68000 Assembly Game - Assembly and C Module - Year 2 Stage 2

Software Development and Cyber Security

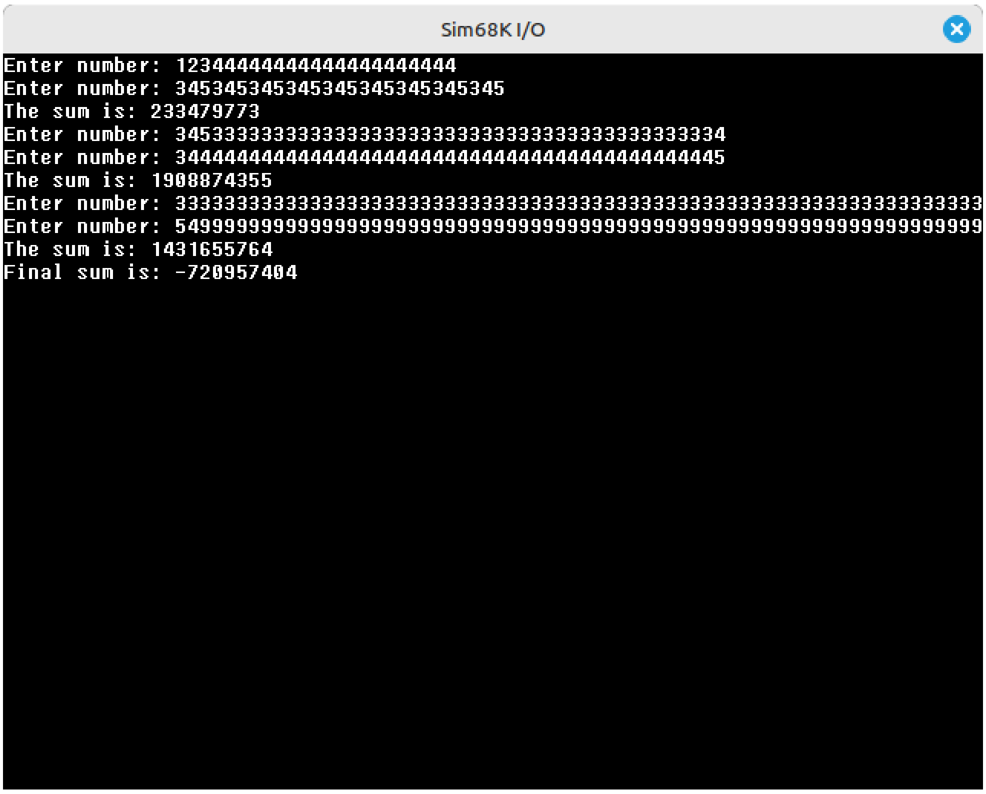
Project Specification: Convert 68000 Assembly Code to x86_64

Project Due: Monday, 28th April 2025 (9:00 AM) – No GitHub commits after this time.
Project Demonstration and Grading: Friday 2nd May 9.00am to 5.00pm
Project Quiz: Friday 2nd May 9.00am to 5.00pm

1. Objective: T The objective of this project is to convert a 68000-assembly program that demonstrates parameter passing via registers and the stack into an equivalent x86_64 assembly program. The focus is on maintaining functionality, efficiency, and correctness while adhering to best coding practices. This project will also highlight security considerations relevant to software developers and cybersecurity professionals, including:

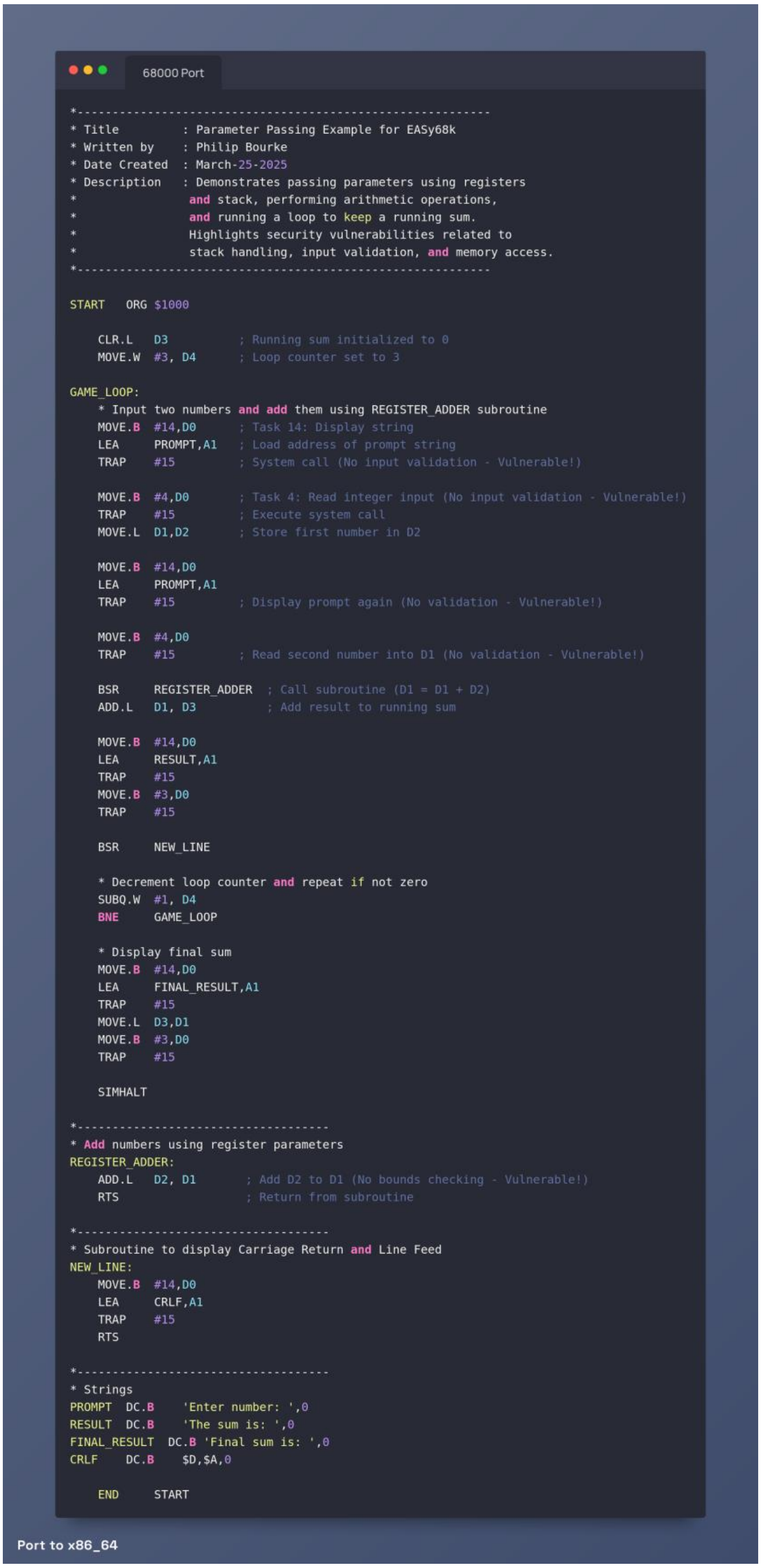
1. The importance of proper stack handling to prevent stack-based buffer overflows.
2. Ensuring proper input validation to avoid unintended code execution.
3. Recognizing how improper memory access can lead to exploits.

Example of one of the major issues to fix (what is this called)



```
Enter number: 12344444444444444444444
Enter number: 345345345345345345345345345
The sum is: 233479773
Enter number: 345333333333333333333333333333333333333333334
Enter number: 344444444444444444444444444444444444444444445
The sum is: 1908874355
Enter number: 333333333333333333333333333333333333333333333333333333333333333333
Enter number: 549999999999999999999999999999999999999999999999999999999999999999
The sum is: 1431655764
Final sum is: -720957404
```

2. Input: The input for this project is the provided 68000 assembly code (image here)



```
*-----------------------------------------------------------
* Title         : Parameter Passing Example for EASy68k
* Written by    : Philip Bourke
* Date Created  : March-25-2025
* Description   : Demonstrates passing parameters using registers
*                 and stack, performing arithmetic operations,
*                 and running a loop to keep a running sum.
*                 Highlights security vulnerabilities related to
*                 stack handling, input validation, and memory access.
*-----------------------------------------------------------

START   ORG $1000

    CLR.L   D3          ; Running sum initialized to 0
    MOVE.W  #3, D4      ; Loop counter set to 3

GAME_LOOP:
    * Input two numbers and add them using REGISTER_ADDER subroutine
    MOVE.B  #14,D0      ; Task 14: Display string
    LEA     PROMPT,A1   ; Load address of prompt string
    TRAP    #15         ; System call (No input validation - Vulnerable!)

    MOVE.B  #4,D0       ; Task 4: Read integer input (No input validation - Vulnerable!)
    TRAP    #15         ; Execute system call
    MOVE.L  D1,D2       ; Store first number in D2

    MOVE.B  #14,D0
    LEA     PROMPT,A1
    TRAP    #15         ; Display prompt again (No validation - Vulnerable!)

    MOVE.B  #4,D0
    TRAP    #15         ; Read second number into D1 (No validation - Vulnerable!)

    BSR     REGISTER_ADDER  ; Call subroutine (D1 = D1 + D2)
    ADD.L   D1, D3          ; Add result to running sum

    MOVE.B  #14,D0
    LEA     RESULT,A1
    TRAP    #15
    MOVE.B  #3,D0
    TRAP    #15

    BSR     NEW_LINE

    * Decrement loop counter and repeat if not zero
    SUBQ.W  #1, D4
    BNE     GAME_LOOP

    * Display final sum
    MOVE.B  #14,D0
    LEA     FINAL_RESULT,A1
    TRAP    #15
    MOVE.L  D3,D1
    MOVE.B  #3,D0
    TRAP    #15

    SIMHALT

*------------------------------------
* Add numbers using register parameters
REGISTER_ADDER:
    ADD.L   D2, D1       ; Add D2 to D1 (No bounds checking - Vulnerable!)
    RTS                  ; Return from subroutine

*------------------------------------
* Subroutine to display Carriage Return and Line Feed
NEW_LINE:
    MOVE.B  #14,D0
    LEA     CRLF,A1
    TRAP    #15
    RTS

*------------------------------------
* Strings
PROMPT  DC.B    'Enter number: ',0
RESULT  DC.B    'The sum is: ',0
FINAL_RESULT  DC.B 'Final sum is: ',0
CRLF    DC.B    $D,$A,0

    END     START
```

3. Output: The expected output of this project will be an x86_64 assembly program that replicates the functionality of the 68000 version, correctly handling parameter passing, arithmetic operations, and maintaining a running sum through a loop.

4. Requirements:

- The x86_64 implementation should match the 68000 version's functionality.
- All variable names, comments, and labels should be appropriately translated to x86_64 syntax.
- Parameter passing via registers and the stack should be properly converted.
- The loop should run three times, keeping a running sum.
- Proper commenting and formatting should be maintained.
- Optimisations should be applied where possible.
- The program should handle errors gracefully.
- Address security concerns where relevant, such as stack handling and buffer overflow risks.

5. Tools and Resources:

- Linux Virtual Machine (VM) Access:
  - Inside SETU Network: https://comp-vcentre.itcarlow.ie/ui
  - Outside SETU Network: https://uag.setu.ie/
    Students can use the Linux VM provided by SETU Carlow for assembly development and testing. Ensure you log in with your SETU credentials. An x86_64 assembly language development environment such as NASM (Netwide Assembler).
- An x86_64 assembly language development environment such as NASM (Netwide Assembler).
- A debugger GDB for testing and troubleshooting the converted code.
- Documentation and references for x86_64 assembly language syntax and instructions.
- The original 68000 assembly code as a reference.

6. Deliverables:

- A complete set of x86_64 assembly files corresponding to the original 68000 assembly code.
- A README file explaining any significant changes or considerations made during the conversion process.
- A test plan and test cases to verify the correctness and functionality of the converted code.
  Test Scripts C (create a separate C file to test assembly code)_
  https://www.tutorialspoint.com/c_standard_library/assert_h.htm
  https://libcheck.github.io/check/index.html
  Assembly Macros
  http://blog.code-cop.org/2015/08/how-to-unit-test-assembly.html
- Capture a brief 10 to 20-second video of software being executed in the command line after it has been ported.

68000 Assembly Game - Assembly and C Module - Year 2 Stage 2

Software Development and Cyber Security

<table>
<tr><th colspan="3">Project Rubric</th></tr>
<tr><th>0 - 35%
(0 - 8)
Basic</th><th>35% - 75%
(8 - 18)
Intermediate</th><th>75% - 100%
(18 - 25)
Advanced</th></tr>
<tr>
<td>• Implementation will achieve minimum functionality.
• Implementation may contain some syntax and/or run-time errors.
• Implementation code will be poorly commented and/or formatted.
• Implementation will contain basic features; application will not be tested properly.
• Implementation code will not follow applicable coding conventions.</td>
<td>• Implementation will achieve expected functionality.
• Implementation will not contain syntax and/or run-time errors.
• Implementation code will be reasonably commented and/or formatted.
• Implementation will contain assignment features.
• Implementation will be tested to a reasonable degree.
• Implementation code will follow appropriate coding conventions.</td>
<td>• Implementation will achieve advanced functionality.
• Implementation will not contain syntax and/or run-time errors.
• Implementation code will be well commented and/or formatted.
• Implementation will contain assignment features.
• Application will be expertly tested.
• Implementation code will follow coding conventions.</td>
</tr>
<tr><td colspan="3">Correctness
• All operations and calculations produce identical results to the original 68000 assembly code.
• The converted code passes all provided test cases without errors or discrepancies.
Code Clarity and Readability
• Variable names, comments, and labels are clear and descriptive.
• The code is well-structured and easy to follow.
• Proper indentation and formatting are used to enhance readability.
Performance Optimization
• The converted code demonstrates optimization techniques where applicable.
• Redundant operations or instructions are eliminated to improve performance.
• Efficient memory usage and register allocation strategies are employed.
Error Handling
• The converted code includes proper error handling mechanisms.
• Potential overflow, underflow, or boundary conditions are checked and handled appropriately.
Documentation and Testing
• The README file provides clear explanations of any significant changes or considerations during the conversion process.
• A comprehensive test plan with test cases is provided to verify the correctness and functionality of the converted code.</td></tr>
</table>