Analysis

Statically investigate the code and look for common mistakes
• stack overflow
• array out of bounds
• accessing uninitialized variables (using default stack value; or uninitialized variables on heap)
• uninitialized pointer
• boolean instead of bitwise
• incorrect pointer usage
• assign instead of compare

Splint: “secure programming lint”
splint myprog.c

Splint also allows you to use code annotations to help reduce false positives (tell it that you know a problem is actually okay) or to help it find more problems (give additional info beyond what C allows, such as invariants)

Memory problems:
• accessing memory that you shouldn't (outside the current function's stack frame, outside a heap pointer's block, freed memory)
• incorrect freeing (double free)
• overlapping src and dest in memcpy
• memory leaks: when program ends, is memory still reachable, definitely lost (leak), indirectly lost

memcheck: memory analysis tool
cc -g -o myprog myprog.c
valgrind --tool=memcheck ./myprog

use –leak-check=full to get more info about memory leaks (like where memory was malloc'd)

Testing

goals:
• find bugs
• exercise different parts of the program

types of testing:
• black-box: based on the specification; try to find inputs that will lead to incorrect outputs; equivalence partitioning and boundary conditions
• white-box: based on the code; try to find inputs that will exercise different parts of the code

we can think of the code as a directed graph and then consider ways of “covering” that graph: statement coverage, branch coverage, path coverage
gcov is a tool that will report statement coverage
   gcc -fprofile-arcs -ftest-coverage myprog.c -o myprog
   ./myprog (this creates myprog.gcda/data and myprog.gcno/graph)
gcov myprog.c (this creates myprog.c.gcov)
gcov -b myprog.c will also give branch coverage info

**Verification**

Testing can only demonstrate the existence of bugs.
But in some cases we want to prove their absence.

We do this by verifying that the code exhibits certain properties that are a “model” of correctness.
Usually we want to prove that safety properties cannot be violated.

```c
int fun (int a, int c) {
    int k = -1;

    if (a < 0) {
        if (c >= 0)
            k = 10 + c - 2 * a;
        else
            k = a * c + 1;
    }
    else
        k = a * 5;
    return k;
}
```

Can we prove that this function always returns a positive value, i.e. k > 0?
To do that we negate the property: if satisfiable, then the property is violated.
In this example, though, there is no way to solve k <= 0 so the property is proven to be true.

**BLAST**: model checking tool that determines reachability
Write your properties in such a way that if you're able to reach the line, something bad happened
put a label on that line, then run BLAST with the name of the source code, the “main” function
(starting point), and label to look for
   • reachable: program is unsafe
   • not reachable: program is safe
   • unknown: program is unsafe