Week of January 25, 2016

Instructions. We will break into groups to discuss the following questions. Please think of as many solutions as you can. Be original! Maybe you will come up with something no one has thought of yet. Be prepared to talk about your solutions with the rest of the section.

Question 1 Software Vulnerabilities (15 min)
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.

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

/* Evaluate a shopping basket with at most 32 food items. */
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 a `\0`.
- `system` runs the shell command given by its first argument.

**Question 2  Reasoning about Memory Safety  (30 min)**

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 == '\0');
}
```

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;
}