Secure Software Programming and Vulnerability Analysis

Christopher Kruegel chris@auto.tuwien.ac.at
http://www.auto.tuwien.ac.at/~chris

Testing and Source Code Auditing

Overview

- When system is designed and implemented
  - correctness has to be tested
- Different types of tests are necessary
  - validation
    - is the system designed correctly?
    - does the design meet the problem requirements?
  - verification
    - is the system implemented correctly?
    - does the implementation meet the design requirements?
- Different features can be tested
  - functionality, performance, security

Testing

- Edsger Dijkstra

  Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

- Testing
  - analysis that discovers what *is* and compares it to what *should be*
  - should be done throughout the development cycle
  - necessary process
  - but not a substitute for sound design and implementation
  - for example, running public attack tools against a server cannot proof that server is implemented secure

Testing

- Classification of testing techniques
  - white-box testing
    - testing all the implementation
    - path coverage considerations
    - faults of commission
    - find implementation flaws
    - but cannot guarantee that specifications are fulfilled
  - black-box testing
    - testing against specification
    - only concerned with input and output
    - faults of omissions
    - specification flaws are detected
    - but cannot guarantee that implementation is correct

Testing

- Classification of testing techniques
  - static testing
    - check requirements and design documents
    - perform source code auditing
    - theoretically reason about (program) properties
    - cover a possible infinite amount of input (e.g., use ranges)
    - no actual code is executed
  - dynamic testing
    - feed program with input and observe behavior
    - check a certain number of input and output values
    - code is executed (and must be available)

Testing

- Automatic testing
  - testing should be done continuously
  - involves a lot of input, output comparisons, and test runs
  - therefore, ideally suitable for automation
  - testing hooks are required, at least at module level
  - nightly builds with tests for complete system are advantageous

- Regression tests
  - test designed to check that a program has not "regressed”, that is, that previous capabilities have not been compromised by introducing new ones

Testing

- Software fault injection
  - go after effects of bugs instead of bugs
  - reason is that bugs cannot be completely removed
  - thus, make program fault-tolerant
  - failures are deliberately injected into code
  - effects are observed and program is made more robust

- Most techniques can be used to identify security problems

Security Testing

- Design level
  - not much tool support available
  - manual design reviews
  - formal methods
  - attack graphs

- Formal methods
  - formal specification that can be mathematically described and verified
  - often used for small, *safety*-critical programs
    e.g., control program for nuclear power plant
  - state and state transitions must be formalized and
    unsafe states must be described
  - model checker can ensure that no unsafe state is reached

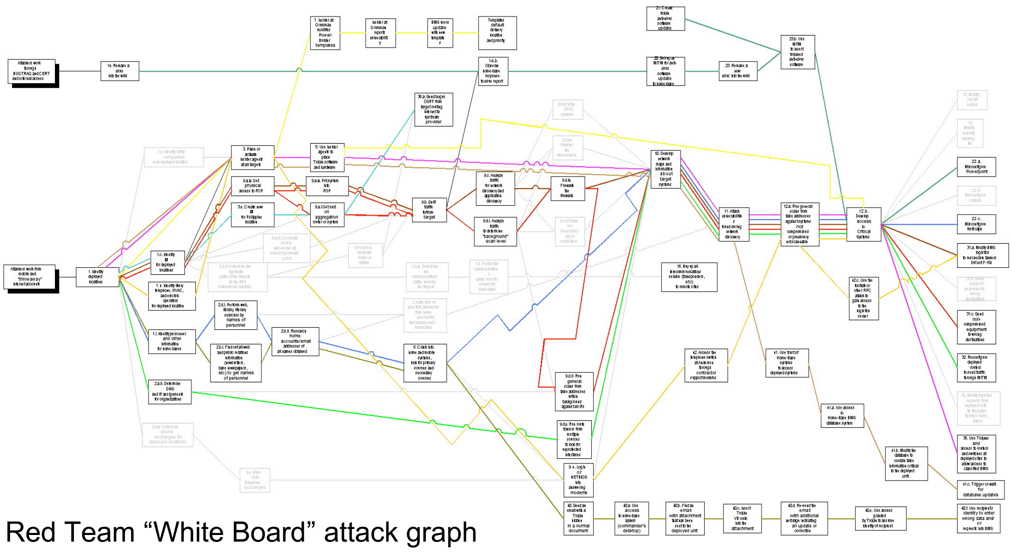
Security Testing

- Attack graph
  - given
    - a finite state model, M, of a network
    - a security property Φ
  - an attack is an execution of M that violates Φ
  - an attack graph is a set of attacks of M

- Attack graph generation
  - done by hand
    - error prone and tedious
    - impractical for large systems
  - automatic generation
    - provide state description
    - transition rules

Security Testing

Automation Systems Group



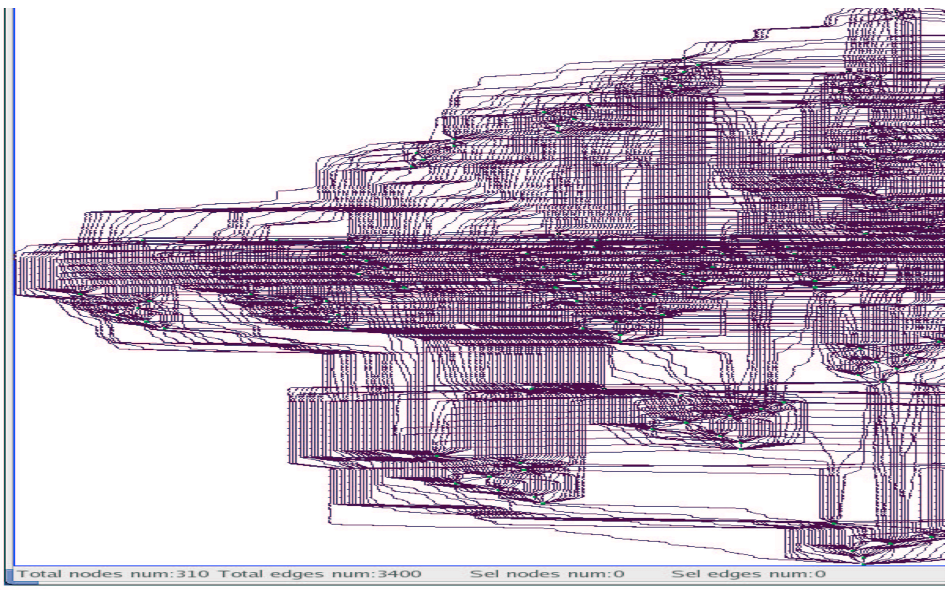
Sandia Red Team "White Board" attack graph from DARPA CC20008 Information battle space preparation experiment

Security Testing

Automation Systems Group

Φ = Attacker gains root access to Host 1.

4 hosts
30 actions
310 nodes
3400 edges



Security Testing

Automation Systems Group

- Implementation Level
  - detect known set of problems and security bugs
  - more automatic tool support available
  - target particular flaws
  - reviewing (auditing) software for flaws is reasonably well-known and well-documented
  - support for static and dynamic analysis
  - ranges from “how-to” for manual code reviewing to elaborate model checkers or compiler extension

Static Security Testing

Automation Systems Group

- Manual auditing
  - code has to support auditing
    - architectural overview
    - comments
    - functional summary for each method
  - OpenBSD is well know for good auditing process
    - 6 -12 members since 1996
    - comprehensive file-by-file analysis
    - multiple reviews by different people
    - search for bugs in general
    - proactive fixes

Static Security Testing

- Manual auditing
  - tedious and difficult task
  - other initiatives were less successful

    - Sardonix

      "Reviewing old code is tedious and boring and no one wants to do it," Crispin Cowan said.

    - Linux Security Audit Project (LSAP)

```
Statistics for All Time
Lifespan |    Rank|Page Views|D/l|Bugs|Support|Patches|Trkr|Tasks
1459 days|0(0.00)|      4,887|  0|0(0)|    0(0)|    0(0)|0(0)| 0(0)
```


Static Security Testing

- Syntax checker
  - parse source code and check for functions that have known vulnerabilities, e.g., `strcpy(), strcat()`
  - also limited support for arguments (e.g., variable, static string)
  - only suitable as first basic check
  - cannot understand more complex relationships
  - no control flow or data flow analysis

  - Examples
    - flawfinder
    - RATS (rough auditing tool for security)
    - ITS4

Static Security Testing

- Annotation-based systems
  - programmer uses annotations to specify properties in the source code (e.g., this value must not be NULL)
  - analysis tool checks source code to find possible violations
  - control flow and data flow analysis is performed
  - problems are undecidable in general, therefore trade-off between correctness and completeness

  - Examples
    - SPlint
    - Eau-claire
    - UNO (uninitialized vars, null-ptr dereferencing, out-of-bounds access)

Static Security Testing

- Model-checking
  - programmer specifies security properties that have to hold
  - models realized as state machines
  - statements in the program result in state transitions
  - certain states are considered insecure
  - usually, control flow and data flow analysis is performed

  - example properties
    - drop privileges properly
    - race conditions
    - creating a secure chroot jail

  - examples
    - MOPS (an infrastructure for examining security properties of software)

Static Security Testing

- Meta-compilation
  - programmer adds simple system-specific compiler extensions
  - these extensions check (or optimize) the code
  - flow-sensitive, inter-procedural analysis
  - not sound, but can detect many bugs
  - no annotations needed

  - example extensions
    - system calls must check user pointers for validity before using them
    - disabled interrupts must be re-enabled
    - to avoid deadlock, do not call a blocking function with interrupts disabled

  - examples
    - Dawson Engler (Stanford)

Static Security Testing

- Model-checking versus Meta-compilation (Engler '03)

- General perception
  - static analysis: easy to apply but shallow bugs
  - model checking: harder, but strictly better once done

- ccNUMA with cache coherence protocols in software
  - 1 bug deadlocks entire machine
  - code with many ad hoc correctness rules
    - `WAIT_FOR_DB_FULL` must precede `MISCBUS_READ_DB`
  - but they have a clear mapping to source code
  - easy to check with compiler

Static Security Testing

- Meta-compilation
  - scales
  - relatively precise
  - statically found 34 bugs, although code tested for 5 years
  - however, many deeper properties are missed
- Deeper properties
  - nodes never overflow their network queues
  - sharing list empty for dirty lines
  - nodes do not send messages to themselves
- Perfect application for model checking
  - bugs depend on intricate series of low-probability events
  - self-contained system that generates its own events

Static Security Testing

- The (known) problem
  - writing model is hard
  - someone did it for a similar protocol than ccNUMA
    - several months effort
    - no bugs
  - use correspondence to auto-extract model from code
- Result
  - 8 errors
  - two deep errors, but 6 bugs found with static analysis as well.
- Myth: model checking will find more bugs
  - in reality, 4x fewer

Static Security Testing

Automation Systems Group

- Where meta-compilation is superior

	Static analysis	Model checking
	Compile ➔ Check	Run ➔ Check
Don't understand?	So what.	Problem.
Can't run?	So what.	Can't play.
Coverage?	All paths! All paths!	Executed paths.
First question:	"How big is code?"	"What does it do?"
Time:	Hours.	Weeks.
Bug counts	100-1000s	0-10s
Big code:	10MLOC	10K
No results?	Surprised.	Less surprised.

Static Security Testing

Automation Systems Group

- Where model-checking is superior

- Subtle errors
  - run code, so can check its implications
  - data invariants, feedback properties, global properties
  - static better at checking properties in code
  - model checking better at checking properties implied by code

- End-to-end
  - catch bug no matter how it is generated
  - static detects ways to cause error
  - model checking checks for the error itself

Dynamic Security Testing

- Run-time checking between operating system and program
  - intercept and check system calls
- Run-time checking between libraries and program
  - intercept and check library functions
  - often used to detect memory problems
    - interception of `malloc()` and `free()` calls
    - emulation of heap behavior and code instrumentation
    - `purify`, `valgrind`
  - also support for buffer overflow detection
    - `libsafe`

Dynamic Security Testing

- Profiling
  - record the dynamic behavior of applications with respect to interesting properties
- Obviously interesting to tune performance
  - `gprof`
- But also useful for improving security
  - sequences of system calls
  - system call arguments
  - same for function calls

Dynamic Security Testing

- Penetration testing
  - explicitly trying to break applications security
  - general tool support available
    - `nessus`
    - `ISS Internet Scanner`
    - `nmap`
  - also tools for available that can test a particular protocol
    - `whisker`
    - `ISS Database scanner`

Summary

- Testing
  - important part of regular software life-cycle
  - but also important to ensure a certain security standard
- Important at design *and* implementation level
  - design
    - attack graphs, formal methods, manual reviews
  - implementation
    - static and dynamic techniques
- Static techniques
  - code review, syntax checks, model checking, meta-compilation
- Dynamic techniques
  - system call and library function interposition, profiling