Keygenning using the Z3 SMT Solver

ExtremeCoders
email: extremecoders@hotmail.com

April 27, 2015

1 Introduction

Quoting Wikipedia, *In computer science and mathematical logic, the satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical firstorder logic with equality.*

Formally speaking, an SMT instance is a formula in first order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable.

This article shows how we can use an SMT solver in developing a key generator. We need to find valid keys while also satisfying certain constraints. We will use the Z3 SMT Solver, as it is quite easy to use and also has a nice python API to code against.

2 The Challenge

While solving a few problems from the TJCTF challenge, I stumbled upon an interesting one. The problem is as follows.

We have been provided with a *jar* (Java Archive) file. The challenge is to find the password which will "unlock" this executable jar file. The *jar* file contains just a solitary *class* within it named *JavaBytecode.class*.

We can decompile this *class* straightaway to proceed. The decompiling proceeds fine which means that the code wasn't obfuscated. Here is the decompiled code. For better understanding of the code, some variable and function names have been renamed. Comments have also been added as and when deemed necessary.

```
import java.io.PrintStream;
import java.util.Scanner;

public class JavaBytecode
{
  public static void main(String[] args)
  {
    Scanner a = new Scanner(System.in);
    System.out.print("Enter passcode: ");
    String b = a.nextLine();
    if (check(b))
      System.out.println("Success!");
    else
      System.out.println("Failed");
```

```
}

// Junk Method
private static void junk(String b)
{
  int i = 0;
  b = b + i;
}

private static boolean check(String pass)
{
  for (int i = 0; i < 10000000; i++)
  {
    junk(pass);
  }

  for (int i = 0; i < pass.length(); i++)
  {
    char c = pass.charAt(i);
    // All characters must lie within 0-9 inclusive
    if ((c < '0') || (c > '9'))
    {
      return false;
    }
  }

  int d = 0;
  // Calculate sum of even positioned digits
  for (int i = 0; i < pass.length() - 1; i += 2)
  {
    d += pass.charAt(i) - '0';
  }

  // Calculates sum of odd position digits
  for (int i = 1; i < pass.length() - 1; i += 2)
  {
    d += (pass.charAt(i) - '0') * 3;
  }

  // rem should equal 9
  int rem = 10 - d % 10;
  int sum = 0;

  // Sum of first 5 digits
  for (int i = 0; i < 5; i++)
  {
    sum += pass.charAt(i) - '0';
  }
```

```
    int prod = 1;
    // Products of 4 digits after the first.
    for (int i = 1; i < 5; i++)
    {
      prod *= (pass.charAt(i) - '0');
    }

    int d1 = pass.charAt(1) - '0';
    int d2 = pass.charAt(2) - '0';
    int d3 = pass.charAt(3) - '0';
    int d4 = pass.charAt(4) - '0';

    // The substring at 5 must equal this hardcoded value
    if (!pass.substring(5, 11).equals("914323"))
      return false;

    if ((Math.abs(d1 + d2 - d3) != 1) || (Math.abs(d1 + d2 - d4) != 1))
      return false;

    // Must contain at least a zero
    if (!pass.contains("0"))
      return false;

    // Last digit of passcode must be 9
    if (pass.charAt(pass.length() - 1) - '0' != rem)
      return false;

    // Sum of first 5 digits must equal 21
    if (sum != 21)
      return false;

    // Product of 4 digits after the first one must equal 480
    if (prod != 480)
      return false;

    return rem == 9;
  }
}
```

3 Analysis of the algorithm

The main function accepts a passcode as a *String*. It is passed to another function *check* which checks whether our passcode is valid or not. At the begin of the function there is a call to *junk* which is called about ten million times. This is nothing but a sort of delay loop and can be ignored.

Next we encounter this snippet which checks whether all the characters of the password are within the range of 0-9.

```
for (int i = 0; i < pass.length(); i++)
{
  char c = pass.charAt(i);
  // All characters must lie within 0-9 inclusive
  if ((c < '0') || (c > '9'))
  {
    return false;
  }
}
```

Next we have two loops. The first one sums up the even positioned digits of the passcode. The next one sums up the odd positioned digits. This last sum is further multiplied by 3.

The total of the above two sums modulo 10 is calculated. We will later see that this remainder must equal 1, *i.e.*, variable *rem* must equal 9.

```
int d = 0;
// Calculate sum of even positioned digits
for (int i = 0; i < pass.length() - 1; i += 2)
{
  d += pass.charAt(i) - '0';
}

// Calculates sum of odd position digits
for (int i = 1; i < pass.length() - 1; i += 2)
{
  d += (pass.charAt(i) - '0') * 3;
}

// rem should equal 9
int rem = 10 - d % 10;
```

Next, there are another pair of loops. The first calculates the sum of the first 5 digits. The next calculates the product of 4 digits after the first. These sum and product must equal 21 and 480 respectively which we will see later.

```
// Sum of first 5 digits
for (int i = 0; i < 5; i++)
{
  sum += pass.charAt(i) - '0';
}

int prod = 1;
// Product of 4 digits after the first.
for (int i = 1; i < 5; i++)
{
  prod *= (pass.charAt(i) - '0');
}
```

At the end, there are some additional checks on the passcode. The substring at 5 must match the hardcoded value 914323. It must contain atleast one zero digit. The last digit of the passcode must be 9. The remaining few checks like the one using *Math.abs* can be understood by going through the code.

```
int d1 = pass.charAt(1) - '0';
int d2 = pass.charAt(2) - '0';
int d3 = pass.charAt(3) - '0';
int d4 = pass.charAt(4) - '0';

// The substring at 5 must equal this hardcoded value
if (!pass.substring(5, 11).equals("914323"))
  return false;

if ((Math.abs(d1 + d2 - d3) != 1) || (Math.abs(d1 + d2 - d4) != 1))
  return false;

// Must contain at least a zero
if (!pass.contains("0"))
  return false;

// Last digit of passcode must be 9
if (pass.charAt(pass.length() - 1) - '0' != rem)
  return false;

// Sum of first 5 digits must equal 21
if (sum != 21)
  return false;

// Product of 4 digits after the first one must equal 480
if (prod != 480)
  return false;

return rem == 9;
```

4 Using Z3

In order to find a key, we can code a bruteforcer which generates keys of increasing lengths until it finds one which satisfies all the checks. However such an approach will be inefficient both in space and time as there are far too many keys to check. Further more memory and time requirements of bruteforce approach will increase exponentially if we need to find keys of increasing lengths.

This is an ideal problem which can be solved using Z3. We need to initialize the parameters and add the constraints. Z3 would automatically try to find a solution respecting all the given constraints. This is our quick n dirty Z3 solver. It will try to find valid keys staring from a length of 11 characters.

```
from z3.z3 import *
```

```
# The digits of the passcode
d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10  = \
Ints('d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 d10')

digits_list = [d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10]

def check(num_digits):
    # Create a new digit
    globals()['d{}'.format(num_digits)] = Int('d{}'.format(num_digits))

    # Append it to the digits list
    digits_list.append(globals().get('d{}'.format(num_digits)))

    # Initialize the solver
    s = Solver()

    # Add the constraints
    # Sum of first 5 digits should equal 21
    s.add(d0 + d1 + d2 + d3 + d4 == 21)

    # Product of 4 digits after the first should equal 480
    s.add(d1 * d2 * d3 * d4 == 480)

    # All digits must be within 0-9 inclusive
    s.add(d0 >= 0, d0 <= 9)
    s.add(d1 >= 0, d1 <= 9)
    s.add(d2 >= 0, d2 <= 9)
    s.add(d3 >= 0, d3 <= 9)
    s.add(d4 >= 0, d4 <= 9)
    for i in range(11, num_digits):
        s.add(  globals().get('d{}'.format(i)) >= 0,
                globals().get('d{}'.format(i)) <= 9
             )

    # Digits from 5 to 10 should be 914323
    s.add(d5 == 9, d6 == 1, d7 == 4, d8 == 3, d9 == 2, d10 == 3)

    # Constraint involving Math.abs
    s.add(Or(d1 + d2 - d3 == 1, d1 + d2 - d3 == -1))
    s.add(Or(d1 + d2 - d4 == 1, d1 + d2 - d4 == -1))

    # Last digit must be 9
    s.add(globals().get('d{}'.format(num_digits)) == 9)

    # ((sum of even pos digits) + (sum of odd pos digits) * 3) % 10 ==1
    s.add((
       reduce(lambda a, b: a+b, digits_list[0:len(digits_list)-1:2])
        +
       reduce(lambda a, b: a+b, digits_list[1:len(digits_list)-1:2]) *3
```

```
        ) % 10 == 1)

    # Check if there is a solution to the problem
    if s.check() == sat:
        m = s.model()
        print 'Found a passcode of length %d digits' %num_digits
        print 'Passcode is', \
            reduce(lambda a, b:a + str(m[b]), ([''] + digits_list))
        return True
    return False


def main():
    # Minimal number of digits is 11
    num_digits = 11
    while check(num_digits) == False:
        print 'No passcode of length %d exists' %num_digits
        num_digits += 1

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

On running our solver we get the following output

```
No passcode of length 11 exists
Found a passcode of length 12 digits
Passcode is 0526891432329
```

The above passcode "unlocks" the jar. Some other valid passcodes of higher lengths are 05286914323029, 025869143232209, 0258691432327299, 02568914323002609. There may also be more valid passcodes for a particular length.

The most notable fact is that Z3 found valid passcodes in quick time, which would not have been possible had we chose a bruteforce approach.

5 Conclusion

SMT Solvers are such an useful tool that they should be a necessary weapon in every reverser's kitty. They can be used whenever we have a type of constraint satisfaction problem. SMT solvers can also be used in symbolic execution, bug finding & fuzz testing, where we need to generate inputs such that a particular branch of code execution is taken, triggering the bug.

6 References

1. z3 Theorem Prover :: https://github.com/Z3Prover/z3
2. z3 API In Python Guide :: http://web.archive.org/web/20150214002728/http://rise4fun.com/z3py/tutorial
3. Z3 - a Tutorial :: http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.225.8231

4. Automated algebraic cryptanalysis with OpenREIL and Z3 (Kao's toy project) :: http://blog.cr4.sh/2015/03/automated-algebraic-cryptanalysis-with.html

5. Taming a Wild Nanomite-protected MIPS Binary With Symbolic Execution :: https://doar-e.github.io/blog/2014/10/11/taiming-a-wild-nanomite-protected-mips-binary-with-symbolic-execution-no-such-crackme/

6. Breaking Kryptonite's Obfuscation: A Static Analysis Approach Relying on Symbolic Execution :: https://doar-e.github.io/blog/2013/09/16/breaking-kryptonites-obfuscation-with-symbolic-execution/

7. Finding unknown algorithm using only input-output pairs and Z3 SMT solver :: http://go.yurichev.com/17268

8. SMT Solving - San Heelan's Blog :: https://seanhn.wordpress.com/category/smt-solving/