Note: You may take no more than five late days on this homework.

Problem 1 Consider the following function, similar to bar in Project One’s target1:

```c
int bar(char *arg)
{
    char buf[20];
    strcpy(buf, arg);
    return 0;
}
```

When compiled with the Boxes compiler, this function disassembles as follows:

```
0x08048434 <bar+0>:       push %ebp
0x08048435 <bar+1>:       mov %esp,%ebp
0x08048437 <bar+3>:       sub $0x28,%esp
0x0804843a <bar+6>:       mov 0x8(%ebp),%eax
0x0804843d <bar+9>:       mov %eax,0x4(%esp)
0x08048441 <bar+13>:      lea -0x14(%ebp),%eax
0x08048444 <bar+16>:      mov %eax,(%esp)
0x08048447 <bar+19>:      call 0x8048348 <strcpy@plt>
0x0804844c <bar+24>:      mov $0x0,%eax
0x08048451 <bar+29>:      leave
0x08048452 <bar+30>:      ret
```

On the x86, this function is vulnerable to a buffer overflow attack. Now suppose that
the x86 architecture were modified so that the stack grew \textit{upwards}: a push instruction
has the effect of increasing the value of the stack pointer by 4, not decreasing it, and
other stack-manipulating instructions behave accordingly.

(a) Give the assembly for the function \texttt{bar} if compiled on the stack-upwards x86.

(b) Give a stack diagram showing the contents of the stack around \texttt{bar} immediately
    after the call-\texttt{strcpy} instruction is executed by \texttt{bar} in the stack-upwards x86.

(c) Is target1 still vulnerable to the \texttt{strcpy} buffer overflow on the stack-upwards x86?
    Why or why not?
Problem 2  Ken Thompson’s paper, “Reflections on Trusting Trust,” describes a technique for installing an undetectable login backdoor by adding a second backdoor to the compiler. The backdoored compiler inserts the appropriate backdoors when compiling the login program and the compiler itself. Once the binary of the compiler, used for bootstrapping future systems, implements the backdoor, any trace of tampering can be removed from the source. In this problem, we will explore a technique for detecting such an attack.

Assume we have two C-language compilers: GCC and Clang. We suspect that nefarious hackers have inserted the Thompson backdoor into the GCC binary on our Linux system, but we believe that these hackers are haven’t managed to corrupt the faculty and students at UIUC responsible for Clang. The entire Linux system doesn’t yet build with Clang, since many programs were written to expect GCC’s extensions to the C language. But we can get Clang to build GCC. (The usual procedure is to compile a new version of GCC using GCC itself; this is known as “self-hosting.”)

Describe how we can reliably detect the presence of a GCC backdoor using the fact that we can compile GCC with Clang.

Problem 3  We would like a firewall ruleset that (1) allows outgoing HTTP (TCP port 80) traffic, but (2) disallows outgoing SSH (TCP port 22) traffic, (3) blocks all other incoming or outgoing connections, and (4) also performs ingress and egress filtering. Assume that the internal network has IP addresses 60.70.*.*.

(a) Using the firewall rule notation given in class, specify a firewall ruleset that implements the policy above.

(b) Suppose users inside the firewalled network wish to make SSH connections to the outside world even though they’re not allowed by policy. Suppose that the administrator of the server (outside the firewall) they’d like to SSH to is cooperative. How can these users and the SSH server administrator conspire to defeat the firewall policy?

(c) What can the firewall administrator do about this?