XIAMEN UNIVERSITY MALAYSIA
厦門大學马来西亚分校

SWE301
Software Verification and Validation
Software Error Mitigation - Lab 3
DYNAMIC ANALYSIS

Introduction to Dynamic Code Analysis

What is Dynamic Code Analysis?

Dynamic Code Analysis, **examines the code while it is running**. It involves observing the real-time behavior of the software in a runtime environment. This method helps identify runtime-specific issues like memory leaks, performance bottlenecks, and race conditions, which are not detectable through static analysis alone. Tools such as GDB, Valgrind, and profiling tools are commonly used for dynamic analysis to capture detailed runtime data.

How it Works

Dynamic Code Analysis works by analyzing a program's behavior during execution, allowing developers to observe how the code interacts with memory, system resources, and other runtime components in real-world or simulated conditions.

Key Steps in Dynamic Code Analysis:

1. **Execution of the Program**: The code is run in an actual or simulated environment. This could involve regular program execution or running specific test cases designed to cover different aspects of the program's functionality. During execution, the dynamic analyzer monitors various runtime behaviors, such as memory allocation, CPU usage, and network interactions.
2. **Instrumentation**: The dynamic analysis tools may instrument the code, meaning they insert additional instructions that allow the tool to track execution paths, variable values, and system calls. This allows for the collection of data about how the software behaves under different conditions.
3. **Detection of Runtime Errors**: Dynamic code analysis is particularly useful for identifying:

    - **Memory Leaks**: Unused memory that is not released.
    - **Buffer Overflows**: Writing data beyond the allocated memory space.
    - **Race Conditions**: Multiple threads accessing shared resources in an unintended sequence.
    - **Security Vulnerabilities**: Like input validation flaws that can be exploited during program execution.

4. **Profiling and Performance Monitoring**: Dynamic analysis tools can profile the application's performance by measuring resource usage, such as CPU, memory, and I/O operations, over time. This helps in identifying performance bottlenecks and optimizing the code.
5. **Detailed Reporting**: Once the analysis is complete, the tool provides a report of the runtime behaviors, including errors detected, performance metrics, and any security issues uncovered. These reports often include backtraces, detailed descriptions of errors, and suggestions for fixes.

How Dynamic Code Analysis Works in GDB

1. **Compilation with Debugging Information**:
   To use GDB effectively, compile your program with debugging symbols included. This is done using the -g flag with the compiler (e.g., g++ -g program.cpp -o program). This allows GDB to access line numbers, variable names, and other useful information during debugging(DZone).
2. **Starting GDB**:
   Launch GDB with the compiled program: gdb ./program. This initializes the GDB session and prepares it to execute the program (NDepend Blog).
3. **Setting Breakpoints**:
   Set breakpoints at specific lines or functions in the code using the break command (e.g., (gdb) break main). This pauses execution when the program reaches these points, allowing for inspection of the program's state (DZone).
4. **Running the Program**:
   Use the run command to start the execution of the program. When the program hits a breakpoint, it will pause, giving you a chance to analyze its state (NDepend Blog).
5. **Stepping Through Code**:
   After hitting a breakpoint, you can step through the code line by line using commands like next (to execute the next line without stepping into functions) or step (to go into functions). This allows you to monitor the flow of execution and observe changes in variable values (DZone).
6. **Inspecting Variables**:
   Use the print command to check the values of variables at any point during execution (e.g., (gdb) print variable_name). This helps in identifying incorrect values or states that might lead to errors (DZone).
7. **Analyzing Crashes and Errors**:
   If the program crashes, GDB will provide information about the crash location. You can use commands like backtrace to see the call stack at the time of the crash, which helps identify the sequence of function calls that led to the error (NDepend Blog)(DZone).
8. **Memory Inspection**:
   GDB also allows inspection of memory addresses and content, which is crucial for detecting issues like segmentation faults or memory corruption. You can use commands like x/10x to examine memory in hexadecimal format (NDepend Blog).

Benefits of Using GDB for Dynamic Code Analysis

- **Real-time Interaction**: GDB allows developers to interactively debug the program, making it possible to quickly identify and resolve issues.
- **Detailed Insights**: It provides deep insights into the program's execution flow and variable states, which is invaluable for diagnosing complex issues that static analysis cannot reveal.
- **Support for Complex Programs**: GDB is especially useful for multi-threaded applications, allowing you to manage and inspect different threads during execution (DZone)(NDepend Blog).

Using GDB effectively for dynamic code analysis can significantly enhance your debugging process, allowing for the identification and rectification of runtime issues that would otherwise be difficult to detect. For more detailed information on using GDB, you can refer to the official GDB documentation and tutorials.

What is a Debugger?

A debugger is a tool used by programmers to test and debug (identify and fix errors) in their code. When writing software, bugs—errors or unintended behavior—can occur. A debugger helps programmers:

- **Find Bugs:** It identifies where the program is going wrong by allowing the user to pause (or break) the program at certain points and inspect the values of variables.
- **Check Logic:** By stepping through the program line by line, a programmer can verify if the code is executing as intended and following the correct logic.
- **Understand Program Flow:** A debugger lets you see the order in which functions and statements are executed, helping to ensure the flow of the program is correct.

Without a debugger, finding issues in a program can be tedious, as you'd have to rely on manually printing variables (using print statements) or guessing.

Overview of GDB Debugger
What is GDB?

GDB (GNU Debugger) is a powerful, command-line debugging tool used by programmers to find and fix issues in their code. It is commonly used for programs written in languages like C, C++, and Fortran.

With GDB, you can:

1. **Step through the program**: Run the program one line at a time, allowing you to observe how each instruction affects the program.
2. **Set breakpoints**: Pause the program at specific points to inspect the state of the program.
3. **Inspect variables**: Check the values of variables at different points during execution to ensure they hold the expected data.
4. **Identify crashes**: When a program crashes, GDB can help pinpoint the exact location and cause of the crash, such as a segmentation fault or a memory error.

This makes GDB invaluable for understanding program behavior and troubleshooting issues efficiently.

Using GDB Debugger:

Free online >> https://www.onlinegdb.com/

Using OnlineGDB (https://www.onlinegdb.com/) is straightforward and provides a convenient way to compile and debug code directly from your browser without any setup. Here's a quick guide on how to use it effectively:

1. **Access the Platform**:
   Open your web browser and go to OnlineGDB. You can use it as a guest, but creating an account allows you to save your code.
2. **Choose Your Language**:
   Select the programming language you want to work with (C, C++, etc.) from the dropdown menu.

3. **Write or Upload Your Code**:
   You can type your code directly into the online editor or upload a file. The editor supports multiple files if your project requires them.
4. **Compile and Run Your Code**:
   Click the **Run** button to compile and execute your code. If there are syntax errors, they will be highlighted in the output section.
5. **Debugging with GDB**:
   - To start debugging, click on the **Debug** button. This allows you to use GDB commands in the console.
   - Set breakpoints by clicking on the line numbers where you want the execution to pause.
   - When the program hits a breakpoint, you can step through the code (step in, step over, step out), inspect variables, and examine the call stack to understand the flow of your program.
6. **Using Watch Expressions**:
   You can add watch expressions to monitor specific variables as your code executes, helping to identify logical errors or bugs more efficiently.
7. **Collaborative Features**:
   If you're working with a team, OnlineGDB supports real-time collaboration, allowing multiple users to debug the same session.

Advantages of OnlineGDB:

- **No Installation Required**: Everything runs in the cloud, so there's no need to install compilers or IDEs locally.
- **Cross-Platform Access**: You can debug from any device with internet access.
- **Immediate Feedback**: The tool provides instant feedback on errors, making it easier to catch bugs quickly For more detailed instructions and features, you can visit OnlineGDB's official site.

EXERCISE 1: Hands-on Demonstration of GDB Debugger

Exercise 1: Infinite Loop Example with Two Variables
This exercise demonstrates how GDB can help you inspect and understand variable values to identify and fix logical issues that cause infinite loops in more complex scenarios.

Code:

```
#include <iostream>
int main() {
   int i = 0;
   int j = 10;
   // This loop will enter an infinite loop due to incorrect condition logic
   while (i < 5 && j > 0) {
      std::cout << "i = " << i << ", j = " << j << std::endl;
      // Only incrementing i and not changing j will cause the loop to run infinitely
      i++;
   }
   std::cout << "Loop terminated." << std::endl;
   return 0;
}
```

Steps to Debug the Program Using GDB:

1. Compile the Program
2. In Debug mode, set a Breakpoint Inside the Loop.
3. Run the Program
4. Use next and print to Inspect Variables: After hitting the breakpoint, use next to step through the loop and print i and print j to inspect the values of both variables.
5. Correct the Code: Modify the loop logic to ensure that j is being decremented along with i, ensuring that the loop will terminate when both conditions are met.
6. Re-run the Program: After fixing the code, compile and debug again to verify that the loop terminates correctly.

Exercise 2: Simple Logic Error (Bubble Sort)
In this exercise, students will debug a sorting algorithm that has a logic error.

Code:

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

void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    std::cout << "Original array: ";
    for (int i : arr)
        std::cout << i << " ";
    std::cout << std::endl;

    bubbleSort(arr);

    std::cout << "Sorted array: ";
    for (int i : arr)
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Instruction to Students:

1. Compile and run the program. You will notice that the program outputs an incorrect sorted array.
2. Use GDB to step through the bubbleSort function and observe the value of arr during each iteration.
3. Identify the bug and fix the issue.

XIAMEN UNIVERSITY MALAYSIA
厦門大學马来西亚分校

PRACTICES

Practice 1 - Detecting and Fixing a Segmentation Fault

Objective

Use GDB to identify and fix a segmentation fault caused by dereferencing a null pointer..

```
#include <iostream>
int main() {
   int* ptr = nullptr;
   *ptr = 10;  // Segmentation fault occurs here
   std::cout << "Value: " << *ptr << std::endl;
   return 0;
}
```

Instructions to Student:

1. Use GDB to set a breakpoint at main().
2. Run the program, inspect the call stack using backtrace (https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_42.html), and examine the cause of the crash.
3. Correct the issue.

Practice 2 - Tracking Down Memory Leaks

Objective

Debug a program with a memory leak by dynamically allocating memory but not deallocating it..

```
#include <iostream>

int main() {
   int* arr = new int[100];
   // Forgot to delete the array, causing a memory leak
   return 0;
}
```

Instruction for students:

Steps:

1. Write a program that allocates memory using new and forgets to free it.

Use GDB to monitor the memory usage and correct the issue by adding the missing delete.

Practice 3 - Fixing Logic Errors in Sorting Algorithms

Objective

Debug a sorting algorithm that doesn't produce the correct result due to a logic error.

```
# void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] < arr[j + 1]) {  // Logic error: should be >
            std::swap(arr[j], arr[j + 1]);
```

```
        }
      }
    }
}
```

Instruction for students:

Steps:

1. Implement a Bubble Sort algorithm with a mistake in the comparison.
2. Use GDB to step through the code, observe how the elements are swapped incorrectly, and fix the logic.

Practice 4: Stack Overflow in Recursive Functions

Objective:

Fix a stack overflow caused by an infinite recursion.

```
#include <iostream>

// Recursive function without a base case
int recursiveFunction(int n) {
   std::cout << "n = " << n << std::endl;
   return recursiveFunction(n - 1);  // Infinite recursion
}

int main() {
   recursiveFunction(5);
   return 0;
}
```

Instructions to Student:

1. Analyse the above recursive function code that causes stack overflow (missing base case).
2. Use GDB to break execution, inspect the call stack, and fix the recursion logic.