Safe Language (Strong Type)

Ada, Perl, Python, Java, C#, and even Visual Basic

- Have automatic bounds checking, and do not have direct memory access

C-derivatives: Rust (Mozilla 2010)

- Designed to be a "safe, concurrent, practical language", supporting functional and imperative-procedural paradigms
- Does not permit null pointers, dangling pointers, or data races
- Memory and other resources are managed through "Resource Acquisition Is Initialization" (RAII).

Go: type-safe, garbage-collected but C-looking language

- Good concurrency model for taking advantage of multicore machines
- Appropriate for implementing server architectures.

Outline

- Safe Programing
- **Software Testing**
- Compiler and System Support

Manual Code Reviews

Peer review

- Very important before shipping the code in IT companies

Code review checklist

- Wrong use of data:

  variable not initialized, dangling pointer, array index out of bounds, …
- Faults in declarations

  undeclared variable, variable declared twice, …
- Faults in computation

  division by zero, mixed-type expressions, wrong operator priorities, …
- Faults in relational expressions

  incorrect Boolean operator, wrong operator priorities, …
- Faults in control flow

  infinite loops, loops that execute n-1 or n+1 times instead of n, ...

Writing Software Tests

Unit tests

- Test individual components or functions of the software in isolation
- Unit tests should cover all code, including error handling

Regression tests

- Test that new code changes do not negatively affect existing functionality
- Verify that the software continues to function correctly after updates

Integration tests

- Test the interaction between multiple software modules or systems
- Ensure that components work together as expected.

Static Analysis

Analyze the source code or binary before running it (during compilation)

- Explore all possible execution consequences with all possible input
- Approximate all possible states
- Identify issues during development, reducing the cost of fixing vulnerability
- Rely on predefined rules or policies to identify patterns of insecure coding practice

Static analysis tools

- Coverity: https://scan.coverity.com/
- Fortify: https://www.microfocus.com/en-us/cyberres/application-security
- GrammarTech: https://www.grammatech.com/

Limitations

- May produce false positives, requiring manual review
- Cannot detect runtime issues, e.g., logical errors, dynamic environment-specific flaws

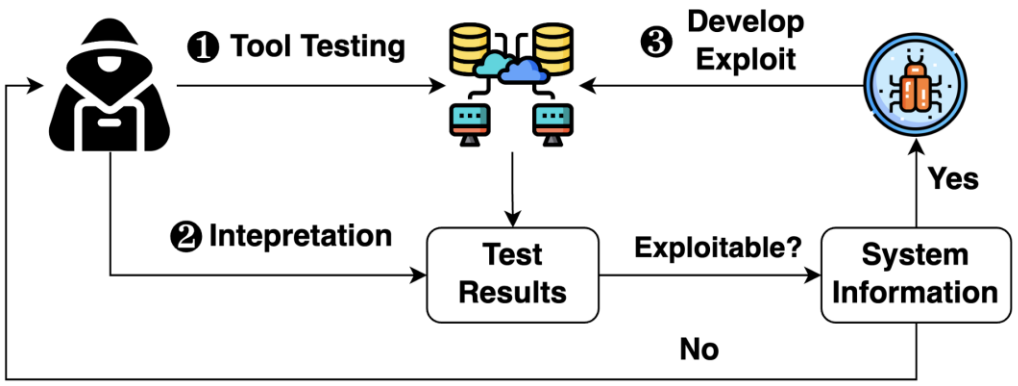
Dynamic Analysis: Penetration Testing

A proactive security assessment method

- Simulate attacks on a system to identify its weakness that is exploitable.
- Identify vulnerabilities before attackers do.
- Ensure compliance with security regulations and improve the overall security posture of systems and applications.

General Procedure

1. Test the system with tools
2. Interpret testing results
3. Check Exploitability
   - Develop the exploit, or
   - Go back to step 1



Dynamic Analysis: Fuzzing

An automated and scalable approach to test software at runtime

- Bombard a program with random, corrupted, or unexpected data to identify how it behaves under unexpected conditions.
- Observe the program for crashes, memory issues or unexpected behaviors.
- Examine failures to determine if they represent exploitable vulnerabilities.

A lot of software testing tools based on fuzzing

- AFL: https://github.com/google/AFL
- FOT: https://sites.google.com/view/fot-the-fuzzer
- Peach: https://wiki.mozilla.org/Security/Fuzzing/Peach

Limitations

- Limited code coverage.
- Require expert analysis to assess whether system crashes are exploitable
- May miss logic flaws that do not result in crashes.

Different Types of Fuzzing Techniques

Mutation-based:

- Collect a corpus of inputs that explores as many states as possible
- Perturb inputs randomly, possibly guided by heuristics, e.g., bit flips, integer increments, substitute with small, large or negative integers.
- Simple to set up. Can be used for off-the-shelf software.

Generation-based

- Convert a specification of input format into a generative procedure
- Generate test cases according to procedure with perturbations
- Get higher coverage by leveraging knowledge of the input format
- Requires lots of effort to set up, and is domain-specific;

Coverage-guided

- Using traditional fuzzing strategies to create new test cases.
- Test the program and measure the code coverage.
- Using code coverage as a feedback to craft input for uncovered code
- Good at finding new states, and combine well with other solutions;