COM-301 Computer Security
Exercise 7: Software Security

November 17, 2023

Software Security

1. Suppose we are building a web application that asks the user for their email address and stores it in a variable m. We want to invoke the shell to send an email message to the email address m, like this:

```
void sendemail(char *m) {
    char cmd[1024];
    sprintf(cmd,"%s",m);
    f = popen(cmd, "w");
    ...
}
```

   (a) Is this code secure in terms of memory safety?

   (b) What checks would you do on m to ensure that no other problem can happen?

2. In the following code, what is the condition on the lengths of s1 and s2 for it to be safe?:

```
char *concat(char *s1, char *s2) {
char result[1024];

for (i=0; s1[i] != '\0'; ++i)
    result[i] = s1[i];

for (j=0; s2[j] != '\0'; ++j)
    result[i+j] = s2[j];

result[i+j] = '\0'
}
```

3. Find the vulnerabilities in the following code:

```
int main (int argc, char **argv) {
    char *items[] = {"boat", "car", "truck", "train"};
    char *newitems;
    newitems = malloc(100);
    int index = GetUntrustedOffset();
    printf("You selected %s\n", items[index-1]);
    free(newitems);
    printf("The next item in the list is %s \n", items[index]);
    *newitems = items[0];
}
```

4. What are the three properties that a mitigation must have?
5. Are each of the following approaches a mitigation mechanism? Justify.
   (a) Inexecutable stack
   (b) Dynamic library linking: Allowing a program to load an external library
   (c) Sandboxing: Running the process in a isolated space
   (d) Compiling with different optimization flags
6. Symbolic execution and dynamic analysis (fuzzing) are two approaches to automatically find bugs. Symbolic execution provides a full path coverage while fuzzing gives partial coverage. Fuzzing may hit a coverage wall and cannot find samples which lead to new coverage. So why is fuzzing more popular in practice? Is there a way to leverage symbolic execution to get better coverage in fuzzing?
7. Branch coverage is a metric to measure how much of the code was executed. Compared to statement coverage which measures if a statement is executed, branch coverage measures if an edge in the control-flow graph is executed. For each conditional jump, branch coverage measures the outgoing edges that are taken (e.g., for an if condition, branch coverage captures if the **if** or the **else** branch was executed). Note that branch coverage is stateless: this means that each branch only remembers if it has been executed or not.
   (a) Branch coverage is incomplete and does not cover all possible execution paths. Explain why branch coverage cannot cover all paths (hint: branch coverage is stateless, reason about paths, not about individual branches).
   (b) Complete the ? instructions in the example below, of a program that has full branch coverage but incomplete path coverage. Add a memory safety bug (e.g., a buffer overflow or an illegal de-reference such as buf[usr1] = usr2) to the program and provide inputs to the program that result in full branch coverage but do not trigger the bug.

```
int example(bool b1, bool b2) {
    int a = 0;
    char c[2];
    ?
    ?
    return c[a];
}
```

8. Fuzzing is an efficient automatic testing technique that scales to large code bases. Modern fuzzing mechanisms leverage branch coverage to record which parts of the program have been executed, mapping fuzzing inputs to coverage. Coverage-guided fuzzers add any input that triggers new coverage to the pool of inputs to perform a mutation. Additionally, these fuzzers record any input that crashes or hangs the program.

   (a) Assume a new seed covers a new path. Fuzzing will continuously mutate this input to trigger different paths and different data-flow along that path. Why is it necessary to generate alternate data-flows to trigger bugs, i.e., why does it not suffice to only generate new paths? (hint: what is the difference between control-flow and data-flow?)

   (b) Fuzzing frequently hits a so-called "**coverage wall**" where it no longer makes progress (i.e., random mutations do not trigger new coverage). What could be the reason for this limitation? (hint: what types of conditions are hard to satisfy for randomly generated input)

   (c) Fuzzing struggles to find crashes in libraries. What could be the reason for the lack of deep coverage when fuzzing the set of exported library functions? (hint: think about a file I/O library that offers open/read/write/close functions; what happens if you only fuzz the read function without prior calls to open?)

9. Sanitization makes bug detection more likely by enforcing certain policies. Commonly used sanitizers enforce memory safety and detect undefined behavior.

   (a) Is sanitization instrumentation helpful to find all types of bugs? Under what circumstances will sanitization be counterproductive?

   (b) Why is address sanitizer needed to detect memory corruption? Explain why and how buffer overflows can be missed without address sanitizer.

   (c) Address sanitizer detects memory corruption by detecting writes to red-zones (8 byte areas directly adjacent to allocated memory with static data). Why is address sanitizer not a mitigation?

10. Several mitigations exist to make exploitation harder. Mitigations must adhere to strict performance criteria as they are always enabled.

(a) ASLR shuffles the address space for each execution. Why can the address space not easily be reshuffled during execution (e.g., after each system call)? Why would it be useful to reshuffle after each system call (think how many stages an attack will have)?

(b) Data Execution Prevention stops code injection attacks. Initial implementations of data execution prevention leveraged segmentation registers and expensive checks to test if a memory region was executable or not. Modern implementations use a page-based mechanism that leverages a bit in the page table to encode execute permissions. Discuss the key advantage and disadvantage of a page-based solution

(c) Stack canaries protect against buffer overflows on the stack and are prone to information leaks. How could an attacker bypass stack canaries in an exploit?