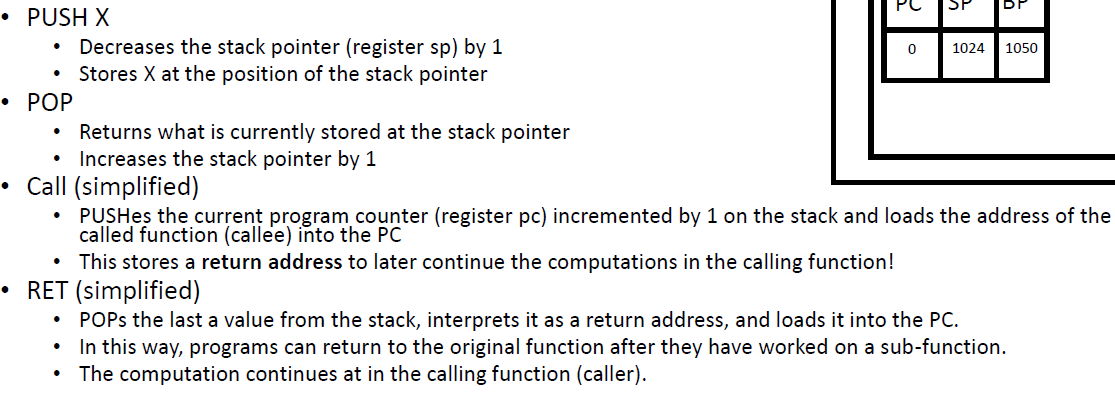
1. **Input Validation and Representation Problems**
2. **API Abuse**
3. **Problems when Using Security Features/Tools**
4. **Parallelism and Consistency Problems**
5. **Error Handling/Output Problems**
6. **Code Quality Problems**
7. **Encapsulation/ Isolation problems**
8. **Dependence on Environment**

IASPEQED



The attacker inputs more characters than the variables are supposed to hold

Insert malicious assembler codes in the sack to overwrite the return address to point to malicious codes.

1. Fat pointers: use the pointer that stores the length of the memory and check it at runtime.

Shortage: decrease the efficiency since it have to be checked at runtime.

1. Stack canaries: add a random value at the stack and its integrity will be checked at runtime.

Shortage: the attacker can insert codes that step-wisely output the contents on the stack.

1. Data execution prevention: mark some of the memories non-executable.

Shortage: just-in-time compilation, machine codes are generated from some intermediate codes which need to be writable and executable.

Return to the lib-c, the attacker can let the codes return to the library codes, using turing-complete gadgets.

1. Address space layout randomization: random the memory areas when the program starts.

Shortage: long nop sequence to increase the probability of execution. Side-channel attack on CPU branch target to leak the addr.

Null dereference

Dalling pointer

Use-after-free double free

Memory leak

The number of format specifiers is different from the arguments. Format string can be generated dynamically. The data are passed by refrence.

%n can cause the number of characters written so far be stored in to the argument

user\_string = xDD xCC xBB x0A %x %x %x %x %x %x %n

**Out-of-bounds write, improper input validation, out-of-bounds read, use after read, null pointer dereference, integer overflow or wrap around, Improper Restriction of Operations within the Bounds of a Memory Buffer, Uncontrolled Resource Consumption,**

Completeness:

A static analysis is complete if it can discover all true properties of the program. In other words, it finds all possible is su es and accurately characterizes all aspects of program behavior.

•

Soundness:

A static analysis is sound if it never produces false positives (no false alarms). In the context of abstract interpretation, this means that if the analysis declares a certain property to be true, then that property is indeed true in the actual program.

If all approximations are under approximations we may preserve completeness.

Algorithm 1: Fuzz testing

Black box Fuzzer

**Preprocess**

**Instrumentation, seed selection, seed trimming, Driver programming (only once)**

**Schedule**

**Goal: analyze currently available information about the configurations and pick a configuration that is likely to lead to the most favorable outcome**

**InputGen**

**White box fuzzers may use access to PUT to find inputs that are accepted e.g. via program analysis**

**InputEval**

**•BugOracles, Execution Optimization, Triage: analyzing and reporting test cases that cause security violations**

**ConfUpdate**

**Evolutionary Algorithm may maintain a seed pool of promising seeds**

**that evolves via biological evolution mechanisms like mutation, recombination,**

**Fuzz Configuration Scheduling (FCS) Problem**

**•time can either be spent on gathering more accurate information on each configuration to inform future decisions (explore) or • on fuzzing the configurations that are currently believed to lead to more favorable outcomes (exploit)**

**the fuzzer taming problem ): process of ranking violating test cases according to uniqueness and severity (determining exploitability of bug)**

**Memory-safe:**

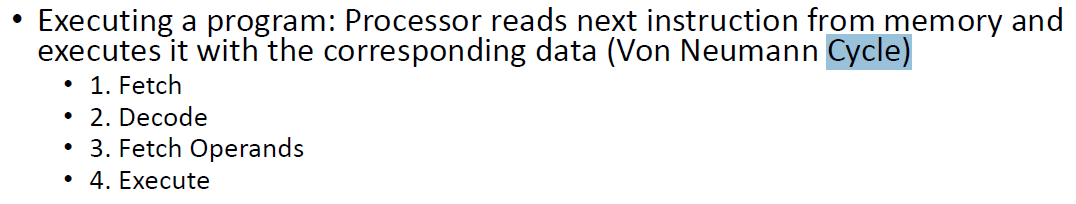
**Pointers only created through standard means**

**Pointers only used to access memories belong to the pointers**

**• Spatial memory safety: pointer does not read or write outside of bounds. • Temporal memory safety: no dangling pointers, i.e. pointers that map to undefined memory regions, no double free**

**Type Safety**

**Operations on the objects are compatible with the type of objects.**



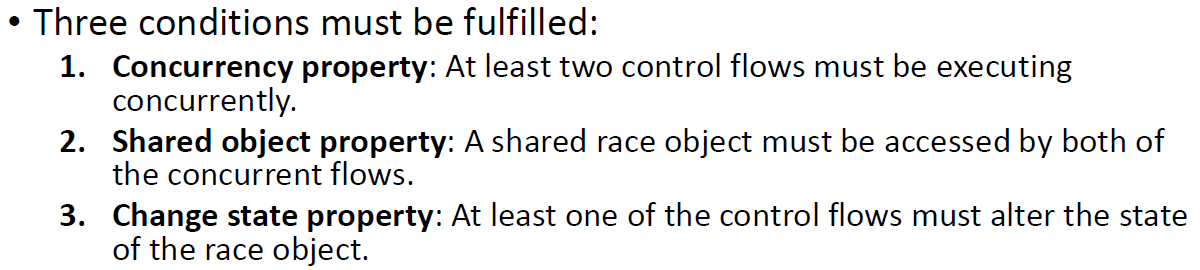
**Fetch decode fetch operands execute**

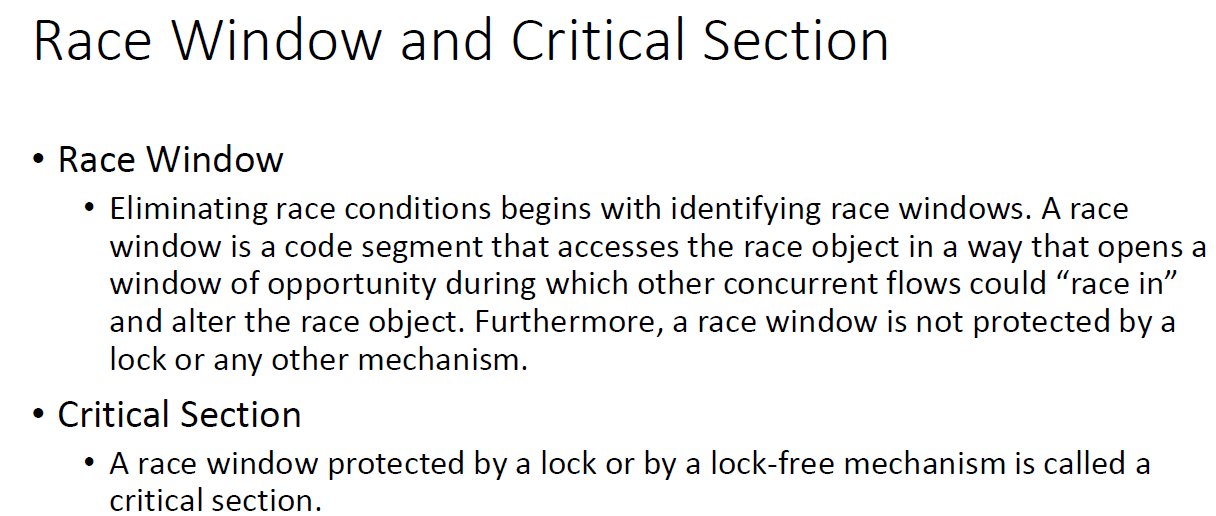
**Fetch decode fetch operands execute**

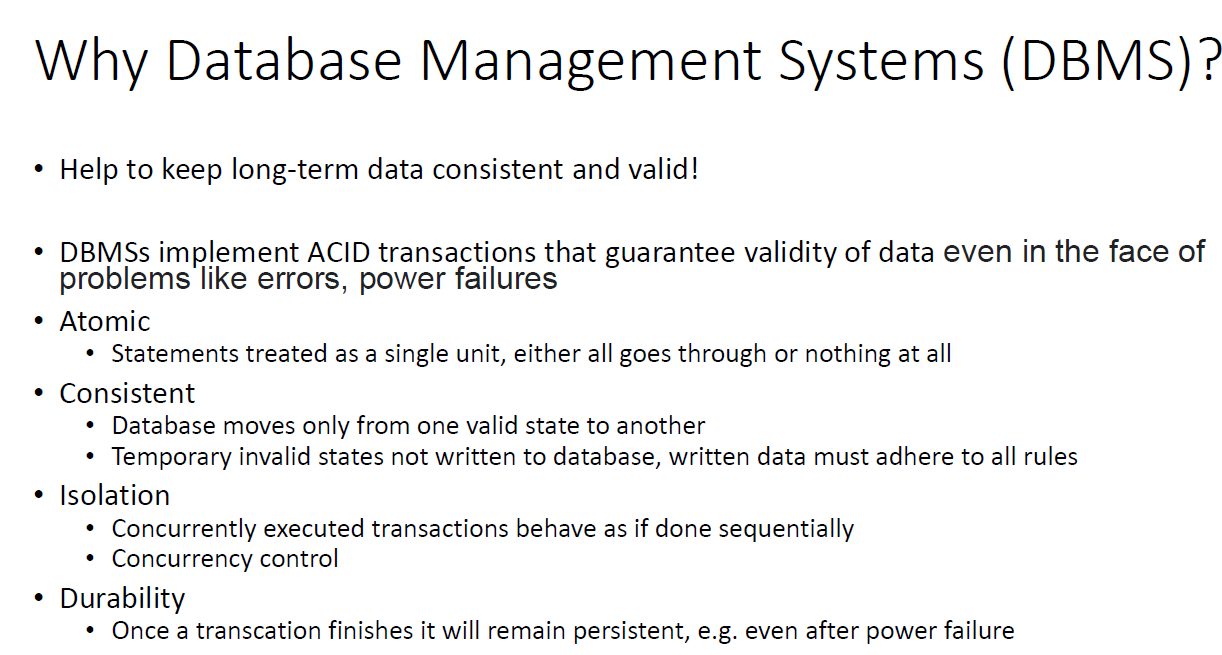
**Fetch decode fetch operands execute**

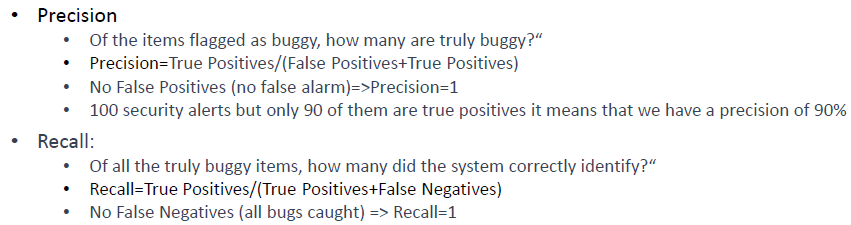
**Amadalh’s law**

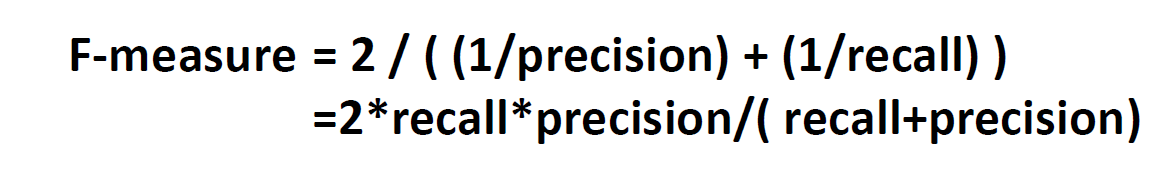
**1/(1-p+p/n)**

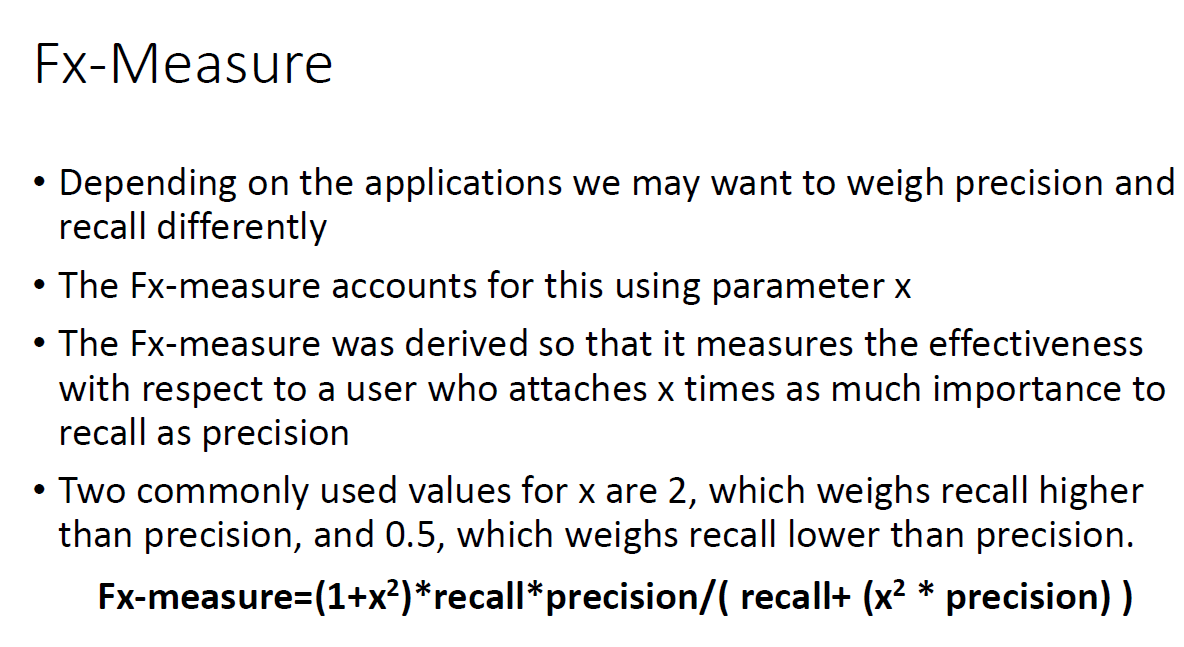












**Very Busy Expressions Analysis**

--> Find expressions that will definitely be evaluated again before its value

•Reduce Code Size

**Live Variables Analysis**

--> A variable is live at a program point if there exists an execution where its value is read later in the execution without it be ing written to in the meantime

•Allocate Registers Efficiently

