Vulnerability Discovery

Dawn Song

dawnsong@cs.berkeley.edu

Malicious Code Defense

- **Vulnerability discovery**
  - **Symbolic execution**
- **The BitBlaze project**
  - **Binary analysis for computer security**
  - **http://bitblaze.cs.berkeley.edu**

IPhone Security Flaw

- **Jul 2007: “researchers at Independent Security Evaluators, said that they could take control of iPhones through a WiFi connection or by tricking users into going to a Web site that contains malicious code. The hack, the first reported, allowed them to tap the wealth of personal information the phones contain.”**

Charles Miller, shown on his iPhone, said that after finding a hole in security, “you were in complete control.”

iPhone attack

- **iPhone Safari downloads malicious web page**
  - **Arbitrary code is run with administrative privileges**
  - **Can read SMS log, address book, call history, other data**
  - **Can perform physical actions on the phone.**
    - » **system sound and vibrate the phone for a second**
    - » **could dial phone numbers, send text messages, or record audio (as a bugging device)**
  - **Can transmit any collected data over network to attacker**

See http://www.securityevaluators.com/iphone/

0days Are a Hacker Obsession

- **An 0day is a vulnerability that's not publicly known**
- **Modern 0days often combine multiple attack vectors & vulnerabilities into one exploit**
  - **Many of these are used only once on high value targets**
- **0day statistics**
  - **Often open for months, sometimes years**

Market for 0days

- **Sell for $10K-100K**
- **Tippingpoint**
- **Eeye**
- **Gleg.net**
- **Dsquare**
- **Idefense**
- **Digital armaments**
- **Breakingpoint**

How to Find a 0day?

- **Step #1: obtain information**
  - **Hardware, software information**
  - **Sometimes the hardest step**
    - **eBay to the rescue**
- **Step #2: bug finding**
  - **Manual audit**
  - **(semi)automated techniques/tools**

The iPhone Story

- **Step #1: WebKit opensource**
  - **svn co http://svn.webkit.org/repository/webkit/trunk WebKit**
- **Step #2: identify potential focus points**
  - **From development site:**
    The JavaScriptCore Tests
    "*If you are making changes to JavaScriptCore, there is an additional test suite you must run before landing changes. This is the Mozilla JavaScript test suite*."
  - **So we know what they use for unit testing**
    - **Use code coverage to see which portions of code is not well tested**
    - **Tools gcov, icov, etc., measure test coverage**

Results

- **59.3% of 13622 lines in JavaScriptCore were covered**
  - **79.3% of main engine covered**
  - **54.7% of Perl Compatible Regular Expression (PCRE) covered**
- **Next step: focus on PCRE**
  - **Wrote a PCRE fuzzer (20 lines of perl)**
  - **Ran it on standalone PCRE parser (pcredemo from PCRE library)**
  - **Started getting errors:**
    PCRE compilation failed at offset 6: internal error: code overflow
- **Evil regular expressions crash mobileSafari**

The Art of Fuzzing

- **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**

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**
  - **GET /index.html HTTP/1.1.1.1.1.1.1.1**

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)**

Approach I: Black-box Fuzz Testing

- **Given a program, simply feed it random inputs, see whether it crashes**
- **Advantage: really easy**
- **Disadvantage: inefficient**
  - **Input often requires structures, random inputs are likely to be malformed**
  - **Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low**

Enhancement I: Mutation-Based Fuzzing

- **Take a well-formed input, randomly perturb (flipping bit, etc.)**
- **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:**
  - **E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/**
  - **Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.**

Example: fuzzing a pdf viewer

- **Google for .pdf (about 1 billion results)**
- **Crawl pages to build a corpus**
- **Use fuzzing tool (or script to)**
  1. **Grab a file**
  2. **Mutate that file**
  3. **Feed it to the program**
  4. **Record if it crashed (and input that crashed it)**

Mutation-based Fuzzing In Short

- **Strengths**
  - **Super easy to setup and automate**
  - **Little to no protocol knowledge required**
- **Weaknesses**
  - **Limited by initial corpus**
  - **May fail for protocols with checksums, those which depend on challenge response, etc.**

Enhancement II: Generation-Based Fuzzing

- **Test cases are generated from some description of the format: RFC, documentation, etc.**
  - **Using specified protocols/file format info**
  - **E.g., SPIKE by Immunity http://www.immunitysec.com/resources-freesoftware.shtml**
- **Anomalies are added to each possible spot in the inputs**
- **Knowledge of protocol should give better results than random fuzzing**

Example: Protocol Description

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation-Based Fuzzing In Short

- **Strengths**
  - **completeness**
  - **Can deal with complex dependencies e.g. checksums**

- **Weaknesses**
  - **Have to have spec 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**

Fuzzing Tools

- **Hackers' job made easy**
- **Input generation**
- **Input injection**
- **Bug detection**
- **Workflow automation**

Input Generation

- **Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)**
  - **Mu-4000, Codenomicon, PROTOS, FTPFuzz**
- **Fuzzing Frameworks: You provide a spec, they provide a fuzz set**
  - **SPIKE, Peach, Sulley**
- **Dumb Fuzzing automated: you provide the files or packet traces, they provide the fuzz sets**
  - **Filep, Taof, GPF, ProxyFuzz, PeachShark**
- **Many special purpose fuzzers already exist as well**
  - **ActiveX (AxMan), regular expressions, etc.**

Input Injection

- **Simplest**
  - **Run program on fuzzed file**
  - **Replay fuzzed packet trace**
- **Modify existing program/client**
  - **Invoke fuzzer at appropriate point**
- **Use fuzzing framework**
  - **e.g. Peach automates generating COM interface fuzzers**

Bug Detection

- **See if program crashed**
  - **Type of crash can tell a lot (SEGV vs. assert fail)**
- **Run program under dynamic memory error detector (valgrind/purify)**
  - **Catch more bugs, but more expensive per run.**
- **See if program locks up**
- **Roll your own checker e.g. valgrind skins**

Workflow Automation

- **Sulley, Peach, Mu-4000 all provide tools to aid setup, running, recording, etc.**
- **Virtual machines can help create reproduceable workload**

How Much Fuzz Is Enough?

- **Mutation based fuzzers may generate an infinite number of test cases... When has the fuzzer run long enough?**

- **Generation based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?**

Example: PDF

- **I have a PDF file with 248,000 bytes**
- **There is one byte that, if changed to particular values, causes a crash**
  - **This byte is 94% of the way through the file**
- **Any single random mutation to the file has a probability of .00000392 of finding the crash**
- **On average, need 127,512 test cases to find it**
- **At 2 seconds a test case, thats just under 3 days...**
- **It could take a week or more...**

Code Coverage

- **Some of the answers to these questions lie in *code coverage***
- **Code coverage is a metric which can be used to determine how much code has been executed.**
- **Data can be obtained using a variety of profiling tools. e.g. `gcov`**

Types of Code Coverage

- **Line/block coverage**
  - **Measures how many lines of source code have been executed.**
- **Branch coverage**
  - **Measures how many branches in code have been taken (conditional jmps)**
- **Path coverage**
  - **Measures how many paths have been taken**

Example

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- **Requires**
  - **1 test case for line coverage**
  - **2 test cases for branch coverage**
  - **4 test cases for path coverage**
    - **i.e. `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`**

Code Coverage

- **Benefits:**
  - **How good is this initial file?**
  - **Am I getting stuck somewhere?**

```
if(packet[0x10] < 7) { //hot path
} else { //cold path

}
```

  - **How good is fuzzer X vs. fuzzer Y**
  - **Am I getting benefits from running a different fuzzer?**
- **Problems:**
  - **Code can be covered without revealing bugs**

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

Fuzzing Rules of Thumb

- **Protocol specific knowledge very helpful**
  - **Generational tends to beat random, better spec's make better fuzzers**
- **More fuzzers is better**
  - **Each implementation will vary, different fuzzers find different bugs**
- **The longer you run, the more bugs you may find**
- **Best results come from guiding the process**
  - **Notice where your getting stuck, use profiling!**
- **Code coverage can be very useful for guiding the process**
- **Can we do better?**

Approach II: Constraint-based Automatic Test Case Generation

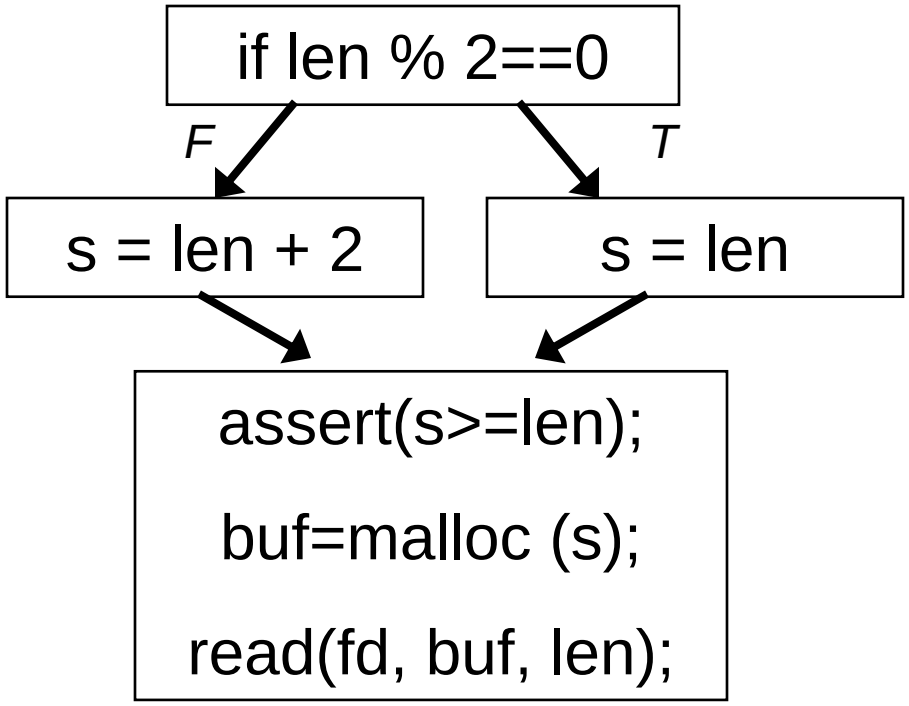
- **Look inside the box**
  - **Use the code itself to guide the fuzzing**
- **Assert security/safety properties**
- **Explore different program execution paths to check for security properties**
- **Challenge:**
  1. **For a given path, need to check whether an input can trigger the bug, i.e., violate security property**
  2. **Find inputs that will go down different program execution paths**

Running Example

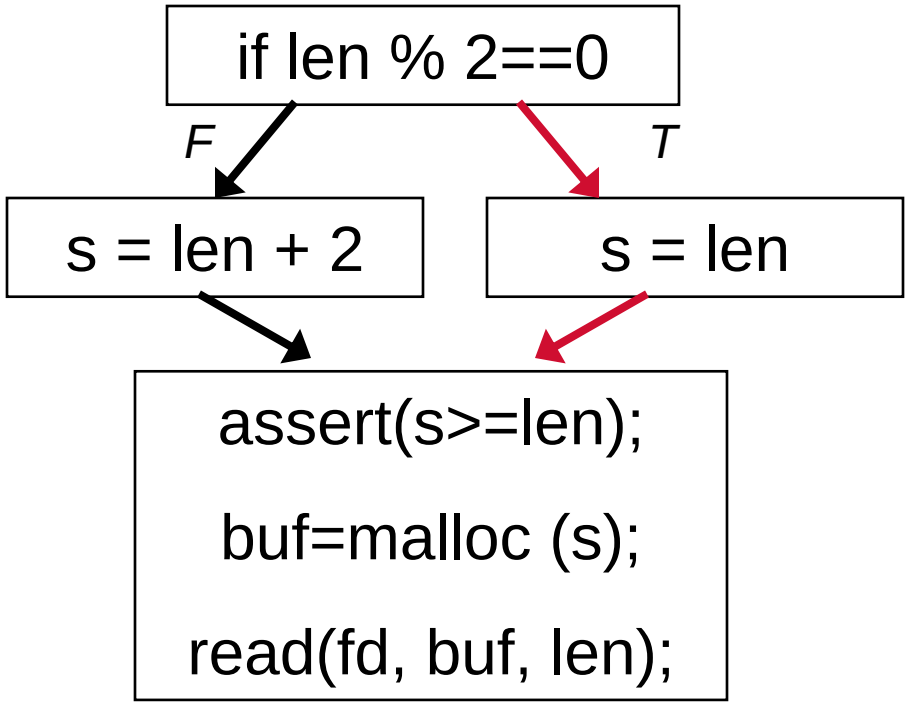
```
f(unsigned int len){
  unsigned int s;
  char *buf;
    if len % 2==0;
    then s = len;
    else s = len + 2;
  buf = malloc(s);
  read(fd, buf, len);
    …
}
```

- Where's the bug?
- What's the security/safety property?
  - s>=len
- What inputs will cause violation of the security property?
  - len = 2^{32} - 1
- How likely will random testing find the bug?

Running Example

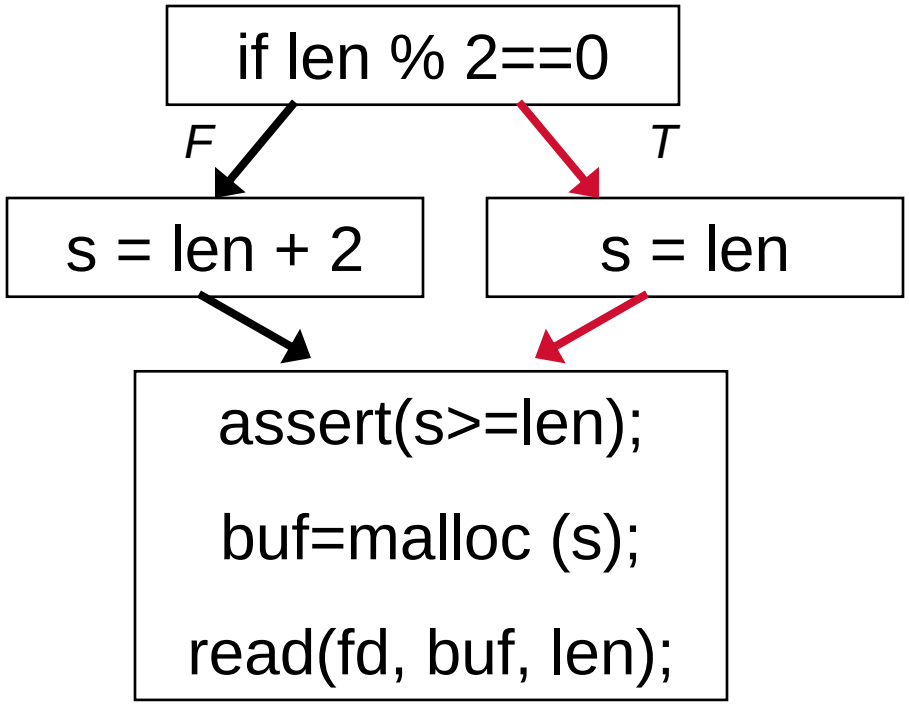


Symbolic Execution



- **Test input len=6**
- **No assertion failure**
- **What about all inputs that takes the same path as len=6?**

Symbolic Execution

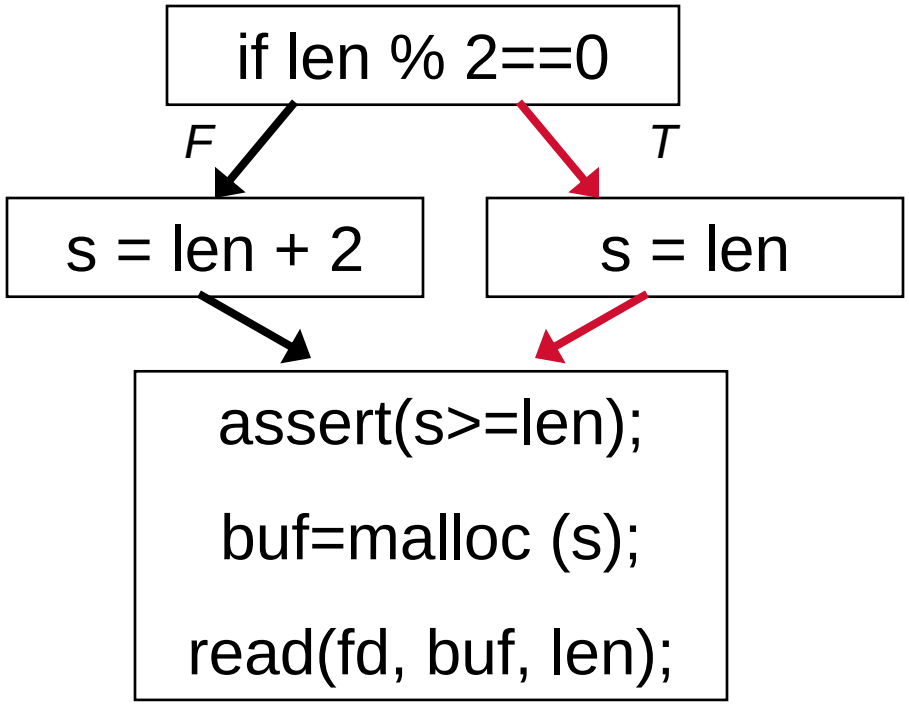


- **What about all inputs that takes the same path as len=6?**
- **Represent len as symbolic variable**

Symbolic Execution

- **Represnset inputs as symbolic variables**
- **Perform each operation on symbolic variables symbolically**
  - **x = y + 5;**
- **Registers and memory values dependent on inputs become symoblic expressions**
- **Certain conditions for conditional jump become symbolic expressions as well**

Symbolic Execution



len % 2 = 0 (path constraint)

s = len

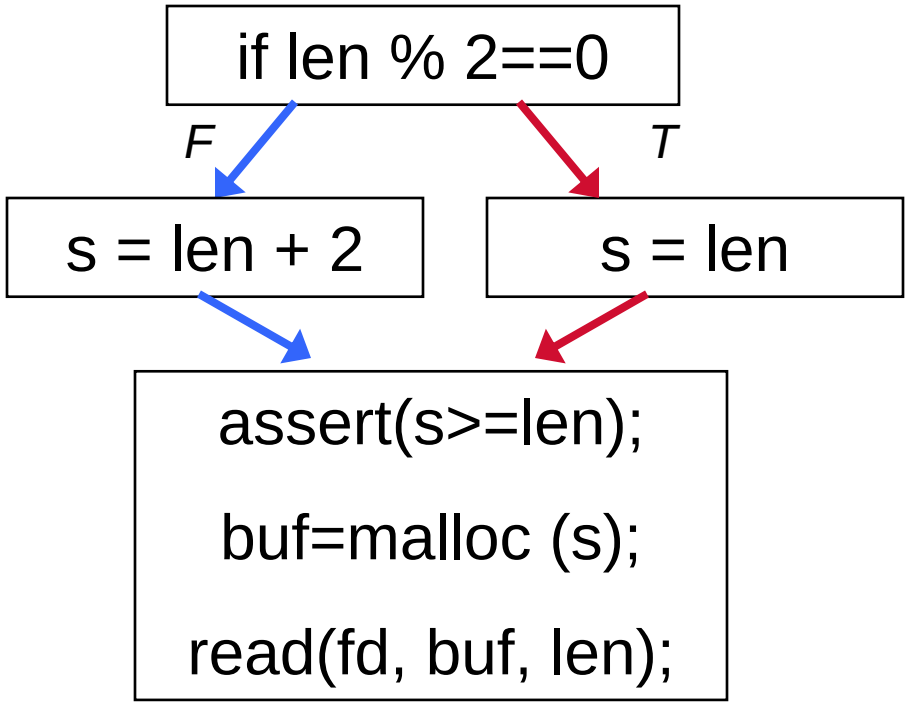
s<len?

- **What about all inputs that takes the same path as len=6?**
- **Represent len as symbolic variable**

Using a Solver

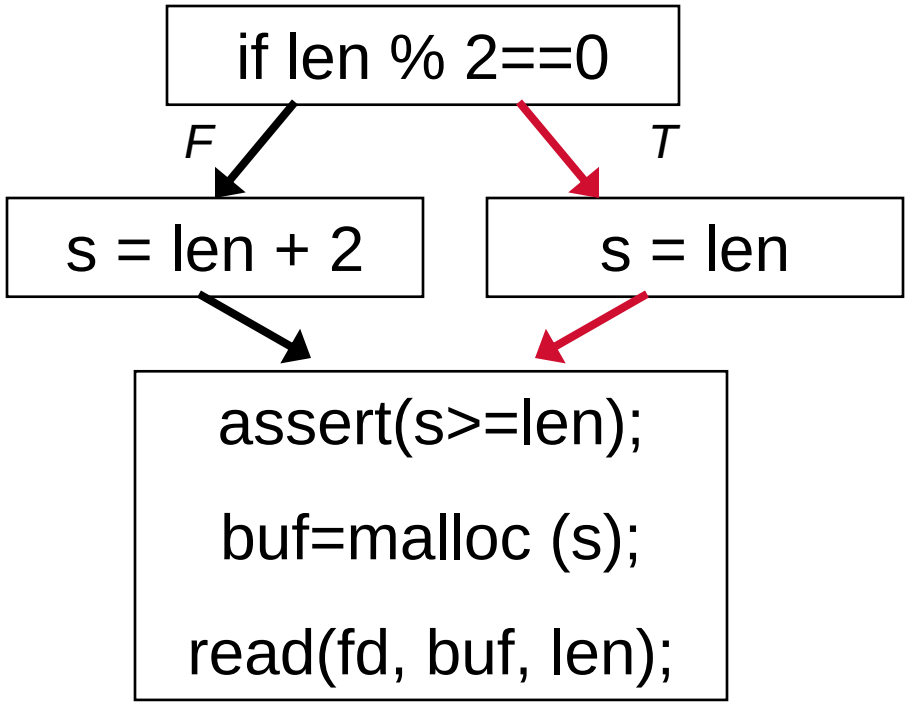
- **Is there a value for len s.t. len % 2 = 0 ^ s = len ^ s < len?**
- **Give the symbolic formula to a solver**
- **In this case, the solver returns No**
  - **The formula is not satisfiable**
- **What does this mean?**
  - **For any len that follows the same path as len = 6, the execution will be safe**
  - **Symbolic execution can check many inputs at the same time for the same path**
- **What to do next?**
  - **Try to explore different path**

How to Explore Different Paths?



- **Previous path constraint: len % 2 = 0**
- **Flip the branch to go down a different path:**
  - **len % 2 != 0**
- **Using a solver for the formula**
  - **A satisfying assignment is a new input to go down the path**

Checking Assertion in the Other Path



len % 2 != 0 (path constraint)

s = len + 2

s<len?

- **Is there a value for len s.t. len % 2 != 0 ^ s = len+2 ^ s < len?**
- **Give the symbolic formula to a solver**
  - **Solver returns satisfying assignment: len = 2^{32} -1**
  - **Found the bug!**

Summary: Symbolic Execution for Bug Finding

- **Symbolicly execute a path**
  - **Create the formula representing: path constraint ^ assertion failure**
  - **Give the solver the formula**
    - **If returns a satisfying assignment, a bug found**
- **Reverse condition for a branch to go down a different path**
  - **Give the solver the new path constraint**
  - **If returns a satisfying assignment**
    - **The path is feasible**
    - **Found a new input going down a different path**
- **Pioneer work**
  - **EXE, DART, CUTE**
- **BitFuzz: binary-based white-box fuzzing**
  - **BitBlaze project (tomorrow)**