МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 «Вычислительные машины, системы и сети»

Отчёт по лабораторной работе № 2

по учебной дисциплине « Анализ защищенности, аудит и управление уязвимостями программно-информационных систем»

по теме Динамический анализ утечек памяти

Выполнили студентки группы М3О-211СВ-24:

Хамдан Шаза

Козлова А.Д

Принял:

Филимонов Илья Андреевич

Москва, [Год]

1. Цель работы

Изучение инструментария Valgrind и Fuzzing для проведения динамического анализа программного

обеспечения, направленного на выявление ошибок управления памятью и потоками, которые

могут быть потенциальными уязвимостями.

2. Использованные инструменты: Valgrind (Memcheck), AFL++

3. Анализ Valgrind:

3.1 код для тестирования:

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

// Структура статуса робота
typedef struct {
    float x, y;           // Позиция в метрах
    float heading;        // Heading в радианах
    float speed;          // Текущая скорость m/s
    int battery_level;    // Процент заряда батареи
} robot_state_t;

// Структура данных датчика
typedef struct {
    float lidar_distance;     // Расстояние до ближайшего препятствия
    float imu_heading;        // IMU heading reading
    int ultrasonic[4];
} sensor_data_t;

// Параметры навигации
typedef struct {
    float target_x, target_y;
    float max_speed;
```

```c
    float safe_distance;
    int emergency_stop;
} nav_params_t;

// VULNERABLE: Buffer overflow in sensor processing
void process_sensor_data(char* sensor_input) {
    char buffer[20];
   snprintf(buffer, sizeof buffer, "%s", sensor_input);
    printf("Processed: %s\n", buffer);
}

float calculate_distance(robot_state_t* robot, float target_x, float target_y) {
    if (robot == NULL) {
        return -1.0f;
    }

    float dx = target_x - robot->x;
    float dy = target_y - robot->y;
    return sqrtf(dx*dx + dy*dy);
}

// VULNERABLE: Integer overflow in battery calculation
int calculate_power_usage(int current, int additional) {
    return current + additional; // CWE-190: Integer overflow possible
}

// функция навигации
int should_stop_for_obstacle(sensor_data_t* sensors, float safe_distance) {
    if (sensors == NULL) return 1;

    for (int i = 0; i < 4; i++) {
        if (sensors->ultrasonic[i] < safe_distance * 100) { // Convert to cm
            return 1; // Stop required
        }
    }
    return 0; // No stop needed
```

```
}

// VULNERABLE: Memory leak in path planning
float* plan_route(robot_state_t* start, nav_params_t* target) {
    float* route = malloc(10 * sizeof(float)); // Allocate memory
    if (route == NULL) return NULL;

    // Simple straight-line path calculation
    for (int i = 0; i < 10; i++) {
        float t = i / 9.0f;
        route[i] = start->x + t * (target->target_x - start->x);
        // Forgot to free this memory - CWE-401
    }
    return route;
}

float calculate_safe_speed(float distance_to_target, float max_speed) {
    if (distance_to_target < 0) return 0.0f;

    // Reduce speed when close to target
    if (distance_to_target < 2.0f) {
        return max_speed * 0.3f;
    } else if (distance_to_target < 5.0f) {
        return max_speed * 0.6f;
    }
    return max_speed;
}

float calculate_turn_angle(robot_state_t* robot, nav_params_t* target) {
    float desired_heading;

    if (robot && target) {
        desired_heading = atan2f(target->target_y - robot->y,
                                 target->target_x - robot->x);
    }
```

```
    return desired_heading - robot->heading;
}

// Main navigation function
void navigate_robot(robot_state_t* robot, sensor_data_t* sensors, nav_params_t* params) {
    if (robot == NULL || sensors == NULL || params == NULL) {
        return;
    }

    // Проверка условий emergency
    if (params->emergency_stop || robot->battery_level < 10) {
        robot->speed = 0.0f;
        return;
    }

    // симуляция
    char sensor_string[] = "Lidar:5.2,IMU:1.57,USS:30,25,40,35";
    process_sensor_data(sensor_string);

    //Рассчитать расстояние до цели
    float distance = calculate_distance(robot, params->target_x, params->target_y);

    // Проверка на наличие препятствий
    if (should_stop_for_obstacle(sensors, params->safe_distance)) {
        robot->speed = 0.0f;
        printf("Emergency stop: obstacle detected!\n");
        return;
    }

    // Рассчитать безопасную скорость
    robot->speed = calculate_safe_speed(distance, params->max_speed);

    // Рассчитать угол поворота
    float turn_angle = calculate_turn_angle(robot, params);

    // Обновление позиции (простое моделирование)
    robot->x += robot->speed * cosf(robot->heading) * 0.1f; // 100ms time step
```

```
    robot->y += robot->speed * sinf(robot->heading) * 0.1f;
    robot->heading += turn_angle * 0.1f;

    //Обновление батареи
    int power_used = calculate_power_usage(robot->battery_level, 1);
    robot->battery_level = power_used > 100 ? 100 : power_used;

    printf("Position: (%.2f, %.2f), Speed: %.2f, Battery: %d%%\n",
           robot->x, robot->y, robot->speed, robot->battery_level);
}

//VULNERABLE: Use after free
void test_use_after_free() {
    robot_state_t* robot = malloc(sizeof(robot_state_t));
    if (robot) {
        robot->x = 0.0f;
        robot->y = 0.0f;

        free(robot);
        for (volatile int i = 0; i < 1000000; i++) {}
        robot->x = 5.0f;
        //printf("Robot X: %.2f\n", robot->x);
    }
}

int main() {
    //
    int* arr = malloc(5 * sizeof(int));
for (int i = 0; i <= 5; i++) { // should be < 5
    arr[i] = i * 2;
}
free(arr);
    //
    printf("=== Robot Navigation System Test ===\n");
    char* leak = malloc(50);
    strcpy(leak, "This memory will never be freed!");
    printf("%s\n", leak);
```

```
// Инициализировать состояние робота, начальные параметры положения
robot_state_t robot = {0};
robot.x = 0.0f;
robot.y = 0.0f;
robot.heading = 0.0f; //ориентация робота Лицом к востоку (x)
robot.speed = 0.0f;
robot.battery_level = 80;

//Инициализировать данные датчика
sensor_data_t sensors = {0};
sensors.lidar_distance = 5.5f;
sensors.imu_heading = 0.0f;
sensors.ultrasonic[0] = 200; // Front
sensors.ultrasonic[1] = 200; // Left
sensors.ultrasonic[2] = 180; // Right
sensors.ultrasonic[3] = 300; // Back

// Установить параметры навигации для цели
nav_params_t params = {0};
params.target_x = 10.0f;
params.target_y = 5.0f;
params.max_speed = 1.5f;
params.safe_distance = 1.5f;
params.emergency_stop = 0;

// Тестовая навигация в течение нескольких циклов
for (int i = 0; i < 5; i++) {
    printf("\n--- Cycle %d ---\n", i + 1);
    navigate_robot(&robot, &sensors, &params);

    //Обновите показания датчика для следующего цикла
    sensors.lidar_distance -= 0.5f;
    sensors.ultrasonic[0] -= 20;
}

printf("\n=== Navigation Test Complete ===\n");
return 0;
```

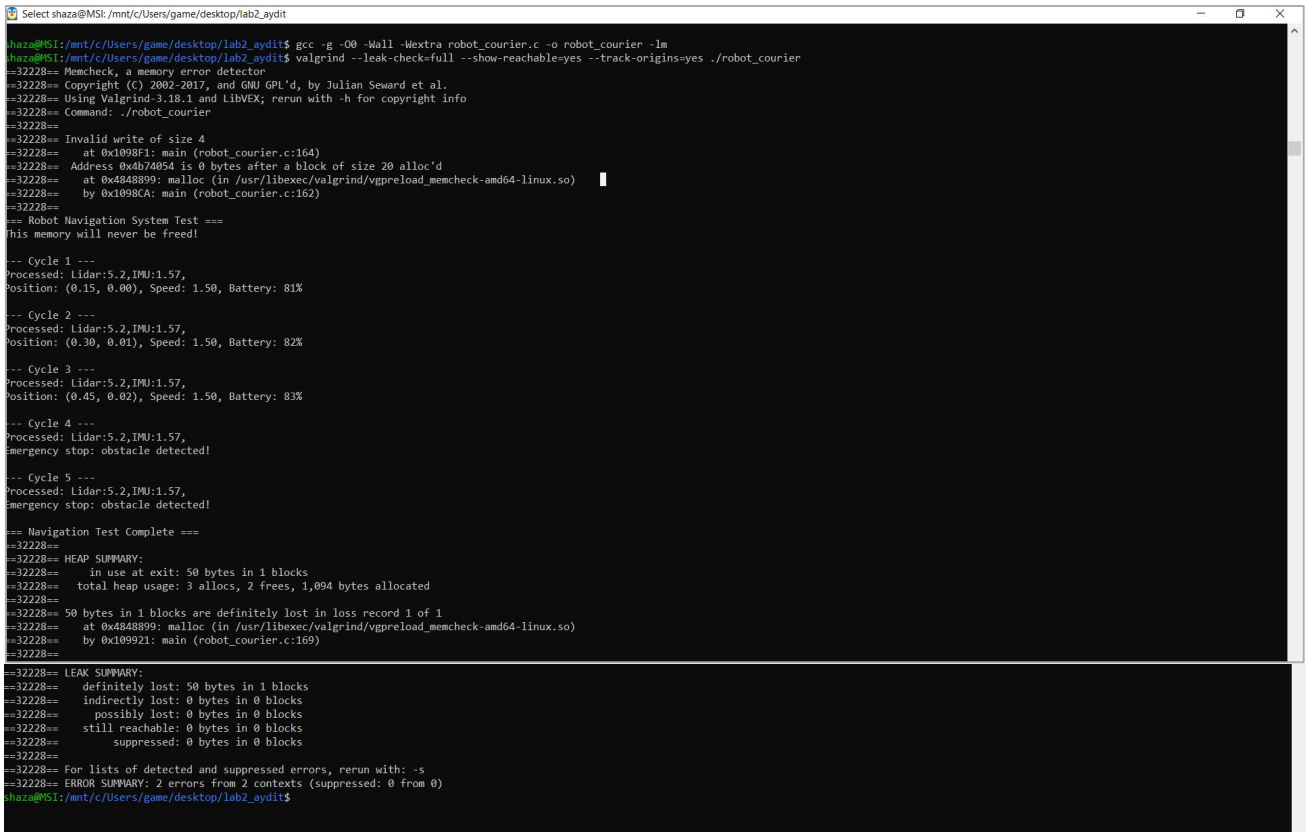
```
}
```

3.2 Используемая инструкция:

- Compiling: gcc -g -O0 -Wall -Wextra robot_courier.c -o robot_courier -lm
- Инструкция по тестированию Valgrind:

valgrind --leak-check=full --show-reachable=yes --track-origins=yes ./robot_courier

3.3 Анализ выходных данных:



- ✓ Ошибка 1:

“Invalid write of size 4” , Степень тяжести (высокая):

Программа записывает данные за пределами выделенной памяти — переполнение буфера.

это произошло в этом блоке кода:

```
161     //
162     int* arr = malloc(5 * sizeof(int));
163 for (int i = 0; i <= 5; i++) { // should be < 5
164     arr[i] = i * 2;
165 }
166 free(arr);
167     //
```

исправление:

```
161     //
162     int* arr = malloc(5 * sizeof(int));
163 for (int i = 0; i < 5; i++) { // should be < 5
164     arr[i] = i * 2;
165 }
166 free(arr);
167     //
```

✓ Ошибка 2:

“50 bytes in 1 blocks are definitely lost” Степень тяжести (средняя)

Программа выделила память и не освободила ее.

это произошло в этом блоке кода:

```
169     char* leak = malloc(50);
170     strcpy(leak, "This memory will never be freed!");
171     printf("%s\n", leak);
```

исправление:

```
169     char* leak = malloc(50);
170     strcpy(leak, "This memory will never be freed!");
171     printf("%s\n", leak);
172     free(leak);
```

дополнительные опасные ошибки, о которых valgrind не сообщил

1. “Use- After- Free” в **test_use_after_free()**
2. “Uninitialized variable” **desired_heading**
3. “Integer Overflow” в **calculate_power_usage()**
4. “Memory leak” в **plan_route()**

4. Анализ Fuzzing:

4.1 код для тестирования:

“Чтобы протестировать программу навигации робота с помощью фаззинга, мы модифицировали код так, чтобы он мог принимать произвольные входные данные от AFL, и намеренно включили несколько уязвимых функций. Это позволяет AFL исследовать различные пути выполнения и выявлять сбои или ошибки памяти.”

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>

/* -------------------- Robot Structures -------------------- */

typedef struct {
    float x, y;
    float heading;
    float speed;
    int battery;
} robot_state_t;

typedef struct {
    float lidar;
    float imu;
    int ultrasonic[4];
} sensor_data_t;

typedef struct {
    float target_x, target_y;
    float max_speed;
    float safe_distance;
} nav_params_t;

/* -------------------- Vulnerable Functions -------------------- */

// (1) VULNERABLE: small static buffer copied with attacker data (buffer overflow)
void process_sensor_string(const char *input) {
    char buf[20]; // small buffer
```

```c
    strcpy(buf, input);   // <-- overflow if input > 19 bytes
    printf("[DBG] SensorString=%s\n", buf);
}

// (2) VULNERABLE: integer overflow
int calc_battery(int current, int cost) {
    return current + cost; // CWE-190
}

// (3) VULNERABLE: use-after-free
void test_use_after_free() {
    robot_state_t *r = malloc(sizeof(robot_state_t));
    r->x = 1;
    free(r);
    r->x = 99;  // <-- use after free
}

// (4) VULNERABLE: memory leak
float *plan_path(robot_state_t *r, nav_params_t *p) {
    float *path = malloc(10 * sizeof(float)); // never freed intentionally
    for (int i = 0; i < 10; i++)
        path[i] = r->x + (i * 0.5f);
    return path; // leak
}

// (5) VULNERABLE: out-of-bounds write
void dangerous_oob() {
    int *arr = malloc(4 * sizeof(int));
    for (int i = 0; i <= 4; i++) { // writes index 4 = OOB
        arr[i] = i * 3;
    }
    free(arr);
}

/* -------------------- Robot Navigation -------------------- */
```

```c
void navigate(robot_state_t *r, sensor_data_t *s, nav_params_t *p) {

    // Stop if obstacle too close
    for (int i = 0; i < 4; i++) {
        if (s->ultrasonic[i] < p->safe_distance) {
            printf("[STOP] Obstacle detected\n");
            r->speed = 0;
            return;
        }
    }

    // Simple navigation update
    float dx = p->target_x - r->x;
    float dy = p->target_y - r->y;
    float dist = sqrtf(dx*dx + dy*dy);

    r->speed = (dist < 2.0f) ? 0.3f : p->max_speed;

    r->x += r->speed * cosf(r->heading) * 0.1f;
    r->y += r->speed * sinf(r->heading) * 0.1f;

    r->heading += (s->imu - r->heading) * 0.05f;

    // battery overflow
    r->battery = calc_battery(r->battery, 2000000000);

    printf("[NAV] pos=(%.2f, %.2f) sp=%.2f batt=%d\n",
           r->x, r->y, r->speed, r->battery);
}

/* -------------------- Main FUZZ Target -------------------- */

int main() {
    unsigned char fuzz[256];
    ssize_t len = read(0, fuzz, sizeof(fuzz));
```

```
if (len <= 0) return 0; // no crash for empty input

// ---------- Map fuzz bytes to sensors + params ----------

sensor_data_t s;
s.ultrasonic[0] = fuzz[0];
s.ultrasonic[1] = fuzz[1];
s.ultrasonic[2] = fuzz[2];
s.ultrasonic[3] = fuzz[3];
s.lidar = (float)fuzz[4];
s.imu = (float)fuzz[5] / 10.0f;

nav_params_t p;
p.target_x = (float)fuzz[6];
p.target_y = (float)fuzz[7];
p.max_speed = (float)fuzz[8] / 5.0f;
p.safe_distance = (float)fuzz[9];

robot_state_t r = {0, 0, 0, 1.0, 50};

// (1) buffer overflow
process_sensor_string((char*)fuzz);

// (2) use-after-free
test_use_after_free();

// (3) memory leak
float *route = plan_path(&r, &p);

// (4) out-of-bounds
dangerous_oob();

// (5) Actual navigation
```

```
    navigate(&r, &s, &p);


    // (6) Crash trigger: if fuzz contains "CRASH"
    if (len >= 5 && memmem(fuzz, len, "CRASH", 5)) {
        int *x = NULL;
        *x = 1; // forced segfault
    }


    return 0;
}
```

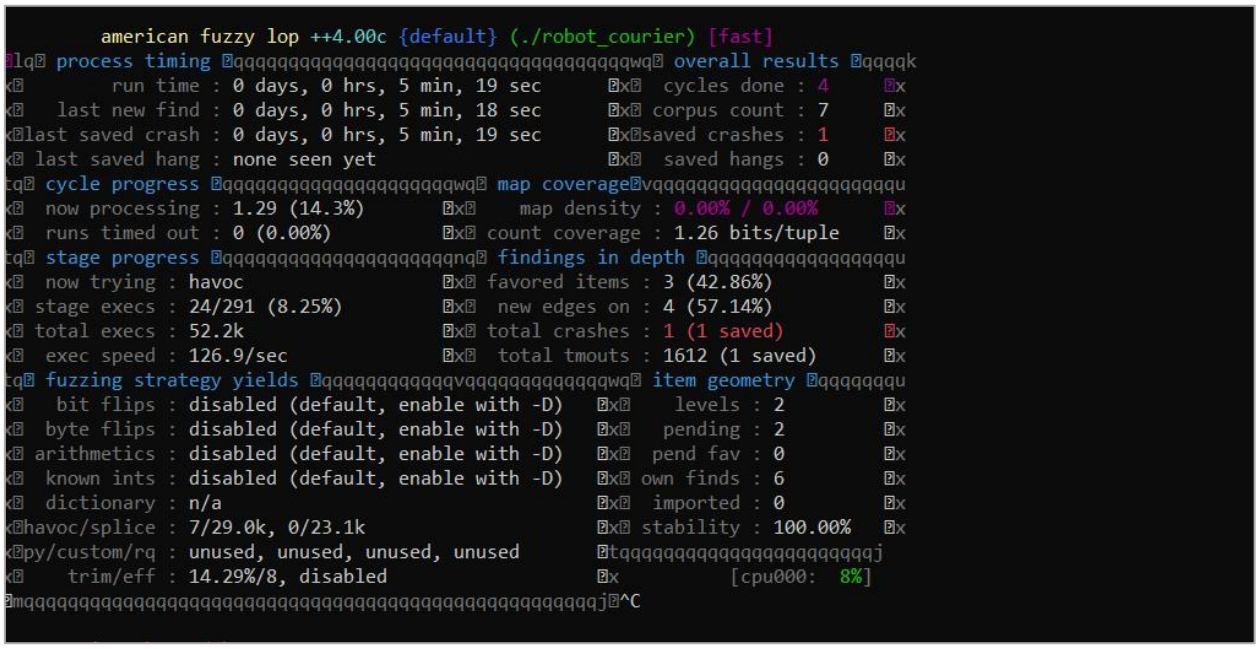
4.2 Используемая инструкция:

- Compiling: afl-gcc -g -O0 -Wall -Wextra robot_courier.c -o robot_courier -lm
- Подготовка входных данных для начального состояния “Fuzzing”:

  mkdir fuzzInput

  echo "AAAA" > fuzzInput/sample.txt
- Запуск Fuzzing: afl-fuzz -i fuzzInput -o out ./robot_courier

4.3 Анализ выходных данных:



- Входных сигналов датчиков НЕ приводило к сбоям.

- saved crashes :1, total crashes : 1

```
if (len >= 5 && memmem(fuzz, len, "CRASH", 5)) {
    int *x = NULL;
    *x = 1;
}
```

Сбой произошёл в строке, использующей функцию memmem().

AFL++ сгенерировал двоичные входные данные, которые привели к чтению memmem() за пределами выделенного буфера.

Кроме того, memmem() была использована без надлежащего объявления, что привело к неопределённому поведению в этой системе.

В результате программа попыталась обратиться к недопустимым адресам памяти, что привело к сбою.

Мы можем заменить нестандартный и небезопасный вызов **memmem()** на собственную функцию ограниченного поиска, которая использует **memcmp()** и явно проверяет размер буфера. Это исключает неопределённое поведение, предотвращает чтение памяти за пределами выделенного пространства и гарантирует полную безопасность кода при фаззинге с произвольными двоичными входными данными.

Вывод

В этом проекте к навигационной программе робота были применены два взаимодополняющих метода динамического анализа: фаззинг-тестирование (AFL++) и анализ памяти (Valgrind). Вместе они дали полную картину надежности программы, безопасности памяти и поведения при сбоях.

Такое сочетание обеспечивает эффективный рабочий процесс поиска и отладки проблем безопасности в программах на языке C, особенно тех, которые взаимодействуют с внешними данными.

Эти методы играют ключевую роль в разработке современного безопасного программного обеспечения и значительно повышают надежность и безопасность программ на языке C/C++.