Module 4
Memory Issues in Linux Applications

Memory management

- Memory management is crucial in programming as it ensures efficient use of system resources, stability, and performance.
- Application programming mostly involves allocation and deallocation of memory resources.
- Proper memory management ensures optimal utilization and helps prevent common memory issues.

Impact of memory issue in application

- **Performance Degradation**: Inefficient memory usage can slow down applications, leading to sluggish user experiences.
- **System Instability and Crashes**: Memory-related errors, like segmentation faults, can cause application crashes or even system-wide instability.
- **Security Vulnerabilities**: Memory vulnerabilities, such as buffer overflows, can be exploited by attackers to compromise system security.
- Common Memory issue
  - Segmentation faults (segfaults)
  - Memory leaks
  - Buffer overflow
  - Use after free (dangling pointer dereference)
- To address concerns stemming from the memory-unsafe nature of C/C++, CISA recommends developers transition to memory-safe programming languages like Rust.

Segmentation faults (segfaults)

- A segmentation fault can happen when a program tries to access a non-existing virtual memory segment or existing virtual memory segment in a different way as defined by its attribute.
    - Execute data in non-executable segment.
    - Write data in read only segment.
- As a consequence, the kernel delivers the SIGSEGV signal to the offending process, and it usually results in the termination of the process.

```
int *example_ptr = NULL; //Point example pointer to NULL (Invalid Memory)
*example_ptr = 5; //(Trying to store '5' in invalid memory location)
```

- Refer segfault example.

Memory Leaks

- A memory leak is a condition that occurs when a program fails to release memory that is no longer needed.
- Memory leaks gradually consume available memory resources over time, potentially causing performance degradation and eventual program crashes.
- Common causes of memory leaks include not deallocating dynamically allocated memory, losing references to memory blocks, and failing to release resources properly.

```c
int main() {
    // Allocate memory for an integer array
    int *arr = (int *)malloc(5 * sizeof(int));
    // Initialize the array
    for (int i = 0; i < 5; i++) {
        arr[i] = i;
    }
    // No explicit free() call to deallocate the memory.
    return 0;
}
```

Buffer overflow

- A buffer overflow is a type of software vulnerability that occurs when a program writes more data to a buffer (a temporary storage area) than it can hold, causing the excess data to overflow into adjacent memory locations.
- Two common types of buffer overflows are stack-based and heap-based, depending on where the buffer is located in memory.
- It can lead to memory corruption, program crashes, or unauthorized access to a system, making it a significant security risk.

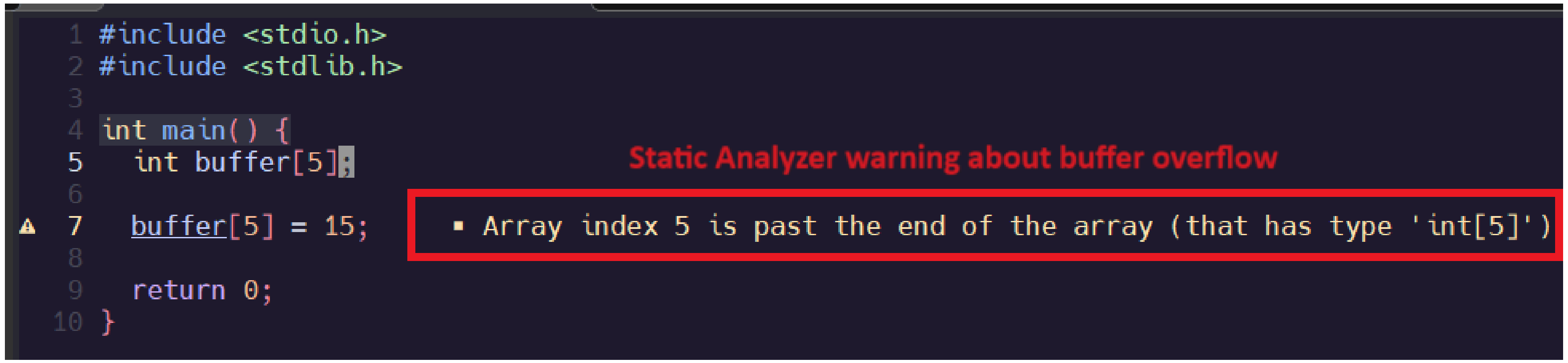
```
// Create an integer array with a size of 5 elements
int buffer[5]

// Writing to the 6th location of buffer will result in a buffer overflow.
buffer[5] = 15;

// Print the content of 'buffer[5]' (this may produce unexpected output)
printf("Buffer[%d]: %d\n", 5, buffer[5]);
```

Memory Debugging Tools

- Detecting and resolving memory issues is crucial for software development and system management.
- Various powerful tools are available to identify, diagnose, and address memory-related problems.
- This section explores these tools and techniques to enhance application and system stability and performance.

Static analysis tools (clang analyzer)

- A static analyzer is a software analysis tool that examines source code or compiled code without executing it.
- A static analyzer performs a comprehensive examination of code, including analysis of code patterns, control flow, and data flow, to identify potential issues, including memory-related problems and vulnerabilities.
- Static analyzers help catch memory issues early in the development process, reducing the likelihood of costly and disruptive issues later.



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int buffer[5];

  buffer[5] = 15;

  return 0;
}
```

Static Analyzer warning about buffer overflow

```
▪ Array index 5 is past the end of the array (that has type 'int[5]')
```

Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- It includes a suite of tools for memory debugging, memory leak detection, and profiling.
    - **Memcheck**: Memory error detector (default tool)
    - **Cachegrind**: Cache profiler
    - **Callgrind**: Call graph profiler
    - **Helgrind**: Thread error detector
    - **Massif**: Heap profiler
- Valgrind operates by running the program in a virtual machine ("valgrind environment") that monitors and analyzes memory and CPU usage.
- Because of the instrumentation added by Valgrind, the execution speed significantly slows down, making it suitable only for a debugging environment.

Valgrind memcheck tool

Memcheck is the default and most widely used tool in Valgrind. All reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. It detects various memory-related errors such as:

- Memory leaks: Identifying memory blocks that were allocated but not freed.
- Uninitialized memory use: Detecting the use of uninitialized values.
- Invalid memory access: Finding out-of-bounds array access and invalid pointer dereferencing.
- Bad frees of heap blocks (double frees, mismatched frees).
- Overlapping source and destination pointers in memcpy and related functions.

```
manas@sandbox:~$ valgrind --tool=memcheck --leak-check=full <program>
```

Valgrind memcheck report

```
manas@sandbox:~/work/Examples/memory_leak$ valgrind --tool=memcheck --leak-check=full -s ./mem_leak
==23087== Memcheck, a memory error detector
==23087== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==23087== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==23087== Command: ./mem_leak
==23087==
Dynamic array allocated with 5 elements
==23087==
==23087== HEAP SUMMARY:
==23087==     in use at exit: 20 bytes in 1 blocks
==23087==   total heap usage: 2 allocs, 1 frees, 1,044 bytes allocated
==23087==
==23087== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23087==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==23087==    by 0x10917E: main (main.c:5)
==23087==
==23087== LEAK SUMMARY:
==23087==    definitely lost: 20 bytes in 1 blocks
==23087==    indirectly lost: 0 bytes in 0 blocks
==23087==      possibly lost: 0 bytes in 0 blocks
==23087==    still reachable: 0 bytes in 0 blocks
==23087==         suppressed: 0 bytes in 0 blocks
==23087==
==23087== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The snapshot of this memcheck report is taken from the memory leak example.

Valgrind with GDB

Valgrind uses a synthetic CPU, not the host CPU, making direct debugging impossible. GDB interacts with Valgrind's gdbserver for full debugging within Valgrind.

- If you want to debug a program with GDB when using the Memcheck tool, start Valgrind like this:

  ```
  valgrind --vgdb=yes --vgdb-error=0 prog
  ```

- Start GDB in another shell.

  ```
  gdb prog
  ```

- Attach the GDB with valgrind gdbserver.

  ```
  (gdb) target remote | vgdb
  ```

You can now debug your program e.g. by inserting a breakpoint and then using the GDB continue command.

Reference: https://valgrind.org/docs/manual/manual-core-adv.html.

Sanitizer

- The Sanitizer suite is a set of runtime analysis tools that helps find common programming mistakes. It can detect issues like memory errors, undefined behaviors, race conditions, and similar bugs.
- Each sanitizer relies on **compiler instrumentation** and **shadow memory** or similar techniques to find issues related to memory, threading, and undefined behaviors in code.
- A range of sanitizers are available for analysing both user-space code and kernel code.
- Major compilers, such as GCC and Clang, provide support for various sanitizers.
    - GCC: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
    - Clang: https://clang.llvm.org/docs/UsersManual.html#id47

Address Sanitizer (ASan)

- AddressSanitizer (ASan) is a runtime memory error detector for C/C++.
- While compiling the program ASan inserts runtime checks into the code to detect memory errors.
- ASan helps identify memory-related issues such as:
    - Use after free (dangling pointer dereference)
    - Heap buffer overflow
    - Stack buffer overflow
    - Global buffer overflow
    - Use after return
    - Use after scope
    - Initialization order bugs
    - Memory leaks

Address Sanitizer usage

- Compile the code with `-fsanitize=address` flag.

```
gcc -fsanitize=address -fno-omit-frame-pointer -g -O1 -o memleak main.c
```

- When a memory error is detected during runtime, ASan will print an error message and a stack trace, indicating where the issue occurred.

```
manas@sandbox:~/work/Examples/memory_leak$ ./memleak
Dynamic array allocated with 5 elements
=================================================================
==23352==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 20 byte(s) in 1 object(s) allocated from:
    #0 0x7f0835dd5867 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x56461c79f1fe in main (/home/manas/work/Examples/memory_leak/memleak+0x11fe)
    #2 0x7f0835b22d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
SUMMARY: AddressSanitizer: 20 byte(s) leaked in 1 allocation(s).
```

- Use ASan during development and testing and disable it in production builds for optimal performance.

Memory Sanitizer (MSan)

- MemorySanitizer (MSan) is a runtime uninitialized memory reads detector for C/C++ programs.
- MSan tracks memory initialization using shadow memory, where each byte is mapped to indicate if it is initialized or uninitialized.
- Compiler instrumentation inserts checks before memory accesses to detect and report the following cases.
    - Uninitialized value was used in a conditional branch.
    - Uninitialized pointer was used for memory accesses.
    - Uninitialized value was passed or returned from a function call.
    - Uninitialized data was passed into some libc calls.
- MSan support is only present in Clang compiler.

```
clang -fsanitize=memory -fsanitize-memory-track-origins -fPIE -pie -fno-omit-frame-pointer -g -O2 <Program File>
```

Thread Sanitizer (TSan)

- ThreadSanitizer is a tool that detects data races in C/C++ program using pthread library.
- A data race occurs when two threads access the same variable concurrently and at least one of the accesses is write.
- Use `-fsanitize=thread` to add compiler instrumentation for TSan.

```
gcc -fsanitize=thread -fno-omit-frame-pointer -g -O1 -o memleak main.c
```

- Upon finding a data race condition, TSan will print the error report.

Undefined Behavior Sanitizer (UBSan)

- UBSAN is a runtime error detection tool that identifies undefined behavior.
    - Signed integer overflow.
    - Invalid Shift operations for example, shifting by a negative or too large number.
    - Dereferencing misaligned or null pointers.
    - Type mismatch or invalid casts between different types.
    - Refer to the exhaustive list of undefined behaviors in this document.
- UBSAN instruments the code at compile-time by adding checks to detect undefined behavior at runtime.
- Enable it by using `-fsanitize=undefined` compiler option. Refer to the GCC or Clang documentation to choose from different "undefined behavior" compiler option.
- When an issue is detected, it logs a detailed message with the type of undefined behavior, source code location, and a backtrace.

libefence

- libefence is a lightweight library that helps to catch buffer overflow in dynamically allocated buffer and use-after-free memory errors.
- libefence allocates extra memory pages around dynamic memory blocks, marking them as unreadable. It triggers a segmentation fault if the program accesses memory beyond its allocated bounds.
- It can either be linked statically in the program or preloaded using `LD_PRELOAD` environment variable.

```
manas@sandbox:~/work/Examples/memory_leak$ sudo apt-get install electric-fence
manas@sandbox:~/work/Examples/memory_leak$ ulimit -c unlimited
manas@sandbox:~/work/Examples/memory_leak$ LD_PRELOAD=libefence.so.0.0 ./mem_leak
  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Dynamic array allocated with 5 elements
Segmentation fault (core dumped)
```

- A coredump is generated upon a segfault. This coredump can be opened with GDB, pinpointing the exact location of the error.

Best Practices for Memory Management

Allocate Memory Dynamically When Needed: Only allocate memory dynamically (e.g., using malloc or new) when necessary. Use stack memory for small, short-lived variables.

Deallocate Memory Properly: Always release dynamically allocated memory using free or delete when you're done with it to prevent memory leaks.

Avoid Manual Memory Management: Use higher-level abstractions and smart pointers (in C++), whenever possible, to manage memory automatically. This reduces the risk of memory leaks and other errors.

Check for Null Pointers: Before dereferencing a pointer, ensure it's not null (i.e., check for null pointer exceptions) to avoid crashes.

Bounds Checking: When working with arrays, use library functions or language features to check bounds (e.g., in C++ use std::vector, in C use strncpy, snprintf, etc.).

Best Practices for Memory Management

Avoid Memory Leaks: Regularly inspect and analyze your code for memory leaks using tools like Valgrind or AddressSanitizer.

Understand Ownership: Clearly define ownership of objects and memory. Understand who is responsible for allocating and deallocating memory and follow ownership patterns consistently.

Defensive Programming: Practice defensive programming by validating input parameters, checking return values from memory allocation functions, and handling errors gracefully.

Code Reviews: Conduct code reviews to catch memory management issues early, as they can be challenging to debug once they occur.

Move to memory-safe programming: Consider prioritizing memory-safe programming languages like Rust over C and C++ for all future development efforts.

References

- Refer to these instructions for enabling coredump.
- Clang Static Analyzer.
    - https://clang-analyzer.llvm.org/
- Valgrind with GDB.
    - https://valgrind.org/docs/manual/manual-core-adv.html.
- Address Sanitizer flags.
    - https://github.com/google/sanitizers/wiki/addresssanitizerflags