Jawale & Dutra
Summer 2019

CS 161
Computer Security

Discussion 2

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

Solution:

1. It is the set of hardware and software on which we depend for correct enforcement of policy. If part of the TCB is incorrect, the system's security properties can no longer be guaranteed to be true. Anything outside the TCB isn't relied upon in any way.
2. Privilege separation and separation of responsibility can help reduce the size of the TCB. You will end up with more components, but not all of them can violate your security goals if they break. The size of the TCB can also be reduced by reducing the application's dependency on third-party components and software.
3. (This list is not necessarily complete)
   (a) The lock, the door, the walls, the windows, the roof, the floor, you, anyone who has a key.
   (b) The bike frame, the bike lock, the post you lock it to, the ground.
   (c) The ticket machines, the tickets, the turnstiles, the entrances, the employees.

(d) The TSA employees, the security gates, the ”one-way” exit gates, the fences surrounding the runway area.

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:

   Solution: Line **15** has a *fencepost* error: the conditional test should be $i < n$ rather than $i <= n$. This example is also commonly called an *off-by-one* error, for obvious reasons. 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.

2. Explanation:

   Solution: At line **20**, there's an error in that the length passed to **snprintf** is *supposed* to be available space in the buffer, 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.

3. Explanation:

   Solution: 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.*

Solution: Some more minor issues concern the **name** strings in **basket** possibly not being correctly terminated with $'\backslash 0's$, 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.

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.

Solution:

False, stack canaries can be defeated if they are revealed by information leakage, or if there is not sufficient entropy, in which case an attacker can guess the value. Remember, the attack just needs to work once in the real world.

Additionally, not all buffer overflows target the `rip`: overflows can still modify local variables (which often encode privileges) without overwriting or bypassing the stack canary.

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

Solution:

True, with format string vulnerabilities, the attacker can learn the contents of the stack frame, other parts of memory, and write to other addresses in memory. Stack canaries won't save you here. Moreover, format string vulnerabilities *can* write, as well as read: a well-crafted `%n` format is sufficient to overwrite parts of the stack.

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

Solution:

False, the definition of the NX bit is that it prevents code from being writable and executable at the same time. An attacker who can write code into memory cannot execute it.

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

Solution:

False. While many attacks rely on writing malicious code to memory and then executing them, other types of attacks which still work in these cases. For example, overflows that target local variables, or attacks like Return Oriented Programming (see the next question). If we make writable parts of memory non-executable, some attacks cannot succeed, but many still may.

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

> **Solution:**
>
> True, Return Oriented Programming is a technique that uses existing instructions already in memory to change the original program flow.

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

> **Solution:**
>
> True, buffer overflow attacks do not work with memory safe languages.

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

> **Solution:**
>
> True, all of these protections can be overcome. See Aleph One's article, on Piazza.

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.

> **Solution:**
>
> An attacker can overwrite the saved frame pointer so that the program uses the wrong address as the base pointer after it returns, crashing the program.

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

> **Solution:**
>
> It doesn't stop an attacker from overwriting the return address. Although if an attacker had absolutely no idea where the return address was, it could potentially detect stack smashing.