Achieving Trusted Systems by Providing Security and Reliability

FORMAL REASONING ON SECURITY VULNERABILITIES USING POINTER TAINEDNESS SEMANTICS

Shuo Chen, Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravishankar K. Iyer
Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign

Motivations
- Develop methods to enable uncovering of and reasoning about security vulnerabilities in applications.
- Analysis of vulnerability reports in Bugtraq and CERT advisories indicates that:
  - Many vulnerabilities due to pointer taintedness: user input value used as a pointer value.
  - A significant portion of vulnerabilities due to errors in incorrect invocations of library functions.
- Following these findings, developed a formal reasoning method to examine possibility of pointer taintedness in application source code.

Formal Reasoning on Security Vulnerabilities using Pointer Taintedness Semantics
- Pointer Taintedness: a pointer value (including a return address) is derived directly or indirectly from user input.
- A unifying perspective for reasoning about security vulnerabilities: a root cause of many vulnerabilities.
- The semantics of pointer taintedness can be formally defined.
- The notion of pointer taintedness enables:
  - Static analysis: reasoning about the possibility of pointer taintedness by source code analysis.
  - Runtime checking: inserting assertions in object code to check pointer taintedness at runtime.
  - Architecture-based checking: enhancing processors to detect pointer taintedness.
- Current focus: extraction of security specifications of library functions based on pointer taintedness semantics.

Extraction of Security Specifications of Library Functions
- A formal approach to reason about potential vulnerabilities in library source code.
- Reasoning based on a hypothetical memory model: a boolean property taintedness associated with each memory location.
- The semantics of pointer taintedness defined in equational logic.
- A theorem prover employed to extract security specifications of library functions.
- Security specifications extracted by the analysis:
  - Expose classes of known security vulnerabilities, such as format string, heap corruption and buffer overflow vulnerabilities.
  - Indicate the possibility of function invocation scenarios that may expose new vulnerabilities.

Pointer Taintedness Caused Vulnerabilities
- Format string vulnerability
  - Taint a pointer argument of print().
- Stack smashing
  - Taint a return address.
- Heap corruption
  - Taint the free-chunk doubly-linked list of the heap.
- Glibc globbing vulnerabilities
  - User input resides in a location that is used as a pointer by the parent function of glob().

Example: Formal Extraction of Security Specifications of strcpy()

Specs Suggested by Theorem Prover
- Suppose when function strcpy() is called, the size of destination buffer (dst) is dstsize, the length of user input string (src) is srclen.
- Specifications that are extracted by the theorem proving approach:
  - srclen <= dstsize
  - The buffers src and dst do not overlap in such a way that the buffer dst covers the NULL-terminator of the src string.
  - The buffers dst and src do not cover the function frame of strcpy.
- Initially, dst is not tainted

Future Directions
- Develop a VCGen (verification condition generator) to facilitate theorem proving (based on program annotations).
- Apply the pointer taintedness analysis to a substantial number of commonly used library functions to extract their security specifications.
- Compiler techniques for generating “guarding code” to provide security for programs.

Example of Format String Vulnerability

Example of Formal Extraction of Security Specifications of strcpy()

Taintedness Semantics
- Store represents a snapshot of the memory state at a point in the program execution.
- Operations on memory locations:
  - The fetch operation $\text{Ftch}(S, A)$ gives the content of the memory address A in store S.
  - The location-taintedness operation $\text{LocT}(S, A)$ gives the taintedness of the location A in store S.
- Operations on expressions:
  - Evaluation operation $\text{Eval}(S, E)$ evaluates expression E in store S.
  - Expression-taintedness operation $\text{ExpT}(S, E)$ computes the taintedness of expression E in store S.
  - E.g., $\text{ExpT}(S, ^*foo + ^*bar) = \ldots = \text{LocT}(S, foo) + \text{LocT}(S, bar)$

Formal Reasoning on Security Vulnerabilities
- Pointer Taintedness: a pointer value (including a return address) is derived directly or indirectly from user input.
- A unifying perspective for reasoning about security vulnerabilities: a root cause of many vulnerabilities.
- The semantics of pointer taintedness can be formally defined.
- The notion of pointer taintedness enables:
  - Static analysis: reasoning about the possibility of pointer taintedness by source code analysis.
  - Runtime checking: inserting assertions in object code to check pointer taintedness at runtime.
  - Architecture-based checking: enhancing processors to detect pointer taintedness.
- Current focus: extraction of security specifications of library functions based on pointer taintedness semantics.