Лекція: System Software-I. Unit 5. Linux Memory Issues Leading to Hard-to-Detect Bugs

1. **Вступ**

У Linux-системах проблеми з пам’яттю є одними з найскладніших для виявлення, особливо коли вони виникають у вже розгорнутому (fielded) програмному забезпеченні. Такі помилки можуть призводити до:
- Непередбачуваних падінь програм (crashes).
- Витоків пам’яті (memory leaks).
- Пошкоджених даних (data corruption).
- Вразливостей безпеки (security vulnerabilities).

У цій лекції ми розглянемо поширені проблеми з пам’яттю, способи їх виявлення та тестові приклади для їх відтворення.

2. Поширені проблеми з пам’яттю

2.1. Некоректні звернення до пам’яті (Incorrect Memory Accesses)

Це ситуації, коли програма намагається читати або записувати дані за межами виділеної пам’яті.

Приклади:
- Звернення до NULL-вказівника (Dereferencing a NULL pointer).
- Використання звільненої пам’яті (Use-after-free).
- Подвійне звільнення пам’яті (Double-free).

Наслідки:
- Segmentation fault (SIGSEGV) – якщо ОС виявляє некоректний доступ.
- Тихий збій (Silent corruption) – якщо помилка не викликає негайного падіння.

2.2. Використання неініціалізованих змінних (Uninitialized Variables)

Якщо змінна не ініціалізована, вона містить "сміттєві" дані зі стеку або купи.

Приклад:

c
```
int *ptr;
printf("%d", *ptr); // Невизначена поведінка!
```

Наслідки:
- Непередбачувані значення.
- Вразливості типу "інформаційне витікання" (information leak).

2.3. Вихід за межі масиву (Out-of-Bounds Memory Accesses)
Спроба доступу до елементів поза виділеним діапазоном.

Приклад:
c
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", arr[10]); // Buffer overflow!

Наслідки:
- Перезапис сусідніх даних.
- Можливість виконання довільного коду (exploitable vulnerability).

3. Тестові приклади для виявлення проблем

Test Case 4, 5, 6: Некоректні звернення до пам’яті
- Test Case 4: Dereferencing a NULL pointer.
- Test Case 5: Use-after-free.
- Test Case 6: Double-free.

Сценарій для Test Case 5 (Use-after-free):
c
char *ptr = malloc(10);
free(ptr);
strcpy(ptr, "test"); // CRASH!

Test Case 7, 8: Помилки ініціалізації
- Test Case 7: Читання неініціалізованої змінної.
- Test Case 8: Використання неініціалізованого вказівника.

Сценарій для Test Case 7:
c
int x;
if (x > 0) { // Undefined behavior!

```
    printf("x is positive");
}
```

Test Case 9, 10: Вихід за межі масиву
- Test Case 9: Stack buffer overflow.
- Test Case 10: Heap buffer overflow.

Сценарій для Test Case 9:

```
c
char buffer[5];
strcpy(buffer, "overflow!"); // Stack corruption
```

4. Витоки пам’яті (Memory Leaks)

Test Case 12: Простий витік пам’яті

```
c
void leak_memory() {
    int *ptr = malloc(100);
    // No free!
}
```

Наслідки: Поступове зростання використання пам’яті (OOM Killer може вбити процес).

Test Case 13: Витік через втрачені вказівники

```
c
void lost_pointer() {
    int *ptr = malloc(100);
    ptr = NULL; // Memory leak!
}
```

Продовження Test Case 13: Витік у циклах

```
c
while (1) {
    char *data = malloc(1024);
    // No free, leak on each iteration!
}
```

5. Невизначена поведінка (Undefined Behavior)
Дії, результати яких не описані в стандарті мови (C/C++):
- Зміна значення const об’єкта.
- Переповнення цілих чисел (integer overflow).
- Використання дикого вказівника (wild pointer).

Приклад:
c
int a = INT_MAX;
a++; // Undefined behavior (signed overflow)

6. Фрагментація пам’яті (Fragmentation)
Фрагментація виникає, коли пам’ять поділена на дрібні блоки, і великі виділення стають неможливими.

Типи фрагментації:
- Зовнішня (External): Вільна пам’ять є, але у розрізнених блоках.
- Внутрішня (Internal): Частина виділеної пам’яті використовується неефективно (напр., через вирівнювання).

Наслідки:
- Збільшення часу виділення пам’яті.
- Помилки malloc() через неможливість знайти достатньо великий блок.

7. Висновки
- Пам’яттєві помилки в Linux важко виявити, оскільки вони часто не викликають негайних збоїв.
- Інструменти налагодження (valgrind, AddressSanitizer, gdb) допомагають знаходити такі проблеми.
- Тестові приклади (Test Cases 4–13) можуть бути використані для перевірки стабільності програм.

Запобігання:
- Використовуйте автоматичні інструменти аналізу пам’яті.
- Уникайте ручного керування пам’яттю (напр., використовуйте std::vector замість масивів у C++).
- Регулярно тестуйте код на наявність витоків та помилок доступу.

Лекція: Debugging Tools for Memory Issues in Linux

1. Вступ
Пам’яттєві помилки є одними з найскладніших для виявлення, але існують інструменти, які допомагають їх знаходити:
- Valgrind – інструмент для динамічного аналізу пам’яті.
- AddressSanitizer (ASan) – швидкий інструмент від Google для виявлення помилок пам’яті.
- Glibc mallopt – налаштування менеджера пам’яті glibc.

2. Valgrind – інструмент для аналізу пам’яті

2.1. Що таке Valgrind?
Valgrind – це інструментарій для динамічного аналізу програм, який допомагає виявляти:
- Витоки пам’яті (memory leaks).
- Некоректні операції з пам’яттю (invalid reads/writes).
- Використання неініціалізованих значень.

2.2. Як встановити Valgrind?
bash
sudo apt-get update
sudo apt-get install valgrind Ubuntu/Debian
sudo yum install valgrind CentOS/RHEL

2.3. Перевірка встановлення Valgrind
bash
valgrind --version

Якщо виводить версію (наприклад, valgrind-3.18.1) – інструмент встановлено правильно.

2.4. Використання GCC та Docker для компіляції
Якщо програма потребує специфічного середовища, можна використати Docker:
bash
docker run -it --rm -v $(pwd):/app gcc:latest bash
cd /app
gcc -g -o my_program my_program.c -g для debug symbols
valgrind ./my_program

3. Тестові приклади для Valgrind

Test Case 0: Простий приклад для перевірки
c
include <stdio.h>
int main() {
    printf("Hello, Valgrind!\n");
    return 0;
}

Запуск:
bash
valgrind --leak-check=full ./my_program

Test Case 1: Читання неініціалізованої пам’яті
c
int main() {
    int x;
    printf("%d", x); // Uninitialized read!
    return 0;
}

Valgrind виведе:

Conditional jump or move depends on uninitialised value(s)

Test Case 5, 6: Вихід за межі масиву (compile-time та dynamic memory)
5: Stack overflow
c
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printf("%d", arr[10]); // Out-of-bounds read
    return 0;
}

6: Heap overflow

c
```
int main() {
    int *arr = malloc(5 * sizeof(int));
    arr[10] = 42;  // Heap buffer overflow
    free(arr);
    return 0;
}
```

Valgrind виведе:

Invalid write of size 4

Test Case 8, 9: Use-After-Free (UAF) та Use-After-Return (UAR)
8: Use-After-Free
c
```
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    *ptr = 10;  // UAF!
    return 0;
}
```

9: Use-After-Return
c
```
int *get_ptr() {
    int x = 5;
    return &x;  // UAR (pointer to stack memory)
}
int main() {
    int *ptr = get_ptr();
    printf("%d", *ptr);  // Undefined behavior!
    return 0;
}
```

Test Case 13: Витік пам’яті через бібліотечний API
c
```
include <stdlib.h>
void leak() {
    malloc(100);  // Never freed!
}
```

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

Valgrind виведе:

100 bytes in 1 blocks are definitely lost

4. Підсумкова таблиця Valgrind

Помилка	**Повідомлення ASan**
Вихід за межі масиву	stack-buffer-overflow / heap-buffer-overflow
Use-After-Free	use-after-free
Use-After-Return	stack-use-after-return
Витік пам’яті	detected memory leaks

5. AddressSanitizer (ASan) – швидкий детектор помилок

5.1. Що таке ASan?
ASan – інструмент від Google, який швидко виявляє помилки пам’яті (менше накладних витрат, ніж Valgrind).

5.2. Компіляція з ASan
bash
```
gcc -fsanitize=address -g -o my_program my_program.c
```

5.3. Тестові приклади з ASan

Test Case 1: Читання неініціалізованої пам’яті
c
```
int main() {
```

```
    int x;
    printf("%d", x);  // ASan: "use-of-uninitialized-value"
    return 0;
}
```

Test Case 2, 3: Переповнення буфера (стек/купа)

```
c
// Stack overflow
int main() {
    int arr[5];
    arr[10] = 1;  // ASan: "stack-buffer-overflow"
    return 0;
}
```

```
c
// Heap overflow
int main() {
    int *arr = malloc(5 * sizeof(int));
    arr[10] = 1;  // ASan: "heap-buffer-overflow"
    free(arr);
    return 0;
}
```

Test Case 8, 9: Use-After-Free та Use-After-Return

```
c
// UAF
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    *ptr = 10;  // ASan: "use-after-free"
    return 0;
}
```

6. Підсумкова таблиця ASan

1. Вихід за межі масиву (stack-buffer-overflow / heap-buffer-overflow)

Stack:

```
#include <stdio.h>

int main() {
   int arr[3];
   arr[3] = 42;  //  stack-buffer-overflow
   return 0;
}
```

Heap:

```
#include <stdlib.h>

int main() {
   int* arr = (int*)malloc(3 * sizeof(int));
   arr[3] = 42;  //  heap-buffer-overflow
   free(arr);
   return 0;
}
```

2. Use-After-Free

```
#include <stdlib.h>

int main() {
   int* ptr = (int*)malloc(sizeof(int));
   free(ptr);
   *ptr = 123;  //  use-after-free
   return 0;
}
```

3. Use-After-Return (stack-use-after-return)

Цей тип помилки виникає, коли ти повертаєш вказівник на локальну змінну:

```
#include <stdio.h>

int* get_ptr() {
   int local = 42;
```

```
    return &local;  // Не можна: local існує тільки до виходу з функції
}

int main() {
    int* p = get_ptr();
    printf("%d\n", *p);  //  stack-use-after-return
    return 0;
}
```

4. Витік пам’яті (detected memory leaks)

```
#include <stdlib.h>

int main() {
    int* ptr = (int*)malloc(100);  //  Виділення
    //  Але нема free()
    return 0;
}
```

Як зібрати з AddressSanitizer:

```
gcc -fsanitize=address -g -O1 example.c -o example
./example
```

ASan видасть повідомлення з типом помилки, адресою, стеком викликів та кодом.

7. Порівняння Valgrind та ASan

Критерій	Valgrind	AddressSanitizer (ASan)
Швидкість	Повільний (~10-20x)	Швидкий (~2x)
Виявлення витоків	Так	Так (з -fsanitize=leak)
Підтримка потоків	Повна	Обмежена
Детекція UAF	Так	Так

8. Glibc mallopt – налаштування менеджера пам’яті
Функція mallopt() дозволяє налаштовувати поведінку malloc():
c
include <malloc.h>
int main() {
    mallopt(M_MMAP_THRESHOLD, 256 * 1024); // Використовувати mmap для виділень >256KB
  return 0;
}

Основні параметри:
- M_MMAP_THRESHOLD – мінімальний розмір для використання mmap.
- M_CHECK_ACTION – рівень деталізації помилок.

9. Висновки
- Valgrind – потужний, але повільний інструмент для детального аналізу.
- ASan – швидкий, добре підходить для тестування під час розробки.
- mallopt – корисний для налаштування алокатора glibc у продуктивних системах.

Рекомендації:
- Використовуйте Valgrind для повного аналізу перед релізом.
- Використовуйте ASan під час розробки для швидкого виявлення помилок.
- Налаштовуйте mallopt для оптимізації виділення пам’яті у критичних додатках.

Лекція: A Brief Look at the Most Recent C Features

1. Вступ
Мова C продовжує розвиватися, і нові стандарти (C11, C17, C23) додають корисні функції, покращуючи безпеку, продуктивність і читабельність коду. У цій лекції ми розглянемо:
- Як визначити версію стандарту C.
- Видалення небезпечних функцій (наприклад, gets()).
- Нові функції з перевіркою меж (bounds-checking).
- Функції, що не повертають значення (_Noreturn).
- Макроси, залежні від типів (type-aware macros).
- Підтримку Unicode.
- Анонімні структури та об’єднання.

2. Як визначити версію стандарту C?
У коді можна використовувати макроси для перевірки стандарту:
c
include <stdio.h>

```
int main() {
   if defined(__STDC_VERSION__)
      printf("C Standard Version: %ld\n", __STDC_VERSION__);
      // C89: не визначено
      // C99: 199901L
      // C11: 201112L
      // C17: 201710L
      // C23: 202311L (останній на даний момент)
   endif
   return 0;
}
```

Приклад виводу:

C Standard Version: 201710L (C17)

3. Видалення небезпечних функцій та зміни

3.1. Видалення gets()
Функція gets() була повністю видалена з C11 через ризик переповнення буфера. Замість неї використовуйте fgets() або gets_s() (C11).

Приклад заміни:
c
```
// Небезпечно (видалено в C11):
// char buffer[100]; gets(buffer);

// Безпечний варіант (C99+):
char buffer[100];
fgets(buffer, sizeof(buffer), stdin);
```

3.2. Зміни в fopen()
У C11 додано режим "x" для fopen(), який забороняє перезапис файлу:
c
FILE *file = fopen("example.txt", "wx"); // Помилка, якщо файл існує
if (!file) {
    perror("Failed to open file");
}

3.3. Функції з перевіркою меж (Bounds-checking, C11)
У <stdio.h> та <string.h> додано безпечні аналоги:
Небезпечна функція	Безпечний аналог (C11)
scanf("%s", buf)	scanf_s("%s", buf, size)
strcpy(dest, src)	strcpy_s(dest, size, src)
strcat(dest, src)	strcat_s(dest, size, src)

Приклад:
c
char dest[20];
strcpy_s(dest, sizeof(dest), "Hello, C11!");

4. Функції, що не повертають значення (_Noreturn)
Модифікатор _Noreturn (або [[noreturn]] у C23) вказує, що функція ніколи не повертає управління (наприклад, завершує програму).

Приклад:
c
include <stdlib.h>
include <stdnoreturn.h>

noreturn void fatal_error(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
    exit(1);
    // Немає return!
}

int main() {
    fatal_error("Something went wrong!");
    // Цей код ніколи не виконається

```
}
```

5. Макроси, залежні від типів (_Generic, C11)
Макрос _Generic дозволяє створювати перевантажені макроси на основі типу аргументу.

Приклад:
```
c
define print_value(x) _Generic((x), \
    int: printf("%d\n", x), \
    float: printf("%f\n", x), \
    char*: printf("%s\n", x) \
)

int main() {
    print_value(42);        // Виведе: 42
    print_value(3.14f);     // Виведе: 3.140000
    print_value("Hello!");  // Виведе: Hello!
    return 0;
}
```

6. Підтримка Unicode (C11/C23)
C11 додає:
- UTF-8 літерали (u8"text").
- 16-бітні (u"text") та 32-бітні (U"text") рядки.
- Типи char16_t та char32_t у <uchar.h>.

Приклад:
```
c
include <uchar.h>

int main() {
    const char *utf8 = u8"Привіт, Unicode!";
    const char16_t *utf16 = u"UTF-16 текст";
    const char32_t *utf32 = U"UTF-32 текст";
    return 0;
}
```

7. Анонімні структури та об’єднання (C11)
Дозволяють звертатися до вкладених полів без вказівки імені структури.

Приклад:
c
include <stdio.h>

```
struct Point {
    union {
        struct { int x, y; }; // Анонімна структура
        int coords[2];
    };
};

int main() {
    struct Point p = { .x = 10, .y = 20 };
    printf("x=%d, y=%d\n", p.x, p.y);  // Без вказівки рівня вкладеності
    printf("coords: %d, %d\n", p.coords[0], p.coords[1]);
    return 0;
}
```

Вивід:

x=10, y=20
coords: 10, 20

8. Приклад коду з новими можливостями (C17/C23)
c
include <stdio.h>
include <stdnoreturn.h>

```
noreturn void exit_program() {
    puts("Exiting...");
    exit(0);
}
```

```
int main() {
    // Використання _Generic
    define print_type(x) _Generic((x), \
        int: puts("int"), \
        float: puts("float") \
    )
    print_type(42);  // Виведе: "int"

    // Анонімна структура
    struct { int a, b; } s = { .a = 1, .b = 2 };
    printf("a=%d, b=%d\n", s.a, s.b);

    exit_program();  // Функція не повертається
    // Тут код не виконається
}
```

9. Висновки
- C11/C17/C23 додають сучасні можливості, зберігаючи сумісність.
- Безпечніші функції (_s, fopen_s) запобігають переповненням буфера.
- _Generic дозволяє створювати "перевантажені" макроси.
- Unicode та анонімні структури покращують читабельність коду.

Рекомендації:
- Використовуйте -std=c11 або -std=c17 при компіляції.
- Уникайте застарілих функцій (gets, strcpy).
- Експериментуйте з новими можливостями (_Generic, _Noreturn).