Lecture 7

maandag 5 december 2022 17:32

"reflect on arguments" question from homework: could be an exam question

pointer analysis usually done in addition to data flow analysis

problems with pointer analysis

assume we use assertions to define a specification

object Circle with variable → radius

```
Circle x = new Circle();
Circle z = ?
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y==1);
```

x may-alias z

result depends on ?

pointer analysis = may-alias analysis

situations like this one make pointer analysis difficult

since there are infinitely many ways of referring to the same memory element

rough solution: simplify program by aggregating groups of pointers into a single pointer

```
char w[100];
   ↓ becomes
char w;
```

if there is pointer aliasing in the original program, then this will still be the case in the simplified program

field-insensitive aggregation: merge all fields of each object $o.F_1 = o.F_2$

field-based aggregation: merge all fields of all objects $o_1.F_1 = o_2.F_1$

field-sensitive: each field of each object kept separate

---

fuzzing: testing 'smartly' for errors
→ there is no 'theory' behind this

+ scales well
+ no false alarms
- more expensive
- not guaranteed to find all bugs
+ does not require code
- test may launch attack (in production environment, so use non-production environment)
- does not (fully) explain cause of bugs

run program with generated input, possibly syntactically/semantically invalid

reduce search space of inputs to 'interesting' e.g. for attacker candidates

testing can be adaptive → previous results influence newly chosen inputs

heuristics:

- testing should be smarter than random/brute force → find input values which are almost good
- code coverage
- configs which found bugs are favorable
- avoid redundancy/useless test cases
- test cases should evaluate sufficiently quickly
- provide feedback to design test cases

code coverage: metric to quantify extent to which a program's code is tested

function coverage: which functions were called?
statement coverage: which statements are executed?
branch coverage: which branches were taken

PUT = program under test

fuzzing: execution of PUT using inputs sampled from input space that protrudes the expected input space of the PUT
fuzz testing: use of fuzzing to test if a PUT violates a security policy

fuzzer: program that performs fuzz testing on a PUT

fuzz campaign: specific execution of a fuzzer on a PUT with a specific security policy