Paxson
Spring 2017

CS 161
Computer Security

Discussion 2

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.
   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 a '\0'.
- **system** runs the shell command given by its first argument.

Question 2 *Buffer Overflow Mitigations* (20 min)

Buffer overflow mitigations generally fall into two categories: (1) eliminating the cause and (2) alleviating the damage. This question is about techniques in the second category.

Several requirements must be met for a buffer overflow to succeed. Each requirement listed below can be combated with a different countermeasure. With each mitigation you discuss, think about *where* it can be implemented—common targets include the compiler and the operating system (OS). Also discuss limitations, pitfalls, and costs of each mitigation.

(a) The attacker needs to overwrite the return address on the stack to change the control flow. Is it possible to prevent this from happening or at least detect when it occurs?

(b) The overwritten return address must point to a valid instruction sequence. The attacker often places the malicious code to execute in the vulnerable buffer. However, the buffer address must be known to set up the jump target correctly. One way to find out this address is to observe the program in a debugger. This works because the address tends to be the same across multiple program runs. What could be done to make it harder to accurately find out the address of the start of the malicious code?

(c) Attackers often store their malicious code inside the same buffer that they overflow. What mechanism could prevent the execution of the malicious code? What type of code would break with this defense in place?

Question 3 *Arc Injection* (15 min)

Imagine that you are trying to exploit a buffer overflow, but you notice that none of the code you are injecting will execute for some reason. How frustrating! You still really want to run some malicious code, so what could you try instead?

Hint: In a stack smashing attack, you can overwrite the return address with any address of your choosing.