Iron Chef Black Hat

Brian Chess, PhD
Sean Fay

Toshinari Kureha
Jacob West

Black Hat
August 2nd, 2007
Las Vegas

Agenda

- Introduction
  - Iron chef concept
  - Introduce chairman, co-host and chefs
  - Reveal secret code base
  - Chefs introduce their tools and techniques
- Contest
  - Chefs look for security holes using their tools / techniques.
- Judging
  - Static analysis findings
  - Runtime analysis findings
  - Winner announced

Iron Chef Concept

- Multiple chefs
- Each an expert in a different area
- Compete to win high scores from judges
  - Partly based on what they produce
  - Partly based on how they present their productions

Chairman, Co-host and Chefs

- **Brian Chess**
- **Sean Fay**
- **Toshinari Kureha**
- **Jacob West**

Secrete Code Base

- **TBA**

Sean Fay – Static Analysis

Where We are Today

- Small coding errors can have a big effect on security
- Typical software development practices don't address the problem
- As a group, developers tend to make the same security mistakes over and over


Husqvarna
137

Static Source Code Analysis

- Benefits
  - 1000x faster than code review
  - Security knowledge built in
  - Consistent
- Limitations
  - Does not understand architecture
  - Does not understand application semantics
  - Does not understand social context

The Many Faces of Static Analysis

- Type checking
- Style checking
- Program understanding
- Program verification / Property checking
- Bug finding
- Security review

Type Checking

- **Taken for granted**
- **Imperfect:**

```
short s = 0;
int i = s; /* the type checker allows this */
short r = i; /* false positive: this will cause a
                type checking error at compile time. */


/* false negative:
   passes type checking, fails at runtime */
Object[] objs = new String[1];
objs[0] = new Object();
```

Style Checking

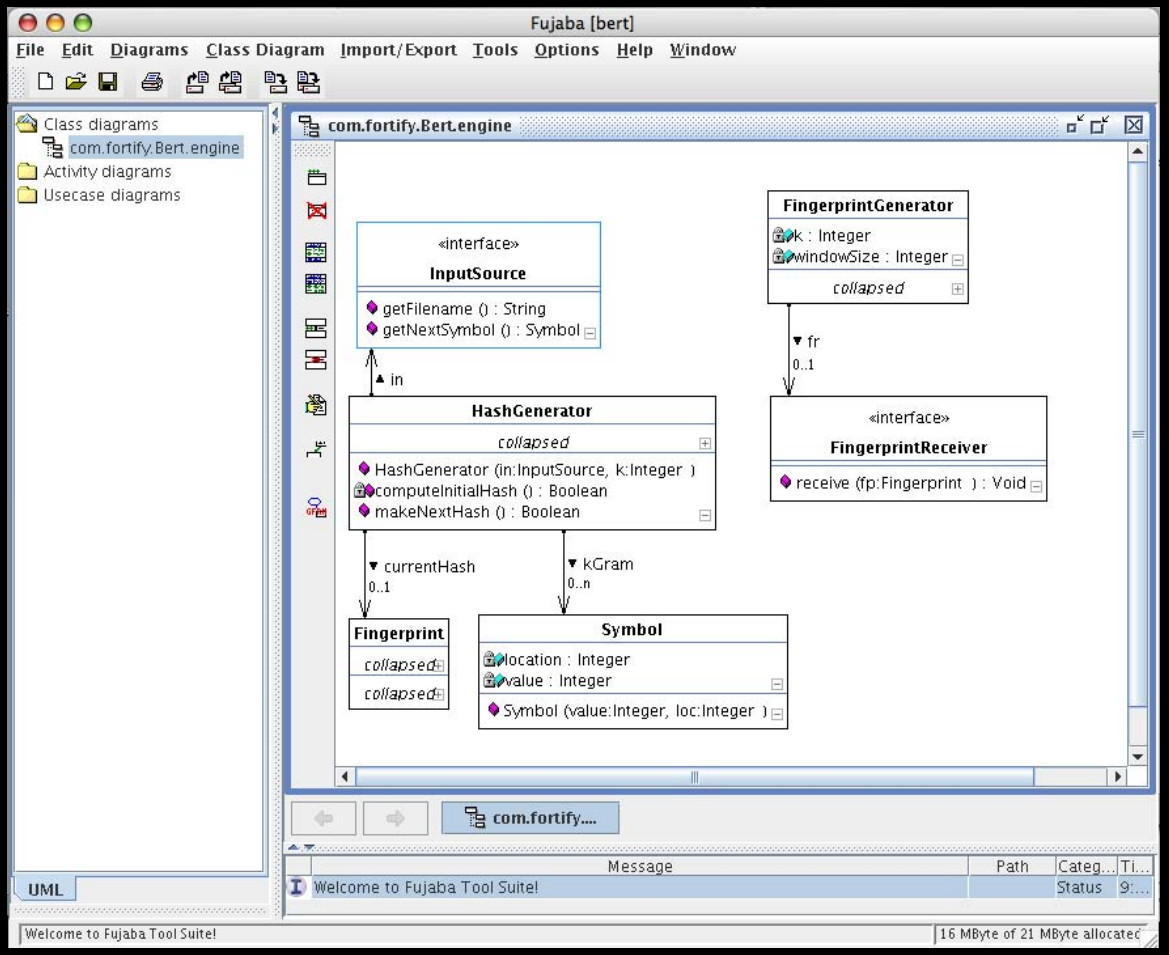
- Pickier than type checker, might look at
  - Whitespace
  - Naming
  - Deprecated functions
- gcc -Wall does some style checking

```
typedef enum { red, green, blue } Color;
char* getColorString(Color c) {
  char* ret = NULL;
  switch (c) {
    case red:
      printf("red");
  }
  return ret;
}
```

- Tools
  - Lint
  - PMD

Program Understanding

- Help make sense of a large codebase
- Tools:
  - Fujaba
  - Klockwork
  - CAST Systems



Program Verification / Property Checking

- *Prove* that a program has particular properties
- Partial specification -> property checking
- Often focuses on *temporal safety properties*
- Example: allocated memory must always be freed

```
inBuf = (char*) malloc(bufSz);
if (inBuf == NULL)
  return -1;
outBuf = (char*) malloc(bufSz);
if (outBuf == NULL)
   return -1; /* memory leak */
```

- Soundness
  - Aspires to “Sound WRT the specification”: reports a bug if one exists
- Tools:
  - Praxis High Integrity Systems
  - PolySpace
  - GrammaTech

Bug Finding

- More sophisticated than a style checker
- Less ambitious than program verification
- Search code for “bug idioms”
- Find high-confidence, low noise results (low false positives)
- Soundness
  - Aspires to “Sound WRT counterexample”: never reports a bug that isn’t a bug
- Example: double checked locking

```
if (fitz == null) {
    synchronized (this) {
      if (fitz == null) {
        fitz = new Fitzer();
      }
    }
}
```

- Tools:
  - FindBugs
  - Coverity
  - Klocwork
  - Prefast

Security Review

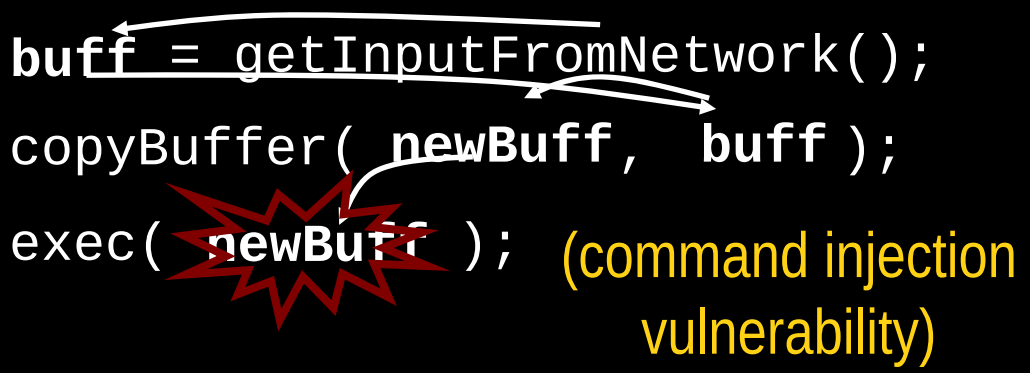
- Focus on finding exploitable code
- Find high-risk code constructs for review (low false negatives)
- Example

```
int main(int argc, char* argv[]) {
  char buf1[1024];
  char buf2[1024];
  char* shortString = "a short string";
  strcpy(buf1, shortString); /* innocuous use of strcpy */
  strcpy(buf2, argv[0]);     /* dangerous use of strcpy */
  ...
```

- Tools
  - RATS, ITS4, FlawFinder
  - Fortify Software and Ounce Labs, ~~Secure Software~~

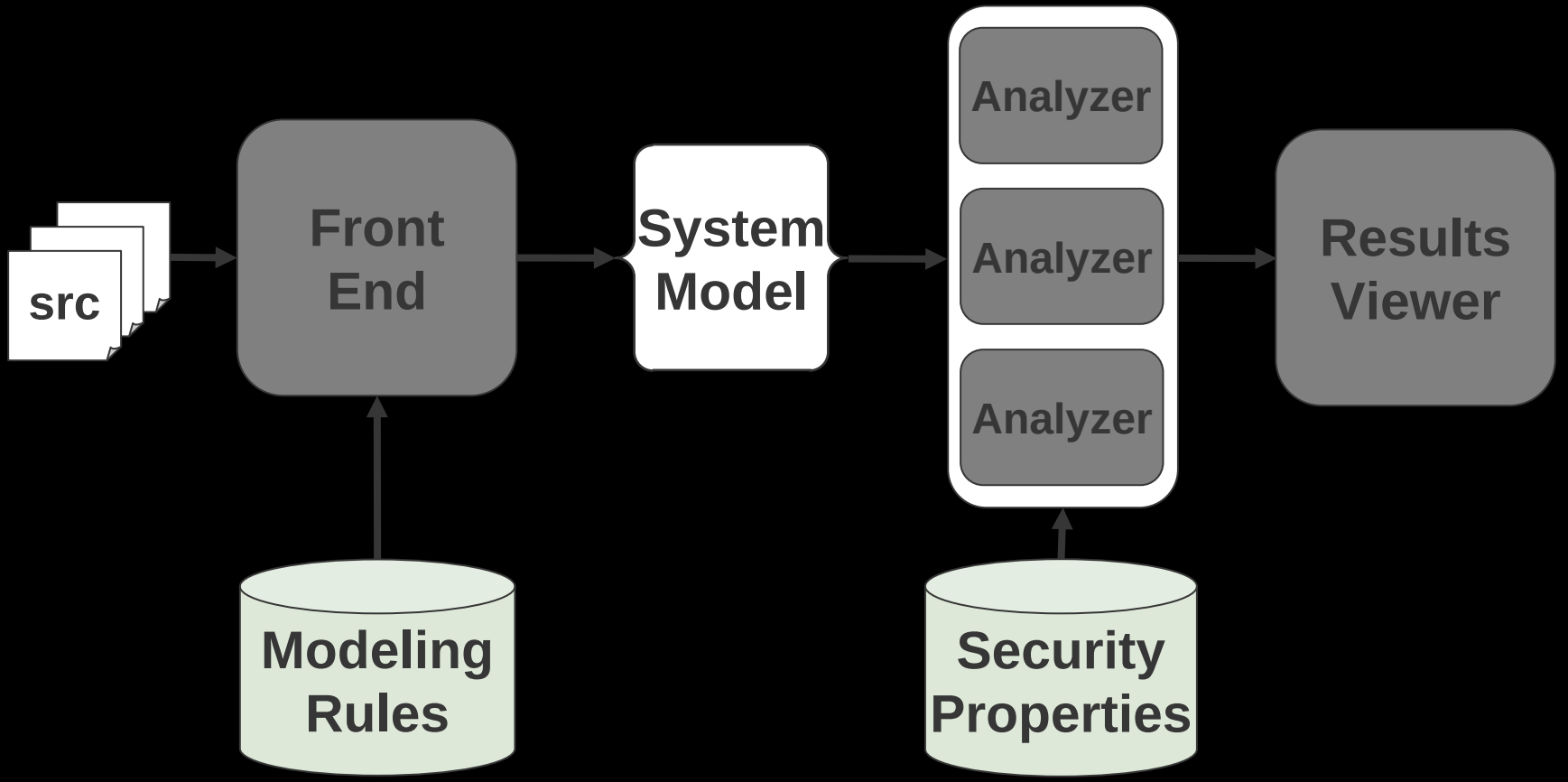
Security Example: Dataflow Analysis

- **Trace potentially tainted data through the program**
- **Report locations where an attacker could take advantage of a vulnerable function or construct**



- **Real-world cases usually cross method boundaries**

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()`

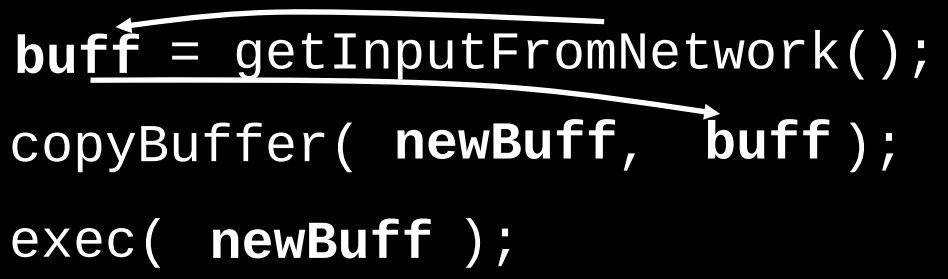
```
buf = realloc(buf, 256);
```

- 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**



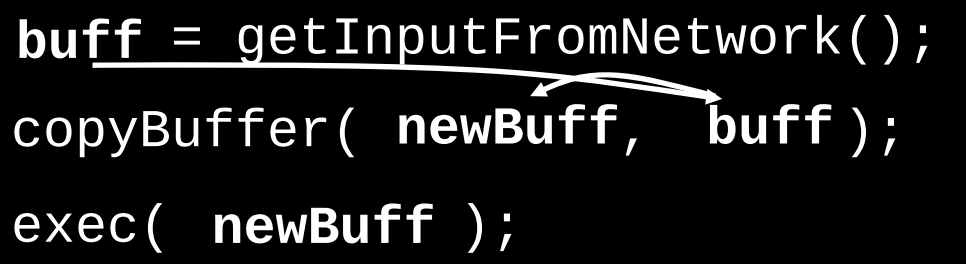
- **Source rule:**

Function: getInputFromNetwork()

Postcondition: return value is tainted

Analysis / Rules: Dataflow Pass-Through Rule

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



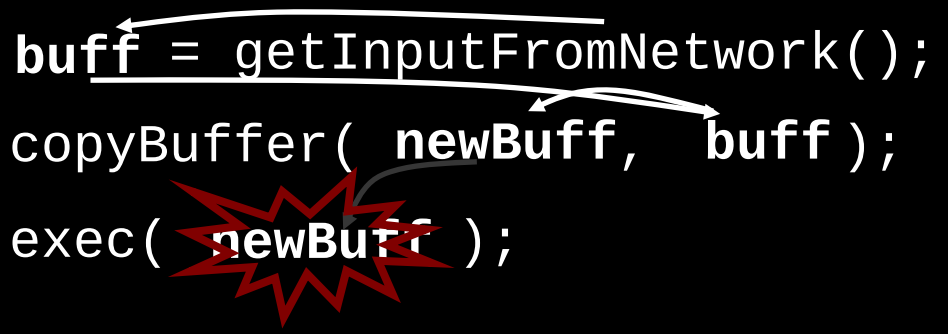
- **Pass-through rule:**

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:**

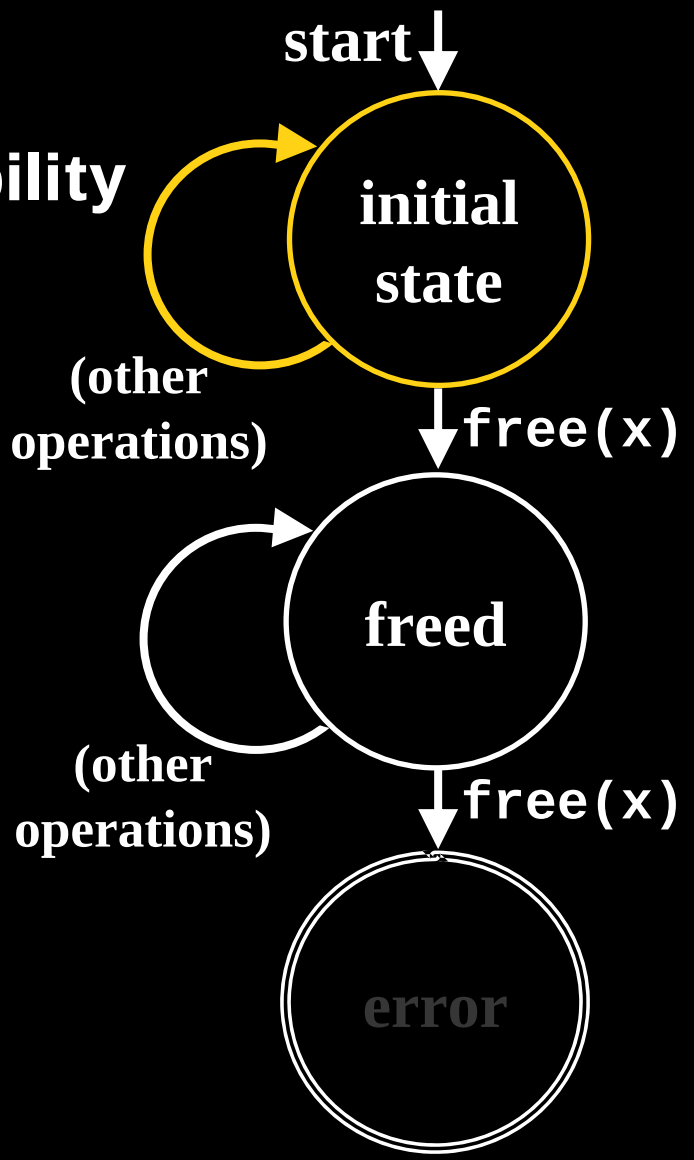
Function: exec()

Precondition: the first argument must not be tainted

Analysis / Rules: Control Flow

- **Look for dangerous sequences**
- **Example: Double-free vulnerability**

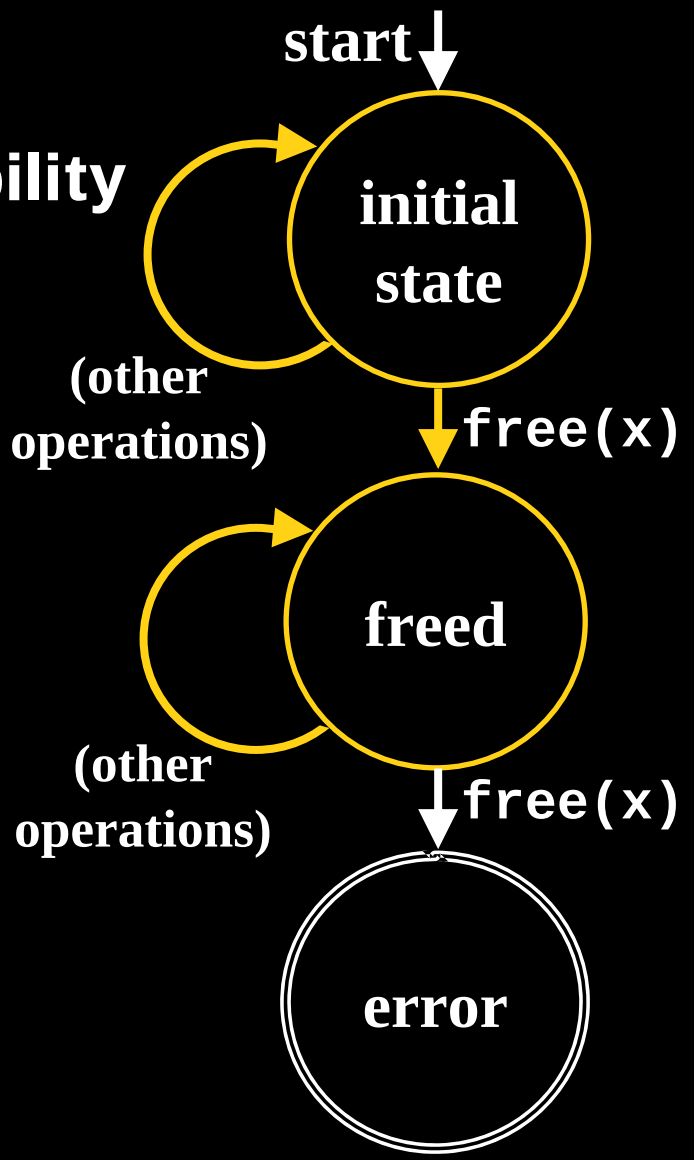
```
while ((node = *ref) != NULL) {
  *ref = node->next;
  free(node);
  if (!unchain(ref)) {
    break;
  }
}
if (node != 0) {
  free(node);
  return UNCHAIN_FAIL;
}
```



Analysis / Rules: Control Flow

- **Look for dangerous sequences**
- **Example: Double-free vulnerability**

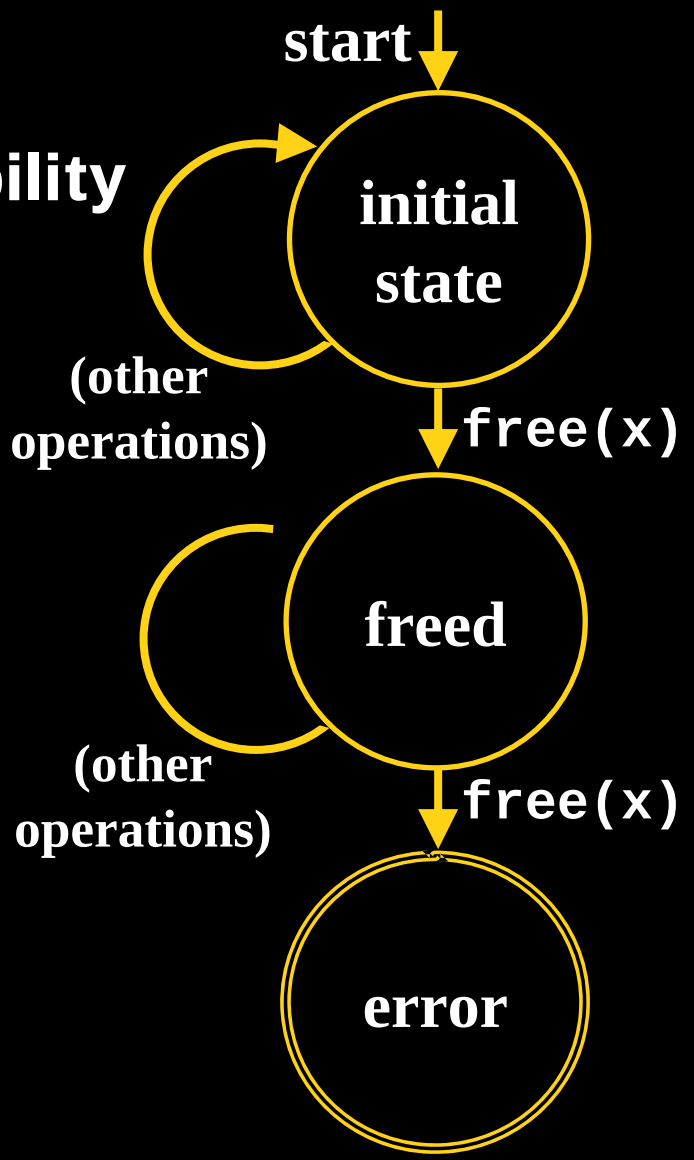
```
while ((node = *ref) != NULL) {
  *ref = node->next;
  free(node);
  if (!unchain(ref)) {
    break;
  }
}
if (node != 0) {
  free(node);
  return UNCHAIN_FAIL;
}
```



Analysis / Rules: Control Flow

- **Look for dangerous sequences**
- **Example: Double-free vulnerability**

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



Critical Attributes

- Capacity
  - Ability to gulp down millions of lines of code
- Rule set
  - Modeling
  - Security properties
- Results management
  - Prioritization of issues
  - Control over what to check for
  - Don’t show the same bad result twice

Common Problems

- False positives
  - Incomplete/inaccurate model
  - Conservative analysis
- False negatives
  - Incomplete/inaccurate model
  - Missing rules
  - “Forgiving” analysis

Untapped Potential

- Customization
  - Check properties that are unique to your application or organization
- Design for testability
  - Write code knowing that it will be checked

Summary

- Lots of static analysis tools for different purposes
- Static analysis is spot-on for security
- Tools make the process more efficient
- Important attributes
  - Language support
  - Analysis techniques
  - Rule set
  - Performance
  - Results management
  - Customization

Toshinari Kureha – Runtime Analysis

My Approach

Traditional Vulnerability Scanner

+

White Box Enabling Tool

Traditional Vulnerability Scanning

- **Traversing The Application**
- **Testing The Application**

Traversing The Application

- Automatic Web Crawl
- Manual Web Crawl
- Importing Web Crawl

Automatic Web Crawl

- Enter Starting URL, Click “Go”
- Advantages
  - Easy to use
  - May be comprehensive
- Disadvantages
  - Cannot crawl complex web applications
  - Make take a long time, looping redundant pages

Manual Web Crawl

- Tell The Crawler How To Crawl – Manually Crawl Once First
- Advantages
  - Can often achieve higher coverage
- Disadvantages
  - Time Consuming

Importing Web Crawl

- Leverage Existing “Crawl”, Such As QA Test Cases
- Advantages
  - No need to additional work
  - High coverage
- Disadvantage
  - May not have an existing crawl

My Vulnerability Scanner

- **Traversing The Application**
  - **Automatic Crawl Support**
  - **Integrated Into QA Environment**
    - Manual Crawl
    - Leverage Existing QA Scripts

Testing The Application

- **Fuzz The Application And Perform**
  - **Signature Analysis**
  - **Behavior Analysis**

Signature Analysis

- **Find Certain Strings In The HTTP Response**
  - **“Unclosed quotation”**
  - **“SQLException”**
  - **“OLE DB Provider”**

“If this string didn’t appear before fuzzing and appears now, then it’s a SQL Injection!”

Behavior Analysis

- **Find Behaviors That Indicate Vulnerability**
  - **E.g. Blind SQL Injection**
    - Inject original clause: id=3
    - Inject true clause: id=3 AND 1=1
    - Inject false clause: id=3 AND 1=0
    - If (original==true && true != false)
      then it's a SQL Injection vulnerability

My Vulnerability Scanner

- **Traversing The Application**
  - **Automatic Crawl Support**
  - **Integrated Into QA Environment**
    - Manual Crawl
    - Leverage Existing QA Test Cases
- **Testing The Application**
  - **Signature Matching**
  - **Behavior Matching**
    - Original Behavior can be derived from QA Test Cases

Vulnerability Scanner Limitations

- Unknown Coverage
- False Positives
- False Negatives
- No Remediation Information

My “White Box” Enabling Tool

- Insert Monitors Inside The Web Application

- What: Report Coverage, Remediation, and API-level Analysis
- Where: Security Relevant Points
- How: Aspect Technology

What The Monitors Do

- Coverage
  - Report Whether This Monitor Was Executed
- Remediation
  - Provide File, Line Number, API Information
- API-Level Analysis
  - Determine Whether Vulnerabilities Are Happening or Can Happen

Where To Inject Monitors

- **Security Relevant APIs**
  - **All Web Inputs**
    - ServletRequest.getParameter(String)
  - **All Sinks**
    - Database: SQLStatement.executeQuery(String)
    - Process: Runtime.exec(String)
    - File: Log.log(String)

How To Inject Monitors

- Use Aspect Oriented Technology
  - AspectJ
  - AspectDNG
- Monitor Code Is Written Via Aspects
- Bytecode Injection – Java Class Files & .NET MSIL

My “White Box” Enabling Tool

- Provides Coverage Information
  - Which Monitors Were Hit
- Provides Remediation Information For Developers
  - Root Cause Filename, Line Number, API Called
- Fewer False Positives
  - Accurate Analysis At API Level
- Fewer False Negatives
  - Analyzes Non-HTTP Level Vulnerabilities

Summary

- Vulnerability Scanner – “Test Driver”
  - Smart Fuzzer
  - Signature & Behavior Analysis
- White Box Enabling Tool – “Analyzer”
  - Inject Monitors
  - Coverage, Remediation, Better Accuracy

Contest

- **Chefs.... BEGIN!**

Judging

- **TBA**

<end>

- PDF for talk available here:
  http://www.fortifysoftware.com/presentations

- Send us e-mail!
  Brian Chess <brian@fortifysoftware.com>
  Sean Fay <sean@fortifysoftware.com>
  Toshinari Kureha <tkureha@fortifysoftware.com>
  Jacob West <jacob@fortifysoftware.com>