Approaches and Tools for code Analysis

Adapted from David A. Wheeler

Outline

- Types of analysis (static/dynamic/hybrid)
  - Some measurement terminology
- Static analysis
- Dynamic analysis (fuzz testing)
- Hybrid analysis
- Operational
- Fool with a tool
- SWAMP
- Adopting tools

Types of analysis

- Static analysis: Approach for verifying software (including finding defects) without executing software
  - Source code vulnerability scanning tools, code inspections, etc.
- Dynamic analysis: Approach for verifying software (including finding defects) by executing software on specific inputs and checking results ("oracle")
  - Functional testing, fuzz testing, etc.
- Hybrid analysis: Combine above approaches
- Operational: Tools in operational setting
  - Minimize risks, report information back, etc.
  - may be static, dynamic, hybrid; often dynamic

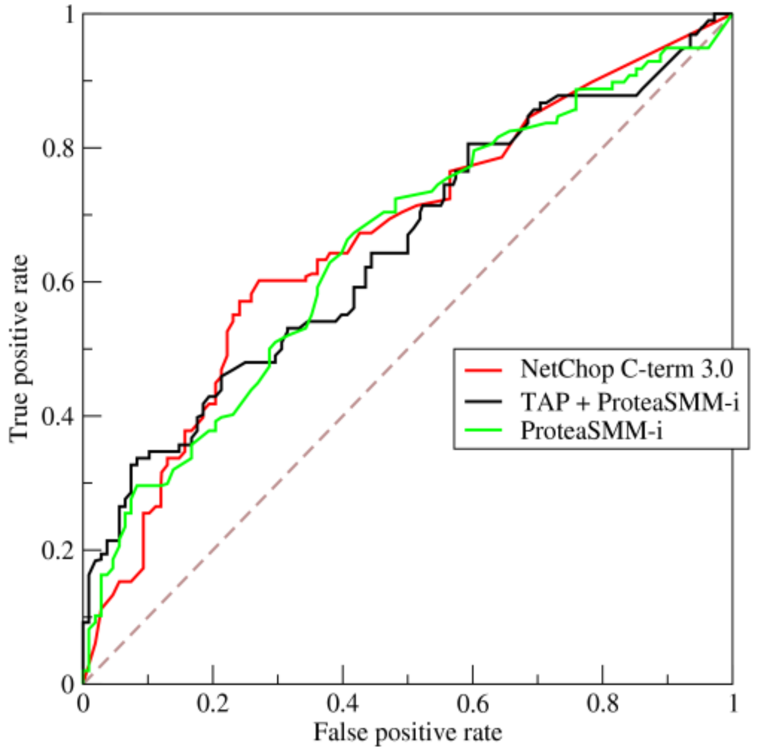
Basic measurement terminology

Analysis/tool report	Report correct	Report incorrect
Reported a defect	**True positive (TP):** Correctly reported a defect	**False positive (FP):** Incorrect, it reported a "defect" that's not a defect ("Type I error")
Did not report a defect (there)	**True negative (TN):** Correctly did not report a (given) defect	**False negative (FN):** Incorrect because it failed to report a defect ("Type II error")

- False positive rate, FPR = #FP/(#TP+#FP) "Probability alert is false"
- True positive rate, TPR = #TP/(#TP + #FN) "% vulnerabilities found" (sensitivity)
- Developers worry about large false positive rate (FPR)
  - "Tool report wasted my time"
- Auditors worry about small or <100% TPR for a given category
  - "Tool missed something important"

Receiver operating characteristic (ROC) curve

- Binary classifiers must generally trade off between FP rates and TP rates
  - To get more reports (larger TP rate), must accept larger FP rate
  - What's more important to you, low FP rate or high TP rate?
- ROC curve (from WW II) graphically illustrates this
- Don't normally know the true values for given tools, but effect is still pronounced
  - Tool developer focus
  - Tool users can configure tool to affect trade-off



Sample ROC curve
[Source: Wikipedia "ROC curve"]

Measurement roll-ups

- Precision (true positive rate) = #TP/(#TP+#FP)
- Recall (sensitivity, soundness, find rate) = #TP/(#TP+#FN)
- F-score (harmonic mean) =
  2 x (Precision x Recall) / (Precision + Recall)
- Discrimination rate = #Discriminations / #Test_Pairs
  - Given a pair of tests (one with defect, one without)
  - Discrimination occurs if tool correctly reports flaw (TP) in test with flaw AND does not when there is no flaw (TN)
- All these values 0..1, where higher is better

Source: CAS Static Analysis Tool Study - Methodology (Dec 2011)

http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf

Some tool information sources

- Software SOAR ("State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation") by David A. Wheeler and Rama S. Moorthy, IDA Paper P-5061
  - http://www.acq.osd.mil/se/docs/P-5061-software-soar-mobility-Final-Full-Doc-20140716.pdf
  - "Appendix E" has a large matrix of different types of tools
- NIST SAMATE (http://samate.nist.gov)
  - "Classes of tools and techniques": http://samate.nist.gov/index.php/Tool_Survey.html
  - Can test tools using Software Assurance Reference Dataset (SARD), formerly known as the SAMATE Reference Dataset (SRD). It's a set of programs with known properties: http://samate.nist.gov/SARD/
- Build security in (https://buildsecurityin.us-cert.gov)
  - Software Assurance (SwA) Technology and tools working group
  - Overview of SwA tools:

    https://buildsecurityin.us-cert.gov/swa/swa_tools.html
  - NAVSEA "Software Security Assessment Tools"

    https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Tools-Paper-2009-03-02.pdf
- NSA Center for Assured Software (CAS)
- OWASP (https://www.owasp.org)

Static analysis

Static analysis: Source vs. Executables

- Source code pros:
  - Provides much more context; executable-only tools can miss important information
  - Can examine variable names and comments (can be very helpful!)
  - Can *fix* problems found (hard with just executable)
  - Difficult to decompile code
- Source code cons:
  - Can mislead tools – executable runs, not source (if there's a difference)
  - Often cannot get source for proprietary off-the-shelf programs
    - Can get for open source software
    - Often can get for custom
- Bytecode is somewhere between

(Some) Static analysis approaches

- Human analysis (including peer reviews)
- Type checkers
- Compiler warnings
- Style checkers / defect finders / quality scanners
- Security analysis:
  - Security weakness analysis - text scanners
  - Security weakness analysis - beyond text scanners
- Property checkers
- Knowledge extraction
- formal methods (separately)

Different people will group approaches in different ways

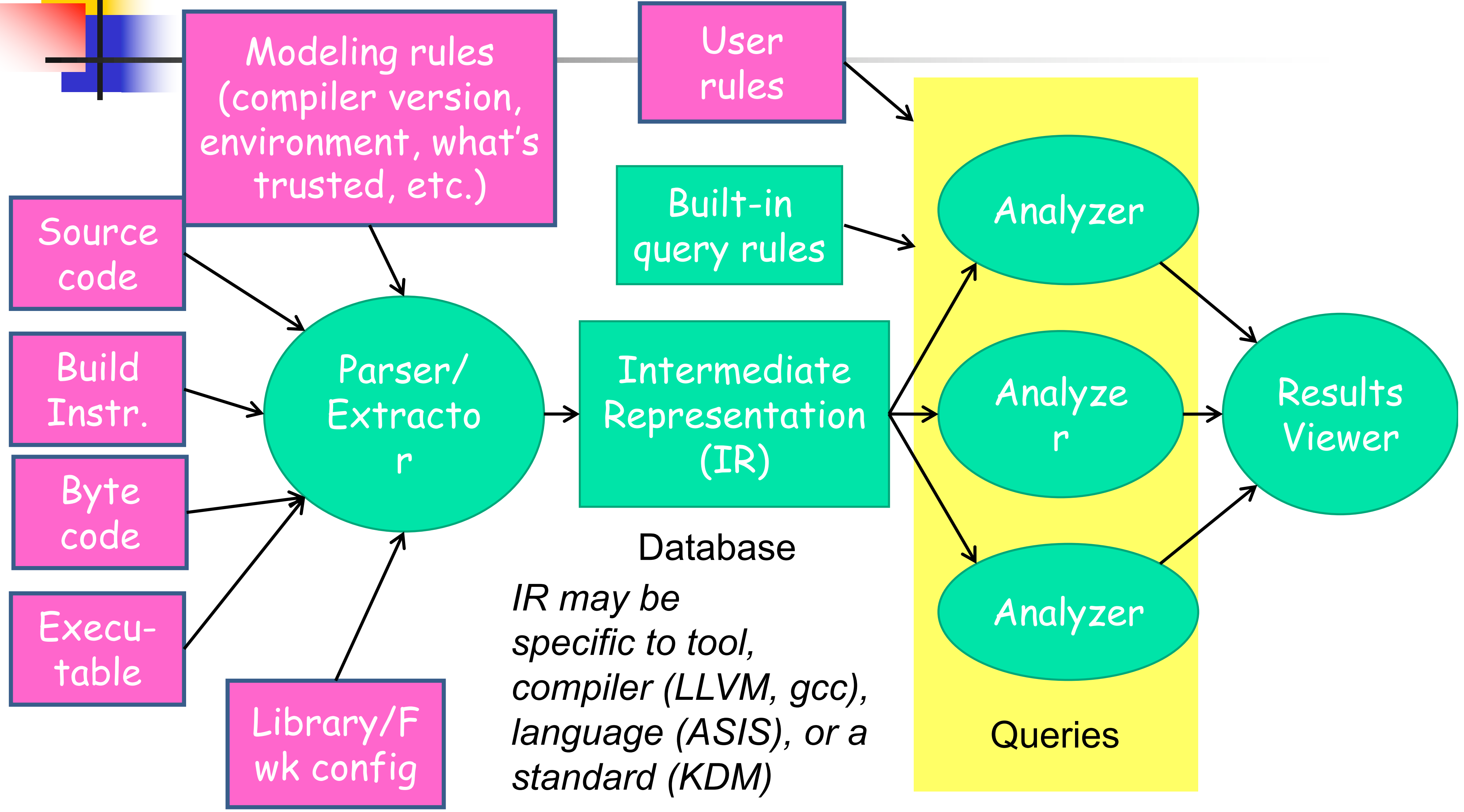
Human (manual) analysis

- Humans are great at discerning context and intent
- Get bored and overwhelmed
- Expensive
  - Especially if analyzing executables
- Can be one person, e.g., "desk-checking"
- Peer reviews
  - Inspections: Special way to use group, defined roles including "reader"; see IEEE standard 1028
- Can focus on specific issues
  - e.g., "Is everything that's *supposed* be authenticated covered by authentication processes?"

Automated tool limitations

- Tools typically don't "understand":
  - System architecture
  - System mission/goal
  - Technical environment
  - Human environment
- Except for formal methods…
  - Most have significant FP and/or FN rates
- Best when *part* of a process to develop secure software, not as the only mechanism

Typical static analysis tool



IR may be specific to tool, compiler (LLVM, gcc), language (ASIS), or a standard (KDM)

Static analysis tools not specific to security can still be useful

- Many static analysis tools' focus is other than security
  - may look for generic defects, or focus on "code cleanliness" (maintainability, style, "quality"etc.) but some defects are security vulnerabilities
  - Reports that "clean" code is easier for other (security-specific) static analysis to analyze (for fewer false positives/negatives)
    - probably easier for humans to review too
    - no hard evidence, though; some would be welcome!
  - Such tools often faster, cheaper and easier
    - many do not need to do whole-program analysis
- Such tools may be useful in reducing as a precursor step before using security-specific tools
- Java users: Consider quality scanners FindBugs or PMD

Type checkers

- Many languages have static type checking built in
  - Some more rigorous than others
  - C/C++ not very strong (and must often work around)
  - Java/C# stronger (interfaces, etc., ease use)
- Can detect some defects before fielding
  - Including some security defects
  - Also really useful in documenting intent
- Work *with* type system – be as narrow as you can
  - Beware diminishing returns

Compiler warnings: Not security-specific but useful

- Where practical, enable compiler/interpreter warnings and fix anything found
  - E.g., gcc "-Wall -pedantic -Wextra", perl's "use strict"
  - Include in implementation/build commands
    - Autoconf: GNU autoconf archive https://www.gnu.org/software/autoconf-archive/
    - AX_CFLAGS_WARN_ALL
    - AX_APPEND_COMPILE_FLAGS([-Wextra])
    - AX_APPEND_COMPILE_FLAGS([-pedantic])
  - "Fix" so no warning, even if technically not a problem
    - That way, any warning is obviously a new issue
- Turn on run-time warnings too:
  - May detect security vulnerabilities
  - Improve other tools' results (fewer false results)
  - Often hard to turn on later
    - Code not written with warnings in mind may require substantial changes before it reports no warnings

Style checkers / Defect finders / Quality scanners

- Compare code (usually source) to set of pre-canned “style” rules or probable defects
- Goal:
  - Make it easier to understand/modify code
  - Avoid common defects/mistakes, or patterns likely to lead to them
- Some try to have low FP rate
  - Don’t report something unless it’s a defect

PMD vs. FindBugs

- Both PMD and FindBugs:
  - Focus on "quality" issues, not security
  - Open source software, available no cost
  - Operate on Java
- PMD
  - Works on (Java) source code
  - Examples: Violation of naming conventions, lack of curly braces, misplaced null check, unnecessary constructor, missing break in switch
  - Also provides Cyclomatic complexity
- FindBugs
  - Works on (JVM) bytecode
  - Examples: equals() method fails on subtypes, clone method may return null, reference comparison of Boolean values, impossible cast, 32bit int shifted by an amount not in the range of 0-31, a collection which contains itself, equals method always returns true
- Neither good at finding security issues – not their purpose

Security defect text scanners

- Scan source code using simple grep-like lexer
  - Typically "know" about comments and strings
  - Look for function calls likely to be problematic
- Examples: RATS, ITS4, Flawfinder (D.A.Wheeler)
- Pros:
  - Fast and cheap
  - Can process partial code (including un-compilable code)
- Cons:
  - Lack of context leads to large FN & FP rates
  - Useful primarily for warning of "dangerous" functions

Security defect finders

- Read software and create internal model of software
- Look for patterns likely to lead to security defects
- Examples
  - Proprietary: HP/Fortify, Coverity
  - OSS: splint (for C), LAPSE+ (for Java)

Analysis approach: Examining structure / method calls

- Warn about calls to gets():

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

Source: Brian Chess and Jacob West

Analysis Approach: Data flow - Taint propagation

- Many tools (static and dynamic) perform "taint propagation"
  - Input from untrusted users ("sources") considered "tainted"
  - Warn/forbid sending tainted data to certain methods and constructs ("sinks")
  - Some operations (e.g., checking) may "untaint" data
- Static analysis:
  - Follow data flow from sources through program
  - Determine if tainted data can get to vulnerable "sink"
- Dynamic analysis (e.g., Perl, Ruby):
  - Variables have "taint" value set when input from some sources
  - Certain operations (sinks) forbid direct use of tainted data
    - Counters accidental use of untrusted and unchecked data
    - esp. useful on injection (SQL, command) and buffer overflow

Taint propagation example

```
buffer = getUntrustedInputFromNetwork();   // Source
copyBuffer(newBuffer, buffer);   // Pass-through
exec(newBuffer); // Sink
```

- Source rule:
  - Function: getUntrustedInputFromNetwork()
  - Postcondition: return value is tainted
- Pass-through rule:
  - Function: copyBuffer()
  - Postcondition: If arg2 tainted, then arg1 tainted
- Sink rule:
  - Function: exec()
  - Precondition: Arg1 must not be tainted

In real code, taint propagation typically flows through many different methods

Source: Brian Chess and Jacob West

Property checkers

- "Prove" that a program has very specific narrow property
- Typically focuses on very specific temporal safety, e.g.:
  - "Always frees allocated memory"
  - "Can never have livelock / deadlock"
- Many strive to be sound ("reports all possible problems")
- Examples: GrammaTech, GNATPro Praxis, Polyspace

Knowledge extraction / program understanding

- Create view of software automatically for analysis
  - Especially useful for large code bases
  - Visualizes architecture
  - Enables queries, translation to another language
- Examples:
  - Hatha Systems' Knowledge Refinery
  - IBM Rational Asset Analyzer (RAA)
  - Relativity MicroFocus (COBOL-focused)

Source/Byte/Binary code security scanners/analyzers - some lists

- http://samate.nist.gov/index.php/Tool_Survey.html
  - Click on “Source Code Security Analyzers”, “Byte Code Scanners”, and “Binary Code Scanners”
- http://www.dwheeler.com/flawfinder

Dynamic analysis

Dynamic analysis' fundamental issue: Cannot test all inputs

- Given trivial program "add two 64-bit integers"
  - Input space = $(2^{64}) (2^{64}) = 2^{128}$ possibilities
- Checking "all inputs" not realistic even in this case
  - Given 4GHz processor & 5 cycles/input (too fast):

    time=2^{128} inputs * (5 cycles/input) * (1 second/(4GHz cycles))

    = 1.35×10^{22} years (13.5 zettayears aka sextillion years)
  - Using 1 million 8-core processors doesn't help:

    time=1.7×10^{15} years (petayears aka quadrillion years)
- Real programs have far more complex inputs
  - Even a 1% sample impossible in human lifetimes

Why dynamic analysis' weakness is especially important to security

- Security (and safety) requirements often have the form "X never happens" (negative requirement)
  - Easier to show there is at least one case where something happens than to show it never happens
- Continuous systems: Check boundaries
  - But digital systems are fundamentally discontinuous
- Dynamic analysis can only be a part of developing secure software process – but has some value

Functional testing for security

- Use normal testing approaches, but add tests for security requirements
  - Test both “should happen” and “should not happen”
  - Often people forget to test what “should *not* happen”
    - “Can I read/write without being authorized to do so?”
    - “Can I access the system with an invalid certificate?”
- Branch/statement coverage tools may warn you of untested paths
- As always, automate and rerun

Web application scanners

- Attempt to go through the various web forms and links
- Send in attack-like and random data
  - Key issues: Input vector (Query string? HTTP body? JSON? XML?), scan barrier, crawl / input vector extraction, which vulnerabilities it detects (and how well)
  - Often build on "fuzzing" techniques (discussed next)
- Shay Chen reviews many in "The Web Application Vulnerability Scanners Benchmark"
  - Compares effectiveness on "WAVSEP". See http://www.sectoolmarket.com and 2014 discussion in http://sectooladdict.blogspot.com/2014/02/wavsep-web-application-scanner.html

Fuzz testing ("fuzzing")

- Testing technique that:
  - Provides (many!) invalid/random input to inputs
  - Monitors program for crashes and other signs of trouble (failing code assertions, appearance of memory leaks)… *not* if the final answer is "correct" (this process is the "oracle")
- Simplifies "oracle" so can create massive data set
- Do not need source, might not even need executable
- Often quickly finds a number of real defects
  - Attackers use it; do not have easy-to-find vulnerabilities
- Can be *very* useful for security, often finds problems
- Typically diminishing rate of return

Fuzz testing history

- Fuzz testing concept from Barton Miller's 1988 class project Univ. of Wisconsin
  - Project created "fuzzer" to test reliability of command-line Unix programs
  - Repeatedly generated random data for them until crash/hang
  - Later expanded for GUIs, network protocols, etc.
- Approach quickly found a number of defects
- Many tools and approach variations created since

Fuzz testing variations: Input

- Test data creation approaches:
  - Mutation based: mutate existing samples to create test data
  - Generation based: create test data based on model of input
    - Including fully random, but that often has poorer coverage
  - May try to create "likely security vulnerability" patterns (e.g. metachars) to increase value
- May concentrate on mostly-valid or mostly-invalid
- Type of input data: File formats, network protocols, environment variables, API call sequences, database contents, etc.
- Input selection may be based on other factors, including info about program (e.g., uncovered program sections)

Fuzz testing variations: oracle

- Originally, just "did it crash / hang"?
- Adding program assertions (enabled!) can reveal more
- Test other "should not happen"
  - Ensure files/directories unchanged if shouldn't be
  - Memory leak (e.g., valgrind)
  - Invalid memory access, e.g., using AddressSanitizer (aka ASan) for C/C++/Objective-C to detect buffer overflows and double-frees
  - More intermediate (external) state checking
  - Final state "valid" (! = "correct")

Sample fuzz testing tools (at least in part)

- CERT Basic Fuzzing Framework (BFF)
  - Built on "zzuf" which does the input fuzzing
- CERT Failure Observation Engine (FOE)
  - From-scratch Windows
- OWASP WebScarab
- Immunity's SPIKE Proxy
- Wapiti
- IBM Security AppScan

There are a huge number of these!

Fuzz testing: Problems

- Fully random often does not test much
  - e.g., if input has a checksum, fuzz testing ends up primarily checking the checksum algorithm
- Fuzz testing only finds "shallow" problems
  - Special cases ("if (a == 2) …") rare in input space
  - Sequence of rare-probability events by "random" input will typically not be covered by testing
  - Can modify generators to increase probability… but you have to know very specific defect pattern before you find defect
  - In general, only a small amount of program gets covered
- Once defects found by fuzz testing fixed, fuzz testing has a quickly diminishing rate of return
  - Fuzz testing is still a good idea… but not by itself

Hybrid analysis

Coverage measures

- Hybrid = Combine static and dynamic analysis
- Historically common hybrid approach: Coverage measures
- "Coverage measures" measure "how well" program has been tested in dynamic analysis (by some measure)
  - Many coverage measures exist
- Two common coverage for dynamic testing:
  - Statement coverage: Which (%) program statements have been executed by at least one test?
  - Branch coverage: Which (%) program branch options have been executed by at least one test?

```
if (a > 0) {   // Has two branches, "true" & "false"
  dostuff();   // Statement coverage 100% with a=1
}
```

- Can then examine what is uncovered (untested)

More hybrid approaches

- Concolic testing ("Concolic" = concrete + symbolic)
  - Hybrid software verification technique that interleaves concrete execution (testing on particular inputs) with symbolic execution
  - Can be combined with fuzz testing for better test coverage to detect vulnerabilities
- Sparks, Embleton, Cunningham, Zou 2007 "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting" http://www.acsac.org/2007/abstracts/22.html
  - Extends black box fuzz testing with genetic algorithm
  - Uses "dynamic program instrumentation to gather runtime information about each input's progress on the control flow graph, and using this information, we calculate and assign it a 'fitness' value. Inputs which make more runtime progress on the control flow graph or explore new, previously unexplored regions receive a higher fitness value. Eventually, the inputs achieving the highest fitness are 'mated' (e.g. combined using various operators) to produce a new generation of inputs…. does not require that source code be available"
- Hybrid approaches are an active research area

More hybrid approaches (2)

- Dao and Shibayama 2011, “Security sensitive data flow coverage criterion for automatic security testing of web applications” (ACM) – proposes new coverage measure, “security sensitive data flow coverage”:

  “This criterion aims to show how well test cases cover security sensitive data flows. We conducted an experiment of automatic security testing of real-world web applications to evaluate the effectiveness of our proposed coverage criterion, which is intended to guide test case generation. The experiment results show that security sensitive data flow coverage helps reduce test cost while keeping the effectiveness of vulnerability detection high.”

Penetration testing (pen testing)

- Pretend to be adversary, try to break in
- Depends on the skills of the pen testers
- Need to set rules-of-engagement (RoE)
  - Problem: RoE often unrealistic
- Really a combination of static and dynamic approaches

Operational

What about when it's fielded?

- Hook into logging systems
  - Make sure your logging system is flexible and can hook into common logging systems
- Support host-based countermeasures
  - e.g., address randomization, etc.
  - Make sure your implementation works on them
  - Microsoft EMET (provide info for it)
- Host-based sandboxing/wrappers
  - SELinux (provide starter policy)
  - Document inputs and outputs (files, ports)
- Network-based measures
  - Firewalls, intrusion detection/prevention systems, NATs
  - Don't assume client IP address you see == IP address client sees

When designing & implementing, prepare for security-related tools in the operational (fielded) setting

Many ways to organize tool types

NIST SAMATE Tool Categories (partial)

- Assurance Case Tools
- **Safer Languages**
- Design/Modeling Verification Tools
- **Source Code Security Analyzers, Byte Code Scanners, Binary Code Scanners**
- **Web Application Vulnerability Scanners**
- Intrusion Detectors
- Network Scanners
- Requirements Verification Tools
- Architecture Design Tools
- **Dynamic Analysis Tools**
- Web Services Network Scanners
- Database Scanning Tools
- Anti-Spyware Tools
- Tool Integration Frameworks

Source: http://samate.nist.gov/index.php/Tool_Survey.html

NAVSEA “Software Security Assessment Tools Review” (2009)

- Static analysis code scanning
- Source code fault injection
- Dynamic analysis
- Architectural analysis
- Pedigree analysis
- Binary code analysis
- Disassembler analysis
- Binary fault injection
- Fuzzing
- Malicious code detector
- Byte code analysis

A fool with a tool… and adopting tools

Fool with a tool is still a fool (1)

- RealNetworks' RealPlayer/Helix Player vulnerabilities:
  - CVE-2005-0455 / iDEFENSE Security Advisory 03.01.05
    char tmp[256]; /* Flawfinder: ignore */
    strcpy(tmp, pScreenSize); /* Flawfinder: ignore */
  - CVE-2005-1766 / iDefense Security Advisory 06.23.05
    sprintf(pTmp, /* Flawfinder: ignore */
  - CVE-2007-3410 / iDefense Security Advisory 06.26.07
    strncpy(buf, pos, len); /* Flawfinder: ignore */
  - Kudos to RealNetworks for revealing what happened!!
- Flawfinder: trivial static analysis tool
  - Lexical scanner for C code, reports vulnerability patterns
  - Comment "Flawfinder: ignore" disables next hit report

Fool with a tool is still a fool (2)

- Flawfinder correctly found the vulnerability!!
  - Someone then modified code, claiming not vulnerable
  - Yet these are obvious – not complex – vulnerabilities
  - Likely told "change code until no problems reported"
- Tools are *useless* unless you understand major types of vulnerabilities and how to fix them
  - Training on *tool* not the issue (this tool trivial to run)
  - Training on *developing secure programs* is critical
    - Must understand tools' purpose and what to do with results
    - e.g., must know what it means and what to do if tool says "potential SQL injection vulnerability at line X"

SWAMP More info: http://continuousassurance.org/

- SWAMP = Software Assurance (SwA) Marketplace
  - DHS-sponsored project
  - Cloud-based tool analysis of submitted software
- Makes it easy to run many SwA tools against software. Users:
  - "Software developers can bring software… to continuously test it against a suite of software assurance tools. We will provide interfaces to common source code repositories and develop common tool reporting formats…"
  - "Software assurance tool developers can run tools in our facility to access a large set of software packages and compare the performance of their tools against other tools."
  - "Software assurance researchers will have a unique set of continuous data to analyze…"
- Initial capability focuses on analyzing C, C++, Java using OSS tools
  - Initial tools: FindBugs, PMD, cppcheck, clang, gcc
  - Focus: Making it *easy* to apply the tools – don't need to install tools, SWAMP figures out how to apply the tools (e.g., through build monitoring), and works to integrate tool results

Adopting tool(s)

- Culture change required
  - More than just another tool
  - Tool will not solve anything in isolation
- Define objectives
  - Create "gate" – soft at first, later "must pass"
- Train before use
  - Esp. software security - types of vulnerabilities, how to fix them
- Start with pilot – small and friendly group
- Start by focusing on relevant, easily-understood
  - Disable detection of most problems at beginning
- Appoint "champion" to advocate
- Later, build on success

Sources: Chess, West, Chou, Ron Ritchey