Fuzzing

Bug Fixing

- Using tools to find bugs
  - Major techniques
  - Some tips on how to use them
- Static Analysis
  - Compile time/source code level
  - Compare code with abstract model
- Dynamic Analysis
  - Run Program/Feed it inputs/See what happens

What is Fuzzing ?

- ➔ Fuzzing or fuzz testing is an automated software testing technique
- ➔ It involves providing invalid, unexpected, or random data as inputs
- ➔ program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks

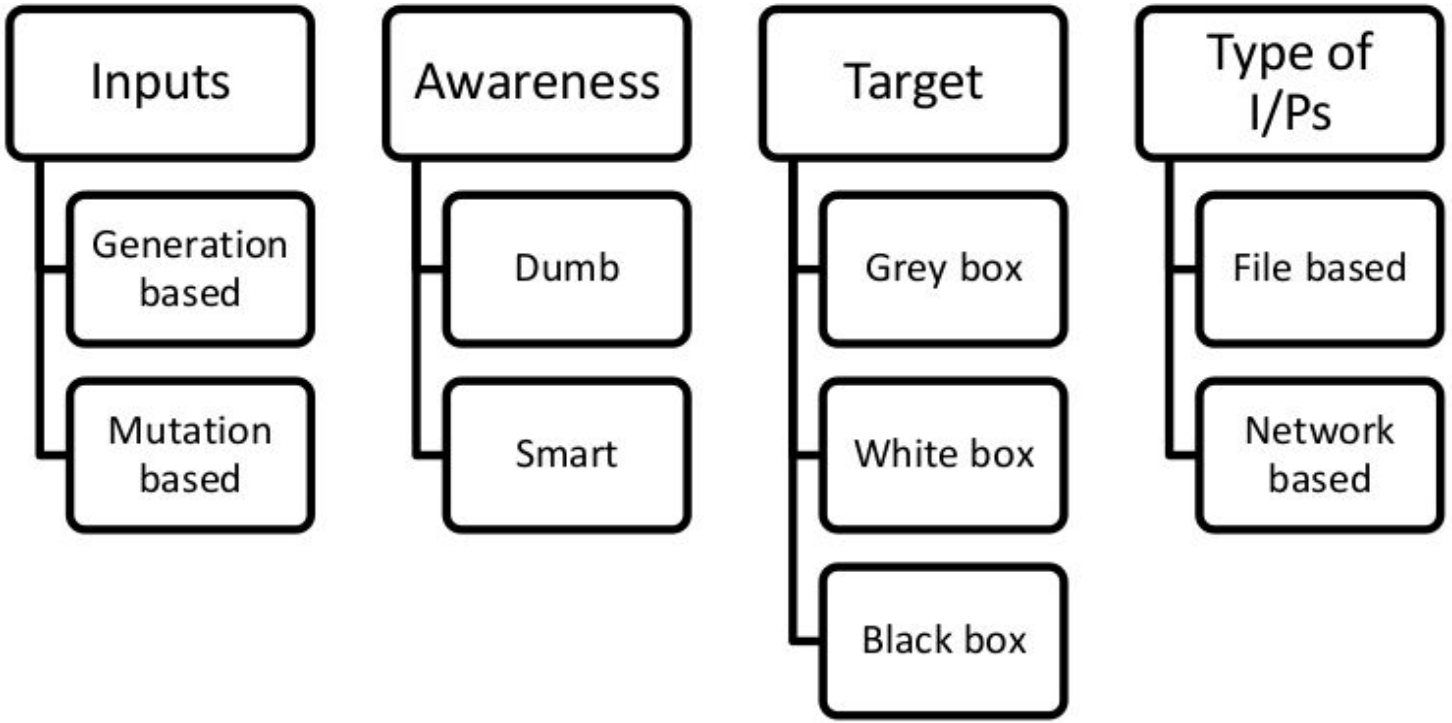
Fuzzing Basics

- Automaticly generate test cases
- Many slightly anomalous test cases are input into a target interface
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, .mpg)
- Or network based…
  - http, SNMP, SOAP
- Or other…
  - e.g. `crashme()`

Trivial Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- Anomalous requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTP/1.1

Types of fuzzers



Types of fuzzers

➔ A fuzzer can be

- **Dumb** or **smart**
  - depending on whether it is aware of input structure

- **Generation-based** or **mutation-based**
  - depending on whether inputs are generated from scratch or by modifying existing inputs,

- **White** , **grey** , or **black-box**
  - depending on whether it is aware of program structure.

Mutation Based Fuzzing -“Dumb Fuzzing”

- ➔ Little or no knowledge of the structure of the inputs is assumed
- ➔ Anomalies are added to existing valid inputs
- ➔ Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)
- ➔ Examples: Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.

Dumb Fuzzing In Short

Strengths

- Super easy to setup and automate
- Little to no protocol knowledge required

Weaknesses

- Limited by initial corpus (code)
- May fail for protocols with checksums, those which depend on challenge response, etc.

Generation Based Fuzzing-“Smart Fuzzing”

- ➔ Test cases are generated from some description of the format: RFC, documentation, etc
- ➔ Anomalies are added to each possible spot in the inputs
- ➔ Knowledge of protocol should give better results than random fuzzing.
- ➔ Can take significant time to set up

Generation Based Fuzzing In Short

Strengths

- Completeness
- Can deal with complex dependencies e.g. checksums

Weaknesses

- Should have specification of protocol.
- Often can find good tools for existing protocols e.g. http, SNMP
- Writing generator can be labor intensive for complex protocols
- The spec is not the code

Challenges

- Mutation based – can run forever. When do we Stop?
- Generation based – stop eventually. Is it enough?
- How to determine if the program did something “Bad”?
- These are the standard problems we face in most automated testing.

Pros and cons of fuzzing

Pros

- Can provide results with little effort: once a fuzzer is up and running, it can be left for hours, days or months to look for bugs with no interaction
- Can reveal bugs that were missed in a manual audit
- Provides an overall picture of the robustness of the target software

Pros and cons of fuzzing

Cons

- **Will not find all bugs**:
  - Fuzzing may miss bugs that do not trigger a full program crash, and may be less likely to trigger bugs that are only triggered in highly specific circumstances

- The crashing test cases that are produced may be difficult to analyse,
  - As the act of fuzzing does not give you much knowledge of how the software operates internally

- Programs with complex inputs can require much more work to produce a smart enough fuzzer to get sufficient code coverage

Open Source Fuzzers

- VUzzer
- Afl-fuzz
- Filebuster
- TriforceAFL
- Nightmare

Regression vs. Fuzzing

- Regression: Run program on many normal inputs, look for badness.
  - Goal: Prevent normal users from encountering errors (e.g. assertions bad).

- Fuzzing: Run program on many abnormal inputs, look for badness.
  - Goal: Prevent attackers from encountering exploitable errors (e.g. assertions often ok)