July 1st: Software Vulnerabilities

Question 1 ***TCB (Trusted Computing Base)*** **(10 min)**

In lecture, we discussed the importance of a TCB and the thought that goes into designing it. Answer these following questions about the TCB:

1. What is a TCB?

2. What can we do to reduce the size of the TCB? Which security principles are heavily connected to designing a TCB?

3. What components are included in the (physical analog of) TCB for the following security goals:

   (a) Preventing break-ins to your apartment

   (b) Locking up your bike

   (c) Preventing people from riding BART for free

   (d) Making sure no explosives are present on an airplane

Question 2 ***Memory Vulnerabilities*** **(25 min)**

For the following code, assume an attacker can control the value of `basket` passed into `eval_basket`. They *cannot* input an arbitrary `n`, however: 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 on the next page.

```c
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:

- **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.

1. Explanation:

2. Explanation:

3. Explanation:

Question 3 *C Memory Defenses* (10 min)

Mark the following statements as True or False, and justify your solution.

1. Stack canaries can protect against all buffer overflow attacks in the stack.

2. A format-string vulnerability *alone* can allow an attacker to overwrite a saved return address even when stack canaries are enabled.

3. If you have data execution prevention/executable space protection/NX bit, an attacker can write code into memory to execute.

4. If you have a non-executable stack and heap, buffer overflows are no longer exploitable.

5. If you have a non-executable stack and heap, an attacker can use Return Oriented Programming.

6. If you use a memory-safe language, buffer overflow attacks are impossible.

7. ASLR, stack canaries, and NX bits all combined are insufficient to prevent exploitation of all buffer overflow attacks.

Short answer!

1. What would happen if the stack canary was between the return address and the saved frame pointer? Assume the canary is impenetrable / un-leakable.

2. What if the canary was *above* the return address instead?