Assignment 2: Safe to the Last Command
15-316 Software Foundations of Security and Privacy

Due: **11:59pm**, Thursday 2/15/18
Total Points: 50

1. **Unfinished business (10 points).** In lecture 7, we discussed two cases of the structural induction used to prove the security of SFI. Complete the inductive case for `while` commands. That is, assuming that Equation 1 is valid for α whenever $0 \leqslant s_l \leqslant (x \,\&\, s_h) \mid s_l \leqslant b_h < U$:

$$\forall i. \neg(s_l \leqslant i \leqslant s_h) \land \mathtt{Mem}(i) = v_i \to [\alpha]\mathtt{Mem}(i) = v_i \tag{1}$$

   Prove that it is also valid for $\mathtt{while}(Q)\,\alpha$.

2. **Bad Rules (10 points).** Proof rules have to be sound, i.e., if all premises are valid then the conclusion must be as well. Show that the following proof rule is unsound by giving a counterexample, i.e. an instance of the proof rule for which **all premises are valid but the conclusion is not valid**. For full credit, you must provide an explanation in words as to why your counterexample demonstrates unsoundness.

$$\text{(R1)}\quad \frac{\Gamma \vdash [\alpha]Q \quad \Gamma, Q \vdash [\beta]P}{\Gamma \vdash [\alpha;\beta]P}$$

3. **Leaky sandbox (20 points).** Consider the following language, which resembles a simplified assembly language.

$\mathtt{and}(x, y)$	Take the bitwise-and of variables x and y, store the result in x
$\mathtt{or}(x, y)$	Take the bitwise-or of variables x and y, store the result in x
$x := y$	Copy the value stored in y to x
$x := \mathtt{Mem}(y)$	Read the memory at address stored in variable y, save result in x
$\mathtt{Mem}(x) := y$	Store the value in y at the address pointed to by x
$\mathtt{if}(Q)\,\mathtt{jump}\,x$	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 Π with the notation Π_i. Note that there are no expressions in this program. 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:

$$\begin{aligned} \mathtt{1:}\quad & \mathtt{and}(x, y) \\ \mathtt{2:}\quad & \mathtt{or}(x, z) \\ \mathtt{3:}\quad & w := x \end{aligned}$$

It is *not* possible to write $w := \mathtt{or}(\mathtt{and}(x, y), z)$ because neither $\mathtt{or}(\mathtt{and}(x, y), z)$ or $\mathtt{and}(x, y)$ is a variable, and updates to variables can only be written with other variables on the right hand side.

Part 1 (10 points). 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 $s_l = \mathtt{0x15316000}$ and $s_h = \mathtt{0x15316fff}$, so the memory sandbox is contained in the range of addresses $\mathtt{0x15316000} - \mathtt{0x15316fff}$.

$$x := \mathtt{Mem}(y) \qquad \text{becomes} \qquad \begin{array}{l} \mathtt{and}(y, \mathtt{0x15316fff}) \\ \mathtt{or}(y, \mathtt{0x15316000}) \\ x := \mathtt{Mem}(y) \end{array}$$

$$\begin{array}{lll} & & \texttt{and}(x, \texttt{0x15316fff}) \\ \texttt{Mem}(x) := y & \text{becomes} & \texttt{or}(x, \texttt{0x15316000}) \\ & & \texttt{Mem}(x) := y \end{array}$$

Additionally, we want to prevent indirect jumps from leaving a code sandbox restricted to the range of instruction addresses `0x00000a00` − `0x00000aff`. So each indirect jump is rewritten as follows.

$$\begin{array}{lll} & & \texttt{and}(x, \texttt{0x00000aff}) \\ \texttt{if}(Q)\,\texttt{jump}\,x & \text{becomes} & \texttt{or}(x, \texttt{0x00000a00}) \\ & & \texttt{if}(Q)\,\texttt{jump}\,x \end{array}$$

Any untrusted code is rewritten using these rules prior to being executed. Unfortunately, we were on a tight deadline and didn't have time to prove that this implementation of SFI is secure.

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.

Part 2 (10 points). Propose an alternative implementation in this language for the policy in Part 1 that is secure. You may assume that the untrusted code is not allowed to modify some variables that you select, but be sure to state any assumptions about what invariants must hold of those variables for your implementation to be secure.

4. ***Jailbreak (10 points).*** The `chroot` system call changes the effective filesystem root for the process that calls it. The main purpose of the call is to create a filesystem sandbox before executing an untrusted piece of code, so that after the code is loaded and run it should not be able to reference files outside of the designated directory tree. For example, if a process has as its current working directory `/new_root`, then after calling `chroot("/new_root")` the following program will fail: `open("../etc/passwd")`. This would be like opening `/../etc/passwd`, which fails because `/` is the root of the entire filesystem and there is no such directory as `/..`

   But calling `chroot("/new_root")` does *not* change the process' current working directory to `/new_root`. So if the current working directory is `/tmp`, then `chroot(’/new_root’); open(’../etc/passwd’, O_RDONLY)` will succeed because the filesystem does not traverse the sandbox directory `/new_root` to open `../etc/passwd` from `/tmp`. This is called "breaking the `chroot` jail", and is a common pitfall with implementations of this type of sandbox.

   For this reason, it is crucial that programs calling `chroot` also call `chdir("/")` immediately afterwards, before calling `open` on any filename and before calling `chroot` **again**.

   Provide a security automaton that captures this policy exactly. For full credit, your solution should clearly state which states are initial, what the transition symbols are, and how they correspond to the requirements on system call identifiers and arguments as described in the previous paragraph.

5. ***(Extra Credit)*** **Tough conditions (5 points).** As discussed in lecture, bounded model checking and symbolic execution can be used to find inputs that drive a program down a particular path. It does this by generating the corresponding path condition, and checking it for satisfiability. If the path condition is satisfiable, then it generates a *model*, or satisfying assignment to the variables. When this assignment is used as the input to the program, it will necessarily end up taking the path used to derive the condition.

   However, this is all contingent on being able to first determine the satisfiability of the path condition and then subsequently generating a satisfying assignment. The decision procedures used to do this are subject to the same laws of computability as any other algorithm, and so there is no guarantee that they will be able to provide answers for every path condition.

   Write a short program for which it is unlikely that a decision procedure will be able to produce satisfying assignments to drive execution down at least one path. Your program is allowed to call outside functions, e.g. `Fib`(n) to return the nth Fibonacci number, but be sure to describe precisely what any such external function computes, and why it is unlikely that a decision procedure will be able to solve the resulting path conditions.