XIAMEN UNIVERSITY MALAYSIA
厦門大學马来西亚分校

SWE301
Software Verification and Validation
Software Error Mitigation - Lab 2
STATIC ANALYSIS

Introduction to Static Code Analysis

What is Static Code Analysis?

Static code analysis is a method of inspecting source code for errors, vulnerabilities, and deviations from best practices **without executing the program**. Unlike dynamic analysis, which evaluates code during runtime, static analysis examines the structure and syntax of code to detect potential issues. This is done using specialized tools that scan the codebase and report issues like coding errors, violations of coding standards, security vulnerabilities, and potential bugs.

How it Works

- Automated Scanning: Static code analysis tools automatically scan the code and highlight problems based on predefined rules or patterns.
- Source Code Examination: The tools review the entire codebase, from logic flows and control structures to variable usage and syntax.
- Rule-Based: These tools rely on sets of rules or coding standards, which may include best practices, common coding mistakes, security guidelines (e.g., OWASP), or company-specific policies.

Role in Software Development

- Find Bugs Early: Static code analysis catches coding issues early in the development process, long before the code is executed or tested. This allows developers to fix problems when they are easier and cheaper to resolve.
- Improve Code Quality: By identifying inefficient code, stylistic inconsistencies, or poorly structured logic, static code analysis improves the maintainability and readability of the code.
- Enforce Coding Standards: These tools help ensure that the code adheres to a company's coding standards or industry guidelines, promoting consistency across the project.

Key Benefits of Static Code Analysis

1. Early Detection of Bugs
   Catching bugs during the development phase prevents them from propagating into production. Static analysis identifies issues like uninitialized variables, null pointer dereferencing, and memory leaks, which can be expensive and difficult to debug later in the development cycle.
2. Security Improvement
   Many static analysis tools are equipped to detect security vulnerabilities like buffer overflows, SQL injection risks, and cross-site scripting (XSS). These tools flag weaknesses that could be exploited by attackers, ensuring that the codebase is secure from potential threats.
3. Reduce Technical Debt
   Technical debt refers to the additional work required to fix problems left in the code. Static code analysis reduces technical debt by catching suboptimal code or shortcuts early on. This minimizes the need for significant refactoring in the future and ensures that the codebase remains clean and maintainable.
4. Save Time and Costs

Fixing bugs early is significantly cheaper than addressing them in later stages of development. By using static code analysis, teams avoid costly rework and time-consuming debugging, which accelerates the development process and ensures timely delivery.

5. Enforces Best Practices
   By adhering to coding standards and best practices, static code analysis promotes cleaner, more efficient, and easier-to-understand code. This leads to better collaboration within the development team and makes it easier for new developers to understand and work on the code.
6. Continuous Quality Control
   Integrated into a Continuous Integration/Continuous Deployment (CI/CD) pipeline, static code analysis ensures that every piece of code meets quality and security requirements before it's merged or deployed. This provides constant feedback to developers and maintains high-quality standards throughout the software lifecycle.

Static code analysis is a powerful tool for improving code quality, enhancing security, and reducing technical debt. By automating code review and providing real-time feedback to developers, it significantly streamlines the development process and ensures that the code is robust, secure, and maintainable from the start.

Overview of PVS-Studio

What is PVS-Studio?

PVS-Studio is a **static code analysis tool** designed to detect potential coding errors, security vulnerabilities, code inefficiencies, and deviations from best practices. It scans the source code of software projects and identifies issues without needing to execute the program. PVS-Studio is used primarily in large and complex software development projects where the early detection of issues is crucial.

Key Features of PVS-Studio:

- **Multi-Language Support**: PVS-Studio supports a variety of programming languages, including:
  - **C and C++**: Versions for Windows, Linux, and macOS.
  - **C#**: For .NET projects.
  - **Java**: For Java projects across multiple platforms.
- **IDE Integration**: The tool integrates seamlessly with popular Integrated Development Environments (IDEs) such as **Visual Studio**, **CLion**, **IntelliJ IDEA**, **Rider**, and others. This allows developers to run the analysis directly within their development environment.
- **Continuous Analysis**: PVS-Studio can be integrated into Continuous Integration (CI) pipelines, ensuring that code is automatically analyzed every time changes are made. This helps developers identify and fix issues early in the development cycle.
- **Wide Range of Diagnostics**: The tool comes with hundreds of built-in diagnostics that check for common coding errors, including:
  - **Bugs**: Detects issues like uninitialized variables, null pointer dereferencing, and buffer overflows.
  - **Code Quality**: Highlights areas for potential refactoring, inefficient code, and overly complex functions.
  - **Security Vulnerabilities**: Identifies potential security risks, such as improper memory management, SQL injections, and others.
  - **Concurrency Issues**: Helps identify threading and synchronization problems in multithreaded applications.
- **Detailed Reports**: PVS-Studio generates detailed reports for developers, providing a description of the error, potential consequences, and suggestions for fixing the issue.

- False Positive Management: The tool allows developers to manage and suppress false positives, ensuring that the results focus on real and actionable problems.
- Incremental Analysis: It supports incremental analysis, meaning that after the initial full scan, only newly modified code will be analyzed. This speeds up the analysis process in large projects.

Why Use PVS-Studio?
PVS-Studio is widely used in industries for several important reasons:

1. Early Detection of Coding Errors
   By using static analysis early in the development cycle, PVS-Studio helps identify bugs before they become costly or difficult to fix. This reduces the overall effort required for debugging and ensures more stable and reliable software.
2. Comprehensive Code Quality Assurance
   PVS-Studio goes beyond bug detection, offering deep insights into code quality, performance, and maintainability. It highlights inefficient code, long functions, and potential architectural problems, promoting best practices and cleaner code.
3. Security Enhancement
   With the rise of cyber threats, PVS-Studio's ability to identify potential security vulnerabilities is invaluable. It can pinpoint risky code, such as memory corruption, input validation flaws, or improper exception handling, which could be exploited in real-world attacks.
4. Improving Codebase Maintainability
   By constantly running checks during the development process, PVS-Studio ensures that the codebase remains clean and easy to maintain. It enforces consistent coding standards and helps teams avoid "technical debt" by addressing issues early.
5. Multithreading Issues Detection
   Detecting concurrency issues in multithreaded applications can be challenging. PVS-Studio specializes in identifying these issues, which are often difficult to find through traditional testing or manual code review.
6. Scalability for Large Projects
   PVS-Studio is designed for use in large-scale projects, where manually reviewing code for bugs and inconsistencies becomes impractical. It can handle codebases with millions of lines of code, making it highly suitable for industrial-scale development.
7. Integration with Development Workflow
   The tool's seamless integration with popular IDEs and CI systems (e.g., Jenkins, TeamCity) ensures that static analysis becomes a natural part of the development process. Automated checks prevent problematic code from being introduced into the main codebase.
8. Widely Trusted in the Industry
   PVS-Studio is trusted by many large organizations and industries, including aerospace, automotive, financial services, and security-focused software companies, to ensure their code is error-free, secure, and optimized.

PVS-Studio offers a robust solution for software developers and companies seeking to improve code quality, security, and maintainability. By integrating seamlessly into the development process and supporting multiple languages, it helps reduce errors, enforce best practices, and catch vulnerabilities early in the software lifecycle.

XIAMEN UNIVERSITY MALAYSIA
厦門大學马来西亚分校

Using PVS Studio:

Free online >> https://godbolt.org/

Using PVS-Studio with **Godbolt** (Compiler Explorer) is a great way to analyze C and C++ code snippets online without needing to set up a local environment. While Godbolt does not directly integrate PVS-Studio, you can still use both tools effectively by following these steps:

Step-by-Step Guide to Using PVS-Studio with Godbolt

1. Access Compiler Explorer (Godbolt)
   Open your web browser and go to the Godbolt Compiler Explorer.
2. Choose Your Compiler
   In the top left corner, select the desired C or C++ compiler from the dropdown menu. You can choose from options like GCC, Clang, or MSVC.
3. Write or Paste Your Code
   In the main code editor area, write your C or C++ code or paste code that you want to analyze.
4. Enable PVS-Studio Analysis
   While you cannot directly run PVS-Studio within Godbolt, you can leverage PVS-Studio's capabilities on your local environment and then compare results:
   1. **Copy Your Code**:
      Copy the code snippet you wrote or pasted in Godbolt.
   2. **Run PVS-Studio Locally**:
      Paste the code into your local development environment where PVS-Studio is installed.
      Set up a new project or file, then use PVS-Studio to analyze the code. This can be done through the IDE or command line, depending on your setup.
5. **Analyze the Results**:
   After running the analysis, PVS-Studio will provide warnings, errors, and suggestions based on the static code analysis.
   Review the results in the PVS-Studio output and take note of any coding issues highlighted by the tool.
6. Compare Results and Refine Your Code
   - Once you have analyzed your code with PVS-Studio, you can return to Godbolt to make adjustments based on the feedback received.
   - Modify your code in Godbolt as necessary to address any warnings or errors identified by PVS-Studio.
7. Testing and Further Analysis
   - After making changes, you can re-test the updated code in Godbolt to ensure it compiles and works as expected.
   - Optionally, you can repeat the process: copy the new code back into PVS-Studio for further analysis and refinement.

Conclusion

While you cannot run PVS-Studio directly within Godbolt, you can effectively use both tools together to analyze and improve your C and C++ code. By leveraging the strengths of PVS-Studio for static analysis and Godbolt for quick compilation and testing, you can enhance your development process significantly.

EXERCISE 1: Hands-on Demonstration of PVS-Studio

1. Analyze Sample Code

To demonstrate the capabilities of PVS-Studio, we'll use a simple C++ code snippet containing intentional errors and vulnerabilities.

Sample Code with Intentional Errors**:**

```
#include <iostream>
void unsafeFunction(int *ptr) {
   // Dereferencing a null pointer
   std::cout << *ptr << std::endl;
}

void memoryLeakFunction() {
   int* array = new int[10]; // Dynamically allocated memory
   // Missing delete statement to free memory
}

int main() {
   int *nullPointer = nullptr;

   // Calling unsafeFunction with a null pointer
   unsafeFunction(nullPointer);

   // Call function that causes memory leak
   memoryLeakFunction();

   return 0;
}
```

2. Run PVS-Studio Analysis

How to Analyze the Code:

1. **Copy the Code**: Paste the sample code into your local IDE where PVS-Studio is installed.
2. **Run the Analysis**: Follow these steps to run the analysis:
   a) Open the project in your IDE.
   b) Use the PVS-Studio menu option to check the solution or project.
3. **View the Results**: PVS-Studio will analyze the code and generate a report highlighting issues.

3. Explanation of Warnings and Errors

After running the analysis, PVS-Studio will display various warnings and errors, which can be categorized into different types:

1. **Bugs**:
   a) **Null Pointer Dereference**: The warning will indicate that the code attempts to dereference a null pointer in the unsafeFunction function. This can lead to undefined behavior and crashes.
   b) **Memory Leak**: The warning will indicate that dynamically allocated memory in memoryLeakFunction is not freed, which can lead to memory leaks.
2. **Vulnerabilities**:
   a) Vulnerabilities typically refer to security flaws. In this case, we have a potential vulnerability due to the null pointer dereference, which can be exploited in a malicious manner if unhandled.

3. **Stylistic Violations**:
    a) PVS-Studio may provide warnings about code style and best practices, such as improper naming conventions or lack of comments. While these are less critical, they contribute to overall code maintainability.

4. Interpreting Diagnostics in PVS-Studio Output
   Each warning/error will include:
   Description: A brief explanation of the issue.
   Location: The file and line number where the issue occurs.
   Severity Level: The level of seriousness (e.g., error, warning, informational).
   Suggestions: Recommended ways to fix the issue.

By using PVS-Studio to identify and resolve issues, developers can significantly improve the quality and robustness of their code, leading to more reliable software applications. Encourage students to adopt static analysis as a regular part of their development workflow to catch such issues early on.

XIAMEN UNIVERSITY MALAYSIA
厦門大學马来西亚分校

PRACTICES

Practice 1

Objective

The objective of this activity is for students to understand how to analyze a codebase using PVS-Studio. They will learn to identify various types of issues, such as bugs, vulnerabilities, and stylistic violations, and understand how to interpret the results provided by the tool.

```
#include <iostream>
#include <vector>

void processData(std::vector<int> data) {
    for (int i = 0; i <= data.size(); i++) {  // Intentional bug: off-by-one error
        std::cout << data[i] << std::endl;  // Will cause an out-of-bounds access
    }
}

void uninitializedVariable() {
    int value;  // Warning: uninitialized variable
    std::cout << "Value: " << value << std::endl;  // Undefined behavior
}

void memoryLeak() {
    int* leak = new int[10];  // Memory leak: no delete
}

int main() {
    std::vector<int> numbers = {1, 2, 3};

    processData(numbers);
    uninitializedVariable();
    memoryLeak();

    return 0;
}
```

Instructions to Student:

1. Analyze the Code
2. Review Warnings and Errors - examine the output provided by PVS-Studio
3. Discuss in group:
   - What each warning/error means.
   - The potential impact of the identified issues (e.g., crashes, undefined behavior, memory consumption).

Practice 2
Objective

The objective of this activity is to familiarize students with the various types of diagnostics provided by PVS-Studio, including bugs, vulnerabilities, and stylistic violations. By analyzing code and categorizing issues based on the diagnostics, students will gain a better understanding of how these warnings relate to code quality.

# PVS-Studio Diagnostics Overview

1. Bugs
Description: Issues that could lead to incorrect behavior or crashes in the program.
Examples
  Null Pointer Dereference: Dereferencing a pointer that is null.
  Array Index Out of Bounds: Accessing an array element outside of its defined range.
2. Vulnerabilities
Description: Security-related issues that may expose the program to attacks.
Examples:
  Buffer Overflow: Writing data beyond the allocated space, potentially allowing code execution.
  Improper Input Validation: Failing to validate input data, which may lead to security breaches.
3. Stylistic Violations
Description: Issues related to coding style and best practices that don't affect functionality but can impact maintainability.
Examples:
  Unused Variables: Variables that are declared but never used.
  Inconsistent Naming Conventions: Naming styles that do not follow established conventions (e.g., camelCase vs. snake_case).
4. Performance Issues
Description: Suggestions for improving performance and efficiency in the code.
Examples:
  Inefficient Loops: Using nested loops that can be optimized.
  Unnecessary Memory Allocation: Allocating memory that could be avoided with better design.
5. Portability Issues
Description: Warnings related to code that may not work on different platforms or compilers.
Examples:
  Use of Non-Standard Functions: Utilizing functions that are not part of the C/C++ standard library.
  Assumptions About Integer Sizes: Making assumptions about the size of data types that may vary between platforms.

```
#include <iostream>

void exampleFunction() {
   int* ptr = nullptr; // Warning: Possible null pointer dereference
   std::cout << *ptr << std::endl; // Dereferencing a null pointer
```

```
    int array[3] = {1, 2, 3};
    for (int i = 0; i <= 3; i++) { // Warning: Array index out of bounds
        std::cout << array[i] << std::endl; // Accessing out-of-bounds
    }

    int unusedVariable; // Warning: Unused variable

    // Potential security issue: no input validation
    std::string userInput;
    std::cin >> userInput; // Missing validation
}

int main() {
    exampleFunction();
    return 0;
}
```

Instructions to Student:

1. Individual Work:
   Analyse and Categorize Issues
   - Run the code in PVS-Studio and review the analysis results.
   - Categorize the warnings and errors based on the diagnostics identified in the document. You should determine whether each issue falls under bugs, vulnerabilities, stylistic violations, performance issues, or portability issues.
2. Group Discussion:
   After completing the categorization, have a discussion with your group where each team member can share their findings and explain:
   The types of issues they encountered and their potential impact on code quality.
   How addressing these issues can improve software reliability, maintainability, and security.