# GLIBC – POSIX Safety Concepts for fdopendir

Presented by Mustafa Acar

# Presentation Outline

### GLIBC - POSIX Safety Concepts

- MT-Safe
- AS-Unsafe (Heap)
- AC-Unsafe (Mem FD)

### Testing Methodology

- Tools used (Valgrind, Helgrind, Custom Tests)
- Test scenarios and code examples

### Results

- MT-Safe behavior
- AS-Unsafe heap leaks
- AC-Unsafe memory and file descriptor issues

### Conclusion

- Evaluation of fdopendir
- Recommendations for usage
- Observations and suggestions

# GLIBC - POSIX Safety Concepts

This presentation focuses on the analysis of **fdopendir** with three concepts. It includes detailed explanations of these concepts and demonstrates the methods used to test them in the context of glibc.

MT SAFE

AS UNSAFE HEAP

AC UNSAFE MEM FD

# MT SAFE

- MT-Safe (Multi-Thread Safe) means the function can safely be called in a multi-threaded environment.
- It ensures no data races or inconsistencies occur, but it doesn’t guarantee **atomicity** or **synchronization** between calls.

```
DIR *
fdopendir (int fd)
{
  char const *name = _gl_directory_name (fd);
  DIR *dirp = name ? opendir (name) : NULL;
  if (dirp != NULL)
    dirp->fd_to_close = fd;
  return dirp;
}
```

# What Makes fdopendir – MT SAFE

**No Access to Static or Global Variables**

- fdopendir operates exclusively on the file descriptor (fd) passed to it.
- It does not access or modify any shared state (e.g., static or global variables).
- Uses thread-safe system calls internally, ensuring no conflicts during concurrent access.

**Re-Entrant Nature**

- The function does not maintain any shared state across calls.
- Every invocation of fdopendir allocates its own resources (e.g., memory for the DIR* object).

**Memory Management**

- fdopendir dynamically allocates memory (malloc) for the directory stream object, but this allocation is local to the thread and not shared.

```
DIR *
fdopendir (int fd)
{
  char const *name = _gl_directory_name (fd);
  DIR *dirp = name ? opendir (name) : NULL;
  if (dirp != NULL)
    dirp->fd_to_close = fd;
  return dirp;
}
```

# AS SAFE

AS-Safe (Asynchronous Signal-Safe) refers to functions that can be safely called from within a signal handler. These functions:

1. **No Shared State**: Do not rely on or modify global or static variables shared between the signal handler and the main program.
2. **Reentrant**: Reentrant means a function can be safely interrupted and called again before it finishes, without breaking its own data or behavior. It doesn't use shared or static data that could get messed up by the interruption.
3. **Minimal Dependencies:** Use only low-level, predictable system calls or operations that are inherently safe during signal handling.

```
DIR *
fdopendir (int fd)
{
  char const *name = _gl_directory_name (fd);
  DIR *dirp = name ? opendir (name) : NULL;
  if (dirp != NULL)
    dirp->fd_to_close = fd;
  return dirp;
}
```

## What Makes fdopendir - AS Unsafe heap

**Signal Interruption Risk:** If a signal interrupts malloc or related functions, it can leave the heap in an inconsistent state.

The signal handler might also call malloc or free, further modifying the heap before the interrupted malloc finishes. This creates overlapping or corrupted memory metadata.

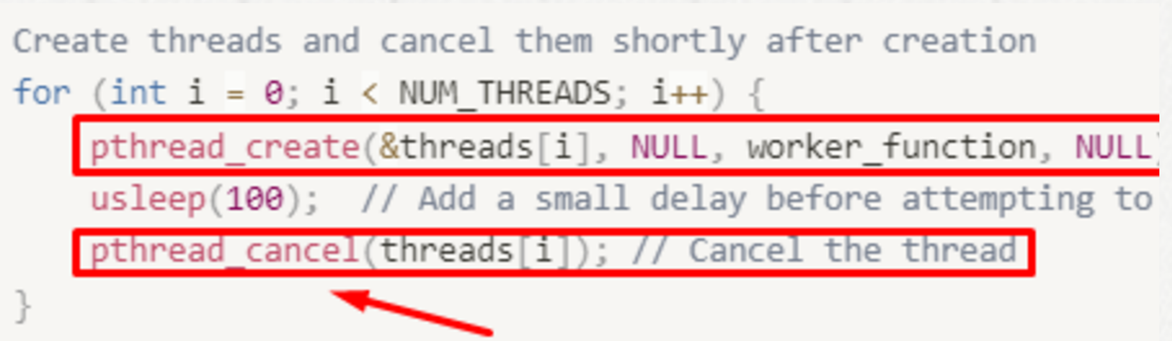
- Subsequent memory allocations or deallocations may fail.
- The program may crash or show undefined behavior due to corrupted heap structures.

```
DIR *
fdopendir (int fd)
{
  char const *name = _gl_directory_name (fd);
  DIR *dirp = name ? opendir (name) : NULL;
  if (dirp != NULL)
    dirp->fd_to_close = fd;
  return dirp;
}
```

# AC SAFE

AC-Safe (Asynchronous Cancellation-Safe) refers to functions that are safe to use when a thread can be asynchronously canceled.

1. **Handle Interruptions Gracefully:** Ensure that resources (memory, file descriptors, etc.) are properly cleaned up if a thread is canceled during their execution.
2. **No Leaks:** Do not leave memory or file descriptors open, even if interrupted mid-execution.(*** use pthread_cleanup_push)
3. **Atomic or Reentrant:** Operate in a way that ensures consistency and avoids partial states, even if canceled.



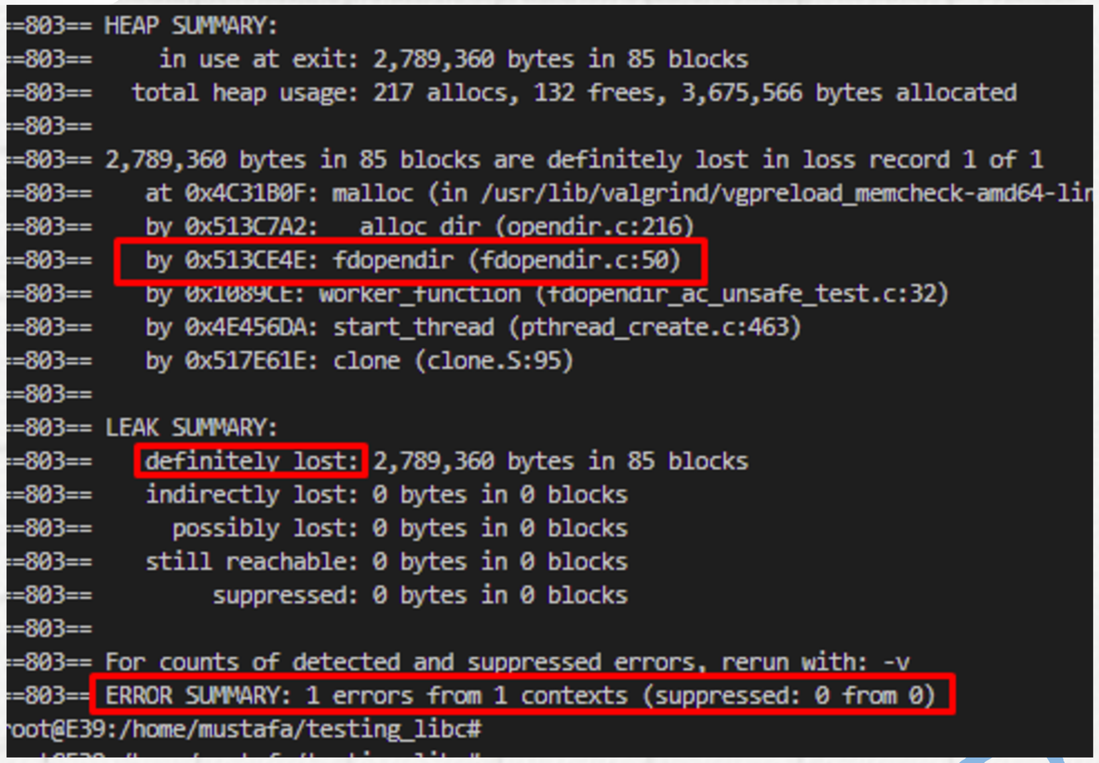
# What Makes fdopendir - AC Unsafe mem fd

**fdopendir** is AC-Unsafe (Mem FD) because:

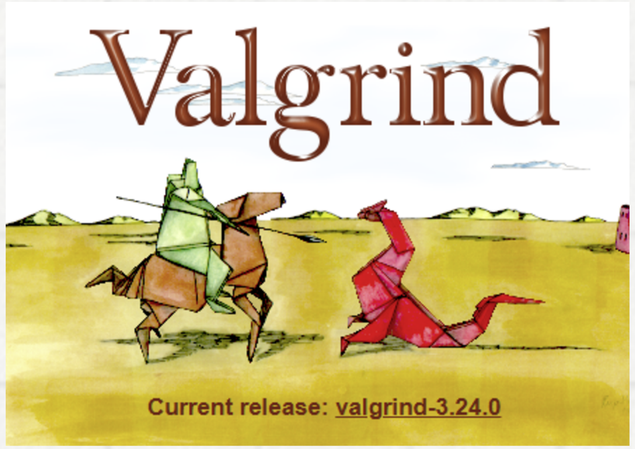
1. **Memory Allocation**: If a thread is canceled during fdopendir, the dynamically allocated memory for the DIR* object is not freed, causing memory leaks.
2. **File Descriptor Leak:** The file descriptor passed to fdopendir remains open if the thread is canceled before closedir is called.
3. **Lack of Cleanup:** fdopendir itself does not handle thread cancellation, so it cannot clean up resources (memory or file descriptors) when interrupted.

**To prevent issues:**

- Use pthread_cleanup_push to ensure resources are freed properly on cancellation.



# Testing Methodology




**1. Tools Used**

- Valgrind (Memcheck):
  - Used to detect memory leaks and ensure proper resource cleanup.
  - Helps identify unfreed memory (malloc) and unclosed file descriptors.
- Valgrind (Helgrind):
  - Used to detect race conditions and verify thread-safe behavior.
  - Confirms whether multiple threads can safely use fdopendir.

```
Compiling...
Running with valgrind --tool=memcheck --leak-check=full --track-origins=yes --trace-children=yes --show-leak-kinds=al
==1324== Memcheck, a memory error detector
==1324== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1324== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1324== Command: ./fdopendir_as_unsafe_heap_test
==1324==
==1324== error calling PR_SET_PTRACER, vgdb might block
==1324==
==1324== HEAP SUMMARY:
==1324==     in use at exit: 0 bytes in 0 blocks
==1324==   total heap usage: 102,430 allocs, 102,430 frees, 107,084,720 bytes allocated
==1324==
==1324== All heap blocks were freed -- no leaks are possible
==1324==
==1324== For counts of detected and suppressed errors, rerun with: -v
==1324== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

# Testing Methodology

**2. Test Scenarios**

**MT-Safe** Test

- Objective: Verify that fdopendir does not cause race conditions or data corruption in a multi-threaded environment.
- Method:
  - Create multiple threads, each calling fdopendir on different file descriptors.
  - Use Helgrind to monitor race conditions.

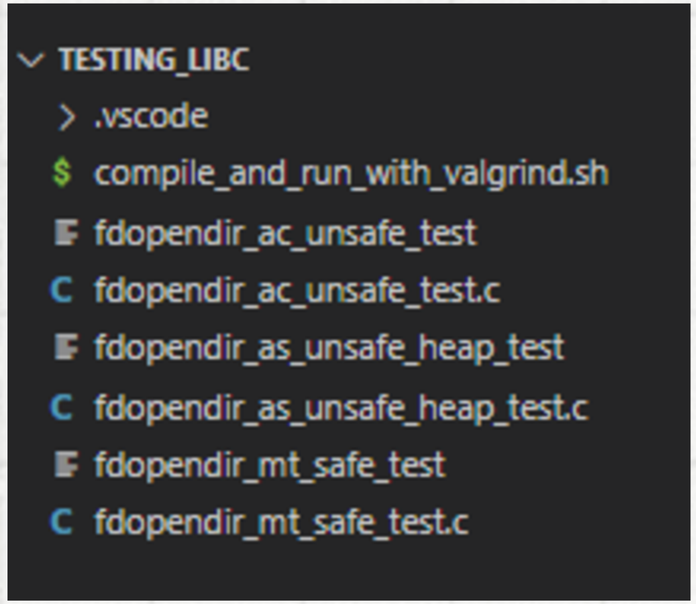
**AS-Unsafe** Test

- Objective: Check for heap corruption or crashes when fdopendir is interrupted by a signal.
- Method:
  - Set up a signal handler that calls fdopendir upon receiving a signal.
  - Send a signal to the process during execution.

**AC-Unsafe** Test

- Objective: Identify resource leaks when fdopendir is interrupted by thread cancellation.
- Method:
  - Cancel a thread during fdopendir execution.
  - Use Valgrind to detect unfreed memory or unclosed file descriptors.

**The scenarios will be shared through my GitHub page.**




```c
/**
 * @brief Signal-sending thread function.
 *
 * Continuously sends SIGUSR1 to the current process until `stop_signal` is set.
 *
 * @param arg Pointer to optional arguments (unused in this function).
 * @return NULL Always returns NULL.
 */
void *signal_sender(void *arg) {
    while (!stop_signal) {
        raise(SIGUSR1); // Trigger the signal
        usleep(10);     // Small delay to avoid overwhelming the process
    }
    return NULL;
}
```

# Results of fdopendir

### MT-Safe Results

- Content:
  - No data races or thread-related issues were detected in fdopendir when used properly with closedir.
  - However, If your program tries to open too many file descriptors simultaneously (e.g., due to NUM_THREADS and NUM_ITERATIONS), it might hit the system's file descriptor limit. When the limit is reached, open() fails, returning -1.

    Solution:

    Increase the file descriptor limit (ulimit -n).

    Reduce NUM_THREADS or NUM_ITERATIONS.
  - Helgrind confirmed it is safe for multi-threaded environments under normal conditions.

### AS-Unsafe Results

- Content:
  - Using fdopendir in signal handlers may caused memory corruption when interrupted.

### AC-Unsafe Results

- Content:
  - Thread cancellation during fdopendir led to memory leaks and unclosed file descriptors.
  - Valgrind reported definitive memory leaks for interrupted calls.

**MT SAFE**

```
Selected file: fdopendir mt safe test.c
Compiling...
Running with valgrind --tool=helgrind --trace-children=yes ...
==4991== Helgrind, a thread error detector
==4991== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==4991== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright in
==4991== Command: ./fdopendir_mt_safe_test
==4991==
==4991== error calling PR_SET_PTRACER, vgdb might block
All threads have completed.
==4991==
==4991== For counts of detected and suppressed errors, rerun with: -v
==4991== Use --history-level=approx or =none to gain increased speed, at
==4991== the cost of reduced accuracy of conflicting-access information
==4991== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1479 from 2
```

**AC UNSAFE**

```
Selected file: fdopendir ac unsafe test.c
Compiling...
Running with valgrind --tool=memcheck --leak-check=full --track-origins=yes --trace-chil
==5039== Memcheck, a memory error detector
==5039== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5039== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5039== Command: ./fdopendir_ac_unsafe_test
==5039==
==5039== error calling PR_SET_PTRACER, vgdb might block
All threads have been joined.
==5039==
==5039== HEAP SUMMARY:
==5039==     in use at exit: 2,756,544 bytes in 84 blocks
==5039==   total heap usage: 227 allocs, 143 frees, 4,003,726 bytes allocated
==5039==
==5039== 2,756,544 bytes in 84 blocks are definitely lost in loss record 1 of 1
==5039==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==5039==    by 0x513C7A2:   alloc_dir (opendir.c:216)
==5039==    by 0x513CE4E: fdopendir (fdopendir.c:50)
==5039==    by 0x1089CE: worker_function (fdopendir_ac_unsafe_test.c:32)
==5039==    by 0x4E456DA: start_thread (pthread_create.c:463)
==5039==    by 0x517E61E: clone (clone.S:95)
==5039==
==5039== LEAK SUMMARY:
==5039==    definitely lost: 2,756,544 bytes in 84 blocks
==5039==    indirectly lost: 0 bytes in 0 blocks
==5039==      possibly lost: 0 bytes in 0 blocks
==5039==    still reachable: 0 bytes in 0 blocks
==5039==         suppressed: 0 bytes in 0 blocks
==5039==
==5039== For counts of detected and suppressed errors, rerun with: -v
==5039== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Conclusion

| Conclusion Points | Details |
| --- | --- |
| **fdopendir MT-Safe** | - fdopendir is MT-Safe.<br>- Does not access shared global or static variables, preventing race conditions.<br>-Each thread call to fdopendir operates independently without interfering with other threads. |
| **fdopendir AS-Unsafe due to Heap Usage** | Relies on malloc for memory allocation, making it vulnerable to heap corruption on interruptions. |
| **fdopendir AC-Unsafe due to Resource Management** | Thread cancellations during fdopendir execution may lead to unfreed file descriptors or memory. |
| **Recommendations** | Avoid using fdopendir in signal handlers. Carefully manage cleanup when handling thread cancellations. |

# Thank you very much!