Assignment 3: The Leaky Sandbox
15-316 Software Foundations of Security and Privacy

1. **Leaky sandbox (30 points).** Consider the following language, which resembles a simplified assembly language.

$\texttt{add}(x, y)$	Add variables x and y, store the result in x
$\texttt{sub}(x, y)$	Subtract variable y from x, store the result in x
$\texttt{mul}(x, y)$	Multiply variables x and y, store the result in x
$\texttt{and}(x, y)$	Take the bitwise-and of variables x and y, store the result in x
$\texttt{or}(x, y)$	Take the bitwise-or of variables x and y, store the result in x
$\texttt{not}(x, y)$	Take the bitwise negation of variable x, store the result in x
$x := c$	Copy a constant c into variable x
$x := y$	Copy the value stored in y to x
$x := *(y)$	Read the memory at address stored in variable y, save result in x
$*(x) := y$	Store the value in y at the address pointed to by x
$\texttt{if}(Q)\,\texttt{jump}\,x$	If Q is true in the current state, jump to the instruction pointed to by x

Programs in this language are sequences of instructions indexed on integers 0 to n, and we refer to the instruction at index i of program α with the notation α_i. Note that there are no expressions other than constants and variables in this language. Instead, results of operations are stored in variables, and can be moved into memory when necessary. Think of variables as acting like registers, so to implement the computation $w := (x \,\&\, y) \mid z$ from our language in lecture we would write the program:

$$\begin{array}{ll} 0: & \texttt{and}(x, y) \\ 1: & \texttt{or}(x, z) \\ 2: & w := x \end{array}$$

Notably, the following are *not* examples of programs:

$w := \texttt{or}(\texttt{and}(x, y), z)$	*because* $\texttt{and}(x, y)$ and $\texttt{or}(\texttt{and}(x, y), z)$ are not variables
$\texttt{add}(x, 1)$	*because* 1 is not a variable
$*(x + 1) := y$	*because* $x + 1$ is not a variable

In each of these cases, the way to correctly express the desired computation would be to break the program into multiple instructions, saving intermediate results in variables. For the third program, this would give:

$$\begin{array}{ll} 0: & \texttt{add}(x, 1) \\ 1: & *(x) := y \end{array}$$

This should remind you of writing assembly code.

Note that just as you should assume that any memory reads outside the bounds of $[0, U]$ will result in an aborted trace, you should assume that any attempt to `jump` to an address outside the bounds of $[0, N)$, where N is the number of instructions in α, will also abort the trace.

The sandbox. We want to implement a sandboxing policy for this language using software fault isolation. So the proposal is to replace all memory read and write operations as follows. Assume that s_l = `0xb00` and s_h = `0xbff`, so the memory sandbox is contained in the range of addresses `0xb00` − `0xbff`.

$$\begin{array}{lll} x := *(y) & \text{becomes} & \begin{array}{l}\texttt{and}(y, \texttt{0xbff}) \\ \texttt{or}(y, \texttt{0xb00}) \\ x := *(y)\end{array} \\ \\ *(x) := y & \text{becomes} & \begin{array}{l}\texttt{and}(x, \texttt{0xbff}) \\ \texttt{or}(x, \texttt{0xb00}) \\ *(x) := y\end{array} \end{array}$$

Additionally, we want to prevent jumps from leaving a code sandbox restricted to the range of instruction addresses 0xa00 − 0xaff, which is where the sandboxed program will be loaded prior to running it. This is more challenging, as the instrumentation that was added to earlier instructions may have changed the address of the original jump target.

To address this, we ensure that prior to running the sandboxed code, a *jump table* has been prepared at memory addresses 0xc00 − 0xcff so that the contents of the jump table at address 0xc00 + x contain the new address of the instruction originally located at x. So if the instruction at 0xa03 were moved to 0xa05, then address 0xc03 will point to 0xa05.

So each indirect jump is rewritten as follows.

$$\begin{array}{lll} \texttt{if}(Q)\,\texttt{jump}\,x & \text{becomes} & \begin{array}{l}\texttt{add}(x, \texttt{0x00000c00}) \\ \texttt{and}(x, \texttt{0x00000cff}) \\ \texttt{or}(x, \texttt{0x00000c00}) \\ x := *(x) \\ \texttt{if}(Q)\,\texttt{jump}\,x\end{array} \end{array}$$

Example. Consider the following program which fills l memory addresses starting from b with zeros:

$$\begin{array}{lll} \begin{array}{rl} \texttt{0}: & i := 0 \\ \texttt{1}: & inc := 1 \\ \texttt{2}: & cur := b \\ \texttt{3}: & zero := 0 \\ \texttt{4}: & done := 9 \\ \texttt{5}: & \texttt{if}(i \geqslant l)\,\texttt{jump}\,done \\ \texttt{6}: & *(cur) := zero \\ \texttt{7}: & \texttt{add}(i, inc) \\ \texttt{8}: & \texttt{add}(cur, inc) \\ \texttt{9}: & \ldots \end{array} & \text{becomes} & \begin{array}{rl} \texttt{a00}: & i := 0 \\ \texttt{a01}: & inc := 1 \\ \texttt{a02}: & cur := b \\ \texttt{a03}: & zero := 0 \\ \texttt{a04}: & done := 10 \\ \texttt{a05}: & \texttt{add}(done, \texttt{0xc00}) \\ \texttt{a06}: & \texttt{and}(done, \texttt{0xcff}) \\ \texttt{a07}: & \texttt{or}(done, \texttt{0xc00}) \\ \texttt{a08}: & done := *(done) \\ \texttt{a09}: & \texttt{if}(i \geqslant l)\,\texttt{jump}\,done \\ \texttt{a0a}: & \texttt{and}(cur, \texttt{0xbff}) \\ \texttt{a0b}: & \texttt{or}(cur, \texttt{0xb00}) \\ \texttt{a0c}: & *(cur) := zero \\ \texttt{a0d}: & \texttt{add}(i, inc) \\ \texttt{a0e}: & \texttt{add}(cur, inc) \\ \texttt{a0f}: & \ldots \end{array} \end{array}$$

The jump table will have the following contents:

jump table address	c00	c01	c01	c03	c04	c05	c06	c07	c08	c09	···
contents	a00	a01	a02	a03	a04	a09	a0c	a0d	a0e	a0f	···

Part 1 (10 points).

Explain why this instrumentation is vulnerable to memory reads and writes outside the memory sandbox, and provide an example program in the language that exploits violates the policy. For full credit, your answer should provide the original program, its modified form after performing the sandboxing operations described above, including instruction addresses, and the state of the jump table when the sandboxed program runs. Be sure to explain in words how your example results in a violation of the sandbox policy.

Solution.

Part 2 (15 points). Propose an alternative implementation in this language for the policy in Part 1 that is secure. Your solution should not introduce new operations to the language, and **for full credit, should not add additional overhead in terms of the number of instructions used for instrumentation**. That is, pointer reads and writes should work out to three instructions after instrumentation, and indirect jumps to five instructions.

You may assume that the program being sandboxed only uses a finite set of variables that are known ahead of time by the sandbox designer; for example, an assumption that "the target program prior to sandboxing will only make use of variables given by the first thirteen lower-case alphabet symbols" is perfectly fine. Be sure to clearly state this and any other assumptions that your solution requires, and in clear language why it prevents your attack from Part 1, and will remain secure against other such attacks.

Solution.