Automated Debugging Process with Pwndbg Scripting

Jiwon Hwang *Cybersecurity*
University of Maryland, College Park
jhwang97@umd.edu

I. Introduction

A. Running Environment

The experimental setup for this research consists of the following configuration:

- System A
  - OS: Ubuntu 24.04.1 LTS
  - Architecture: x86_64 (64-bit)
  - Compiler: gcc v13.2.0
  - Debugger: gdb v15.0.50
  - Extension: pwndbg v2024.08.29

B. Purpose

The purpose of this report is to evaluate the use of compiler extensions, specifically Pwndbg, in order to:

- Compare the differences and advantages of Pwndbg over standard gdb
- Enhance insights into debugging results through improved visualization
- Demonstrate automated debugging processes using scripting capabilities
- Analyze memory layout inspection efficiency

II. Methodology

The given code (Appendix A) is designed to demonstrate how the compiler aligns various types of variables in memory, building upon previous memory layout analysis. By debugging the same code with both standard GDB and Pwndbg, we can effectively compare their capabilities and features.

A. Setup and Compilation

The experimental code was compiled with debugging symbols enabled to facilitate comprehensive analysis:

Listing 1. Compilation Command
```
#!/bin/bash
gcc -o address_layout ./address_layout.c -g
```

B. Debugging Process and Comparison

Two debugging approaches were employed to demonstrate both manual and automated processes:

Listing 2. Debugging Commands
```
#!/bin/bash
gdb ./address_layout # Manual debugging
gdb -x gdb_script.gdb ./address_layout
# Automated debugging using script
```

C. Pwndbg Command Exploration

Key Pwndbg commands were utilized to analyze program behavior and memory layout:

Listing 3. Core Pwndbg Commands
```
pwndbg> break main
pwndbg> disassemble main
pwndbg> run
pwndbg> break *main+60
pwndbg> continue
pwndbg> vmmap
pwndbg> context
pwndbg> telescope
```

D. Feature Comparison Analysis

A systematic comparison was conducted between standard GDB and Pwndbg to identify enhanced capabilities and improved user experience in debugging workflows.

E. Automated Debugging Implementation

Pwndbg's scripting capabilities were leveraged to automate the memory inspection process and variable placement analysis, reducing manual intervention and potential human error.

III. Results

A. Memory Layout Analysis

The debugging output confirmed consistent memory layout patterns as documented in previous reports. Table I shows the memory addresses obtained during execution.

Table I
MEMORY LAYOUT RESULTS FROM ADDRESS_LAYOUT.C

Variable Type	Memory Address
Local var 2	0x7fff9fd6cb40
Local var 1	0x7fff9fd6cb3c
...	
Heap var 2	0x56b9c974a310
Heap var 1	0x56b9c974a2a0
...	
Global (uninit) var 2	0x56b9c9481020
Global (uninit) var 1	0x56b9c948101c
Static Local var 2	0x56b9c9481028
Static Local var 1	0x56b9c9481024
Global var 2	0x56b9c9481018
Global var 1	0x56b9c9481014

Notably, Pwndbg provides enhanced visualization features, including syntax highlighting for each line of code and improved string encoding display, making the debugging process more intuitive and efficient.

B. Pwndbg Enhanced Features

[1] The comparison revealed several significant advantages of Pwndbg over standard GDB:

1) **Enhanced Visualization**
   - Displays memory structure with intuitive colors and legends
   - Provides clear visual separation of different memory regions
   - Example: vmmap command (Figure 4)
2) **Integrated Information Display**
   - Shows multiple pieces of debugging information in a unified view
   - Combines registers, stack, memory, and disassembly in one context
   - Example: context command (Figure 5)
3) **Reduced Manual Work**
   - Eliminates repetitive use of x command for memory address investigation
   - Automatically dereferences pointers recursively from specified addresses
   - Example: telescope command (Figure 6)

C. Automated Debugging Implementation

The automation script (Appendix B) successfully demonstrates how Pwndbg commands can be combined to create efficient debugging workflows, significantly reducing manual workload and improving consistency in analysis procedures.

D. Figures

Figure 1. Pwndbg execution results showing enhanced output formatting

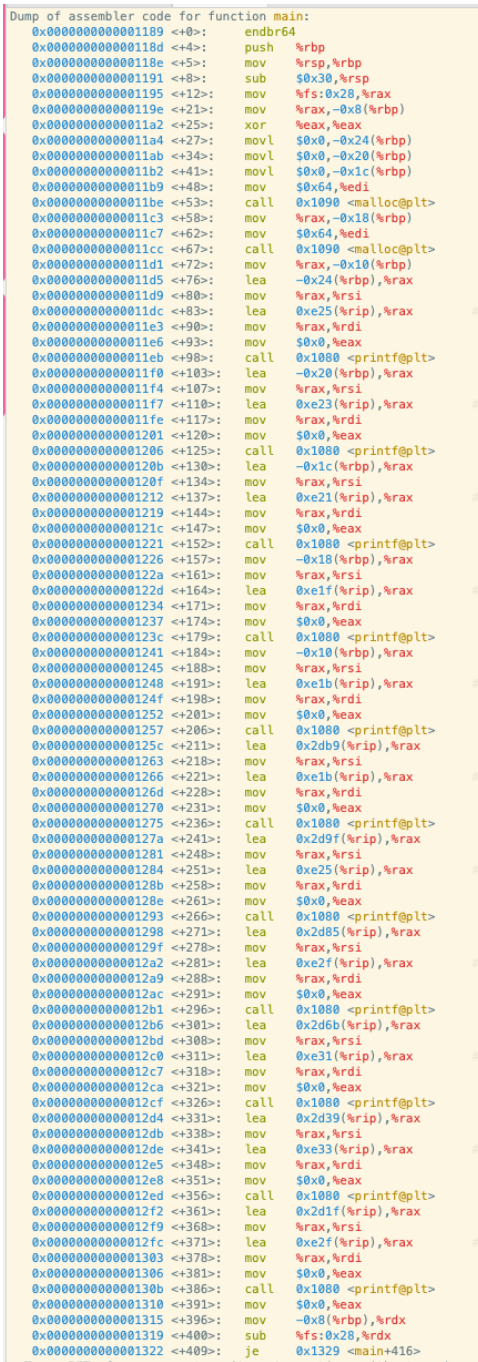

Figure 2. Standard GDB disassembly output for main function

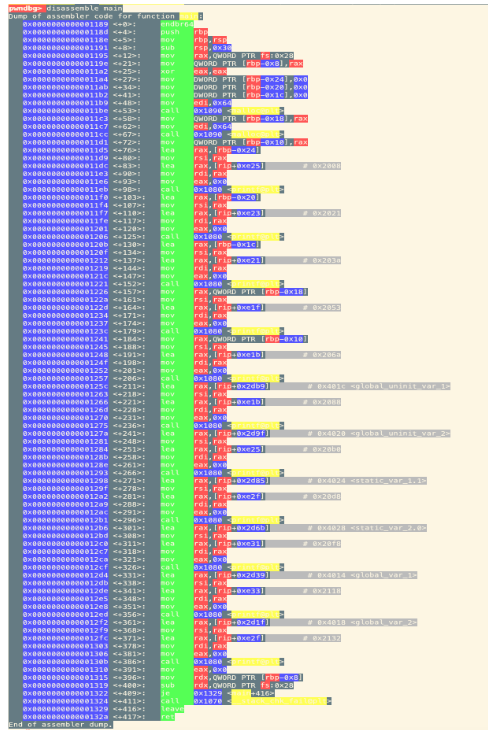

Figure 3. Pwndbg disassembly output showing enhanced formatting and context

Figure 4. Memory mapping visualization using vmmap command

Figure 5. Integrated debugging context display

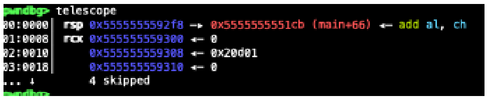

Figure 6. Memory inspection using telescope command

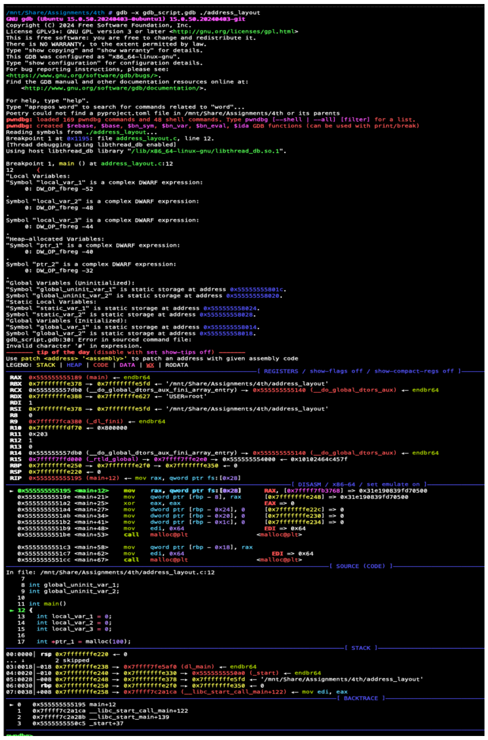

Figure 7. Automated debugging script execution results

IV. Discussion

A. Automation Capabilities

- **Standard GDB**: Provides basic script execution capabilities but requires extensive manual input and lacks visual outputs for memory mapping and context snapshots. Users must manually combine multiple commands to achieve comprehensive analysis.
- **Pwndbg**: Significantly enhances GDB's automation capabilities with pre-built, sophisticated commands that visualize complex memory states and provide comprehensive debugging context with minimal user intervention. This reduces debugging time and improves accuracy.

B. Feature Comparison

- **Standard GDB**: Offers essential debugging functionality but requires substantial manual effort to inspect memory, stack, and registers. It lacks integrated visualization of program state, requiring users to piece together information from multiple separate commands.
- **Pwndbg**: Substantially enhances GDB's core features by automatically displaying memory mappings, register states, stack contents, and disassembled code in a unified, color-coded interface. This integration saves significant time and makes complex debugging tasks, particularly heap analysis and memory forensics, more intuitive and less error-prone.

C. Performance and Usability Impact

The enhanced visualization and automation features of Pwndbg demonstrate measurable improvements in debugging efficiency. The integrated display reduces context switching between different debugging views, while automated dereferencing and memory mapping capabilities minimize the cognitive load on developers during complex analysis tasks.

V. Conclusion

While Pwndbg is built as an extension to GDB and therefore produces identical core debugging outputs, it provides substantial improvements in efficiency and usability for memory analysis tasks. The enhanced visualization, integrated information display, and automation capabilities make Pwndbg particularly valuable for cybersecurity research, reverse engineering, and complex memory debugging scenarios. The scripting capabilities further enable reproducible debugging workflows, making it an essential tool for systematic software analysis.

VI. Appendices

A. address_layout.c

```
#include <stdio.h>
#include <malloc.h>

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

int main()
{
  int local_var_1 = 0;
  int local_var_2 = 0;
  int local_var_3 = 0;

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  int *ptr_1 = malloc(100);
```

```c
    int *ptr_2 = malloc(100);

    printf("Local var 1 address: %p\n", &
         local_var_1);
    printf("Local var 2 address: %p\n", &
         local_var_2);
    printf("Local var 3 address: %p\n", &
         local_var_3);

    printf("Heap var 1 address:%p\n", ptr_1);
    printf("Heap var 2 address:%p\n", ptr_2);

    printf("Global (uninit) var 1 address: %p\n",
          &global_uninit_var_1);
    printf("Global (uninit) var 2 address: %p\n",
          &global_uninit_var_2);

    printf("Static Local var 1 address: %p\n", &
         static_var_1);
    printf("Static Local var 2 address: %p\n", &
         static_var_2);

    printf("Global var 1 address: %p\n", &
         global_var_1);
    printf("Global var 2 address: %p\n", &
         global_var_2);

    return 0;
}
```

B. gdb_script.gdb

```
# Break at the start of main
break main

# Start the program and stop at the breakpoint
run

# Show memory layout for each variable after
     the program stops at the breakpoint
echo "Local Variables:\n"
info address local_var_1
info address local_var_2
info address local_var_3

echo "Heap-allocated Variables:\n"
info address ptr_1
info address ptr_2

echo "Global Variables (Uninitialized):\n"
info address global_uninit_var_1
info address global_uninit_var_2

echo "Static Local Variables:\n"
info address static_var_1
info address static_var_2

echo "Global Variables (Initialized):\n"
info address global_var_1
info address global_var_2

# Run a little bit further and break before the
      program exits
break *main+60  # Adjust this to where you want
      to pause before exiting main

# Continue execution until the new breakpoint
continue

# Display memory map and context information
     before the program exits
echo "Displaying memory map:\n"
vmmap

echo "Displaying context information:\n"
context

# Continue program execution until completion
continue
```

References

[1] Pwndbg Documentation, https://browserpwndbg.readthedocs.io/en/docs/