Assignment 1: Safety & Proof
15-316 Software Foundations of Security and Privacy

Total Points: 50

1. **The least significant vulnerability (15 points)** In lecture we saw several examples of widely-used memory safety defenses that fall short of providing a rigorous guarantee. But even defenses that seem strong in principle can leave unprotected vulnerabilities if the defense itself is not implemented correctly.

   Consider a memory safety policy that aims to restrict an untrusted program from reading or writing outside the range `0x8000300`–`0x8000400`. Such a policy would be important on a platform that does not provide a virtual memory system, so all processes must share the same address space, and each is given a specific range to work with. To enforce this policy, a runtime monitor interprets each statement of a C-like language, keeping track of the values of variables and pointers. Whenever a statement would cause a pointer to be dereferenced, the monitor checks its state to determine the address that the pointer refers to, and terminates the untrusted program if executing the statement would violate the above policy.

   For example, suppose that a malicious program author knew that a variable `y` had been allocated at `0x80003F2`, `0xF` bytes below the top of the policy region `0x8000400`. The following attempt to violate the policy by reading as many bytes past `y` would fail. As shown on the right, the monitor keeps track of each variable's address and value, so before a statement executes the monitor can terminate the program, as it would do before the third line is executed: the monitor's state reflects that `x` points to `0x8000401`, so dereferencing this pointer would violate the policy.

```
int *x = &y;
x += 0xF;
printf("Secret is: %d\n", *x);
```

	Monitor state			
	`x`	`&x`	`y`	`&y`
initial	`0x0000000`	`0x8000300`	`0x0000000`	`0x80003F2`
after line 1	`0x80003F2`	`0x8000300`	`0x0000000`	`0x80003F2`
after line 2	`0x8000401`	`0x8000300`	`0x0000000`	`0x80003F2`

   However, while the monitor was created to enforce this policy on programs that use 64-bit integers, it was accidentally compiled with a 32-bit compiler. This means that the monitor's state is stored in 32-bit words, and so the monitor's state for every variable and address is truncated to 32 bits.

   Show how this can be exploited by providing a program that will violate the policy when executed alongside the buggy monitor, by writing to address `0x8000480`. If your exploit cannot write to `0x8000480`, but it can write to a different address that violates the policy, then identify the concrete address that it can write to. Remember to also explain in clear language how your program takes advantage of the bug, and why you think that it will violate the policy. If it clarifies your answer, please provide a statement-by-statement summary of the monitor's state as in the example above.

2. **Peirce's arrow (25 points)** Peirce's arrow is a binary operation over propositions that is true exactly when both of its arguments are false. Formally, its semantics are:

$$I \models P \downarrow Q \text{ iff } I \not\models P \text{ and } I \not\models Q$$

First, provide inference rules for Peirce's arrow by filling the elided content below (5 points).

$$(\downarrow\text{L}) \ \frac{\cdots}{\Gamma, P \downarrow Q \vdash \Delta} \qquad (\downarrow\text{R}) \ \frac{\cdots}{\Gamma \vdash P \downarrow Q, \Delta}$$

Make sure that there are no propositional connectives like conjuction, disjunction, and negation in the premises of your rules. For example, the following rule would be invalid because it uses conjunction and negation in the premise:

$$(\downarrow\text{L}) \ \frac{\Gamma, \neg P \wedge \neg Q \vdash \Delta}{\Gamma, P \downarrow Q \vdash \Delta}$$

Then, prove that your rules are sound by arguing rigorously, and separately for each rule, that the validity of the premises implies the validity of the conclusion (20 points).

Solution.

3. **Functional completeness (10 points)** There is more to Peirce's arrow than initially meets the eye: it is one of only two propositional operators that is *functionally complete*, so it plays an important role in several practical applications of logic. Functional completeness means that all possible operations over propositions can be defined in terms of Sheffer stroke. Thus, we would not lose anything essential if we were to insist that formulas only contain atomic propositions and $\downarrow$ connectives, and this would allow us to define semantics and proof rules for just this one operator.

   The following equivalences show how the propositional connectives covered in lecture can be expressed in this way.

negation $\neg P \leftrightarrow P \downarrow P$	*implication* $P \rightarrow Q \leftrightarrow ((P \downarrow P) \downarrow Q) \downarrow ((P \downarrow P) \downarrow Q)$
conjunction $P \wedge Q \leftrightarrow (P \downarrow P) \downarrow (Q \downarrow Q)$	*disjunction* $P \vee Q \leftrightarrow (P \downarrow Q) \downarrow (P \downarrow Q)$

   Show that the above equivalence for disjunction ($\vee$) is valid by using your rules from the previous question to write a sequent calculus proof. Note that it can be difficult to typeset proofs that use $\leftrightarrow$R, so you may break your answer into two proofs instead, showing $P \vee Q \rightarrow (P \downarrow Q) \downarrow (P \downarrow Q)$ and $(P \downarrow Q) \downarrow (P \downarrow Q) \rightarrow P \vee Q$ as separate trees.

   Solution.