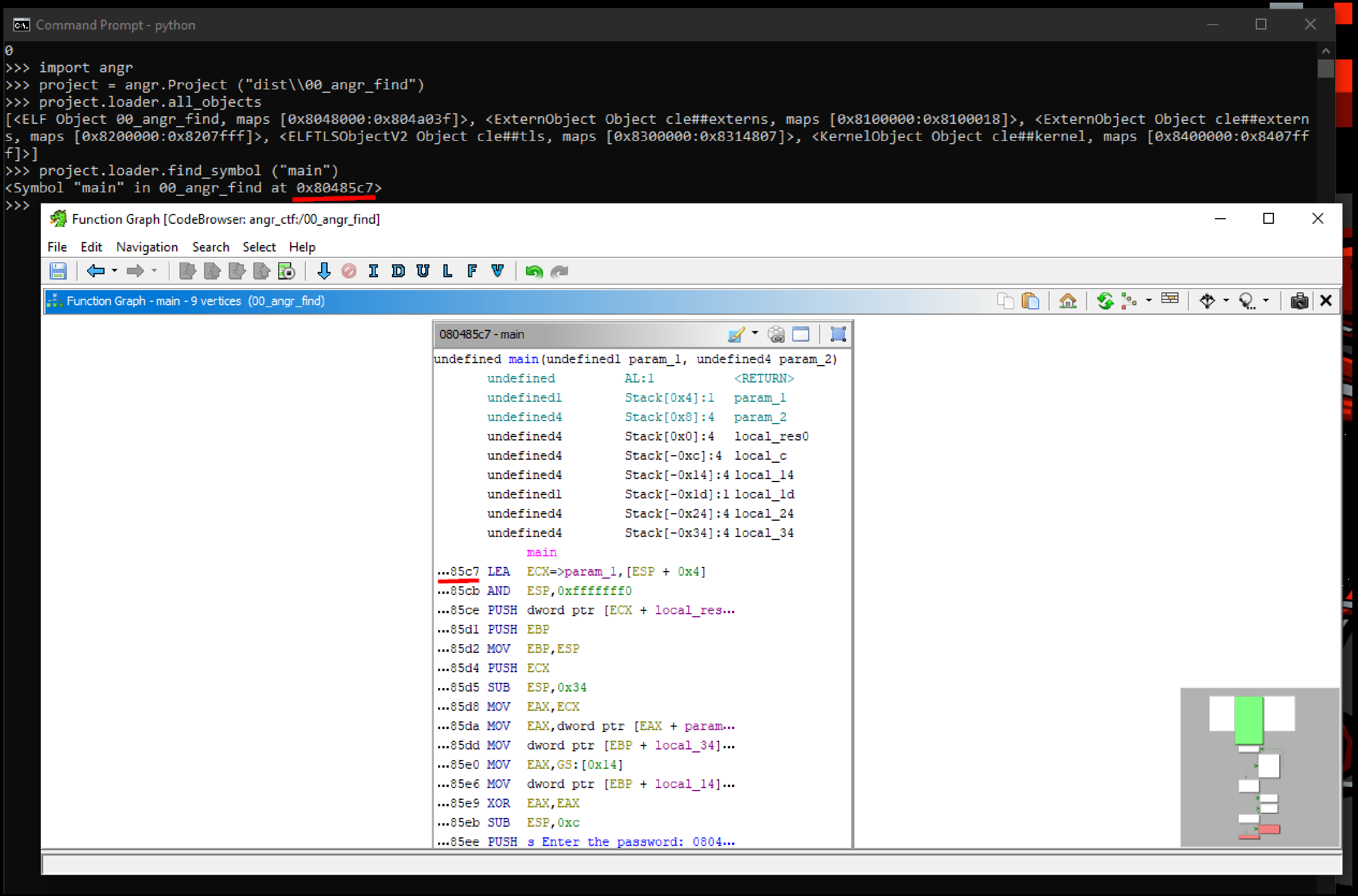


??? It doesn’t work (or I don’t know how to use it)

https://docs.angr.io/en/latest/getting-started/helpwanted.html#angr-management

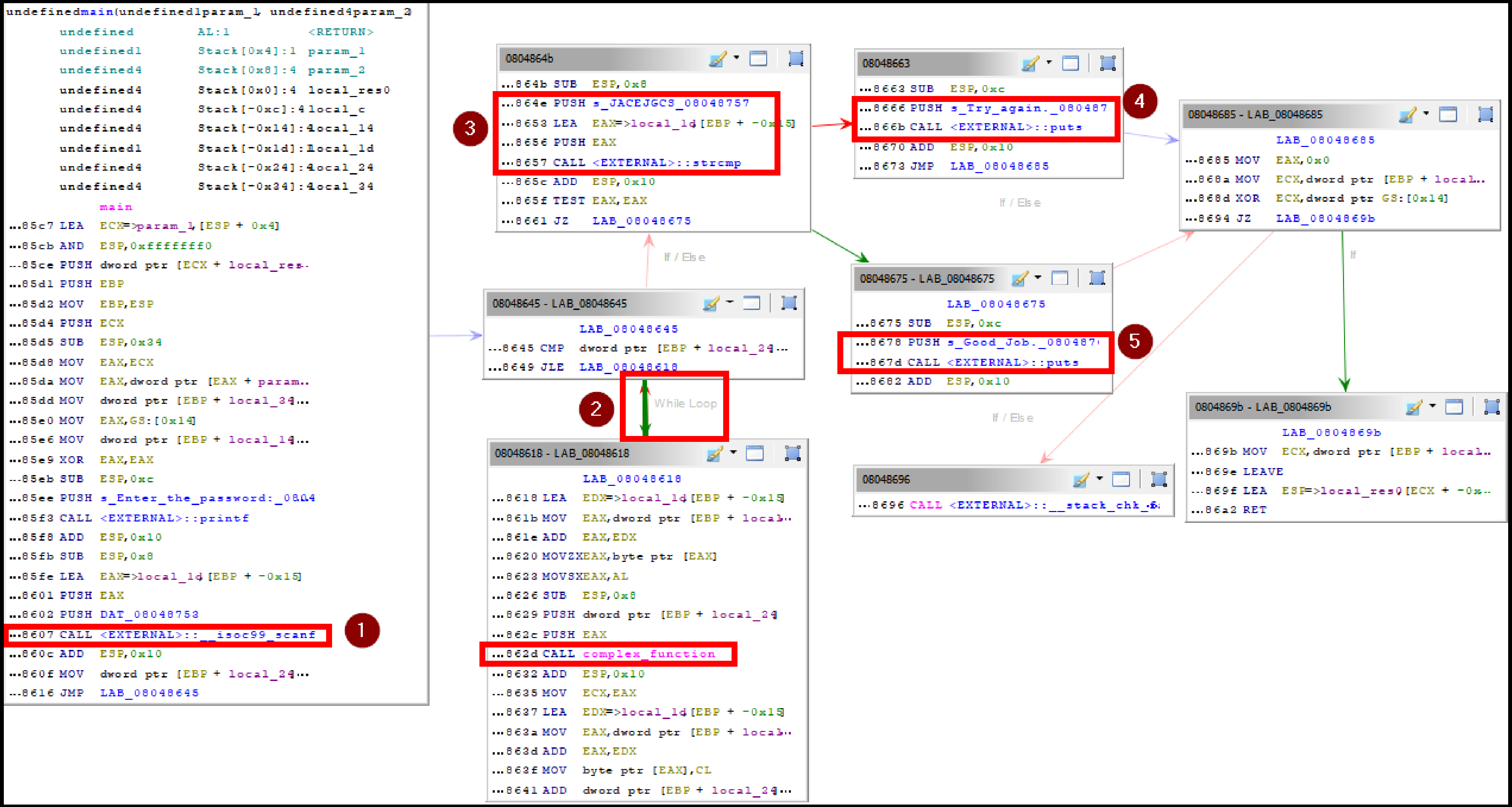
angr CTF

00_angr_find // Option 3: Ghidra + angr CLI



angr CTF

00_angr_find // Option 3: Ghidra + angr CLI



angr CTF

00_angr_find // Option 3: Ghidra + angr CLI // Ghidra analysis

- Important points discovered with Ghidra:
  1. The program gets some **user input** using **`scanf()`**
  2. A loop transforms that input using **`complex_function()`**
  3. The result is compared with the global variable **`s_JACEJGCS`**
  4. If the comparison **fails**, the program prints “**Try again**”
  5. If the comparison **succeeds**, the program prints “**Good job**”

- angr takes care of symbolizing this simple scanf
- We can ask angr to gather the constraints that must be satisfied to print “Good Job”

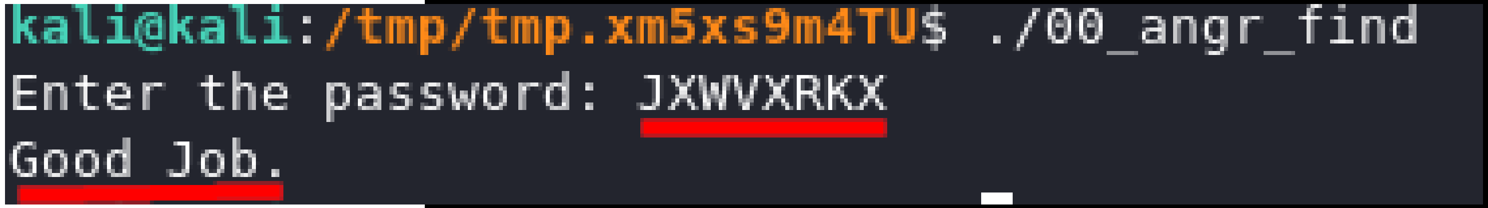
angr CTF

00_angr_find // Option 3: Ghidra + angr CLI // angr modelling and execution

```
>>> initial_state = project.factory.entry_state ()
>>> simulation = project.factory.simgr (initial_state)
>>> target_address = 0x8048675 # Address to print "Good Job"
>>> avoid_address = 0x8048663 # Not needed now, but good to optimise
>>> simulation.explore (find = target_address, avoid = avoid_address)
WARNING  | (...)
<SimulationManager with 16 deadended, 1 found, 1 avoid>
>>> simulation.found
[<SimState @ 0x8048675>]
>>> solution.posix.dumps (0) # STDIN
b'JXWVXRKX'
>>> # Simulations that failed to meet the requirements
>>> simulation.deadended[12].posix.dumps (0)
b'AAAAAB\x00\x00'
>>> simulation.deadended[13].posix.dumps (0)
b'AAAAAA[\x00'
>>> simulation.deadended[14].posix.dumps (0)
b'AAAAAAC\x00'
>>> simulation.deadended[15].posix.dumps (0)
b'AAAAABA['
```

Different stashes depending on the simulation’s state

Solution to this challenge



angr CTF

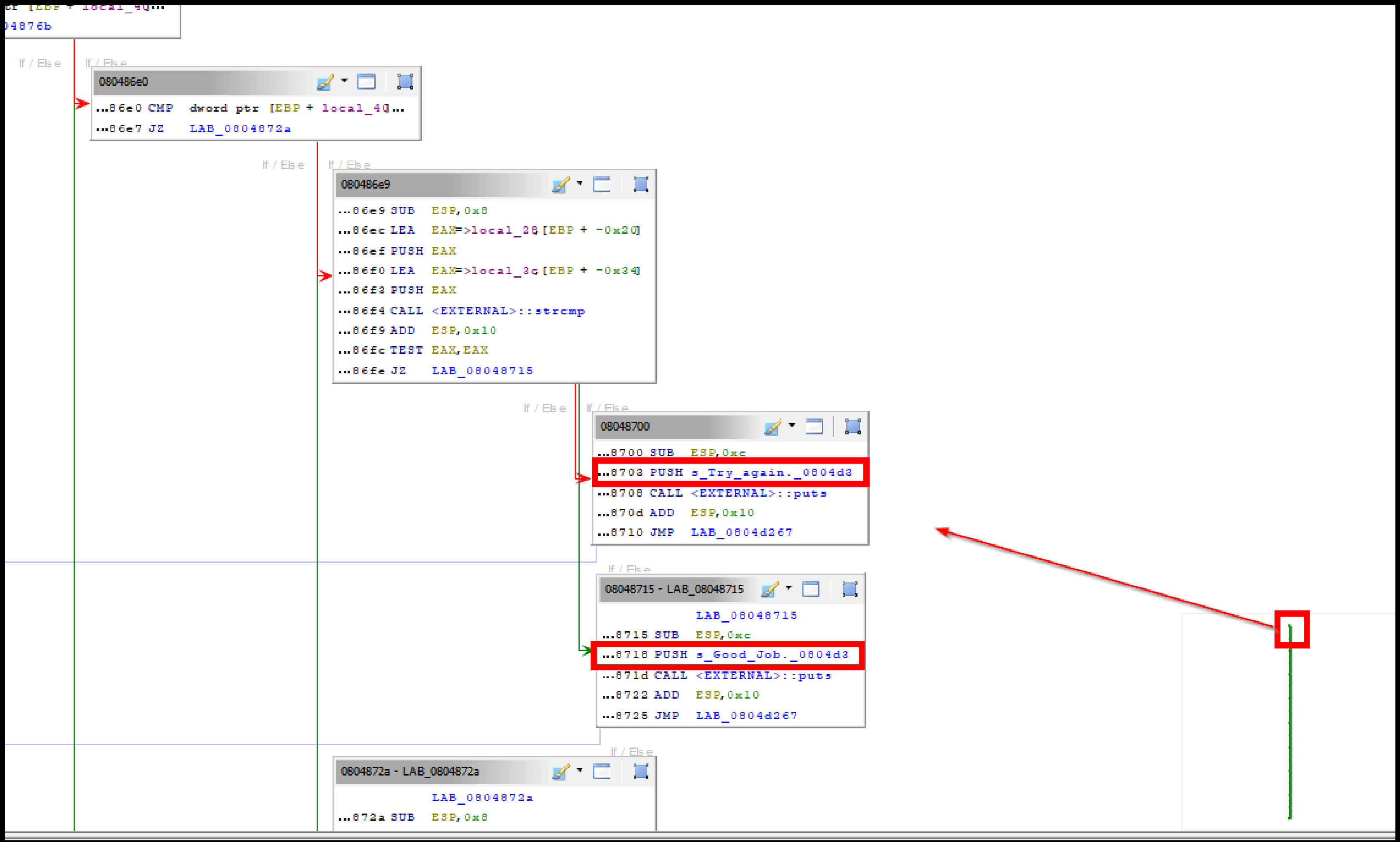
02_angr_find_condition // Ghidra

Next example: 02_angr_find_condition

- Huge if-else tree
- All statements end on:
  - “Good Job”

  or

  - “Try again”



angr CTF

02_angr_find_condition // angr

- angr lets us add conditions to avoid paths, aside from of hardcoded addresses
  - take the current state as a parameter
  - return True (the state fulfills the condition) or False
- We can also use STDIN or STDOUT for these conditions
- QUIZ: What should these functions return?

```
def is_successful (state):
    stdout_output = state.posix.dumps (sys.stdout.fileno ())
    return ???

def should_abort (state):
    stdout_output = state.posix.dumps (sys.stdout.fileno ())
    return ???

simulation.explore (find = is_successful, avoid = should_abort)
```

API docs: https://docs.angr.io/en/latest/api.html#state

angr CTF

02_angr_find_condition // angr // solution

```
>>> initial_state = project.factory.entry_state ()
>>> simulation = project.factory.simgr (initial_state)
>>> def is_successful (state):
...     stdout_output = state.posix.dumps (sys.stdout.fileno ())
...     return b'Good Job.' in stdout_output
>>> def should_abort (state):
...     stdout_output = state.posix.dumps (sys.stdout.fileno ())
...     return b'Try again.' in stdout_output
...
>>> simulation.explore (find = is_successful, avoid = should_abort)
<SimulationManager with 1 found, 17 avoid>
>>> simulation.found[0].posix.dumps (sys.stdin.fileno ())
b'HETOBRCU'
```

angr CTF

03_angr_symbolic_registers // Ghidra

Next example: 03_angr_symbolic_registers

- Complex `scanf()`

```
                     get_user_input                          XREF[4]:     Entry Point(*), main:0804897b
                                                                          08048afc, 08048be0(*)
0804890c 55              PUSH       EBP
0804890d 89 e5           MOV        EBP,ESP
0804890f 83 ec 18        SUB        ESP,0x18
08048912 65 8b 0d        MOV        ECX,dword ptr GS:[0x14]
         14 00 00 00
08048919 89 4d f4        MOV        dword ptr [EBP + local_10],ECX
0804891c 31 c9           XOR        ECX,ECX
0804891e 8d 4d f0        LEA        ECX=>local_14,[EBP + -0x10]
08048921 51              PUSH       ECX
08048922 8d 4d ec        LEA        ECX=>local_18,[EBP + -0x14]
08048925 51              PUSH       ECX
08048926 8d 4d e8        LEA        ECX=>local_1c,[EBP + -0x18]
08048929 51              PUSH       ECX
0804892a 68 93 8a        PUSH       s_%x_%x_%x_08048a93                  = "%x %x %x"
         04 08
0804892f e8 9c fa        CALL       <EXTERNAL>::__isoc99_scanf           undefined __isoc99_scanf()
         ff ff
```

angr CTF

03_angr_symbolic_registers // angr // How to create a symbol

- We can manually recreate the state after scanf() with our symbolic values
- **EAX**, **EBX** and **EDX** contain the read values (`%x` → lowercase hex values)
- Symbols are handled by **claripy** (angr's solver engine)

- An example on how to create a symbol and assign it to EAX:

```
>>> import claripy
>>> eax = claripy.BVS ('<symbol_name>', 32) # registers are 32-bit long
>>> initial_state.regs.eax = eax
```

angr CTF

03_angr_symbolic_registers // angr // **Solution**

```
>>> start_address = 0x08048946 # Right after get_user_input()
>>> initial_state = project.factory.blank_state (addr = start_address)
>>> eax = claripy.BVS ('password0', 32) # registers are 32-bit long
>>> ebx = claripy.BVS ('password1', 32) # registers are 32-bit long
>>> edx = claripy.BVS ('password2', 32) # registers are 32-bit long
>>> initial_state.regs.eax = eax
>>> initial_state.regs.ebx = ebx
>>> initial_state.regs.edx = edx
>>> simulation = project.factory.simgr (initial_state)
>>> simulation.explore (find = 0x080489e9, avoid = 0x080489dc)
<SimulationManager with 1 found, 3 avoid>
>>> "%x" % simulation.found[0].solver.eval (eax)
'b9ffd04e'
>>> "%x" % simulation.found[0].solver.eval (ebx)
'ccf63fe8'
>>> "%x" % simulation.found[0].solver.eval (edx)
'8fd4d959'
```

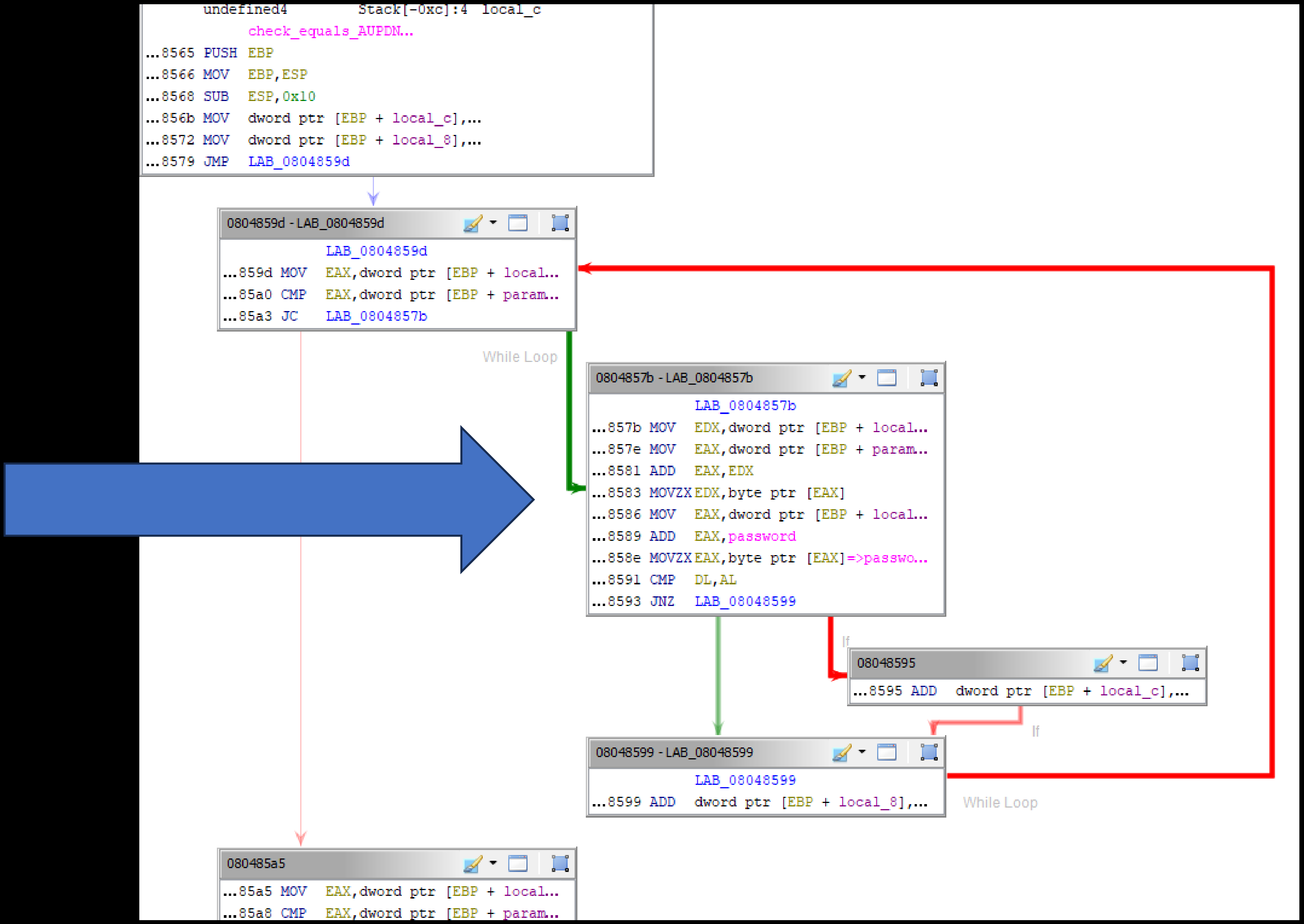
angr CTF

08_angr_constraints // path explosion

- Comparing each character of the password leads to **path explosion**

- 2 branches to explore for each of the 16 characters: $2^{16} = 65536$ **states** to explore

- Solution: stop *before* this check and constraint it manually



angr CTF

08_angr_constraints // manual constraint

- Tip: the CPython REPL takes ages to finish
  - Better use a standalone script (or maybe another REPL - ?)

```
start_addr = 0x0804863C # Right after scanf()

initial_state = project.factory.blank_state (
   addr = start_addr,
   add_options = { angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY }
)

passwd_size = 16 # 16 symbolic Bytes for the password
password = claripy.BVS ('password', 8 * passwd_size)
passwd_address = 0x804a040 # Made-up address to "store" the user input
initial_state.memory.store (passwd_address, password)

target_addr = 0x08048685 # After the loop, but before check_equals_()
simulation = project.factory.simgr (initial_state)
simulation.explore (find = target_addr)
# <SimulationManager with 2 active, 30 deadended, 1 found>
solution_state = simulation.found[0]

constrained_bv = solution_state.memory.load (passwd_address, passwd_size)
target_passwd = b'OSIWHBXIFOQVSBZB'

solution_state.add_constraints (constrained_bv == target_passwd)

solved_passwd = solution_state.solver.eval (password, cast_to = bytes)
print (solved_passwd)
```

angr CTF

10_angr_simprocedures // Same problem, different solution

- The same problem can also be solved by implementing the problematic function Python using **SimProcedures**
- Useful to avoid uninteresting or already tested functions
  - the less code we symbolize, the better
- angr already implements many common library functions as SimProcedures: https://github.com/angr/angr/tree/master/angr/procedures
- Very easy to implement:

```
class Whatever (SimProcedure):
  def run (self, arg):
     return arg + 3

project.hook_symbol (<target>, Whatever ())
```

angr CTF

10_angr_simprocedures // Running our SimProcedure

```
>>> project = angr.Project (".\\dist\\10_angr_simprocedures")
>>> # This scanf is a simple and we don't need to manually constrain the password buffer
>>> initial_state = project.factory.entry_state ()
>>> class CustomSimproc (angr.SimProcedure):
...     def run (self, string_ptr, length):
...         string = self.state.memory.load (string_ptr, length)
...         return claripy.If ( # The expression can be symbolic, hence claripy being used here
...             string == b'ORSDDWXHZURJRBDH',
...             claripy.BVV (1, 32), # 32-bit vector // TRUE
...             claripy.BVV (0, 32)  # 32-bit vector // FALSE
...         )
...
...
>>> project.hook_symbol ('check_equals_ORSDDWXHZURJRBDH', CustomSimproc ())
0x80485f5
>>> simulation = project.factory.simgr (initial_state)
>>> simulation.explore (find = is_successful, avoid = should_abort)
>>> simulation.found[0].posix.dumps (0)
b'MSWKNJNAVTTOZMRY'
```

angr CTF

17_angr_arbitrary_jump // Finding a Buffer Overflow

- Theory: if a **jump destination is unconstrained**, that means we (maybe) overwrote it with **user input** → possible buffer overflow
- Unconstrained states are normally discarded by angr
- Let's practice with a basic buffer overflow:

```
void read_input(void)

{
  undefined local_24 [32];

  __isoc99_scanf("%s",local_24);
  return;
}
```

angr CTF

17_angr_arbitrary_jump // Finding unconstrained jumps

```
>>> project = angr.Project (".\\dist\\17_angr_arbitrary_jump")
>>> symbolic_input = claripy.BVS ("input", 100 * 8) # Simulate a big user input
>>> initial_state = project.factory.entry_state (stdin = symbolic_input)
>>> simulation = project.factory.simgr ( # Manually create the needed stashes
...     initial_state,
...     save_unconstrained = True, # Needed to check for unconstraied EIP
...     stashes = {
...       'active' : [initial_state],
...       'unconstrained' : [],
...       'found' : [],
...       'not_needed' : []
...     }
...   )
>>> # explore() doesn't work automatically anymore. We must do it manually
>>> while (simulation.active or simulation.unconstrained) and (not simulation.found):
...     for state in simulation.unconstrained:
...         simulation.move ('unconstrained', 'found')
...     simulation.step ()
...
<SimulationManager with 2 found>
```

angr CTF

17_angr_arbitrary_jump // Build exploit

```
<SimulationManager with 2 active>
<SimulationManager with 2 active>
WARNING  | 2024-02-27 21:55:04,795 | angr.engines.successors | Exit state has over 256 possible solutions. Likely unconstrained; skipping. <BV32 Reverse(input_2253_800[511:480])>
WARNING  | 2024-02-27 21:55:06,045 | angr.engines.successors | Exit state has over 256 possible solutions. Likely unconstrained; skipping. <BV32 Reverse(input_2253_800[511:480])>
<SimulationManager with 2 unconstrained>
<SimulationManager with 2 found>
<SimulationManager with 2 found>
>>> _
```

```
>>> # Build exploit
>>> solution_state = simulation.found [0]
>>> solution_state.add_constraints (solution_state.regs.eip == 0x4d435250) # Target function
>>> for byte in symbolic_input.chop (bits = 8): # Account for bad chars
...     solution_state.add_constraints (
...             byte >= b'A',
...             byte <= b'Z'
...     )
>>>  solution_state.solver.eval (symbolic_input)
0x43434343434343434343434343434343434343434343434343434343434343434343434343434343505243 4d434343...
```

Resources

- https://docs.angr.io
- https://www.youtube.com/watch?v=yRVZPvHYHzw
- https://github.com/jakespringer/angr_ctf
- https://blog.notso.pro/2019-03-20-angr-introduction-part0/
- https://www.youtube.com/watch?v=TlEjgqSXYNE

- angr API docs: https://docs.angr.io/en/latest/api.html
- claripy API docs: https://docs.angr.io/projects/claripy/en/latest/api.html
- CLE API docs: https://api.angr.io/projects/cle/en/latest/