Writing Secure Software

Kehuan Zhang

IERG4130 2022

The meaning of Secure Software

- From angle of developers:
  - Goal: No vulnerability in their source code
  - Basic Techniques
    - Be aware of the best practices (and common pitfalls)
    - Tools to perform security analysis (at source code level)
    - Testing (with source code, i.e., white box)
- For third parties:
  - Goals
    - No vulnerability in a shipped code
    - No malicious code in the shipped code
  - Basic Techniques
    - Binary code analysis
    - Testing (without source code, i.e., grey or black box)

Why it is hard to write secure code?

- It is easy for Human beings to make mistakes
  - Lazy to have a complete and rational thinking and analysis
  - Or to perform security checking (either manually or automatically)
- Lack of knowledge and training on secure coding
  - E.g., what are the common vulnerabilities, whether my code can be attacked
- Languages like C/C++ are tricky in order to write secure code
- Security is not top priority in most cases
  - Time to market, Functionalities, etc., may be more important

Common Vulnerabilities and Best Practice

- What we had already learned
  - Best practice
    - Follow secure design principles
    - We have already covered about ten principles (recall?)
  - Vulnerabilities
    - Buffer Overflow
    - Integer overflow
    - Heap overflow
    - Format String
- Some new things
  - Validate all user inputs
  - Perform static and dynamic analysis

Validate User Inputs

- You should always assume that users are untrustable
  - They may be malicious by nature
  - Or just innocent but naive users exploited by attacks
- Thus you should not easily trust all user inputs
  - They may be malformed (by accident, say a typo)
  - They may be malicious and had been crafted carefully to launch attack
- How to validate user inputs? There are two strategies:
  - White-list based
    - Clearly define patterns or a list of legitimate inputs
    - Reject all that does not satisfy these predefined patterns
    - Rejection by default (if a input is not on the list)
  - Black-list based
    - Maintain a list of values (or patterns) of illegal inputs
    - Reject if any input is in that list
    - Contrary to white-list strategy, black-list based approach will accept inputs by default (if they did not hit any pattern or rules)
  - Compare the advantages and disadvantages of both methods?

Validate user inputs (cont.)

- Typical things to validate
  - Data types: e.g., string vs. integer
  - Data ranges: e.g., whether negative number is acceptable
  - Data numbers: recall the **format string** attack
  - Data patterns: e.g., E-mail address, student ID, HKID number, etc.
    - Can be specified and checked via **regular expressions**
  - Data encodings: same string may looked differently when decoded using different schemes (like ASCII and Unicode)
    - An Phishing attack: `https://thehackernews.com/2017/04/unicode-Punycode-phishing-attack.html`
  - Data integrity (sometimes): e.g., cookie in Web applications
- Sanitize ambiguous inputs
  - A given user input may be interpreted differently under different context, so it is important to remove such ambiguity
    - E.g., “<script>” could be a normal string, or a HTML tag to include external JavaScript
    - If only a normal string is expected, then we need to remove such an ambiguity by convert ‘<’ to < and ‘>’ to >

Overview of Static Program Analysis

- What is static analysis?
  - To verify software properties without execution
  - Compilers use lots of static analysis
    - E.g., for code optimization, like dead-code ellimination
  - Security community also have used static analysis at many places
    - To Detect vulnerabilities
- Static analysis is mostly done on source code
  - Relatively easy, since lots of information is still available
    - Especially: data type and code structure (considering loops, function scopes)
- But static analysis can also be done over binary code
  - When source code is not available
  - More challenging (why?)
  - To extract useful information to help dynamic analysis (hybrid analysis)
  - Or to verify the equivalence between binary code and source code
    - Compiler (or libraries / SDKs to be linked) may be malicious
    - Compiler may be wrong (especially for code optimization)

What will static analysis do?

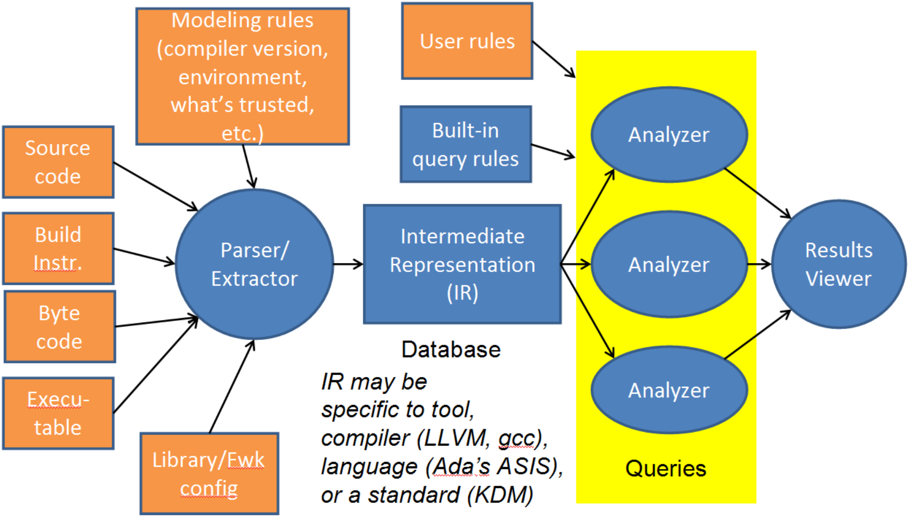
- Type checking
  - Type checking can be useful to eliminate simple vulnerabilities
  - Static type checking languages vs. dynamic type checking languages
    - E.g., Java/C# is static type checking, Python is dynamic
    - C? also static typing, but it is too flexible to perform type casting. Recall the concept of pointer.
- Style checking
  - Checking for best practices or common vulnerable patterns
  - E.g., call to insecure functions, variable initializations, etc.
  - Or the way to organize the code to make it easier to understand and analysis (like prevent use of `eval()` function)
- Program verification / property checking / Bug finding
  - E.g., loop invariants, executation timing invariance, etc.
  - Or: ensure the confidentiality of secret data
- More advanced: program understanding
  - E.g., try to detect the discrepancy between code and comments
- One trend: Machine Learning + Lots of open source code = ?

Strength and Weakness of Static Analysis

- Strength
  - Fast when compared to manual code review and inspection
  - Fast when compared with dynamic analysis
    - No need to really run on concrete data / user inputs
  - Generally better coverage than dynamic analysis
    - In dynamic analysis, each execution can only track one execution path
    - But a typical program many huge diferent paths (combination explosion)
- Weakness
  - Cannot handle Logic or algorithm error
    - Too high level mistakes to understand for such kind of tools
  - Statical Undecidability
    - Some values may be undecidable only until runtime
    - Has to do approximimation
    - But approximation can introduce problems
    - The concept of *false positive* and *false negative*

Typical Structure of a Static Analysis Tool

- Very Similar to a compiler



Be careful to the warnings given by a compiler

- Actually compiler contains many static code analysis)
- Warnings given by compilers may be relevant to security vulnerabilities
    - There may be a vulnerability sitting behind those warnings
    - So try to take a look and think possible reasons (instead of overlooking them)

Two typical analysis tasks

- Control Flow Analysis
  - What is control flow and control flow graph?
    - It is important to understand the concept of **Basic Block**
  - What information or vulnerabilities can be acquired via control flow analysis?
    - E.g., backdoor
    - Can you think about the control flow graph of a backdoor vulnerability
    - Or dangerous statement sequences
- Data Flow Analysis
  - Trace the propagation of data inside a program
    - Via some kind of virtual execution
  - One famous technique is called **taint analysis**
    - It can be used to track the usage of certain data
    - Basic concepts: taint souce, taint sink, taint rules
    - E.g., code injection attack: trace propagation of unstructed inputs
    - E.g., password leakage: trace properties of user password

Overview of Dynamic Program Analysis

- Dynamic analysis: verifying software by executing it on specific inputs and check results
  - Debugging
- Somehow related to testing
  - But testing emphasizes more on the results of function correctness
  - Dynamic analysis may cover security (i.e., to find vulnerabilities)
  - Fuzz Testing (Fuzzing): testing with random inputs
- Major draw-backs of dynamic analysis
  - Huge input space and execution paths to test
    - A integer (on 32-bit CPU) may have 2^{32} possbilities
    - A program with 20 branch points will have 2^{20} different execution paths
    - For real world programs, it is generally impossible to check all inputs and executaiton paths
- Coverage of dynamic analysis
  - Statement Coverage: every line should be visited
  - Branch Point Coverage: every branch condition should be reached
  - Branch Coverage: every branch should be tested
  - Complete Execution Path Coverage: –> too expensive in real world!

Fuzz Testing (Fuzzing)

- Try to solve a key problem of dynamic analysis: efficiency
  - By feeding program-under-test with large number of **random** inputs
- Rational behind?
  - Developers will often focus on legitimate and well-formatted inputs
  - Thus malformed inputs are more efficient to reach corner cases that were mostly ignored
  - Those corner cases generally may cause crashes, memory leaks, or bypass input checking
- Random inputs can also simplify inputs generation and save lots of time
- Proven to be very useful in finding security problems
  - Very popular nowadays

Strategies to get inputs for Fuzzing

- The efficiency on generating valid inputs are very important
  - Limited testing time vs. higher code coverage
- Fully random
  - Relatively low efficient, many contain lots of invalid data discarded before it reach code at later stage
  - So can only detect very shallow problems
- Mutation based
  - Require some sample inputs, then mutate values of those samples
- Generation based
  - Require a model of inputs
  - To generate random inputs based on that model
- Rule of Thumb: the more information about the input generation, the better
  - Thus generation based strategy is relatively better
  - And the key is how to get more information about inputs
  - Here comes the static analysis
    - E.g., get the branch conditions –> propagate back to determine the corresponding inputs