Fuzzing

What is fuzzing?

- A kind of **random testing**
- **Goal**: make sure certain **bad things don't happen, no matter what**
  - **Crashes, thrown exceptions, non-termination**
  - All of these things can be the foundation of security vulnerabilities
- **Complements functional testing**
  - Test features (and lack of misfeatures) directly
  - Normal tests can be starting points for fuzz tests

Kinds of fuzzing

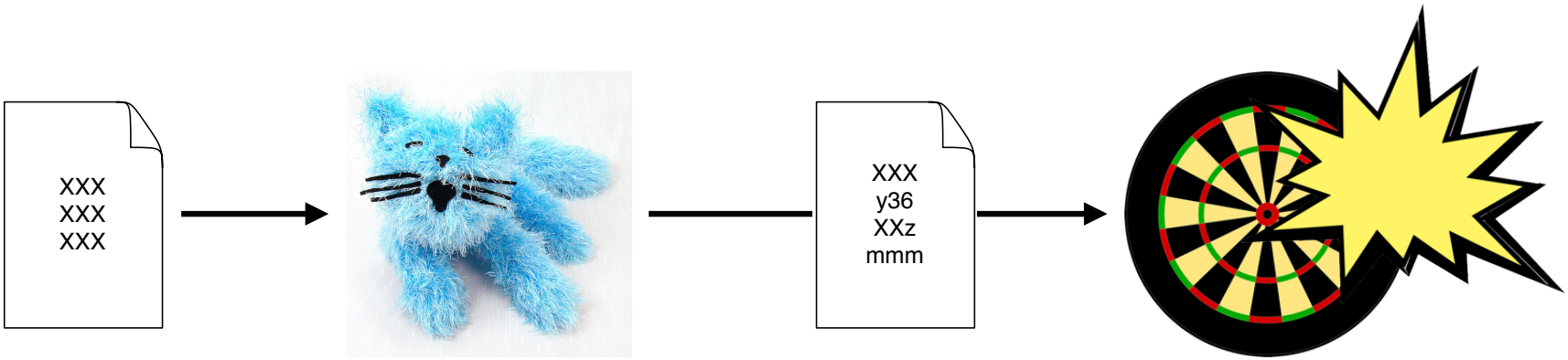
- **Black box**
  - The tool knows nothing about the program or its input
  - **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky
- **Grammar based**
  - The tool generates input informed by a grammar
  - **More work to use**, to produce the grammar, but **can go deeper** in the state space
- **White box**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed
  - Often **easy to use**, but **computationally expensive**

Fuzzing inputs

- **Mutation**
  - Take a **legal input and *mutate* it**, using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar
- **Generational**
  - **Generate** input from scratch, e.g., from a **grammar**
- **Combinations**
  - Generate initial input, mutate^N, generate new inputs, …
  - Generate mutations according to grammar

File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- See **what happens**



Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based*, black box fuzzer
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

https://code.google.com/p/ouspg/wiki/Radamsa https://code.google.com/p/ouspg/wiki/Blab

Example: American Fuzzy Lop

- It is a *mutation-based*, white-box fuzzer. Process:
  - **Instrument target** to gather run-time information
    - Tuple of **<ID of current code location, ID last code location>**
  - **Run a test. Mutate** test input to create a new one *if unseen tuple generated*; otherwise discard the test
    - Mutations include bit flips, arithmetic, other standard stuff
  - Periodically **cull** gathered tests, to avoid local minima

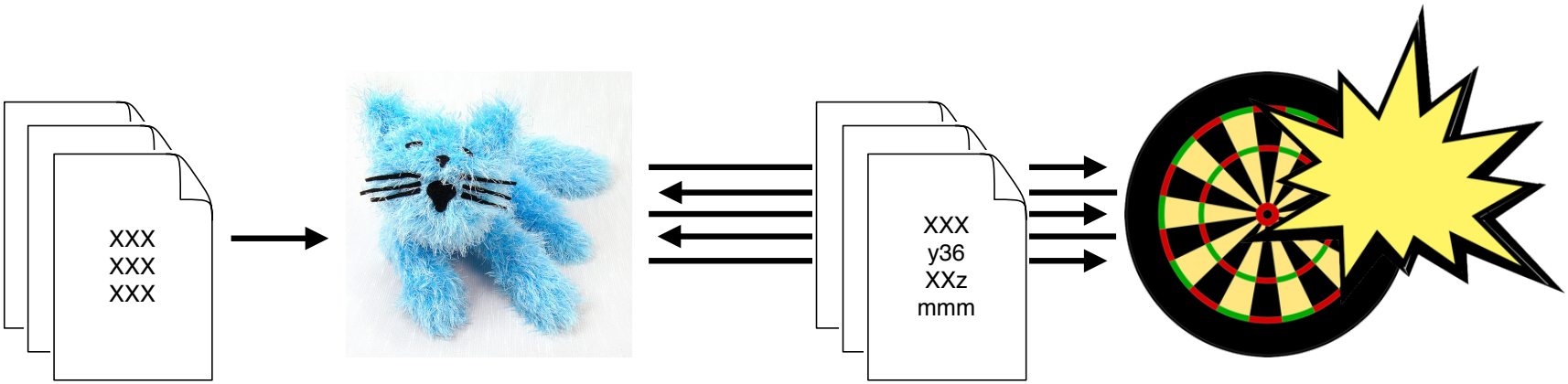
See also Microsoft's SAGE, covered in the unit on symbolic execution

```
% afl-gcc -c … -o target
% afl-fuzz -i inputs -o outputs target
afl-fuzz 0.23b (Sep 28 2014 19:39:32) by <lcamtuf@google.com>
[*] Verifying test case 'inputs/sample.txt'...
[+] Done: 0 bits set, 32768 remaining in the bitmap. …
————————
Queue cycle: 1n time : 0 days, 0 hrs, 0 min, 0.53 sec …
```

https://code.google.com/p/american-fuzzy-lop/

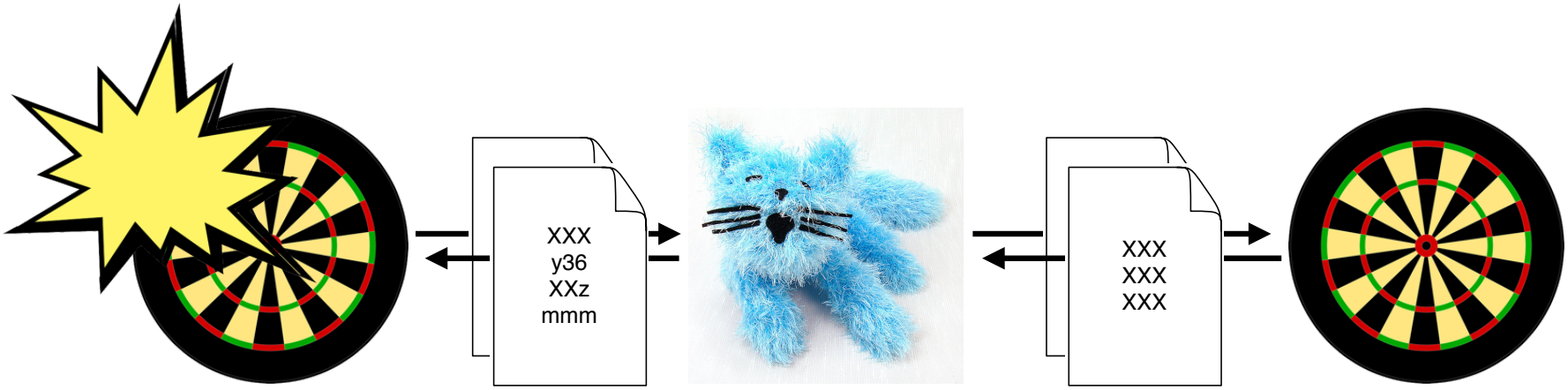
Network-based fuzzing

- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)



Network-based fuzzing

- **Act as a "man in the middle"**
  - mutating inputs exchanged between parties (perhaps informed by a grammar)



Example: SPIKE

- **SPIKE** is *fuzzer creation kit*, providing an C-language API for writing fuzzers for network based protocols

```
s_size_string("post",5);
s_block_start("post");
s_string_variable("user=bob");
s_block_end("post");
spike_tcp_connect(host,port);
spike_send();
spike_close_tcp();
```

backpatch for length field
… for this block
prefix of string to fuzz
end of block; include length
connect to server
send this buffer
close connection

http://www.immunitysec.com/downloads/SPIKE2.9.tgz
http://resources.infosecinstitute.com/intro-to-fuzzing/

Example: Burp Intruder

- Burp **automates customized attacks against web applications**
- Similar to SPIKE in allowing the user to **craft the template** of a request, but **leave "holes"** (called ***payloads***) for fuzzing
  - Nice GUI front end
- Integrates with the rest of the Burp Suite, which includes a proxy, scanner, spider, and more

http://portswigger.net/burp/intruder.html

Dealing with crashes

- You **fuzz**. A **crash** occurs. **Questions**:
- What is the **root cause** (so it can be fixed)**?**
  - Is there a way to **make the input smaller**, so it is more understandable?
  - Are **two or more crashes signaling the same bug?**
    - Yes, if they “minimize” to the same input
- Does the crash signal an **exploitable vulnerability**?
  - Dereferencing NULL is rarely exploitable
  - Buffer overruns often are

Finding memory errors

1. **Compile** the program with **Address Sanitizer** (ASAN)
   - Instruments accesses to arrays to check for overflows, and use-after-free errors
   - https://code.google.com/p/address-sanitizer/
2. **Fuzz it**
3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

- Similarly, you can *compile with other sorts of error checkers* for the purposes of testing
  - E.g., `valgrind` memcheck http://valgrind.org/

A few other fuzzers

- **CERT Basic Fuzzing Framework** (BFF)
  - Based in part on **Zzuf** (http://caca.zoy.org/wiki/zzuf)
  - Found bugs in Adobe Reader and Flash Player, Apple Preview and QuickTime, and others
  - http://www.cert.org/vulnerability-analysis/tools/bff.cfm?
- **Sulley** (http://code.google.com/p/sulley/) provides lots of extras to manage the testing process:
  - watches the network and methodically maintains records
  - instruments and monitors the health of the target, capable of reverting to a known good state
  - detects, tracks and categorizes detected faults
  - fuzzes in parallel, if desired

There are many more …

Summary

- Penetration testers **simulate real attackers**
  - Try to find exploitable vulnerabilities in complete systems
- Penetrations signal **real problems**
  - **Lack of penetrations is not proof of impossibility**
- Pen testers **employ a variety of tools**
  - Scanners, proxies, exploit injectors, fuzzers
- And require **ingenuity** and **guile**