Using z3 to solve crackme

Julien Bachmann
@milkmix_

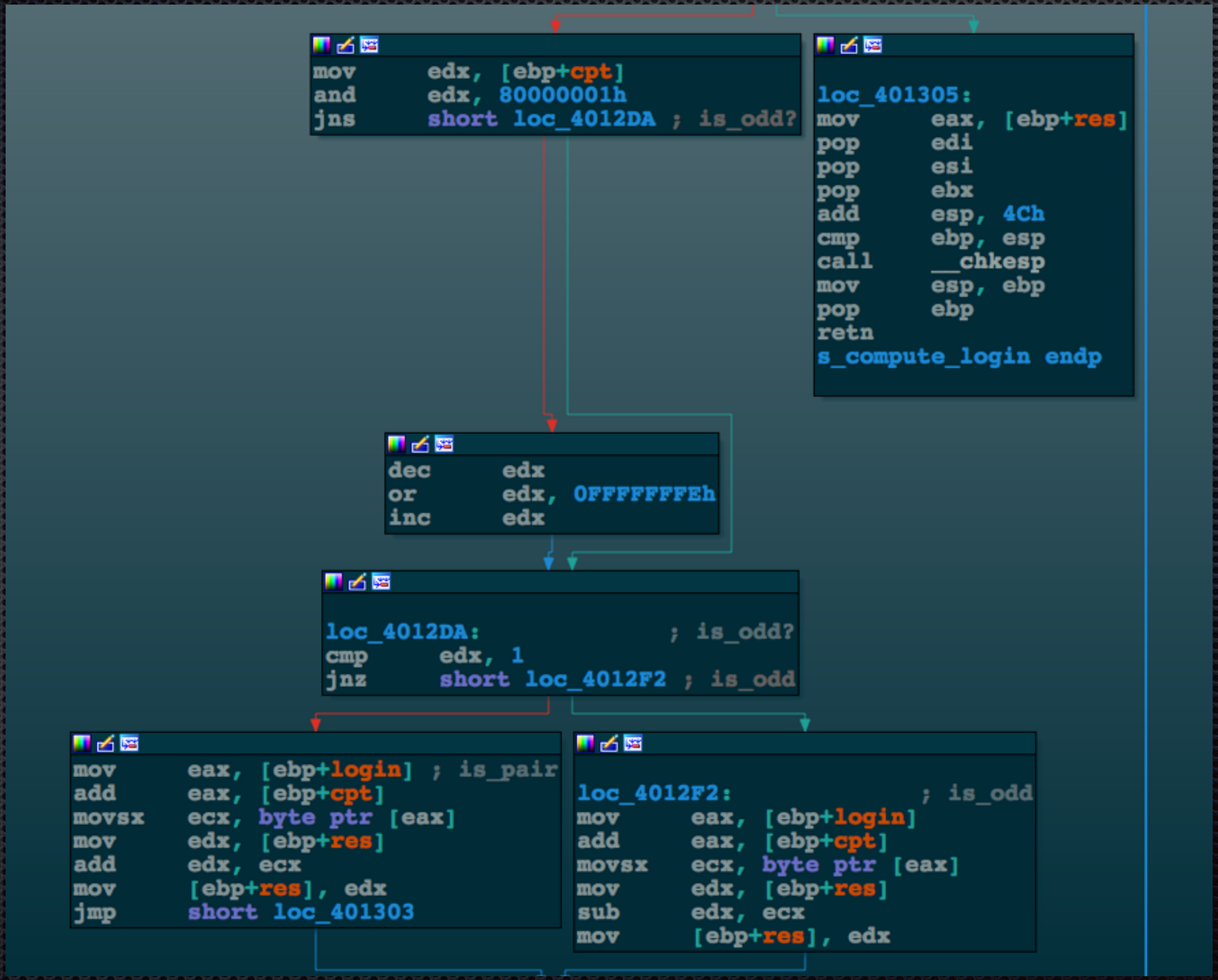
how | irc, con and ctf

- Some have been talking about it for a long time
- Lately : Defcon'15 CTF *fuckup* challenge
  - *"The flag is: z3 always helps"*
  - solved by teammate using… z3 !

use case | standard crackme

- Pretty simple crackme
- No anti-reverse engineering protections
- Need to have *id/serial* tuple that matches the criteria

use case | standard crackme



use case | reverse and reimplement

- Inputs should be alphanumeric strings between 6 and 9 characters
- All distinct
- Sums of both strings characters should be equal
- `compute_serial == compute_id`
- Serial should have increasing order at even positions, decreasing at odd ones

z3 | so what is it?

- z3 is an SMT solver
  - *Satisfiability Modulo Theory*
  - an extension of SAT solvers
  - give it an equation and it can tell you if solvable or not
  - even give you an answer
    - not necessarily the best one

z3 | so what is it?

- Example usages
  - solving Sudoku
  - solving factorisation of large number into primes numbers

z3 | so what is it?

- Example usages
  - solving Sudoku
  - solving factorisation of large number into primes numbers

lame

not sure about that one…

z3 | so what is it?

- For me it is more an Cyber Oracle
  - honestly, I didn’t looked at all the theory and maths behind

z3 | installation

- Open sourced by Microsoft
  - *yeah, for real !*
  - https://github.com/Z3Prover/z3

z3 | types

- Constraints can only be applied to z3 data types
- Numbers
  - *Int, Real, Bool*
- Define multiples
  - *Ints*
  - *Reals*

```
>>> from z3 import *
>>> x = Int('x')
>>> y = Real('y')
>>> a, b = Ints('a b')
```

z3 | types

- Closest to our potentials cases
- CPU registers !
  - *BitVec*
- Extendable
  - *ZeroExt*
  - *SignExt*

```
>>> from z3 import *
>>> eax = BitVec('eax', 32)
>>> rax = ZeroExt(32, eax)
>>> eax.size()
32
>>> rax.size()
64
```

z3 | types

- Warning !
- *Int* are infinite numbers
- *BitVec* are wrapping, like registers

z3 | operators

- Standard ones
  - **+**, **-**, *****, **==**, …
  - *RotateLeft, RotateRight*
- Constraints
  - *And, Or*
  - *ULT, UGT*
  - *Distinct*
  - *…*

z3 | solver

- The class you will be using the most
  - ***add*** : add a constraint to the equation
  - ***push/pop*** : store current state of the constraints
  - ***prove*** : check if given equation is always true
  - ***check*** : validate if solution exists
  - ***model*** : if solvable, return **a** solution
  - ***simplify*** : simplify current equation

z3 | solver

```
>>> from z3 import *
>>> x, y = Ints('x y')
>>> s = Solver()
>>> s.add(x + 2 * y == 2)
>>> s.check()
sat
>>> s.model()
[y = 0, x = 2]
```

```
>>> from z3 import *
>>> x, y = Ints('x y')
>>> prove((x + y) < (x * y))
counterexample
[y = -8, x = 5]
```

crackme | time to solve it

```python
def generate_string(base, length):
    return [Int('%s%d' % (base, i)) for i in range(length)]

def alpha(c):
    return And(97 <= c, c <= 122)
```

```python
def constraint_serial(values):
    res = []
    res.append(values[0] > values[-1])
    for i in range(1, len(values) - 2):
        if i % 2:
            res.append(values[i] > values[i + 2])
        else:
            res.append(values[i] < values[i + 2])
    return res
```

```python
def std_sum(values):
    res = IntVal(0)
    for i in range(0, len(values)):
        res += values[i]
    return res
```

crackme | time to solve it

```
s = Solver()
id = generate_string('x', 7)
serial = generate_string('y', 7)

s.add(Distinct(id + serial))

s.add(And(map(alpha, id)))
s.add(And(map(alpha, serial)))

s.add(constraint_serial(serial))

s.add(std_sum(id) == std_sum(serial))
s.add(compute_id(id) == compute_serial(serial))

if s.check() == unsat:
    print "[+] no solution can be found"
    exit(1)

while s.check() == sat:
    print_model(s.model(), id, serial)
    s.add(And([x != s.model()[x] for x in id]))
```

conclusion | awesome

- Quite useful tool when
  - brute force would take too long
  - problem can easily be put in the form of equations
- Can be applied to
  - auto-ROP to solve constraints on registers
  - concolic execution (symbolic+concrete)
    - check Quarkslab Triton