Question 1  

For the following code, assume an attacker can control the value of `basket` passed into `eval_basket`. The value of `n` is constrained to correctly reflect the number of elements in `basket`.

The code includes several security vulnerabilities. **Circle three such vulnerabilities** in the code and **briefly explain** each of the three.

```
struct food {
  char name[1024];
  int calories;
};

/* Evaluate a shopping basket with at most 32 food items.
   Returns the number of low-calorie items, or -1 on a problem. */
int eval_basket(struct food basket[], size_t n) {
  struct food good[32];
  char bad[1024], cmd[1024];
  int i, total = 0, ngood = 0, size_bad = 0;
  if (n > 32) return -1;
  for (i = 0; i <= n; ++i) {
    if (basket[i].calories < 100)
      good[ngood++] = basket[i];
    else if (basket[i].calories > 500) {
      size_t len = strlen(basket[i].name);
      snprintf(bad + size_bad, len, "%s", basket[i].name);
      size_bad += len;
    }
    total += basket[i].calories;
  }
  if (total > 2500) {
    const char *fmt = "health-factor — calories %d — bad-items %s";
    fprintf(stderr, "lots of calories!");
    snprintf(cmd, sizeof cmd, fmt, total, bad);
    system(cmd);
  }
  return ngood;
}
```

**Reminders:**

- `strlen` calculates the length of a string, not including the terminating '\0' character.
- `snprintf(buf, len, fmt, ...)` works like `printf`, but instead writes to `buf`, and won't write more than `len - 1` characters. It terminates the characters written with
• system runs the shell command given by its first argument.

**Solution:** There are significant vulnerabilities at lines 15/17, 20, and 31.

Line 15 has a fencepost error: the conditional test should be \( i < n \) rather than \( i \leq n \). The test at line 13 assures that \( n \) doesn’t exceed 32, but if it’s equal to 32, and if all of the items in `basket` are “good”, then the assignment at line 17 will write past the end of `good`, representing a buffer overflow vulnerability.

At line 20, there’s an error in that the length passed to `snprintf` is supposed to be available space in the buffer (which would be `sizeof bad - size_bad`), but instead it’s the length of the string being copied (along with a blank) into the buffer. Therefore by supplying large names for items in `basket`, the attacker can write past the end of `bad` at this point, again representing a buffer overflow vulnerability.

At line 31, a shell command is run based on the contents of `cmd`, which in turn includes values from `bad`, which in turn is derived from input provided by the attacker. That input could include shell command characters such as pipes (`'|'`) or command separators (`'::'`), facilitating command injection.

Some more minor issues concern the `name` strings in `basket` possibly not being correctly terminated with `\0`'s, which could lead to reading of memory outside of `basket` at line 19 or line 20.

Note that there are no issues with format string vulnerabilities at any of lines 20, 29, or 30. For each of those, the format itself does not include any elements under the control of the attacker.
Consider the following C code:

```c
int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        if (issafe(s[j])) {
            s[i] = s[j];
            i++; j++;
        } else {
            j++;
        }
    }
    return i;
}

int issafe(char c) {
    return ('a' <= c && c <= 'z') || ('0' <= c && c <= '9') || (c == '.');
}
```

We’d like to know the conditions under which `sanitize` is memory-safe, and then prove it. On the next page, you can find the same code again, but with blank spaces that you need to fill in (a-f). You don’t need to prove the safeness of `issafe`.

Recall our proving strategy from lecture:

1. Identify each point of memory access
2. Write down the precondition it requires
3. Propagate the requirement up to the beginning of the function

*Hint:* Propagating the requirement up to the beginning of the function is more involved than in lecture. Here you need to reason about the properties that hold about the array indices after they are modified.
/* (a) requires: */

int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {

        /* (b) invariant: */

        if (issafe(s[j])) {

            /* (c) invariant: */

            s[i] = s[j];
            i++; j++;

            /* (d) invariant: */
        } else {
            j++;

            /* (e) invariant: */
        }
    }

    /* (f) invariant: */

    return i;
}
Solution:

Note: you might be able to develop the necessary invariants using simpler or more direct analytical approach. Here we strive to proceed very methodically using a large number of small steps, as that’s the most clear and least error-prone process. Such care can become particularly important when attempting to reason about more complex code. But if you can work out proofs correctly in a more direct manner, that’s fine. The bottom line is making sure you have a correct proof!

Since the goal is to prove memory safety, the first step is to identify points in the code that can violate this property, and write down the preconditions for these points of interest. This step establishes what we ultimately need to prove.

In this particular example, only the array accesses $s[i]$ or $s[j]$ can violate the memory safety property. Let’s write down the preconditions for statements involving these memory accesses.

```c
/* requires: s!= NULL && 0 <= j < size (s) */
issafe(s[j])
```

Note that we only wrote down the condition for this particular array access, ignoring everything else in the program. Similarly,

```c
/* requires: s != NULL && 0 <= j && j < size (s) */
issafe(s[j])
```

We wrote the above preconditions using compound relationals (like $a \leq b < c$). While more succinct, those are actually harder to reason about because they involve more than one condition, and during the reasoning process we may know that part of the compound holds, but not yet be sure we have the right form for the other part. So we rewrite them in expanded form:

```c
/* requires: s!= NULL && 0 <= j && j < size (s) */
issafe(s[j])
```

...  

```c
/* requires: s != NULL && 0 <= i && i < size (s) */
s[i] = s[j]
```

Now, we need to propagate the implications of these requirements up to the function precondition. That is, given we need those preconditions to hold, what does that ultimately translate into in terms of the precondition for calling the function? When dealing with loops, this propagation step involves reasoning about code that comes after the memory accesses as well.

The procedure for propagation is to walk through the code writing invariants at each point in the code. Recall that an invariant is what is guaranteed to be true of the state of variables whenever we reach that point in execution. In developing these invariants, we will establish elements of them by imposing preconditions earlier in
the code’s execution, ultimately up to the calling of the function itself. Doing so can take repeated passes through a set of candidate invariants, firming up each element in an invariant as we can reason it must hold, or perhaps adjusting it if we discover we didn’t get it quite right the first time.

How do we construct the invariants? One point of confusion is: “for some invariants, isn’t part of the invariant what we are trying to prove?” For example, how do we know what state of variables we can guarantee to hold at point (b) in this code? We start off with a framing of what we believe the invariant will look like, marking each uncertain element with “?”, and then set about successively examining the invariants to determine which parts we can directly establish. Thus, we start off with parts marked does it hold?, and update those to it holds as we work through the code.

Let’s now walk through the code and illustrate the above process.

Step 1:

(a) To begin, we don’t have any clear picture yet regarding the precondition for the function, so this is simply “?”. We will come back to this later once we have worked out what needs to be propagated.

(b) The loop condition ensures \( j < n \). This will clearly always hold at this point. Note that while the loop initialization ensures that the first time executing we have \( i = j = 0 \), that does not necessarily hold for subsequent iterations (indeed, we would not expect it to), so we don’t include that fact in our draft invariant here.

So for this part, for our first step we have the precondition required by the next statement, marked as uncertain, and the fact ensured by the loop condition, which is not uncertain.

(c) The preceding if statement doesn’t modify \( j \), so we can include the loop condition at this point too. The rest of our candidate invariant here comes from the precondition for the assignment.

(d) Here we take the invariant from above and update it to reflect that \( i \) and \( j \) have been incremented. (There’s actually a subtle point here: incrementing can cause variables to overflow. Therefore, what we’ve written down here won’t in fact necessarily hold. We return to this point later, as an example of how we can sometimes have to fix up a candidate invariant when we discover we can’t prove it.)

(e) The reasoning here is similar to that for (d), except only \( j \) has been incremented.

(f) At this point, we take what we can logically establish given that we know either (d) holds or (e) holds, but we don’t know which one. Thus, our candidate invariant is the minimum that follows from both of them, giving us something that is true regardless of which of the if-else branches is taken. (This is why our
candidate invariant for (f) doesn’t make mention of constraints on i, since (e) doesn’t provide any such constraints.)

/* (a) requires: ? */

int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: j < n && 0 <= j? && j < size(s)? && s != NULL? */
        if (issafe(s[j])) {
            /* (c) Invariant: j < n && 0 <= i? && i < size(s)?
               && 0 <= j? && j < size(s)? && s != NULL? */
            s[i] = s[j];
            i++; j++;
            /* (d) Invariant: j <= n && 0 < i? && i <= size(s)?
               && 0 < j? && j <= size(s)? && s != NULL? */
        } else {
            j++;
            /* (e) Invariant: j <= n && 0 < j?
               && j <= size(s)? && s != NULL? */
        }
    }
    /* (f) Invariant: j <= n && 0 < j?
        && j <= size(s)? && s != NULL? */
    }
    return i;
}

Step 2: Right now there are a lot of “?”s. We can do a bit of initial simplification regarding s != NULL?. Since the code never alters s itself, the only way to ensure that s != NULL holds is that it’s a precondition for calling the function. That gives us this simple update:

/* (a) requires: s != NULL && ? */

int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
while (j < n) {
    /* (b) Invariant: j < n && 0 <= j? && j < size(s)? && s != NULL */
    if (issafe(s[j])) {
        /* (c) Invariant: j < n && 0 <= i? && i < size(s)?
        && 0 <= j? && j < size(s)? && s != NULL */
        s[i] = s[j];
        i++; j++;
        /* (d) Invariant: j <= n && 0 < i? && i <= size(s)?
        && 0 < j? && j <= size(s)? && s != NULL */
    } else {
        j++;
        /* (e) Invariant: j <= n && 0 < j? && j <= size(s)? && s != NULL */
    }
    /* (f) Invariant: j <= n && 0 < j? && j <= size(s)? && s != NULL */

    return i;
}

That is, we added to the function’s precondition, and removed the “?”s around the
s != NULL parts of the invariants, since given the precondition, we’re now confident
that those hold.

Step 3: The next simplification is to observe that we often want to establish condi-
tions like 0 <= i? or 0 < i?. The first of these follows immediately because i and
j are type size_t, so we can mark those as resolved (remove the “?”).

Note that that observation does not quite allow us to do away with the second “?”,
because simply the fact that i is of type size_t does not guarantee that it is strictly
larger than 0. Indeed, even if it was larger than zero before being incremented, it
might overflow and become zero. That observation suggests that some of our original
candidate invariants had flaws (things we might not be able to prove). Accordingly,
we decide to relax those invariants that have conditions like to 0 < i? to instead have
0 <= i?. These latter then can be changed to 0 <= i because of the observation
that i’s type is size_t.

In general, this notion of making an invariant more relaxed is an important reason-
ing technique to keep in mind. Sometimes the immediate constraint implied by a
statement is more narrow than what we need for our overall reasoning. Note that
“relax” means “make broader”; you have to ensure that relaxing the constraint has
not undermined the original preconditions for memory safety.
Another note: had the variables instead been of type int, then the reasoning here would have been more difficult. Indeed, we would need to resort to induction to reason about requirements that ensure that the condition holds. We will see an example of using induction below in Step 8.

This leads then to:

```c
int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: j < n && 0 <= j && j < size(s)? && s != NULL */
        if (issafe(s[j])) {
            /* (c) Invariant: j < n && 0 <= i && i < size(s)? && 0 <= j && j < size(s)? && s != NULL */
            s[i] = s[j];
            i++; j++;
        } else {
            j++;
            /* (e) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */
        }
        /* (d) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */
    } else {
        j++;
        /* (e) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */
    }
    /* (f) Invariant: j <= n && 0 <= j && j <= size(s)? && s != NULL */
    return i;
}
```

Step 4: When we inspect the above, what’s left are questions about the relationship between i and j to size(s). This part requires some thought. In particular, we have some candidate invariants inside the loop that want to provide conditions on i, but the candidate invariants at the top of the loop and at its bottom do not discuss i. That’s going to make it hard to reason about properties of i inside the loop. So we extend the other candidate invariants to include information about i as well, by propagating the conditions we need inside the loop. This gives us:

```c
/* (a) requires: s != NULL && ? */
```
int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: j < n && 0 <= i && i < size(s)?
           && 0 <= j && j < size(s)? && s != NULL */
        if (issafe(s[j])) {
            /* (c) Invariant: j < n && 0 <= i && i < size(s)?
               && 0 <= j && j < size(s)? && s != NULL */
            s[i] = s[j];
            i++; j++;
        } else {
            j++;
            /* (d) Invariant: j <= n && 0 <= i && i <= size(s)?
               && 0 <= j && j <= size(s)? && s != NULL */
        } else {
            j++;
            /* (e) Invariant: j <= n && 0 <= i && i <= size(s)?
               && 0 <= j && j <= size(s)? && s != NULL */
        }
        /* (f) Invariant: j <= n && 0 <= i && i <= size(s)?
           && 0 <= j && j <= size(s)? && s != NULL */
    }
    return i;
}

This change may strike you as obvious, and you may be wondering why we didn’t include these statements about i in those other candidate invariants from the start. The reason is that initially we didn’t have any particular requirements for memory safety at those points that directly needed a precondition on i. It’s only as we try to establish properties on i inside the loop that we now see we have to reason about i at the loop’s beginning and end. (Including i from the get-go would certainly be a reasonable short-cut to take, as long as you’re always careful about verifying all of the steps in your reasoning.)

Step 5: Now we get to the crux of it. In (b), we have an invariant that we want to know always holds when beginning a new loop iteration. To repeat it:

(b) Invariant: j < n && 0 <= i && i < size(s)?
    && 0 <= j && j < size(s)? && s != NULL */
How can we establish that the “?” will hold? For i, it’s not clear how to proceed, but we note that we have some additional information already established about j, namely \( j < n \) (first clause in the invariant). So if we happen to have \( n \leq \text{size}(s) \) then it logically follows that \( j < \text{size}(s) \).

Here we can inspect the code and note that \( n \) never changes. Thus, if we’d like \( n \leq \text{size}(s) \), that will need to be a precondition for calling the function.

This gives us:

```c
/* (a) requires: \( s \neq \text{NULL} \) && \( n \leq \text{size}(s) \) && ? */
int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: \( j < n \) && 0 <= i && i < \text{size}(s)\)?
            && 0 <= j && j < \text{size}(s) && s \neq \text{NULL} */
        if (issafe(s[j])) {
            /* (c) Invariant: \( j < n \) && 0 <= i && i < \text{size}(s)\)?
                && 0 <= j && j < \text{size}(s) && s \neq \text{NULL }*/
            s[i] = s[j];
            i++; j++;
        } else {
            j++;
        }
        /* (d) Invariant: \( j <= n \) && 0 <= i && i <= \text{size}(s)\)?
            && 0 <= j && j <= \text{size}(s) && s \neq \text{NULL }*/
    }
    return i;
}
```

Step 6: All that’s left is that pesky \( i < \text{size}(s) \)!. Here we need to bring a bit of reasoning to the table: why do we think it’ll be the case that \( i \) never exceeds \( \text{size}(s) \)? Well, because it looks like \( i \) never exceeds \( j \), and we know (now) that
j never exceeds \text{size}(s)$. So we \textit{add to our candidate invariants} the notion that i never exceeds j, which so far isn’t proven:

\[
\begin{align*}
/* (a) \text{requires: } s \neq \text{NULL} \&\& n \leq \text{size}(s) \&\& ? */ \\
\end{align*}
\]

int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: j < n \&\& 0 \leq i \&\& i < \text{size}(s) \&\& i < j? \\
            \&\& 0 \leq j \&\& j < \text{size}(s) \&\& s \neq \text{NULL } */
        if (issafe(s[j])) {
            /* (c) Invariant: j < n \&\& 0 \leq i \&\& i < \text{size}(s) \&\& i <= j? \\
               \&\& 0 \leq j \&\& j < \text{size}(s) \&\& s \neq \text{NULL } */
            s[i] = s[j];
            i++; j++;
        } else {
            /* (d) Invariant: j <= n \&\& 0 \leq i \&\& i < \text{size}(s) \&\& i <= j? \\
               \&\& 0 \leq j \&\& j <= \text{size}(s) \&\& s \neq \text{NULL } */
            j++;
        }
    } /* (e) Invariant: j <= n \&\& 0 \leq i \&\& i < \text{size}(s) \&\& i <= j? \\
           \&\& 0 \leq j \&\& j <= \text{size}(s) \&\& s \neq \text{NULL } */
}

/* (f) Invariant: j <= n \&\& 0 \leq i \&\& i < \text{size}(s) \&\& i <= j? \\
       \&\& 0 \leq j \&\& j <= \text{size}(s) \&\& s \neq \text{NULL } */

return i;
}

Step 7: At this point, it’s helpful to review some of the internal logic for this new condition. Let’s assume for the moment that (b) has been fully established. Then (c) follows immediately, because there are no changes to i or j at the intervening step. (d) also follows immediately, since both i and j are changed by the same amount.

(e) is trickier. j has increased, but i hasn’t. Normally we might think well surely then if we had i <= j going into that increment, we will have it after doing the increment of j. But here we have to consider \textit{overflow}, always a subtle part of reasoning about code. The key insight is that for j to overflow, it must have been equal to the largest expressible unsigned value just prior to the overflow. However, another part
of (b) (which remember we’re assuming for now has been fully established) tells us 
\( j < \text{size}(s) \). Since the size of a memory region can’t exceed what’s expressible in 
a \text{size}_t variable, that condition ensures that \( j \) is \textsf{less} than the largest expressible 
unsigned value. Therefore, overflow can’t occur; and therefore, we will have \( i \leq j \) 
at (e) if (b) has been fully established.

Given that reasoning for (d) and (e), we know we then also have it at (f).

Step 8: Almost done! Okay, how do we establish that \( i \leq j \) indeed holds at (b)?
When reasoning about invariants at the beginning of loops, we can need to employ 
\textit{induction}. The \textit{base case} is the first entry into the loop. Here, we can use the loop’s 
initialization, which is that both \( i \) and \( j \) are 0. For that, we have \( i \leq j \) — so far, 
so good.

The inductive step has two parts. First, we need to show that if we assume that the 
invariant at the beginning of the loop holds, then given that we can show that the 
invariant at the end of the loop (f) holds. It should be straightforward for you to 
confirm that this is indeed the case. Second, we need to show that if we assume the 
invariant at the end of the loop \textit{plus} the loop condition, then we can establish that 
the invariant at the beginning of the loop holds. That is, we assume both:

\[
\begin{align*}
\text{(f) Invariant: } & j \leq n \land 0 \leq i \land i < \text{size}(s) \land i \leq j \\
& \land 0 \leq j \land j < \text{size}(s) \land s \neq \text{NULL} \ *
\end{align*}
\]

and \( j < n \). Again, it should be clear upon inspection (including the precondition 
that \( n \leq \text{size}(s) \)) that these two when combined indeed give us:

\[
\begin{align*}
\text{(b) Invariant: } & j < n \land 0 \leq i \land i < \text{size}(s) \land i \leq j \\
& \land 0 \leq j \land j < \text{size}(s) \land s \neq \text{NULL} \ *
\end{align*}
\]

That completes the reasoning. We’ve established by induction that \( i \leq j \) holds in 
(b), and from that and \( j < \text{size}(s) \) we have also established \( i < \text{size}(s) \). So we 
don’t need any more preconditions for the function, and the final answer is:

\[
\begin{align*}
\text{(a) requires: } & s \neq \text{NULL} \land n \leq \text{size}(s) \\
\end{align*}
\]

---

```c
int sanitize(char s[], size_t n) {
    size_t i = 0, j = 0;
    while (j < n) {
        /* (b) Invariant: j < n \land 0 \leq i \land i < \text{size}(s) \land i \leq j \\
        \land 0 \leq j \land j < \text{size}(s) \land s \neq \text{NULL} \ */
        if (issafe(s[j])) {
            /* (c) Invariant: j < n \land 0 \leq i \land i < \text{size}(s) \land i \leq j \\
            \land 0 \leq j \land j < \text{size}(s) \land s \neq \text{NULL} \ */
        }
    }
}
```
s[i] = s[j];
i++; j++;

/* (d) Invariant: j <= n && 0 <= i && i <= size(s) && i <= j
   && 0 <= j && j <= size(s) && s != NULL */
} else {
    j++;

    /* (e) Invariant: j <= n && 0 <= i && i < size(s) && i <= j
       && 0 <= j && j <= size(s) && s != NULL */
}

/* (f) Invariant: j <= n && 0 <= i && i < size(s) && i <= j
   && 0 <= j && j <= size(s) && s != NULL */

} return i;
}

Phew!

We can summarize the overall reasoning strategy using the following template, which is elaborated a bit from the one presented in lecture:

(1) Identify any memory accesses that could have safety issues.

(2) Write down preconditions for these.

(3.1) Write down candidate invariants that will satisfy these preconditions.

(3.2) Repeatedly identify any elements of those invariants that you can directly reason about and mark them as no longer uncertain. This may involve adding preconditions to the function.

(3.3) For what remains, determine what sort of constraints one might add to the candidate invariants to enable proving the missing pieces, and also possibly what conditions to relax (make broader). This may enable going back to 3.2.

(3.4) Use induction to indirectly reason about remaining elements. From here, one can go to 3.2 or 3.3.

(4) Once you’ve resolved all of the candidate invariants, you’re done.

A final note: do not hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.