Nick Anderson, 09/06/2013
Application Security - Project 1, Turing Complete Sandbox Documentation

This is a python implementation of the Turing complete sandbox for the Application Security course at NYU-Poly.

This program takes as input the name of a program file which the sandbox will run. While the name of this file does not need to have any specific structure, it does need to exist in the same directory as the python script. As a convention, scripts written for this sandbox have been utilizing the '.in' suffix. To run one of the sample programs, at the console simply type: python sandbox.py name_of_script.in. As an example, running the Fibonacci script happens as follows:

```
python sandbox.py fibo.in
```

A discussion regarding the Turing completeness of this sandbox as well as the sample execution of the included scripts is at the end of this document, after the finer points of the language have been disclosed.

Sanbox Notes:

- All numeric values used in this sandbox are integers. This sandbox does not support floating point anything. Thus, any division performed that would normally be a fraction will be rounded off according to the Python programming language.
- Example programs for this sandbox are included in the directory where this document lives. They are 'fibo.in', 'countdown.in', 'inner_loop.in', 'ifelse.in', and 'while_ifelse.in'. These programs exemplify some of the idiosyncrasies of this sandbox programming language.
- This sand box takes from the command line one parameter, being the program which the sandbox should run. The structure of the program given to the sandbox should contain one to many of the following types of commands.
- Although all of the below documentation utilizes upper casing for the accepted commands, lower casing will be counted as valid syntactically, e.g. 'add 2 1' is an acceptable command.
- Additionally, the structure of the programs may be whatever you like so long as the commands utilized are in the accepted commands detailed later in this document. This language additionally disregards any extraneous spacing, tabbing, or newlines. Thus the physical structure of the language can have any amount of spacing you desire.
- Variables may be used without being assigned in any arithmetic expression, however variables must be assigned before attempting to print them. For example

  ```
  VAR ADD X 1
  VAR ASGN X Y
  ```

  Are both valid statements, however the following is an incorrect statement

  ```
  VAR PRINT Z
  VAR PRINT TMP
  ```

As neither 'Z' nor 'TMP' have been seen before. Both of these statements will throw an execution error, as the variable has not been seen previously, either in an assignment statement or in an arithmetic operation.

- Any command which does not adhere to any of the formats listed below will be discarded as syntactically incorrect.

Output Operations:

This sandbox supports writing to standard out via the use of the command 'PRINT'. The print command will take one of three arguments – A string, an integer, or variable with the 'VAR' declaration preceding the name of the variable you wish to output

```
PRINT VAR X        # Prints the value currently stored in the variable X
PRINT 1            # Prints the character '1'
PRINT “Hello”      # Prints the string ‘Hello’
```

As mentioned above, it is an error to print a variable which has not yet been assigned, or been used in any arithmetic statements.

Variables:

Reserved words are any mnemonics used for a command. These words may not be used for variable assignment. The detailed list is: ADD, SUB, MUL, DIV, EXP, MOD, VAR, WHILE, ENDWHILE, IF, ENDIF, ELSE …

All variables will be initialized to the value 0. All arithmetic operations on a variable will store the result in that variables memory. For example,

```
VAR ADD X 3
PRINT VAR X
```

Will print out the value '3'. Whereas the following code

```
VAR ASGN X 5
VAR MUL X 4
PRINT VAR X
```

Will output the value '20'. Lastly, any variable arithmetic performed before the right hand sided variable has been assigned will initialize the right hand sided variable to 0 for addition/subtraction and 1 for multiplication/division. For example

```
VAR X = 10
VAR ADD X Y
PRINT VAR X
```

Will print the value ‘10’, further

```
VAR X = 4
VAR DIV X Y
PRINT VAR X
```

Will output the value ‘4’. Below is a detailed list of all variable operations supported.

Variable Logic Operations:

```
VAR ASGN X 10    # Assigns the value of X to be 10
VAR ADD X 2      # Adds two to the value of X and stores the result in X
VAR ADD X Y      # Adds the value of Y to the value of X and stores the result in X
VAR SUB Z 2      # Subtracts 2 from the value of Z and stores the result in Z
VAR SUB Z X      # Subtracts the value of X from Z and stores the result in Z
VAR MUL Y 3      # Multiplies the value of Y by 3 and stores the result in Y
VAR MUL Z X      # Multiplies the value of Z by X and stores the result in Z
VAR DIV X 4      # Divides the value of X by 4 and stores the result in X
VAR DIV X Z      # Divides the value of X by Z and stores the result in X
VAR EXP X 4      # Exponentiates the value of X by 4 and stores the result in X
VAR EXP X Z      # Exponentiates the value of X by Z and stores the result in X
VAR MOD Z 6      # Divides the value of Z by 6, and stores the remainder in Z
VAR MOD Z X      # Divides the value of Z by X, and stores the remainder in Z
```

Loops:

Loop statements are constructed similar to a while loop in either C or Python is constructed. Currently the only operations supported by this language for control logic are greater than, and less than. They are signified by typing LT or GT respectively. For example, to loop 10 times in this language, one would write the script.

```
VAR ASGN CNT 0
WHILE VAR CNT LT 10
  PRINT VAR CNT
  VAR ADD CNT 1
ENDWHILE
```

It is worth noting here, that any while loop must be closed with an 'ENDWHILE' statement, signifying the end of the while loop commands. Additionally, in order to use a variable inside of the control logic for a while loop, one must specify the variable being used with a prefix of 'VAR', as written above. Below is a more detailed listing of the loop commands.

Looping Commands:

```
WHILE ‘condition’   # Loops over all code between 'LOOP' and 'END' 12 times
‘looped inst’       # Commands which will occur once every iteration.
ENDWHILE            # Ends the while loop.  This line is required
```

IF Statements:

If statements are similar to while loops, in that the logic statements only support less than, and greater than operations, and are similarly specified by 'LT' or 'GT' respectively. Further, IF statements must be completed with an ENDIF statement, and any 'else' logic must exist between the opening 'IF' statement, and the closing 'ENDIF' statement. An example if statement is written below

```
VAR ASGN X 10
IF VAR X GT 5
  PRINT VAR X
ELSE
  PRINT “X WAS NOT GREATER THAN 5”
ENDIF
```

Below is a more detailed listing of the 'IF' and 'ELSE' commands.
Logical Branch Constructs:

```
IF ‘condition’          # Signifies the beginning of a logical branch
‘commands’              # Commands that will run if the IF condition passes
ELSE                    # Signifies the alternate branch of the IF statement
‘alternate commands’    # Commands which run only if the IF condition fails
ENDIF                   # Ends the if statement.  This line is required
```

Arithmetic:

Any purely arithmetic statement will simply compute the solution and print the value. Any variable arithmetic will compute the solution and store the answer in the left hand side variable. Below is the detailed listing of all arithmetic operations supported

Basic Arithmetic Operations:

```
ADD 1 2     # Adds the values 1 and 2
SUB 1 2     # Subtracts the values 1 and 2
MUL 1 2     # Multiplies the value 1, by the value 2
DIV 1 2     # Divides 1 by the integer 2
EXP 1 2     # Raises 1 to the 2nd power
MOD 2 4     # Computes 2 modulo 4, similar to the % operator in Python
```

Turing Completeness

To argue that my sandbox is Turing complete, we note that this sandbox implements all of the functionality in some form of equivalence as another Turing complete programming language. The language used for this equivalence comparison is the Intel 8088 Architecture instruction set, as this is a Turing complete instruction set. Firstly, regarding the ability to access and reference arbitrary memory location, my sandbox implements the functionality of variables. Referencing variables happens through the use of formal variable names, and thus we can assign values to memory locations, and print the value of a memory location through the use of the variable name. Although this sandbox does not have a formal structure that would directly equate to a stack, one could be simulated through the use of compound expressions including variable assignment statements, and the while loop construct.

Secondly, the Intel 8088 architecture implements all of the basic arithmetic instructions that one would expect a basic process to understand. This is accomplished in my sandbox by implementing very basic arithmetic operators that equate to the Intel 8088 instruction set, such as addition, subtraction, multiplication, and division. Furthermore, though my sandbox does not have bit shifts, these operations can be simulated through proper arithmetic sequences, such as division or multiplications by powers of two respectively.

Lastly, to implement conditional jumps in my sandbox I made use of the familiar IF ELSE statement. This simulates the conditional logic implemented in the Intel 8088 via the use of jmp statements. Because the IF ELSE statements were implemented recursively, we can further nest as many if else statements withing one another as we desire. Additionally, the implementation of a jmp like structure is aided by the while loop construct implemented in my sandbox. This allows for repetitive execution of logic, as well as the potential for infinite loops.

Sample Execution of Sandbox Scripts

Along with this sandbox there are a few sample scripts to aid the user in understanding how to write programs for the sandbox. The fibo.in script will compute and display the 'nth' Fibonacci number. To change which Fibonacci number is computed open the script and change the value of the variable 'fibo' to the nth Fibonacci number you wish to compute.

The countdown.in script will count down from 10 to 1, and print out the number after each iteration through the loop. To change the value the script counts down from, simply open the countdown.in script with a text editor and change the value of the 'CNT' variable.

The inner_loop.ing, ifelse.in, and while_ifelse.in scripts simply illustrate that the sandbox is capable of nested logical constructs, such as inner while loops, or if else statements inside of if statements or while loops. These programs can be run from the command line in the same fashion as the fibo and countdown programs.

Lastly the 'data.in' script was simply a collection of arbitrary commands the sandbox uses designed to test the robustness of the sandbox. It is included simply for an additional script that can be run to test the sandboxes functionality.