Secure Turing Complete Sandbox Challenge

Assignment 1, Part 1

Application Security CS-GY 4753

Prof. Justin Cappos

Author: Casey McGinley

Overview

My approach to this assignment was to take an existing computational resource and restrict its functionality to “safe” operations. To that end, I wrote my sandbox in Python. Essentially, my sandbox acts as a gatekeeper between the Python interpreter and some user-supplied code.

My sandbox allows the supplied program access only to basic functions and structures that Python supports. This framework is flexible enough to be considered Turing-complete (as I will explain in a later section), but it does remove many Python capabilities, most notably all I/O (aside from the ability to print to standard out) and any system level hooks/calls. Also, particularly problematic commands/functions like eval, exec, and execfile have also been removed from the execution environment of code in the sandbox.

Also, in order to prevent any attempts to get the sandbox to overwrite important data, the sandbox only reads from a pre-defined filename, which the user MUST use if they want the sandbox to run at all. The program source is in sboxed_program.lpy (lpy, for Limited Python) and the data is in sboxed_data.dat

I created a "blacklist" of unacceptable keywords/commands (import, exec, del, from) and a "whitelist" of acceptable functions from the __builtins__ module. Import and from would allow the user's program to bring in other modules, modules which may have some dangerous functions. So, to keep things simple, I just removed the import functionality. Exec I removed as I don't want the user having any access to commands or functions which arbitrarily process code. I tokenized the program using Python's tokenize module to first check for any blacklisted keywords. If any were found, the user was notified and an Exception was raised.

To enforce my whitelist of builtin functions, I created an explicit dictionary of all functions I wanted the sandboxed program to have access to. These included some basic and important functions like the typing functions str and int, but not too much beyond that. There are likely other "safe" functions that could be added, but they wouldn't do anything for the "Turing-completeness" of my sandbox, so they can be passed over for now. Notably, the function calls eval and execfile were not included in my whitelist, since, as I mentioned, these can be particularly problematic as they allow for the arbitrary execution of code. Additionally, important constants like False, True, and None were also included. Finally, I also set the __builtins__ entry in this dictionary explicitly to None so that all other builtin functions that haven't been whitelisted are inaccessible.

Since data could potentially be stored in a file separate from the program source, I added logic to read the data file entirely into memory at the beginning of the sandbox's runtime. This data is then stored as a string into the same dictionary I described above.

Assuming the program passes the tokenization portion where I test for banned keywords, we can now safely execute the file. Note that the dictionary I've defined is passed along with the filename to execfile. This dictionary establishes the variables/constants/functions available in the execution environment of the code given to my sandbox.

Examples

The examples here are very straightforward.

For Example 1 (examples/example1.lpy), I start with a base case of $2^0=1$, and then I multiply this result by two repeatedly inside a while loop. The result is printed each time and the loop finally breaks when we hit 128.

For Example 2 (examples/example2.lpy), I maintain a variable for the current number (num) and the previous (prev). I setup a counter and setup my while loop to stop when the counter hits 10. On each iteration I print the current number and then compute the next by adding the current and the previous. I then set the previous to be the current and the current to be the next (using a temporary variable to accommodate the switch) and finally increment the counter.

Turing Complete

We consider again the limitations I have imposed on Python. None of them have any bearing on Python's ability to loop, store data in variables, or perform conditional logic. All I have restricted is I/O operations, system calls, and calls to external modules. All primitive data types (and a few non-primitive) are accessible, all arithmetic operations can still be performed, variable assignment is unaffected, and repetition and conditional logic are still present. In truth, these three characteristics (variable assignment, repetition, conditional logic) are really enough for a language to be Turing complete.

We consider the Turing machine components: tape, head, state register, table of instructions. The tape is made of cells each of which can contain a value, and the tape is infinite in this theoretical model. Variable assignment is a clear analog to this, as we can assign some variable to a given value or structure to persist that value, and although variable assignment is not truly infinite, in most cases it might as well be since it would be difficult to make enough variables to reach the end of machine's memory and/or storage capacity. The head of the Turing machine may move left or right along the tape, meaning cells may be returned to and certain conditions can be repeated since the head is not limited to movement in a single direction on the tape. This is essentially the concept of repetition, which in Python can be achieved both through for or while loops, as well as recursion. The state register essentially allows for more than one condition to play into a single logical action (e.g. the state AND the current cell determine the course of action). This functionality is of course also seen in Python's complex conditional statements (e.g. and, or, etc.). Finally, we have the table of instructions, which determine the behavior of the head and any changes to the state and the current cell at any given instance. This table of instructions is essentially equivalent to the conditional logic and flow control that Python uses. Conditional statements determine if a Python program proceeds, if it skips some segment, or if it goes back and repeats a previous segment, just as a given instruction from this table determines where the head moves to next and how values and state are affected.

Clearly, Python has analogs for the four main components of a classical Turing machine, and as I described above, my restrictions on Python had no impact on the essential operations of these analogs.

Thus, as Python is Turing-complete and as my restrictions do no affect the Python components which make it Turing-complete, my sandbox must be Turing-complete.