Automated Vulnerability Identification and Patch Generation System

Background

- "Shellcode" often means a generic payload for the exploitation, so its goal is to launch an interactive shell (via eg: return to libc/ROP attack) as a result
- A vulnerability is a bug(system isn’t behaving as it’s supposed to) that manifests itself as an opportunity for exploitation
  - Buffer overflow - control flow hijacking attack
  - Format string - allows arbitrary read/write and thus arbitrary execution (fgets() function can read data without a format specifier)
- Preventive measures (Partial)
  - ASLR (Address space layout randomization)
  - Stack Canaries
  - NX stack

Background

- Programs (can open source or not)
- General analyst tools and techniques
  - gdb/pwndbg: dynamic/objdump: static (both disassemble the code)
  - static analysis: cppcheck
  - IDA Pro/Ghidra for reverse engineering
- Dynamic analysis (manual) can identify bugs, but not suitable for large programs
- **Fuzzing** is an automated software testing technique for bug finding

Background - Fuzzing

- Providing invalid, unexpected, or random data as inputs to a target application which is then subsequently monitored for exceptions like overflows or memory leaks
- Based on their purpose, fuzzers can be general purpose or they can be attuned to perform well in certain domains. Each fuzzer has its own strategy (random, context-free grammar, mutation) to generate inputs

Background - Fuzzing Terminologies

- **Corpus** is a set of inputs for the fuzz target
- **Harness** wrapper to fuzz the component of the target
- **Code coverage** is a metric used to judge the degree to which the source code of a program is executed for an application
- Code coverage is essential to the success of coverage-guided fuzzing, and it is often achieved through **instrumentation**.
- By inserting instrumentation code to a program, the fuzzer can track the execution flow exercised by an input and determine whether the input covers a new part of the program (which has not been executed before).
- Instrumentation is expensive and instrumenting every basic block ensures full visibility

Background - Fuzzer Categorization

- Black box : completely unaware of the internal program structure
  - Radamsa
- Grey box : leverages instrumentation to gain information about the program
  - American Fuzzy Lop (AFL) : fork() child processes and feed input
  - libFuzzer (good for detecting memory leaks and out-of-memory bugs)
  - Honggfuzz (suitable for IOT firmware's)
- Have already found many real-world CVEs in a variety of software (CVE-list of AFL-Fuzz, OSS-Fuzz(cloud based)) and already part of SDL of many products
- White-box : leverages program analysis to gain information
  - Symbolic execution

Background - Symbolic execution

- **Symbolic execution** is a "targeted fuzzing" that specifically hits certain symbolic values.
- When we "symbolically" execute a program, a symbolic executor tracks symbolic states rather than concrete input by analyzing the program and generates a set of test cases that can reach (theoretically) all paths existing in the program.
- KLEE well known Symbolic execution engines.

Concrete Execution(Fuzzing)

```
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b < 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

Concrete execution

a = 5 = c = 1
x = y = z = 0
true
x = -2

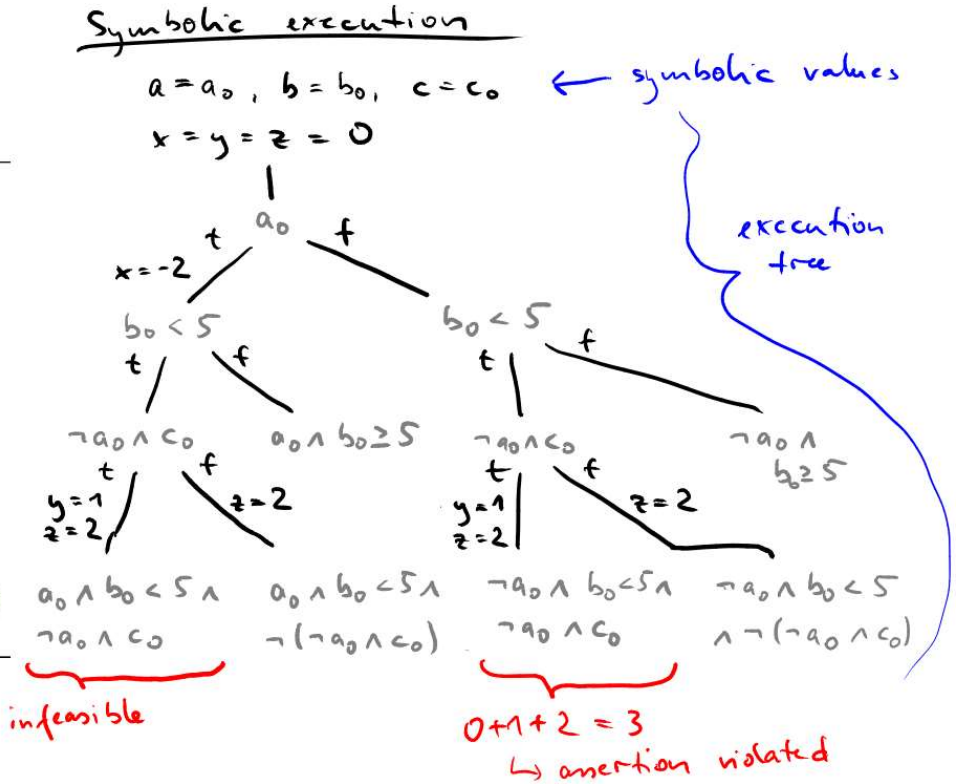
true
false

z = 2

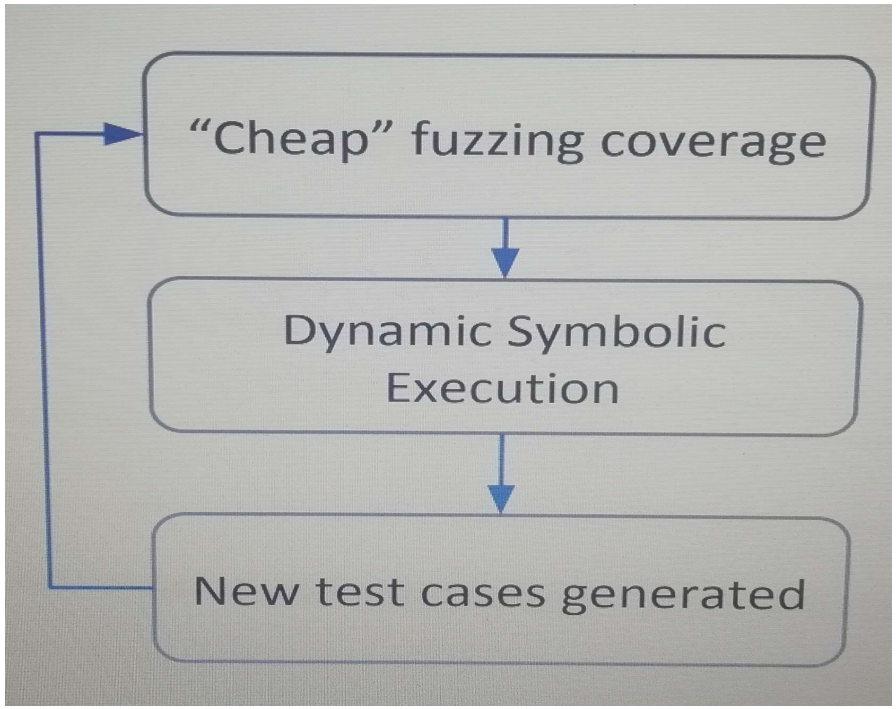
-2 + 0 + 2 ≠ 3 ✓

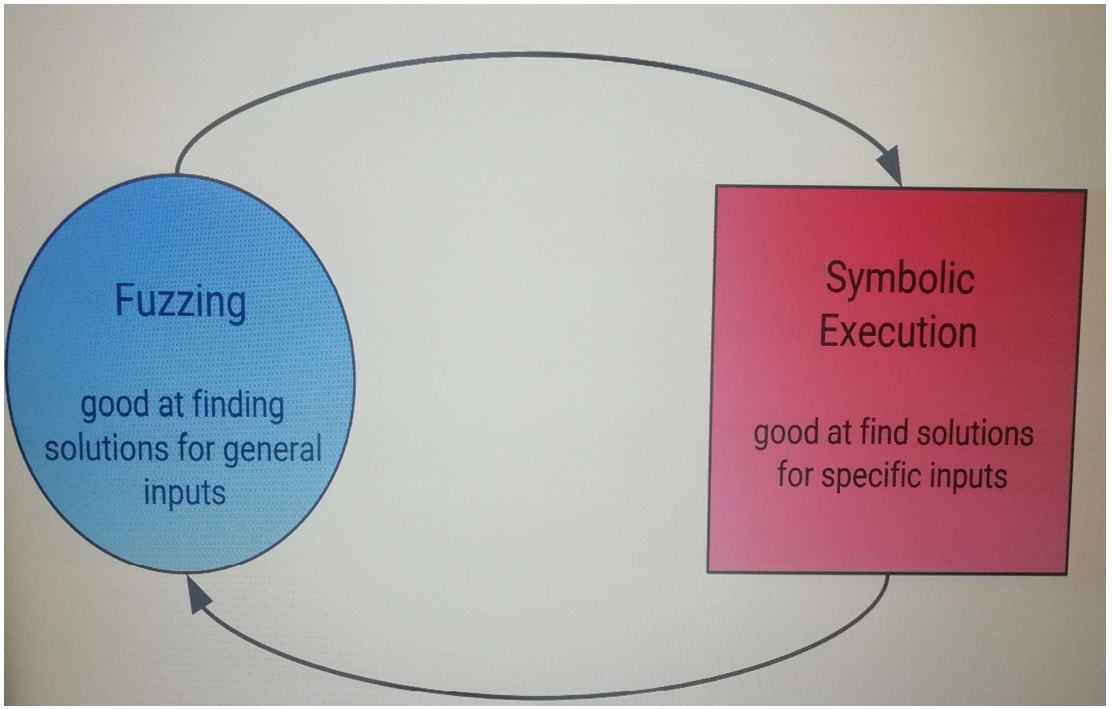
Symbolic Execution

```
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b < 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```



Concolic Execution - Fuzzing + Symbolic Execution





- Eg: Driller = AFL(Fuzzer) + Angr(Symbolic engine)
- Symbolic execution state space reduced, and path explosion reduces

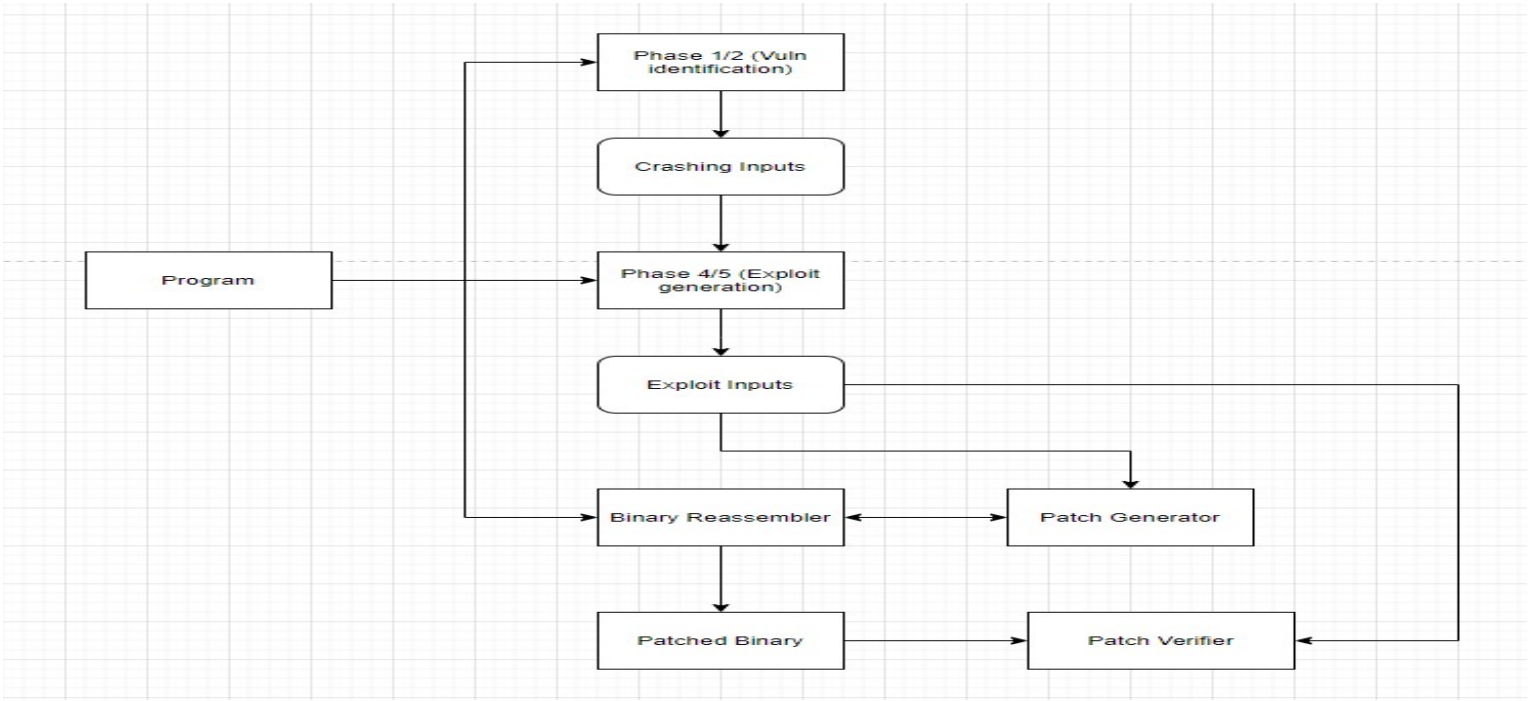
Focused Vulnerabilities

- Heap Overflow – Use after free, Double free etc.
- Stack Overflow
- Format String Vulnerabilities
- Command Injection
- Null pointer dereference
- Integer/float overflow
- Detection of arbitrary read and write primitives

Automatic Exploit Generation

- The automatic exploit generation (AEG) is a problem of finding program inputs that result in a desired exploited execution state
- Most of the analyst starts with a manual/semi automated exploit generation solution and later solution can be extended to a fully automated one
- AEG produces two types of exploits
  - **Return-to-stack Exploit**.
  - **Return-to-lib Exploit**
- Eg : Mayhem

Automated Patch Generation



- Initially we will identify a manual/semi automated patch generation solution and later solution can be extended to a fully automated one
- Explore jump patching, OSSPatcher, approachs taken by Ramblr and 0patch for Microsoft