How I Learned to Stop Fuzzing and Find More Bugs

Jacob West
Fortify Software

August 3-5, 2007
Las Vegas

Agenda

- Introduction to fuzzing
  - What we mean by fuzzing
  - Challenges with fuzzing
- Introduction to static analysis
  - How static analysis works
  - Examples of bugs static analysis is good at finding
  - Untapped potential: Customization
- Experiment
  - Fuzzing versus static analysis
- Conclusion

What is Fuzzing?

- Encompasses runtime testing that attempts to induce faults in software systems by inputting random or semi-random values
- Introduced by Barton Miller at the University of Wisconsin, Madison in 1990 (cs.wisc.edu/~bart/fuzz/)

Examples of Tools

- We're talking about tools such as:
  - SPIKE
    www.immunitysec.com/resources-freesoftware.shtml
  - Peach
    http://peachfuzz.sourceforge.net
  - PROTOS
    http://www.ee.oulu.fi/research/ouspg/protos/
  - ... and many more
- But not specialized black box scanning tools:
  - Cenzic
  - SPI Dynamics (except SPI Fuzzer)
  - Watchfire

The Inventor's Thoughts on Fuzzing

- 1990: "[Fuzzing] is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs…"
- 1995: "While [fuzzing] is effective in finding real bugs in real programs, we are not proposing it as a replacement for systematic and formal testing."
- 2000: "Simple fuzz testing does not replace more extensive formal testing procedures."

- Barton Miller

Woulda, Coulda, Shoulda

SC-L Digest, Vol 3, Issue 118:

- **“I would assume that "smart" fuzzing could have lots of manipulations of the HH:mm:ss.f format, so this might be findable using black box testing.”**

- Steve Christey

Woulda, Coulda, Shoulda

http://blogs.msdn.com/sdl

- "It turns out none of the .ANI fuzz templates had a second "anih" record.

  This is now addressed, and we are continually enhancing our fuzzing tools to make sure they add manipulations that duplicate arbitrary object elements better."

  - Michael Howard

How Fuzzing Works

- Identify sources of input to a program
- Permute or generate pseudorandom input
- Use an oracle to monitor for failures
- Record the input and state that generate faults

Input Sources: File Formats

1. **Identify all valid file formats (e.g. JPG, TIFF, PDF, DOC, XLS)**
2. **Collect a library of valid files**
3. **Malform a file**
4. **Cause the program to consume the file and observe its execution for problems**

Input Sources: Protocols

- **Create bogus messages (e.g. SMPT, TCP/IP, RPC, SOAP, HTTP)**

- **Record-fuzz-replay**
  1. **Run a sniffer**
  2. **Collect a few thousand messages**
  3. **Fuzz the messages**
  4. **Replay the fuzzed messages**

Dumb Fuzzing

- Dumb fuzzing: Modify data randomly
  - Most input will be invalid
  - Makes good error handling test cases
  - Takes a long time to enumerate valid test cases
  - May test the validation logic of high-level protocols instead of the underlying application

Smart Fuzzing

- **Smart fuzzing: Aware of data structure**
  - **Altering content size**
  - **Replacing null-terminated strings**
  - **Altering numeric values or flipping signs**
  - **0, 2^n +/- 1**
  - **Adding invalid headers, altering header values, duplicating headers, …**

Challenging Questions with Fuzzing

- Microsoft SDL mandates that you run 100,000 iterations per file format/parser.
- If you find a bug, you reset to 0 and start running another 100,000 with a new seed.
- Why? Does this get you what you need?

- How many input sources were missed?
- How much of the program was tested?
- How long did the tests take to run?
- How good were the tests?

Challenge: Nebulous File Formats / Protocols

- No problem for a standard Web application
- What about proprietary interfaces?
  - Web Service APIs
  - Network servers
  - Thick client software
- Difficult to enumerate input sources to fuzz
- Even harder to generate valid input

- Requires customization
  - Tool must be tuned to specific input sources and formats

Challenge: Program Semantics / Reachability

- **Example:**

```
if (!strcmp(input1, “static_string”) {
  strcpy(buffer2, input2);
}
```

- **Need to provide value of `input1` equal to “static_string” and large value of `input2`**
- **Requires N*M random inputs to reach bug guarded by two-variable conditions**
- **May be hard to satisfy some conditionals**
- **Requires customization**
  - **Number of input values needed must be narrowed**

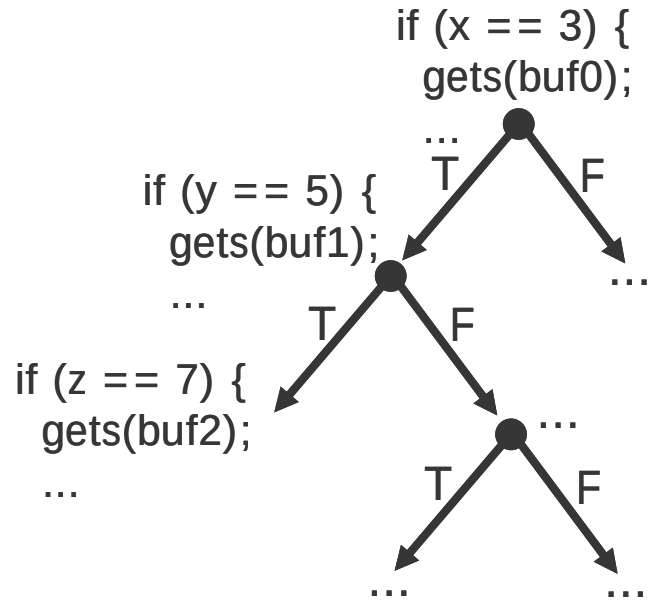
Shallow Bugs versus Deep Bugs

- **Fuzzing focuses on shallow bugs**

```
foo(int x, int y, int z) {
if (x == 3) {      //p = 1/2^32
  gets(buf0);

  if (y == 5) {    //p = p * 1/2^32
    gets(buf1);

    if (z == 7) { //p = p * 1/2^32
       gets(buf2);
    }
  }
}
}
```

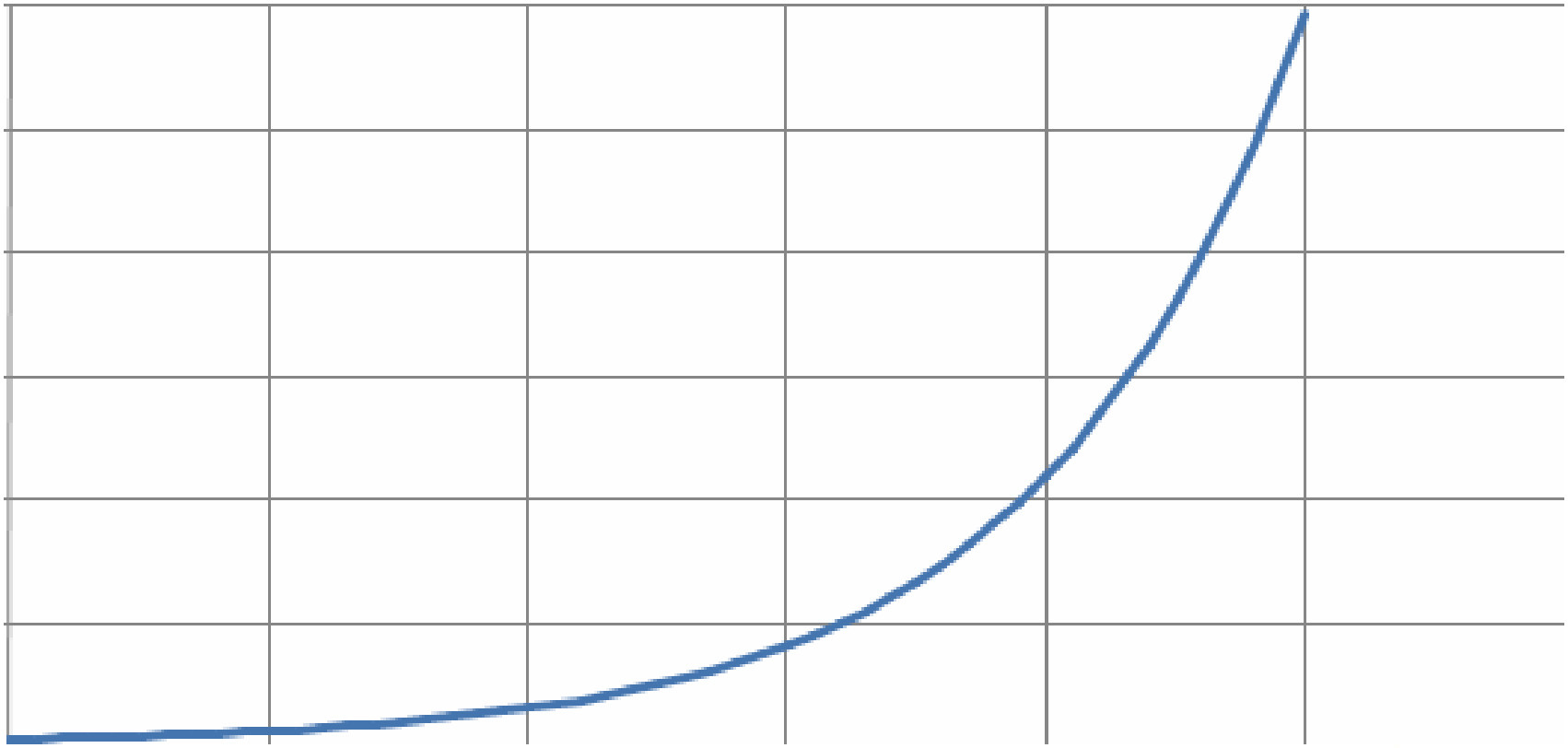


# of random values of x, y and z to reach each state:

gets(buf0) =	4,294,967,296
gets(buf1) =	18,446,744,073,709,551,616
gets(buf2) =	79,228,162,514,264,337,593,629,020,928

Runs Necessary to Reach State Z

- Each conditional adds exponentially to the number of input permutations required to hit a bug
- Running time for the fuzz tests increases accordingly



Example: Vulnerabilities by Conditional Depth

- **wu-ftpd 2.6.0–Buffer Overflow–extensions.c:**
  `strcpy(curptr->dirname, cwd);`
  Conditional depth: 4
- **wu-ftpd 2.6.0–format string–ftpd.c:**
  `vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf)-4 : sizeof(buf), fmt, ap);`
  Conditional depth: 3
- **OFBiz 1.5–XSS–CommonEvents.java:**
  `out.println(responseString);`
  Conditional depth: 4

Challenge: Difficult-to-Reach States

- **Airline booking system – overbooked flight**
  - **Difficult for a fuzzer to induce**
- **Example:**

```
if (flight.seatsAvailable() == 0) {
  // echo user input to error page – XSS vulnerability
  ...
  out.println("Flight "
              + request.getParameter("flightNumber")
              + " is overbooked. Please search again.");
  ...
}
```

Challenge: Identifying Errors

- Error reporting conventions differ
- Good design guidelines often require programs to mask errors and error details

- Requires customization
  - Better oracle
  - Binary instrumentation
  - ...

Finding Bugs with Fuzzing

Spectrum of ease of detection with fuzzing:

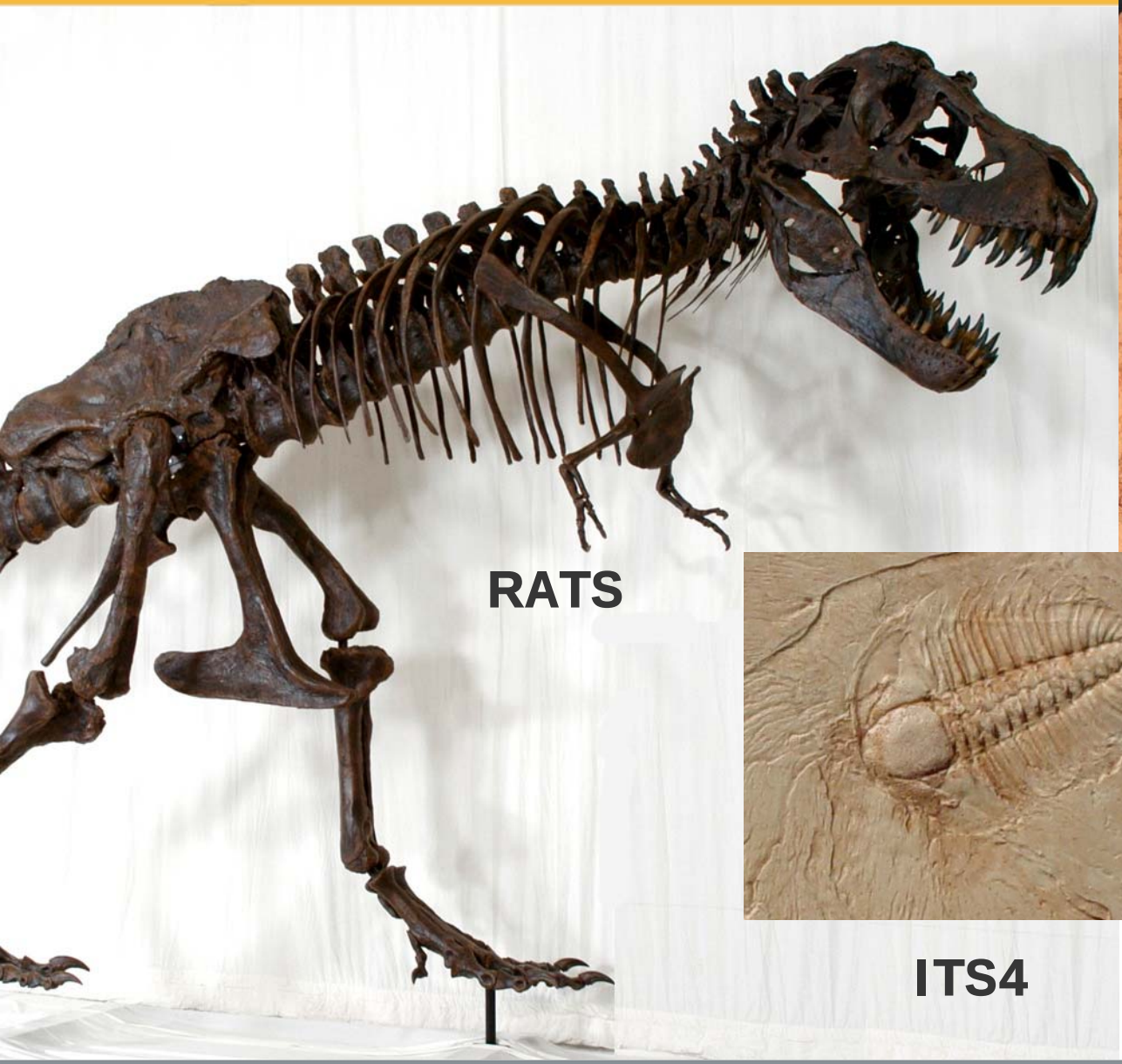
Easy: Shallow cross-site scripting vulnerability (shallowest bugs never leave the client à la JavaScript)

Hard: Many nested conditionals that checks for hard-to-reach states like the overbooked flight

Fuzzing Summary

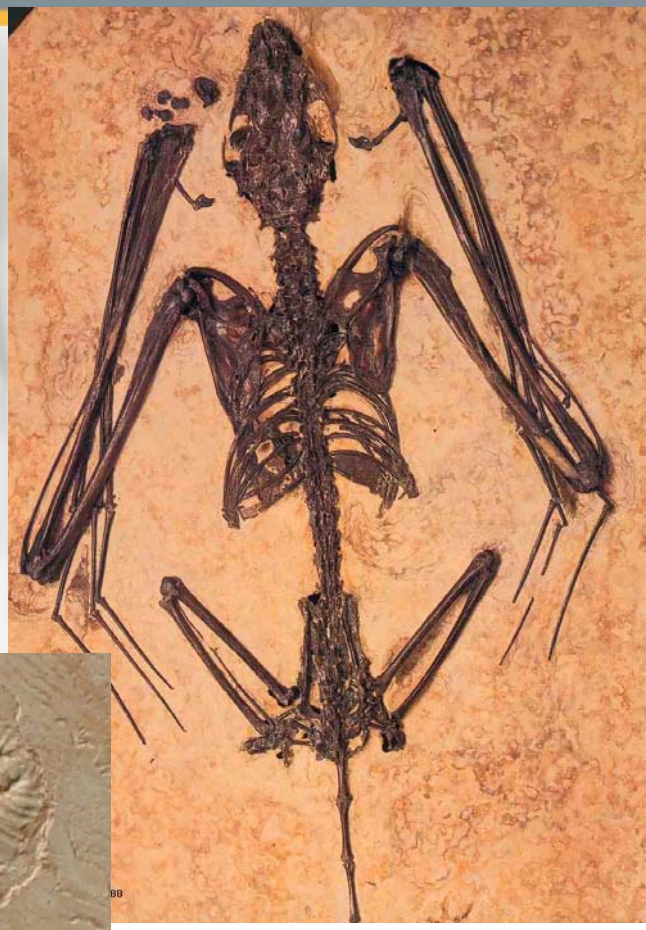
- **Advantages**
  - Requires least effort to find a bug
  - Verifiable and reproducible at runtime
  - Scalable to programs that use the same file format or protocol
- **Disadvantages**
  - Very costly to achieve completeness
  - Increasing coverage increases runtime (sometimes exponentially)
  - May miss bugs due to inadequate oracle

Prehistoric Static Analysis Tools



RATS

ITS4



Flawfinder

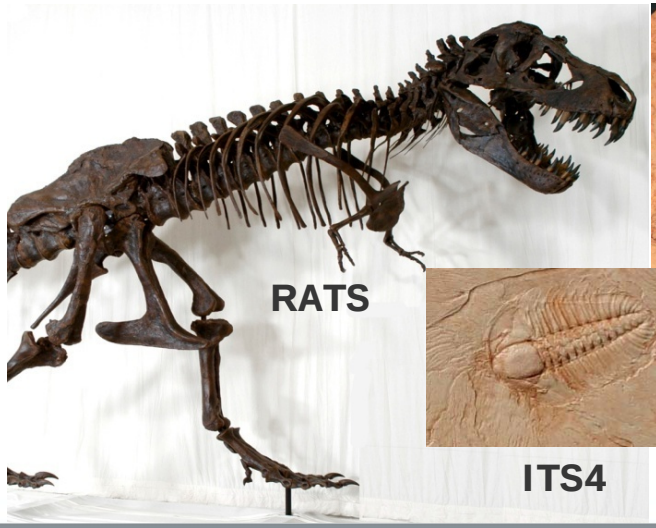
Prehistoric Static Analysis Tools

(+) Good

- Help security experts audit code
- Repository for known-bad coding practices

(-) Bad

- NOT BUG FINDERS
- Not helpful without security expertise



Misconceptions Prevail

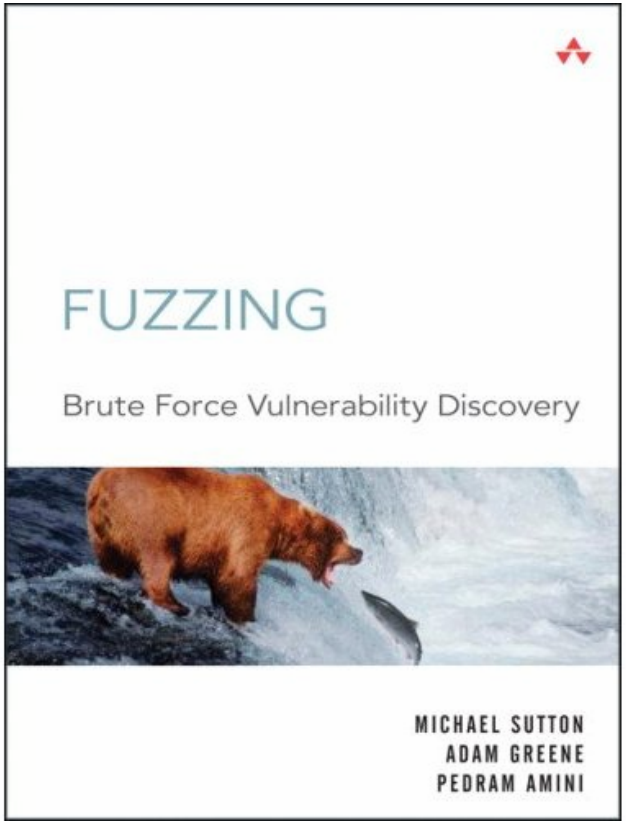
***Fuzzing*, Page 4:**

Low priority

```
int main(int argc, char** argv)
    char buffer[10];
    strcpy(buf1, “test”);
  }
```

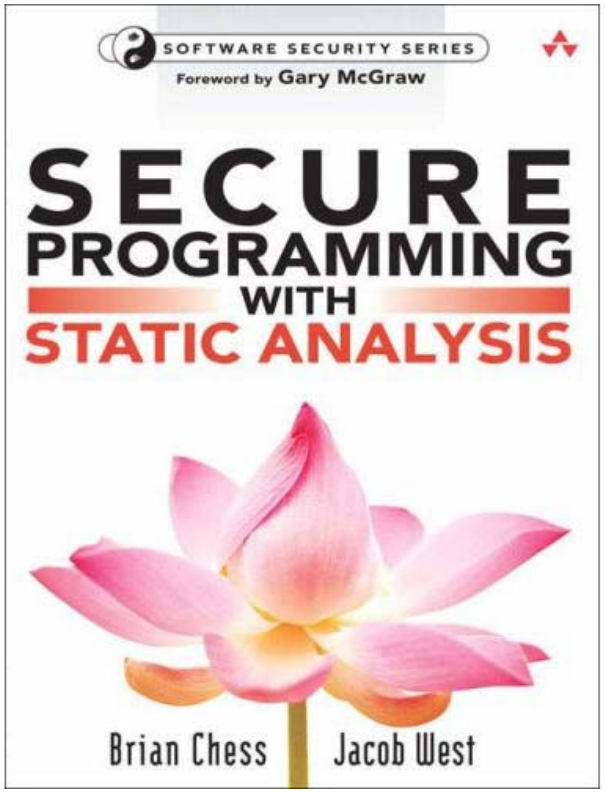
High priority

```
int main(int argc, char** argv)
    char buffer[10];
    strcpy(buf1, argv[1]);
  }
```



Static Analysis Is Good For Security

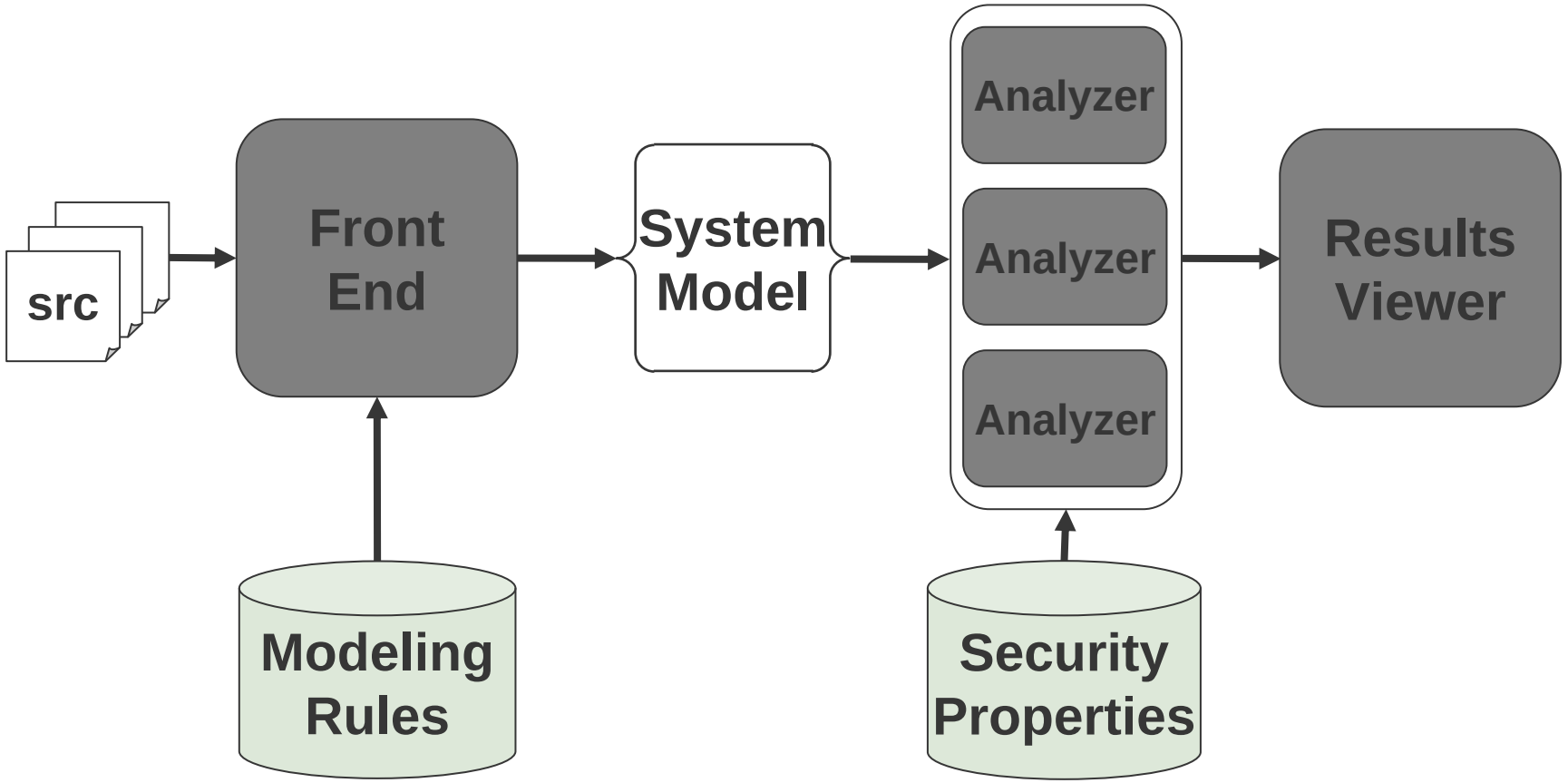
- Fast compared to manual review
- Fast compared to testing
- Complete, consistent coverage
- Brings security knowledge with it
- Makes security review process easier for non-experts
- Useful for all kinds of code, not just Web applications



Static Analysis: No Silver Bullet

- **Human limitations**
  - Requires access to code
  - User must understand code
- **Tool limitations**
  - Does not understand architecture
  - Does not understand application semantics
  - Does not understand social context

A Peek Inside a Static Analysis Tool



Parsing

- Language support
  - One language/parser is straightforward
  - Lots of combinations is harder
- Could analyze compiled code…
  - Everybody has the binary
  - No need to guess how the compiler works
  - No need for rules
- …but
  - Decompilation can be difficult
  - Loss of context hurts
  - Want to report line numbers

Analysis / Rules: Structural

- **Identify bugs in the program's structure**
- **Example: calls to `gets()`**
- **Structural rule:**

```
FunctionCall: function is [name == "gets"]
```

Analysis / Rules: Structural

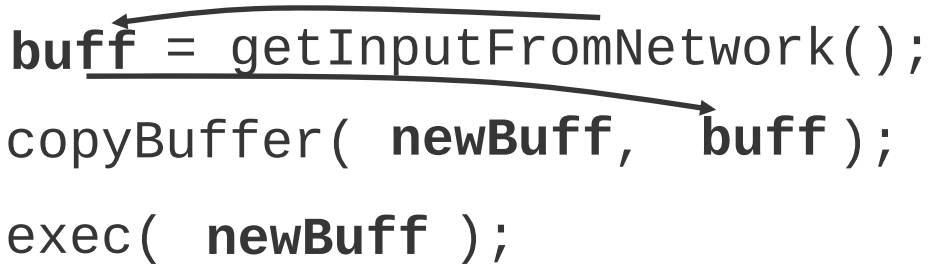
- **Identify bugs in the program's structure**

- **Example: memory leaks caused by `realloc()`**
  `buf = realloc(buf, 256);`
- **Structural rule:**

```
FunctionCall c1: (
  c1.function is [name == "realloc"] and
  c1 in [AssignmentStatement: rhs is c1 and
         lhs == c1.arguments[0]
        ]
)
```

Analysis / Rules: Dataflow Source Rule

- Following interesting values through the program
- Example: Command injection vulnerability

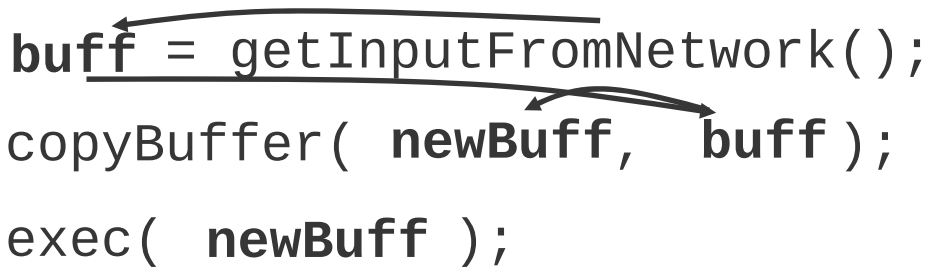


- Source rule:

```
Function: getInputFromNetwork()
Postcondition: return value is tainted
```

Analysis / Rules: Dataflow Pass-Through Rule

- **Following interesting values through the program**
- **Example: Command injection vulnerability**



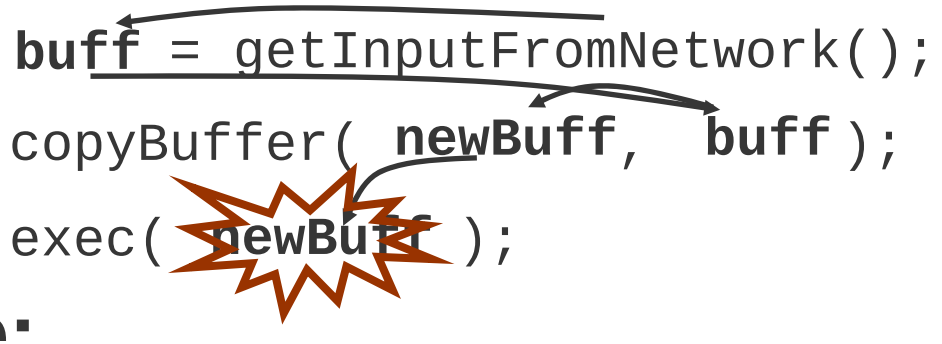
- **Pass-through rule:**

Function: copyBuffer()

Postcondition: if the second argument is tainted, then the first argument becomes tainted

Analysis / Rules: Dataflow Sink Rule

- **Following interesting values through the program**
- **Example: Command injection vulnerability**



- **Sink rule:**

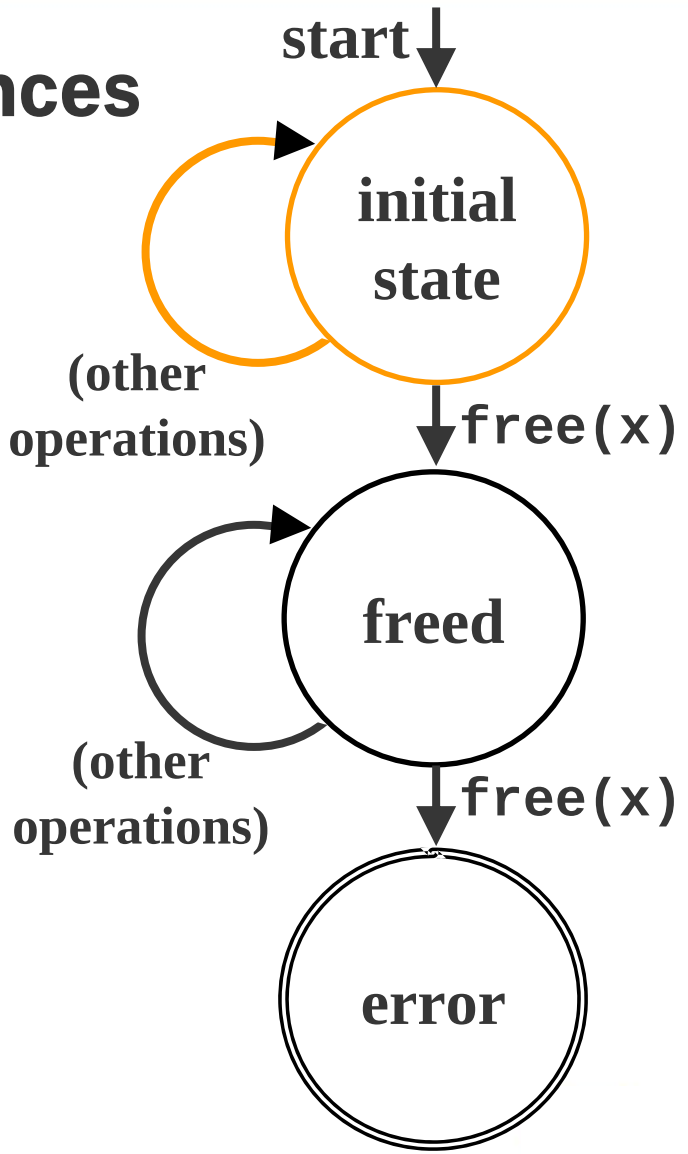
Function: exec()

Precondition: the first argument must not be tainted

Analysis / Rules: Control Flow

- Look for dangerous sequences
- Example: Double-free

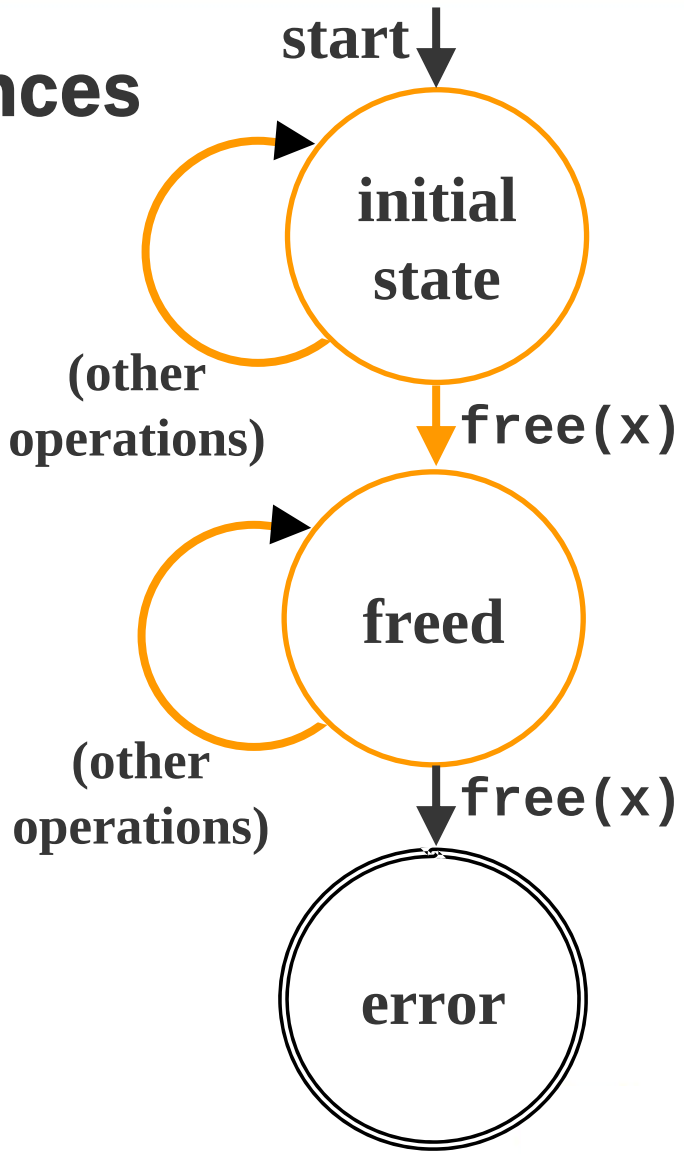
```
while ((node = *ref) != NULL) {
    *ref = node->next;
    free(node);
    if (!unchain(ref)) {
      break;
    }
}
if (node != 0) {
    free(node);
    return UNCHAIN_FAIL;
}
```



Analysis / Rules: Control Flow

- Look for dangerous sequences
- Example: Double-free

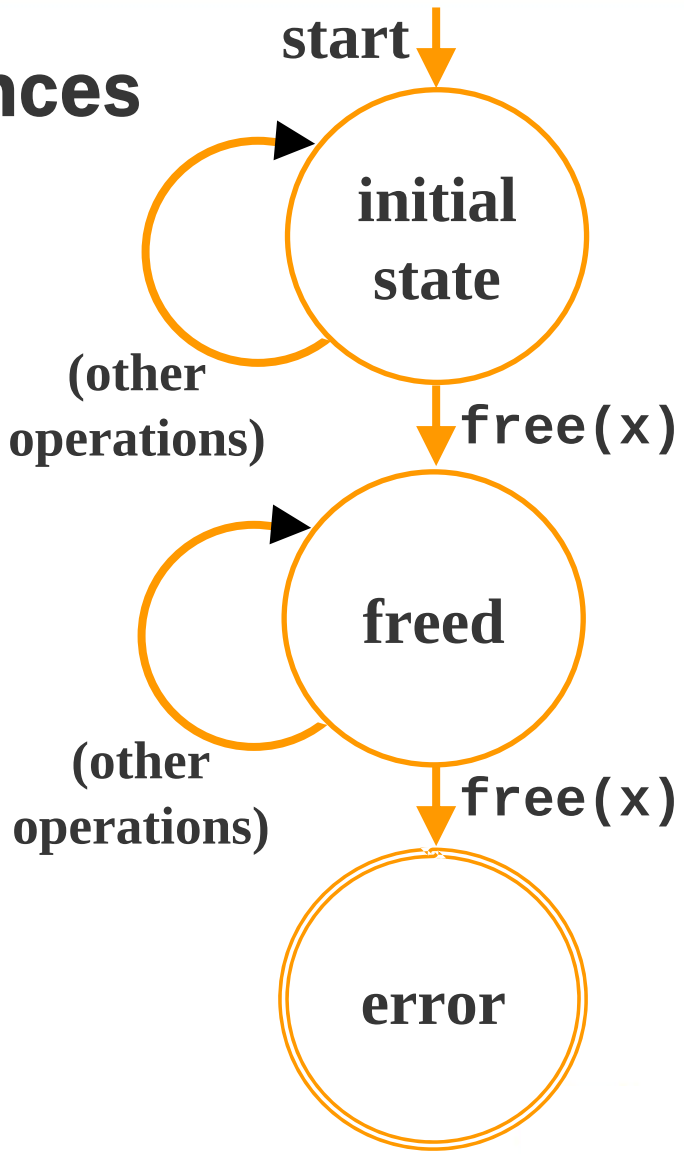
```
while ((node = *ref) != NULL) {
  *ref = node->next;
  free(node);
  if (!unchain(ref)) {
    break;
  }
}
if (node != 0) {
  free(node);
  return UNCHAIN_FAIL;
}
```



Analysis / Rules: Control Flow

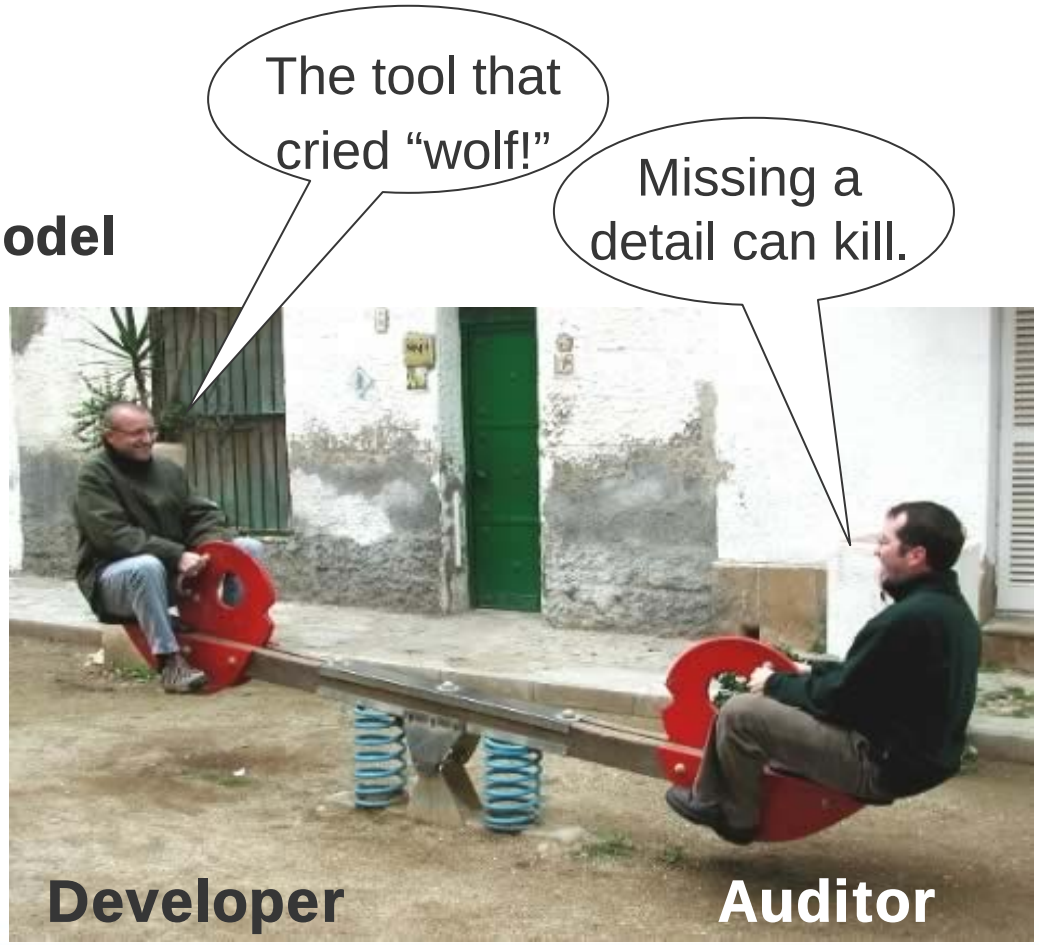
- Look for dangerous sequences
- Example: Double-free

```
while ((node = *ref) != NULL) {
  *ref = node->next;
  free(node);
  if (!unchain(ref)) {
    break;
  }
}
if (node != 0) {
  free(node);
  return UNCHAIN_FAIL;
}
```



Only Two Ways to Go Wrong

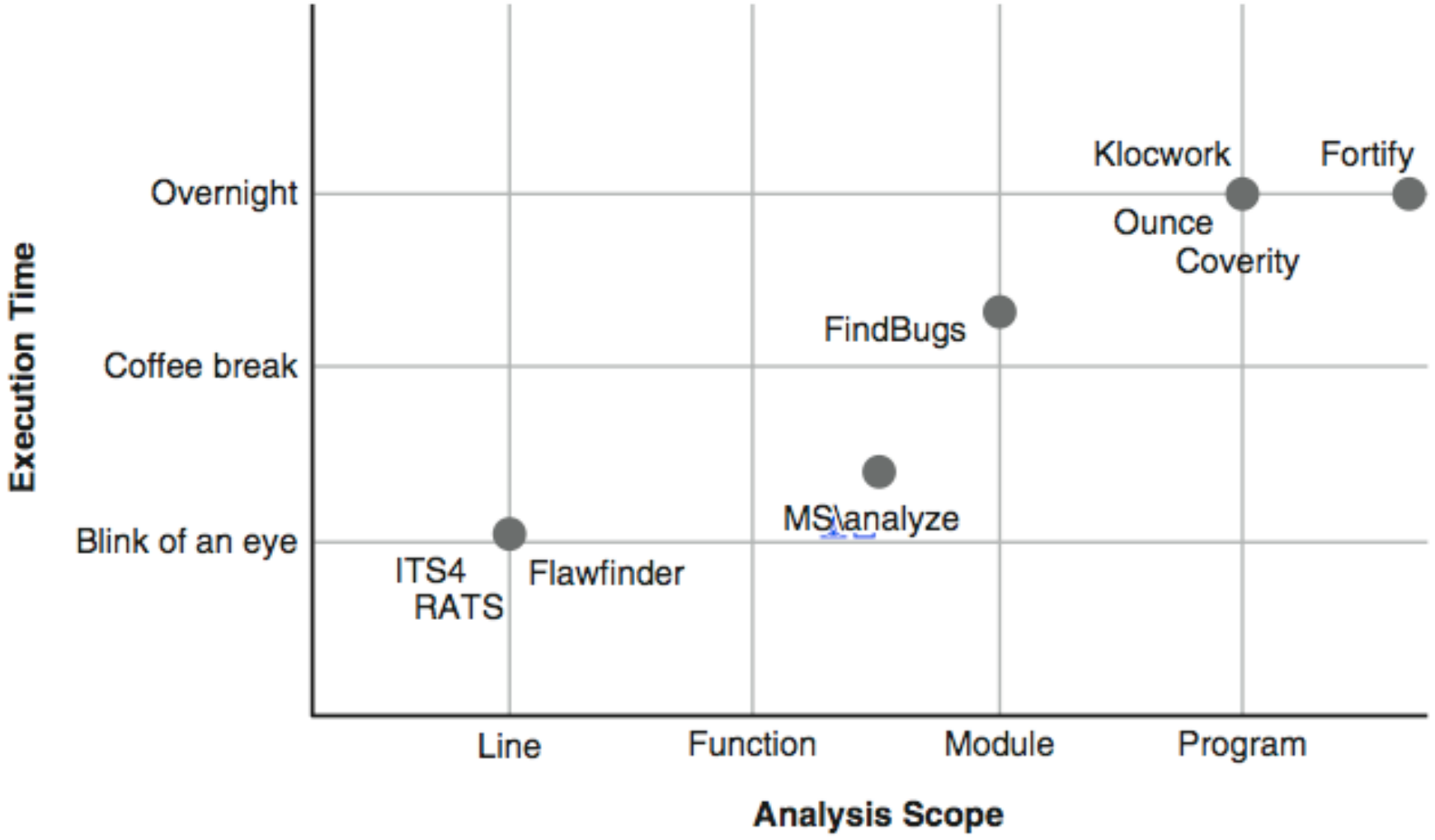
- **False positives**
  - **Incomplete/inaccurate model**
  - **Conservative analysis**
  - **Missing rules**
- **False negatives**
  - **Incomplete/inaccurate model**
  - **"Forgiving" analysis**
  - **Missing rules**



Untapped Potential: Customization

- Improve tool understanding of the program
  - Model the behavior of third-party libraries
  - Describe program semantics
- Identify program-specific vulnerabilities
  - Call out targets for manual review
  - Enforce specific coding standards
  - Find vulnerabilities in custom interfaces

Scope vs. Performance



Experiment

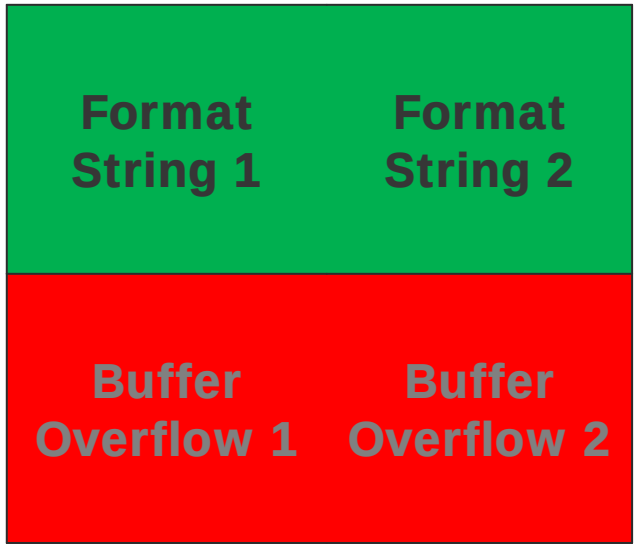
- **Select project: open source mail daemon**
  - **qwik-smtpd version .3**
  - **Contains multiple known vulnerabilities**
- **Select tools: fuzzing and static analysis**
  - **Fuzzing: @stake SMTP Fuzz 0.9.16**
    - Customized for SMTP protocol
  - **Static analysis: Fortify**
    - The one we have sitting around
- **Collect data:**
  - **Run fuzzing tool on SMTP protocol**
  - **Run static analysis tool on C source code**

What We Found

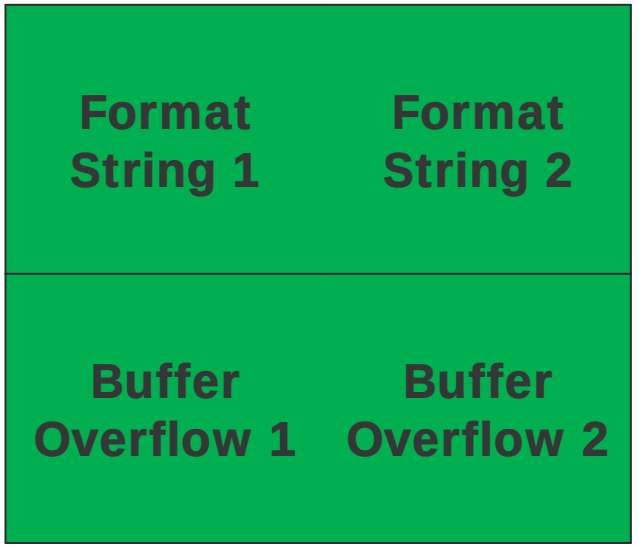
- Identified four remotely exploitable bugs
  - Two buffer overflows
  - Two format string vulnerabilities
- And numerous other locally exploitable vulnerabilities, including:
  - Buffer overflows
  - Format string vulnerabilities
  - Command injection
  - Memory errors
  - Resource leaks
  - ...

Results

- Fuzzing found both remotely exploitable format string bugs, but missed both remotely exploitable buffer overflows
- Static analysis: Found all four vulnerabilities



Fuzzing



Static Analysis

Conclusions

- **Fuzzing…**
  - **Found exploitable vulnerabilities fast**
  - **Missed critical bugs within its reach**
  - **Missed vulnerabilities from non-SMTP sources**
  - **Would miss bugs behind complex conditions (bugs hidden behind multiple header conditions)**

Advantages of Fuzzing over Static Analysis

- **Less involved**
  - **Does not require access to or understanding of code**
- **Access to context**
  - **Does not requires customization to understand program semantics and context**
- **The last step**
  - **Produces a demonstrable exploit or test case without further human efforts**

Advantages of Static Analysis Over Fuzzing

- **Thoroughness**
  - Considers every source of input
  - Considers every path through the program
- **Speed**
  - Doesn’t require running the code
  - Customization has almost no impact on performance
- **Visibility**
  - Identifies vulnerabilities hidden by error handling
  - Finds vulnerabilities evidenced through that may be hidden

Summary

- Static analysis is spot-on for security
- Important attributes
  - Language support
  - Analysis techniques
  - Rule set
  - Performance
  - Results management
- Customization
  - Describe program semantics
  - Model program context

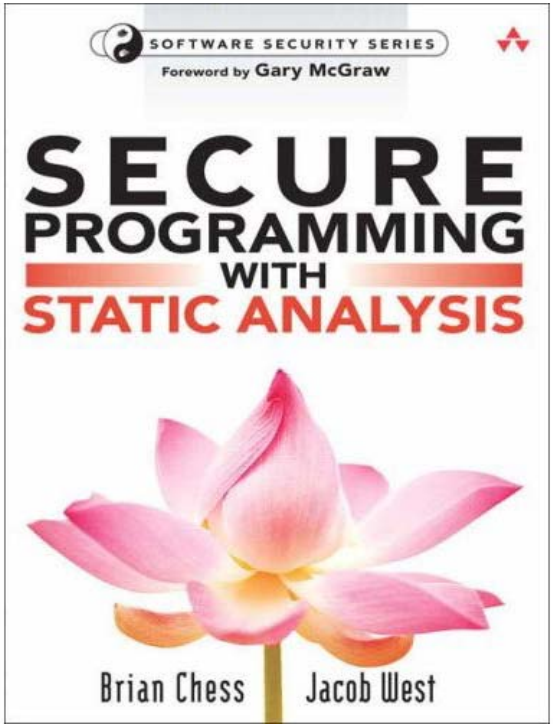
<end>

- PDF of talk will be available here:

  http://www.fortify.com/presentations

- Send me email!

  Jacob West <jacob@fortify.com>



Secure Programming with Static Analysis