monitoring tool vulnerabilities

Nikolaos Mouzakitis

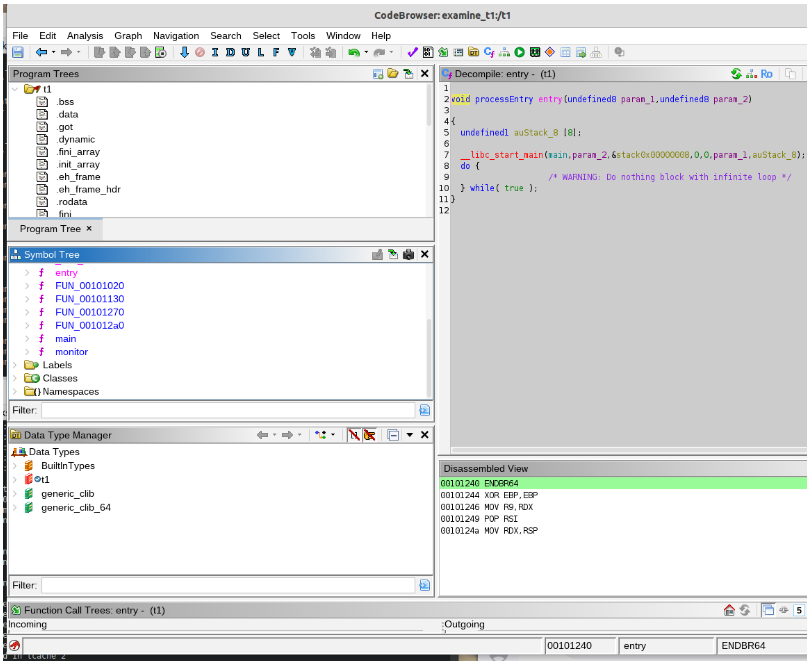
Date Last Edited: March 25, 2025

Abstract

This report provides a security assessment of the directory monitoring application used by system administrators. The application is designed to monitor changes in directory structures, but several vulnerabilities have been identified that could lead to security risks. This report describes each vulnerability, its impact, a proof of concept (PoC) for exploitation, and recommendations for mitigation.

I. Decompilation using Ghidra 11.3.1

The program was decompiled using the NSA's Ghidra tool[1], and two basic functions were identified and examined. The decompilation code is presented on the Appendix. On the decompiled code as provided by the ghidra app, the only modification made was the renaming part of the functions/variables to support better understanding in code examination.



```

void processEntry entry(undefined8 param_1,undefined8 param_2)

{
  undefined1 auStack_8 [8];

  __libc_start_main(main,param_2,&stack0x00000008,0,0,param_1,auStack_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}

```

```
00101240 ENDBR64
00101244 XOR EBP,EBP
00101246 MOV R9,RDX
00101249 POP RSI
0010124a MOV RDX,RSP
```

FIGURE 1. ghidra's usage for decompilation

A. Decompiled code overview

Observing the decompiled code from the Appendix, function `monitor` is designed to monitor a directory (dirname) for changes in its contents. It uses `opendir`, `readdir`, and `closedir` to count files in the directory periodically (every 5 seconds). The function also interacts with shared memory via the pointer `smaddr`.

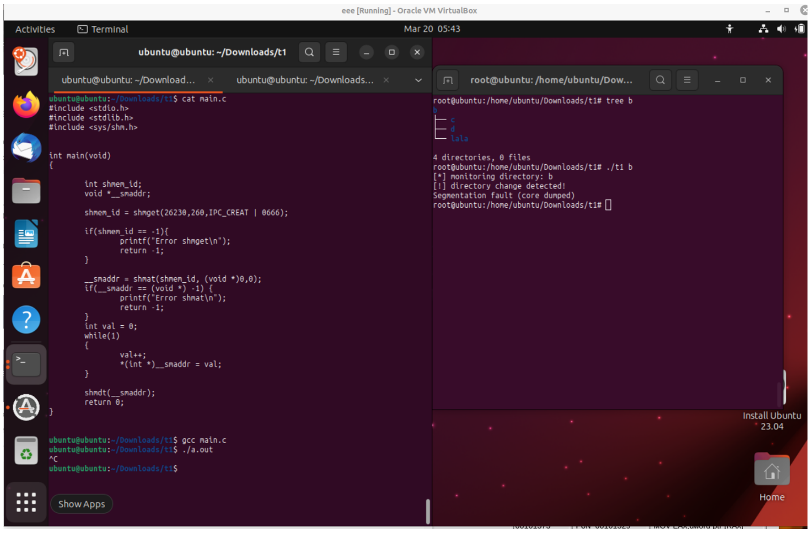
The function dynamically allocates memory for tracking directory entries. It compares the current state of the directory with the previous state to detect changes based on the number of the directories discovered to be present. Memory management is performed using `malloc`, `realloc`, and `free`.

II. Vulnerability 1: Unbounded memory allocation

A. Technical description

The application uses a shared memory segment created via the `shmget()` system call. The shared memory segment is used to store an integer value that determines the size of memory allocations in the `monitor.c` file. However, the shared memory segment is not properly initialized, and its contents can be manipulated by external processes. This can lead to unbounded memory allocation, as the application relies on the value stored in the shared memory segment to determine the size of memory allocations.

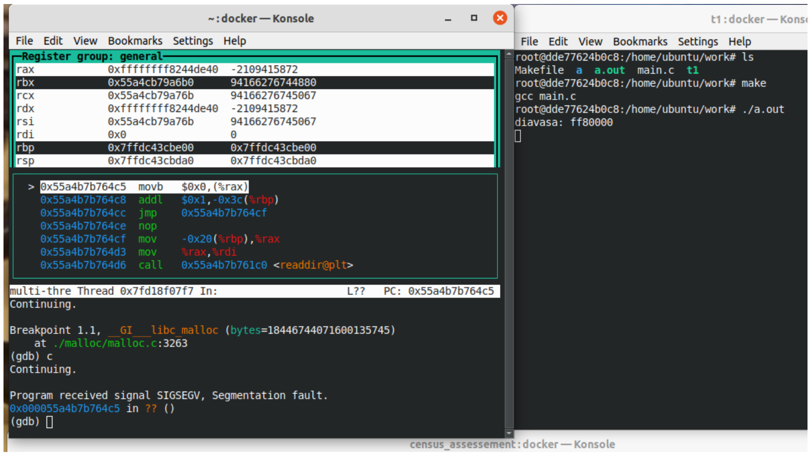
The vulnerability is located in the `main.c` file, specifically in the `shmget()` call, and in the `monitor.c` file(line 21) where the shared memory value is used for memory allocation.


eee [Running] - Oracle VM VirtualBox
Activities
Terminal
Mar 20 05:43
ubuntu@ubuntu: ~/Downloads/t1
ubuntu@ubuntu: ~/Download...
ubuntu@ubuntu: ~/Downloads...
root@ubuntu: /home/ubuntu/Dow...
Install Ubuntu 23.04
Home
Show Apps

```
ubuntu@ubuntu:~/Downloads/t1$ cat main.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>


int main(void)
{

        int shmem_id;
        void *__smaddr;

        shmem_id = shmget(26230,260,IPC_CREAT | 0666);

        if(shmem_id == -1){
                printf("Error shmget\n");
                return -1;
        }

        __smaddr = shmat(shmem_id, (void *)0,0);
        if(__smaddr == (void *) -1) {
                printf("Error shmat\n");
                return -1;
        }
        int val = 0;
        while(1)
        {
                val++;
                *(int *)__smaddr = val;
        }

        shmdt(__smaddr);
        return 0;
}


ubuntu@ubuntu:~/Downloads/t1$ gcc main.c
ubuntu@ubuntu:~/Downloads/t1$ ./a.out
^C
ubuntu@ubuntu:~/Downloads/t1$
```

```
root@ubuntu:/home/ubuntu/Downloads/t1# tree b
b
├── c
├── d
└── lala

4 directories, 0 files
root@ubuntu:/home/ubuntu/Downloads/t1# ./t1 b
[*] monitoring directory: b
[!] directory change detected!
Segmentation fault (core dumped)
root@ubuntu:/home/ubuntu/Downloads/t1# 
```

FIGURE 2. Screenshot from t1 application getting killed.


~ : docker — Konsole
File Edit View Bookmarks Settings Help
t1 : docker — Konsole
census_assessement : docker — Konsole

```
┌Register group: general──────────────────────────────
rax            0xffffffff8244de40  -2109415872
rbx            0x55a4cb79a6b0      94166276744880
rcx            0x55a4cb79a76b      94166276745067
rdx            0xffffffff8244de40  -2109415872
rsi            0x55a4cb79a76b      94166276745067
rdi            0x0                 0
rbp            0x7ffdc43cbe00      0x7ffdc43cbe00
rsp            0x7ffdc43cbda0      0x7ffdc43cbda0

  > 0x55a4b7b764c5  movb   $0x0,(%rax)
    0x55a4b7b764c8  addl   $0x1,-0x3c(%rbp)
    0x55a4b7b764cc  jmp    0x55a4b7b764cf
    0x55a4b7b764ce  nop
    0x55a4b7b764cf  mov    -0x20(%rbp),%rax
    0x55a4b7b764d3  mov    %rax,%rdi
    0x55a4b7b764d6  call   0x55a4b7b761c0 <readdir@plt>

multi-thre Thread 0x7fd18f07f7 In:                    L??   PC: 0x55a4b7b764c5
Continuing.

Breakpoint 1.1, __GI___libc_malloc (bytes=18446744071600135745)
    at ./malloc/malloc.c:3263
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x000055a4b7b764c5 in ?? ()
(gdb) 
```

```
root@dde77624b0c8:/home/ubuntu/work# ls
Makefile  a  a.out  main.c  t1
root@dde77624b0c8:/home/ubuntu/work# make
gcc main.c
root@dde77624b0c8:/home/ubuntu/work# ./a.out
diavasa: ff80000
```

FIGURE 3. gdb recreation of allocating unbounded alloc.

B. Impact

An attacker or another process can map the same shared memory segment into its address space and write arbitrary values to it. This can cause the application to allocate excessive amounts of memory, leading to resource exhaustion or a crash of the application. This could result in a denial of service (DoS) condition, affecting the system's availability.

C. Proof of Concept

A small program can be developed to map the shared memory segment and write incrementing integer values to its starting address. The following code demonstrates how to manipulate the shared memory segment:

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

```

```c
int main() {
    int shmid = shmget(0x6676, 260, 0666);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    int *shmaddr = (int *)shmat(shmid, NULL, 0);
    if (shmaddr == (void *)-1) {
        perror("shmat");
        return 1;
    }

    *shmaddr = 0xfff7000;

    shmdt(shmaddr);
    return 0;
}
```

When this program is executed, the application will allocate excessive memory based on the manipulated value, potentially causing a crash.

D. Recommendation

To mitigate this vulnerability, the following measures should be implemented:

- Range validation: Before allocating memory, validate that the value from the shared memory segment falls within a reasonable range.
- Restricted access: Modify the `shmget()` call to restrict access to the shared memory segment. For example:

  ```c
      int shmid = shmget(0x6676, 260, IPC_CREAT | 0600);
  ```

  This ensures that only the root user has read and write access to the shared memory segment.

III. Vulnerability 2: Double-free error when deleting directories under the monitored directory

A. Technical description

The application attempts to free already deallocated memory when subfolders within the monitored directory are deleted. This results in a double-free error, which can lead to undefined behavior, including crashes or potential security vulnerabilities. The issue occurs in the `monitor.c` file, specifically in the memory deallocation logic.

B. Impact

A double-free error can corrupt the memory allocator's internal state and this results to a denial of service by crashing the application.

C. Proof of Concept

To trigger the double-free issue, follow these steps:

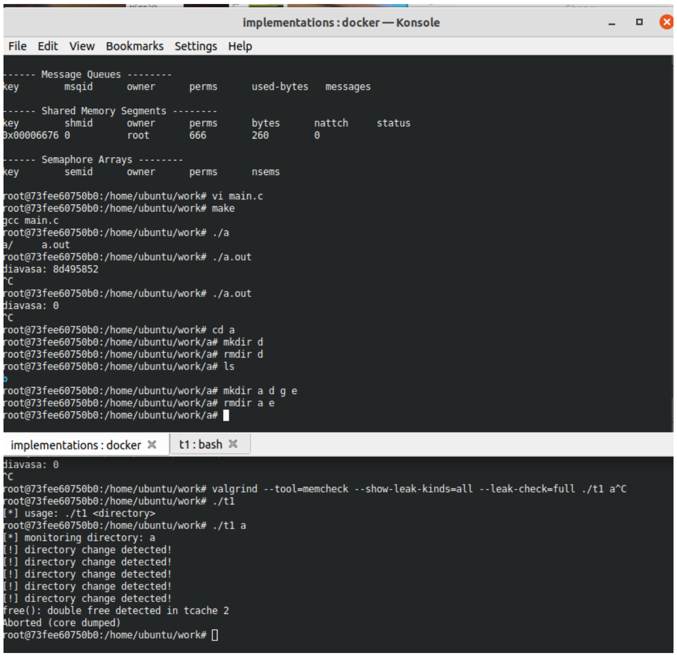
1) Start the application and monitor a directory with subfolders.
2) Delete one or more subfolders while the application is running.
3) Observe the application attempting to free already deallocated memory, leading to a crash.

D. Recommendation

To prevent this issue, a defensive programming practice should be adopted by setting pointers to `NULL` after freeing them. For example:

```c
free(*(void **)((long)new_pointer + (long)local_40 * 8));
*(void **)((long)new_pointer + (long)local_40 * 8) = NULL;
```

This ensures that any subsequent calls to `free()` on the same pointer will have no effect.

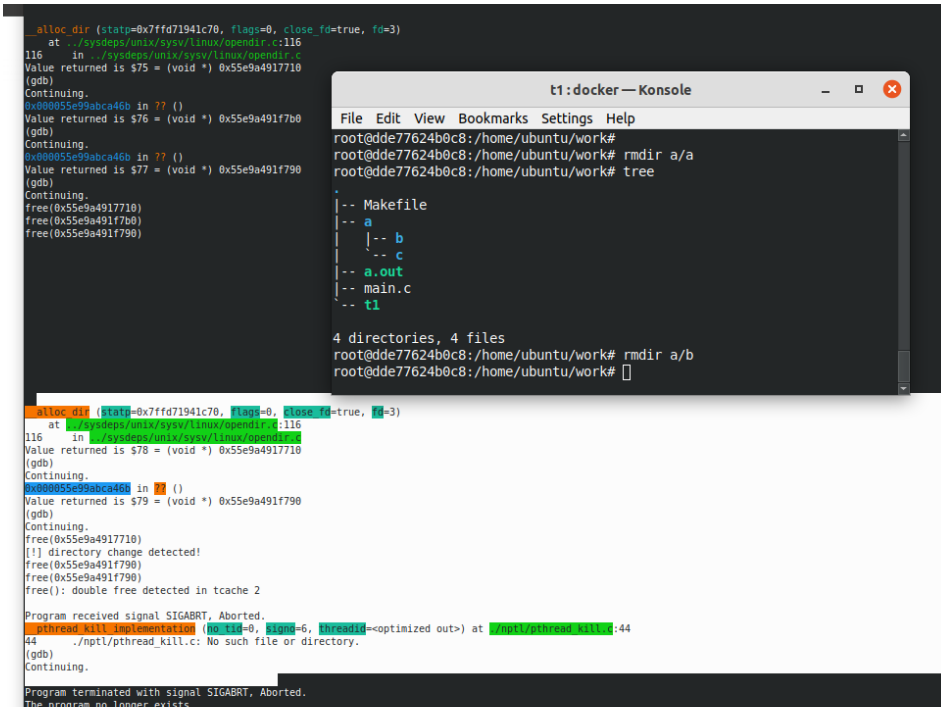


implementations : docker — Konsole
File Edit View Bookmarks Settings Help
implementations : docker
t1 : bash

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00006676 0          root       666        260        0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

root@73fee60750b0:/home/ubuntu/work# vi main.c
root@73fee60750b0:/home/ubuntu/work# make
gcc main.c
root@73fee60750b0:/home/ubuntu/work# ./a
a/     a.out
root@73fee60750b0:/home/ubuntu/work# ./a.out
diavasa: 8d495852
^C
root@73fee60750b0:/home/ubuntu/work# ./a.out
diavasa: 0
^C
root@73fee60750b0:/home/ubuntu/work# cd a
root@73fee60750b0:/home/ubuntu/work/a# mkdir d
root@73fee60750b0:/home/ubuntu/work/a# rmdir d
root@73fee60750b0:/home/ubuntu/work/a# ls
b
root@73fee60750b0:/home/ubuntu/work/a# mkdir a d g e
root@73fee60750b0:/home/ubuntu/work/a# rmdir a e
root@73fee60750b0:/home/ubuntu/work/a# 
```

```
diavasa: 0
^C
root@73fee60750b0:/home/ubuntu/work# valgrind --tool=memcheck --show-leak-kinds=all --leak-check=full ./t1 a^C
root@73fee60750b0:/home/ubuntu/work# ./t1
[*] usage: ./t1 <directory>
root@73fee60750b0:/home/ubuntu/work# ./t1 a
[*] monitoring directory: a
[!] directory change detected!
[!] directory change detected!
[!] directory change detected!
[!] directory change detected!
[!] directory change detected!
free(): double free detected in tcache 2
Aborted (core dumped)
root@73fee60750b0:/home/ubuntu/work# 
```

FIGURE 4. **Double free observation**



```
__alloc_dir (statp=0x7ffd71941c70, flags=0, close_fd=true, fd=3)
    at ../sysdeps/unix/sysv/linux/opendir.c:116
116     in ../sysdeps/unix/sysv/linux/opendir.c
Value returned is $75 = (void *) 0x55e9a4917710
(gdb)
Continuing.
0x000055e99abca46b in ?? ()
Value returned is $76 = (void *) 0x55e9a491f7b0
(gdb)
Continuing.
0x000055e99abca46b in ?? ()
Value returned is $77 = (void *) 0x55e9a491f790
(gdb)
Continuing.
free(0x55e9a4917710)
free(0x55e9a491f7b0)
free(0x55e9a491f790)
```

```
root@dde77624b0c8:/home/ubuntu/work#
root@dde77624b0c8:/home/ubuntu/work# rmdir a/a
root@dde77624b0c8:/home/ubuntu/work# tree
.
|-- Makefile
|-- a
|   |-- b
|   `-- c
|-- a.out
|-- main.c
`-- t1

4 directories, 4 files
root@dde77624b0c8:/home/ubuntu/work# rmdir a/b
root@dde77624b0c8:/home/ubuntu/work# 
```

```
__alloc_dir (statp=0x7ffd71941c70, flags=0, close_fd=true, fd=3)
    at ../sysdeps/unix/sysv/linux/opendir.c:116
116     in ../sysdeps/unix/sysv/linux/opendir.c
Value returned is $78 = (void *) 0x55e9a4917710
(gdb)
Continuing.
0x000055e99abca46b in ?? ()
Value returned is $79 = (void *) 0x55e9a491f790
(gdb)
Continuing.
free(0x55e9a4917710)
[!] directory change detected!
free(0x55e9a491f790)
free(0x55e9a491f790)
free(): double free detected in tcache 2

Program received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=<optimized out>) at ./nptl/pthread_kill.c:44
44      ./nptl/pthread_kill.c: No such file or directory.
(gdb)
Continuing.

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
```

FIGURE 5. **GDB examination of double free-ing the pointer.**

IV. Vulnerability 3: Memory leak

A. Technical description

A memory leak occurs during the first iteration of the directory monitoring loop in the `monitor.c` file. When the application processes directory entries, memory is allocated for each entry using `malloc()`, but this memory is not freed specific on the first iteration. This results in a memory leak proportional to the number of directories present in the monitored folder.

B. Utilization of Valgrind to detect leakage

Next, Valgrind[2] has been utilized to check for memory leakage issues. The **t1** application was executed with 2 different configurations. In the next snippet, are presented the findings by initializing to 5, the first 4bytes of the shared memory segment which monitor.c file uses for allocating memory for each present directory. Examination took place by having a monitored directory with seven and two subfolders.

Listing 1. monitoring memory leakage in a directory with 7 folders

```
        root@340eed153787:/home/ubuntu/work# valgrind --leak-check=full --track-origins=yes ./t1 a
==355525== Memcheck, a memory error detector
==355525== Copyright (C) 2002-2022, and GNU GPL’d,␣by␣Julian␣Seward␣et␣al.
==355525==␣Using␣Valgrind-3.19.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==355525==␣Command:␣./t1␣a
==355525==
[*]␣monitoring␣directory:␣a
[!]␣directory␣change␣detected!
^C==355525==
==355525==␣Process␣terminating␣with␣default␣action␣of␣signal␣2␣(SIGINT)
==355525==␣␣␣␣at␣0x4934E36:␣clock_nanosleep@@GLIBC_2.17␣(clock_nanosleep.c:78)
==355525==␣␣␣␣by␣0x4939C96:␣nanosleep␣(nanosleep.c:25)
==355525==␣␣␣␣by␣0x4939BCD:␣sleep␣(sleep.c:55)
==355525==␣␣␣␣by␣0x109586:␣???␣(in␣/home/ubuntu/work/t1)
==355525==␣␣␣␣by␣0x109692:␣???␣(in␣/home/ubuntu/work/t1)
==355525==␣␣␣␣by␣0x487BA8F:␣(below␣main)␣(libc_start_call_main.h:58)
==355525==
==355525==␣HEAP␣SUMMARY:
==355525==␣␣␣␣␣in␣use␣at␣exit:␣1,139␣bytes␣in␣9␣blocks
==355525==␣␣␣total␣heap␣usage:␣26␣allocs,␣17␣frees,␣99,657␣bytes␣allocated
==355525==
==355525==␣35␣bytes␣in␣7␣blocks␣are␣definitely␣lost␣in␣loss␣record␣1␣of␣3
==355525==␣␣␣␣at␣0x4843828:␣malloc␣(in␣/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==355525==␣␣␣␣by␣0x10946A:␣???␣(in␣/home/ubuntu/work/t1)
==355525==␣␣␣␣by␣0x109692:␣???␣(in␣/home/ubuntu/work/t1)
==355525==␣␣␣␣by␣0x487BA8F:␣(below␣main)␣(libc_start_call_main.h:58)
==355525==
==355525==␣LEAK␣SUMMARY:
==355525==␣␣␣␣definitely␣lost:␣35␣bytes␣in␣7␣blocks
==355525==␣␣␣␣indirectly␣lost:␣0␣bytes␣in␣0␣blocks
==355525==␣␣␣␣␣␣possibly␣lost:␣0␣bytes␣in␣0␣blocks
==355525==␣␣␣␣still␣reachable:␣1,104␣bytes␣in␣2␣blocks
==355525==␣␣␣␣␣␣␣␣␣suppressed:␣0␣bytes␣in␣0␣blocks
==355525==␣Reachable␣blocks␣(those␣to␣which␣a␣pointer␣was␣found)␣are␣not␣shown.
==355525==␣To␣see␣them,␣rerun␣with:␣--leak-check=full␣--show-leak-kinds=all
==355525==
==355525==␣For␣lists␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-s
==355525==␣ERROR␣SUMMARY:␣1␣errors␣from␣1␣contexts␣(suppressed:␣0␣from␣0)
```

Listing 2. monitoring memory leakage in a directory with 2 folders

```
root@340eed153787:/home/ubuntu/work# valgrind --leak-check=full --track-origins=yes ./t1 a
==355530== Memcheck, a memory error detector
==355530== Copyright (C) 2002-2022, and GNU GPL’d,␣by␣Julian␣Seward␣et␣al.
==355530==␣Using␣Valgrind-3.19.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==355530==␣Command:␣./t1␣a
==355530==
[*]␣monitoring␣directory:␣a
[!]␣directory␣change␣detected!
^C==355530==
==355530==␣Process␣terminating␣with␣default␣action␣of␣signal␣2␣(SIGINT)
==355530==␣␣␣␣at␣0x4934E36:␣clock_nanosleep@@GLIBC_2.17␣(clock_nanosleep.c:78)
==355530==␣␣␣␣by␣0x4939C96:␣nanosleep␣(nanosleep.c:25)
==355530==␣␣␣␣by␣0x4939BCD:␣sleep␣(sleep.c:55)
==355530==␣␣␣␣by␣0x109586:␣???␣(in␣/home/ubuntu/work/t1)
==355530==␣␣␣␣by␣0x109692:␣???␣(in␣/home/ubuntu/work/t1)
==355530==␣␣␣␣by␣0x487BA8F:␣(below␣main)␣(libc_start_call_main.h:58)
==355530==
==355530==␣HEAP␣SUMMARY:
==355530==␣␣␣␣␣in␣use␣at␣exit:␣1,114␣bytes␣in␣4␣blocks
==355530==␣␣␣total␣heap␣usage:␣8␣allocs,␣4␣frees,␣66,756␣bytes␣allocated
==355530==
==355530==␣10␣bytes␣in␣2␣blocks␣are␣definitely␣lost␣in␣loss␣record␣1␣of␣3
==355530==␣␣␣␣at␣0x4843828:␣malloc␣(in␣/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==355530==␣␣␣␣by␣0x10946A:␣???␣(in␣/home/ubuntu/work/t1)
```

```
==355530==␣␣␣␣by␣0x109692:␣???␣(in␣/home/ubuntu/work/t1)
==355530==␣␣␣␣by␣0x487BA8F:␣(below␣main)␣(libc_start_call_main.h:58)
==355530==
==355530==␣LEAK␣SUMMARY:
==355530==␣␣␣␣definitely␣lost:␣10␣bytes␣in␣2␣blocks
==355530==␣␣␣␣indirectly␣lost:␣0␣bytes␣in␣0␣blocks
==355530==␣␣␣␣␣␣possibly␣lost:␣0␣bytes␣in␣0␣blocks
==355530==␣␣␣␣still␣reachable:␣1,104␣bytes␣in␣2␣blocks
==355530==␣␣␣␣␣␣␣␣␣suppressed:␣0␣bytes␣in␣0␣blocks
==355530==␣Reachable␣blocks␣(those␣to␣which␣a␣pointer␣was␣found)␣are␣not␣shown.
==355530==␣To␣see␣them,␣rerun␣with:␣--leak-check=full␣--show-leak-kinds=all
==355530==
==355530==␣For␣lists␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-s
==355530==␣ERROR␣SUMMARY:␣1␣errors␣from␣1␣contexts␣(suppressed:␣0␣from␣0)
```

After validating that memory leak is directly related to the number of directories present on the monitored folder on the first code iteration of `monitor.c`, in the next step the detection of the vulnerability inside the code follow. The occurance of this memory leak during the first iteration of the directory monitoring loop specifically because

- On the first iteration of the main loop in function `monitor()`, `local_4c` is initialized to 0. When directory entries are processed, memory is allocated for each entry using malloc() and stored in the `new_pointer` array. However, since `local_4c` is 0, the code skips the memory deallocation step in the else block (lines 57-59). As a result, the memory allocated for the directory entries in the first iteration is never freed.

This behavior constitutes a memory leak because the dynamically allocated memory for `oti_na_nai` in the first iteration is not released, even though it is no longer accessible after the first iteration. Over time, repeated execution of this function could lead into memory consumption and resource exhaustion.

C. Impact

Over time, repeated execution of the monitoring function can lead to memory consumption and resource exhaustion, potentially causing the application or system to become unresponsive.

D. Proof of Concept

To observe the memory leak, monitor a directory with multiple subfolders and check the memory usage using `Valgrind`. The following command can be used:

```
valgrind --leak-check=full --track-origins=yes ./t1 /path/to/dir
```

The output will show memory blocks that are definitely lost, indicating a memory leak.

E. Recommendation

To mitigate this issue, ensure that all allocated memory is freed, even if no changes are detected in the directory structure. Modification in the code is required to free memory in all cases, not just when changes are detected. As a mitigation countermeasure it should be ensured that all the `tmp2` number of allocations are getting freed instead of doing the deallocations this way. Or in a different fashion incorporate two if statements instead of the if-else structure with the assignment of `tmp2` to `loc_4c` in between them.

For example:

```
for (int i = 0; i < tmp2; i++) {
    free(*(void **)((long)new_pointer + i * 8));
}
```

V. Vulnerability 4: Integer overflow on tmp

A. Technical description

The variable `tmp` is used to dynamically allocate memory for storing file names in a directory. It is doubled (`tmp = tmp << 1`) whenever the number of files exceeds the current capacity (file `monitor.c` line 37). However, there is no check to ensure that `tmp` does not exceed the maximum value of an `int`. If `tmp` becomes too large, it will overflow, leading to undefined behavior. On some architectures (f.e, ARM with overflow trapping) or with certain compiler flags like `-ftrapv`), signed integer overflow can trigger exceptions or program crashes. Even on architectures where overflow wraps around, the behavior is unpredictable and can lead to security vulnerabilities.

B. Impact

An integer overflow in `tmp` can lead to undefined behavior, including program crashes or heap corruption. The inconsistency in how different architectures handle overflow makes this a significant security risk.

C. Recommendation

To prevent this issue, add bounds checking before doubling `tmp` or use a larger data type like `size_t` or even better `uint64_t` to avoid overflow entirely. Additionally, a consideration of utilizing compiler flags on compilation such as `-ftrapv` or `-fsanitize=undefined` for the detection and prevention of integer overflows.

VI. Vulnerability 5: Infinite loop without exit condition

A. Technical description

The `monitor` function contains an infinite loop with no clear exit condition:

```
    while (true) {
        ...
    }
```

This makes it difficult to terminate the program gracefully. If the program is forcibly terminated via `Ctrl+C` or a `kill`, it will not execute cleanup code, leading to resource leaks and potential security issues.

B. Impact

Abnormal termination can result in:

- Memory leaks (allocated memory is not freed).
- Shared memory leaks (shared memory segments are not detached or destroyed).
- File descriptor leaks (open files are not closed).

C. Recommendation

To address this issue, implement a mechanism to exit the loop gracefully and perform proper cleanup before termination. For example:

- Use a flag variable to control the loop.
- Handle signals (e.g., `SIGINT`, `SIGTERM`) to perform cleanup.
- Ensure all resources get properly freed or closed.

VII. Vulnerability 6: Shared memory segment is not configured for its application logic

A. Technical description

Another vulnerability of the program, which is directly connected to the improper configuration of the shared memory segment, is that if `init_smaddr` is set to 0, the `strncpy` function on line 42 of *monitor.c* does not copy any data. However, in the next line it still writes a null-terminator at the calculated memory location (init_smaddr - 1). Since (init_smaddr - 1) evaluates to *0xFFFFFFFF* (a large invalid address), this can cause memory corruption by overwriting unintended areas of memory.

B. Implications

This vulnerability can lead to several consequences:

- **Memory corruption**: Writing the null-termination character to an unintended location can corrupt adjacent memory, leading to unpredictable program behavior or crashes.
- **Security risks**: If an attacker can control the value of `init_smaddr`, they may exploit this vulnerability to overwrite critical memory regions.
- **Data integrity issues**: The program may fail to properly monitor the directory, as the shared memory segment is not being used as intended (writing out of bounds).

C. Recommendation

To address this issue, the followings can be implemented:

- **Input validation-bound check for strncpy()**: It has to be ensured that `init_smaddr` is validated before use. If it is read from shared memory, verification that it contains a valid value for should be performed.
- **Error handling**: Error handling logic should be added to detect and respond to invalid values of `init_smaddr`, such as logging an error message or termination of the program gracefully.
- **Default values**: Initialization of the `init_smaddr` to a safe default value if it is read as 0 or invalid.

VIII. Vulnerability 7: Shared memory leak on directory open failure

A. Technical Description

When the program attempts to monitor a non-existent directory, the `opendir` function fails, and the program prints an error message before exiting. However, the shared memory segment created earlier in the program is not properly cleaned up. This results in a shared memory leak, as shown in the following example:

```
root@3ef1beec6f09:/home/ubuntu/work# ./t1 sdfsd
[*] monitoring directory: sdfsd
opendir: No such file or directory
root@3ef1beec6f09:/home/ubuntu/work# ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00006676 6          root       666        260        0
```

In this case, the shared memory segment (with key `0x00006676` and shmid `6`) remains allocated in the system even after the program terminates. However in this particular case the program uses always the constant value of `0x6676` for the key as can be observed in `main.c` file. The same behavior is present even if the program does execute and monitors a directory, but gets terminated by using (`ctrl-C` or `kill`).

B. Impact

The failure to clean up the shared memory segment in the event of an error has the following consequences:

- **Resource leak**: The shared memory segment remains allocated in the system, consuming resources until it is manually cleaned up or the system reboots.
- **Security risk**: If the program is run repeatedly with invalid directory names, multiple shared memory segments may accumulate, potentially exhausting system resources. Although in the examined case, the executable program always uses the same shared memory key.
- **System administration overhead**: System administrators must manually identify and clean up orphaned shared memory segments using tools like `ipcrm`.

C. Recommendation

To address this issue, the program should execute proper cleanup in all error scenarios, including when `opendir` fails. For example:

- Inclusion of cleanup code after the `opendir` failure check:

```
__dirp = opendir(dirname);
if (__dirp == NULL) {
    perror("opendir");
    // Clean up shared memory segment
    shmdt(smaddr);
    shmctl(shmid, IPC_RMID, NULL);
    return;
}
```

- Use of a structured approaches of resource management, such as wrapping shared memory allocation and cleanup in dedicated functions.
- Consideration of using a signal handler to ensure cleanup is performed even if the program terminates unexpectedly.

By implementing these countermeasures, the program can avoid shared memory leaks and ensure proper resource management in all scenarios.

IX. Vulnerability 8: Double free error when deleting the monitored directory

A. Technical description

The vulnerable code monitors a directory in an infinite loop, dynamically allocating memory for directory entries. When the target directory is deleted during runtime, the program fails to handle the cleanup correctly, leading to a double-free error.

B. Impact

- Leads into double-free error, which corrupts heap metadata (glibc's `malloc/free` structures).
- Lead into crashes(in the examined case).

C. Proof of Concept

1) Run the program targeting a directory.
2) Delete the monitored directory while the program is running.
3) Observation of the crash (`free(): double free detected`).

D. Recommendation

In order to avoid this vulnerability as noted in earlier sections, all the pointers freed should be set to NULL in order to avoid these issues.

X. Vulnerability 9: Symlink testing

A. Technical Description

The program monitors a target directory but it does not perform any validation whether target path is symbolic link before processing it. This could allow attackers to redirect directory scans to unintended locations (e.g., `/etc` or `/home/ubuntu`) by replacing a monitored directory with a symlink during runtime.

B. Testing Methodology

1) Created a symlink from the monitored directory to restricted locations:

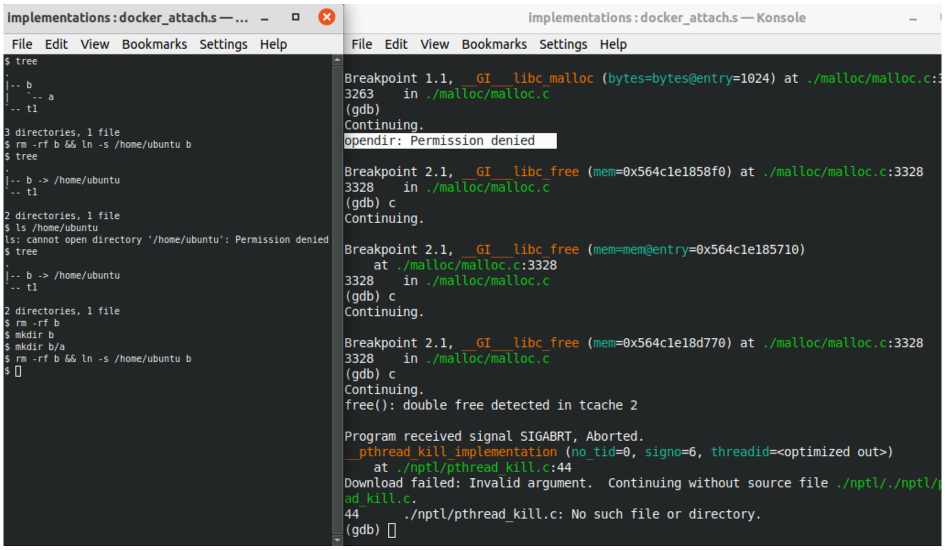
```
        rm -rf b && ln -s /home/ubuntu b  # a root directory
        rm -rf b && ln -s /etc b      # a world-readable directory
```

2) Executed the program as a non-root user:

```
        ./monitor b/
```

C. Findings

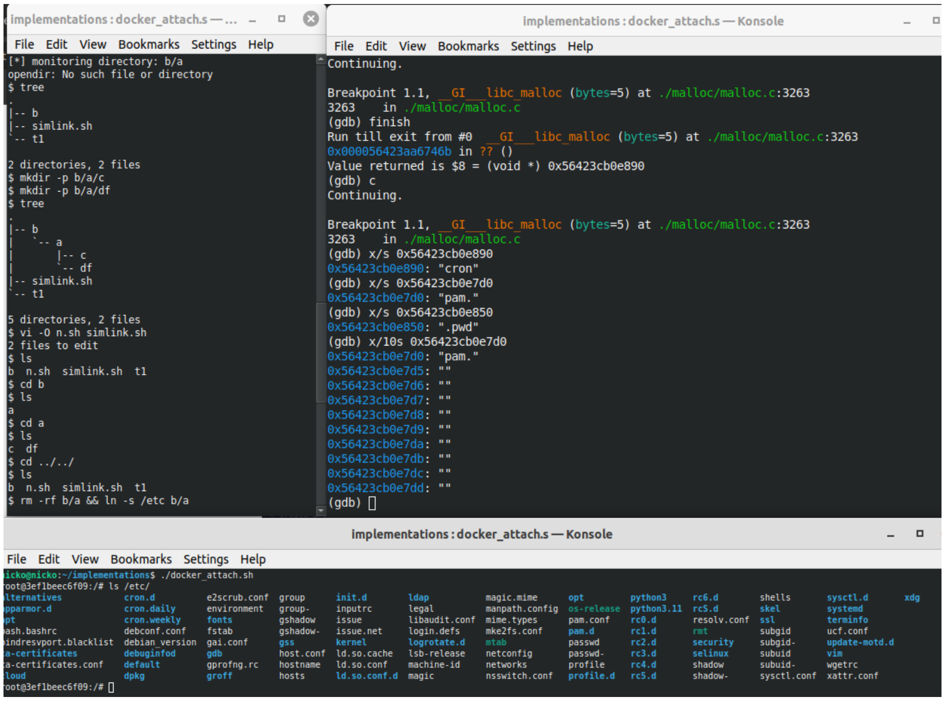
- **Restricted directory (`/home/ubuntu`)**:
  - Program returned `opendir: Permission denied`.
  - Confirmed it respects the filesystem permissions (shown safe behavior).
- **World readable directory (`/etc`)**:
  - Program listed contents of `/etc` (e.g., `cron`, `pam.d`).
  - **Vulnerability**: Successfully followed symlinks to unintended paths.


implementations : docker_attach.s — ...
File Edit View Bookmarks Settings Help
implementations : docker_attach.s — Konsole
File Edit View Bookmarks Settings Help

```
$ tree
.
|-- b
|   `-- a
`-- t1

3 directories, 1 file
$ rm -rf b && ln -s /home/ubuntu b
$ tree
.
|-- b -> /home/ubuntu
`-- t1

2 directories, 1 file
$ ls /home/ubuntu
ls: cannot open directory '/home/ubuntu': Permission denied
$ tree
.
|-- b -> /home/ubuntu
`-- t1

2 directories, 1 file
$ rm -rf b
$ mkdir b
$ mkdir b/a
$ rm -rf b && ln -s /home/ubuntu b
$ 
```

```
Breakpoint 1.1, __GI___libc_malloc (bytes=bytes@entry=1024) at ./malloc/malloc.c:3
3263    in ./malloc/malloc.c
(gdb)
Continuing.
opendir: Permission denied

Breakpoint 2.1, __GI___libc_free (mem=0x564c1e1858f0) at ./malloc/malloc.c:3328
3328    in ./malloc/malloc.c
(gdb) c
Continuing.

Breakpoint 2.1, __GI___libc_free (mem=mem@entry=0x564c1e185710)
    at ./malloc/malloc.c:3328
3328    in ./malloc/malloc.c
(gdb) c
Continuing.

Breakpoint 2.1, __GI___libc_free (mem=0x564c1e18d770) at ./malloc/malloc.c:3328
3328    in ./malloc/malloc.c
(gdb) c
Continuing.
free(): double free detected in tcache 2

Program received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=<optimized out>)
    at ./nptl/pthread_kill.c:44
Download failed: Invalid argument.  Continuing without source file ./nptl/./nptl/p
ad_kill.c.
44      ./nptl/pthread_kill.c: No such file or directory.
(gdb) 
```

FIGURE 6. Following a simlink as root.


implementations : docker_attach.s — ...
File Edit View Bookmarks Settings Help
implementations : docker_attach.s — Konsole
File Edit View Bookmarks Settings Help
implementations : docker_attach.s — Konsole
File Edit View Bookmarks Settings Help

```
[*] monitoring directory: b/a
opendir: No such file or directory
$ tree
.
|-- b
|-- simlink.sh
`-- t1

2 directories, 2 files
$ mkdir -p b/a/c
$ mkdir -p b/a/df
$ tree
.
|-- b
|   `-- a
|       |-- c
|       `-- df
|-- simlink.sh
`-- t1

5 directories, 2 files
$ vi -O n.sh simlink.sh
2 files to edit
$ ls
b  n.sh  simlink.sh  t1
$ cd b
$ ls
a
$ cd a
$ ls
c  df
$ cd ../../
$ ls
b  n.sh  simlink.sh  t1
$ rm -rf b/a && ln -s /etc b/a
```

```
Continuing.

Breakpoint 1.1, __GI___libc_malloc (bytes=5) at ./malloc/malloc.c:3263
3263    in ./malloc/malloc.c
(gdb) finish
Run till exit from #0  __GI___libc_malloc (bytes=5) at ./malloc/malloc.c:3263
0x000056423aa6746b in ?? ()
Value returned is $8 = (void *) 0x56423cb0e890
(gdb) c
Continuing.

Breakpoint 1.1, __GI___libc_malloc (bytes=5) at ./malloc/malloc.c:3263
3263    in ./malloc/malloc.c
(gdb) x/s 0x56423cb0e890
0x56423cb0e890: "cron"
(gdb) x/s 0x56423cb0e7d0
0x56423cb0e7d0: "pam."
(gdb) x/s 0x56423cb0e850
0x56423cb0e850: ".pwd"
(gdb) x/10s 0x56423cb0e7d0
0x56423cb0e7d0: "pam."
0x56423cb0e7d5: ""
0x56423cb0e7d6: ""
0x56423cb0e7d7: ""
0x56423cb0e7d8: ""
0x56423cb0e7d9: ""
0x56423cb0e7da: ""
0x56423cb0e7db: ""
0x56423cb0e7dc: ""
0x56423cb0e7dd: ""
(gdb) 
```

```
icko@nicko:~/implementations$ ./docker_attach.sh
oot@3ef1beec6f09:/# ls /etc/
lternatives           cron.d         e2scrub.conf  group      init.d        ldap          magic.mime      opt         python3      rc6.d        shells       sysctl.d          xdg
pparmor.d             cron.daily     environment   group-     inputrc       legal         manpath.config  os-release  python3.11   rcS.d        skel         systemd
pt                    cron.weekly    fonts         gshadow    issue         libaudit.conf mime.types      pam.conf    rc0.d        resolv.conf  ssl          terminfo
ash.bashrc            debconf.conf   fstab         gshadow-   issue.net     login.defs    mke2fs.conf     pam.d       rc1.d        rmt          subgid       ucf.conf
indresvport.blacklist debian_version gai.conf      gss        kernel        logrotate.d   mtab            passwd      rc2.d        security     subgid-      update-motd.d
a-certificates        debuginfod     gdb           host.conf  ld.so.cache   lsb-release   netconfig       passwd-     rc3.d        selinux      subuid       vim
a-certificates.conf   default        gprofng.rc    hostname   ld.so.conf    machine-id    networks        profile     rc4.d        shadow       subuid-      wgetrc
loud                  dpkg           groff         hosts      ld.so.conf.d  magic         nsswitch.conf   profile.d   rc5.d        shadow-      sysctl.conf  xattr.conf
oot@3ef1beec6f09:/# 
```

FIGURE 7. Following a simlink as a normal user.

D. Impact

Scenario	Risk
Symlink to /etc	Information disclosure (configs, user accounts)
Symlink to /root (as root)	Privilege escalation

E. Proof of Concept (PoC)

```
# Step 1: Redirect monitored directory to /etc
ln -sf /etc b

# Step 2: Run program (observe /etc contents in memory)
./monitor b/
```

GDB output:

```
(gdb) x/s 0x56423cb0e890
0x56423cb0e890: "cron"  # From /etc/cron.d/
```

F. Recommendations

- Use O_NOFOLLOW when opening directories:

```
    int fd = open(dirpath, O_RDONLY | O_NOFOLLOW | O_DIRECTORY);
    if (fd < 0) { /* Handle error */ }
    DIR *dirp = fdopendir(fd);
```

- Add post-readdir() validation:

```
    struct stat st;
    if (fstat(dirfd(dirp), &st) == -1 || !S_ISDIR(st.st_mode)) {
        /* Directory was replaced with symlink! */
    }
```

XI. Observed vulnerabilities

The vulnerabilities observed so far were triggered by:

- writing malicious data to shared memory which program reads in each loop execution in monitor.c and allocates memory from the reading.
- the double-free memory deallocation, which occured by a programming mistake and depending on the addresses of the allocated memory triggers after deletion of folders on the monitoring directory.
- memory leak during the first iteration of the main monitoring loop if there are present directories inside the monitored folder. The memory leakage happens to be proportional to the number of directories that exist on the monitoring folder.
- an unchecked integer overflow (mentioned in the previous section).
- the infinite loop of the main monitoring activity without exit condition.
- the utilized shared memory segment is lacking of initial configuration related to the application code.
- the shared memory allocated segment is not destroyed after program's termination, consisting a shared memory leak.
- double-free error occurance when the monitored directory is deleted. Not present error handling logic for edge cases.
- program follows symbolic links when opening monitored directories, allowing redirect directory scans to unintended locations. This can lead to information disclosure or privilege escalation if executed as root.

XII. Comments and observations

A. Improper handling of malloc() return values

In the source file monitor.c, specifically at line 24, the function malloc() is invoked without verifying its return value. According to the Linux manual page, malloc() returns NULL on failure, which can occur due to insufficient memory or other system-level errors. While `malloc()` may also return `NULL` for a successful allocation with a size of zero, this is not applicable in this context, as the function is called with a non-zero argument.

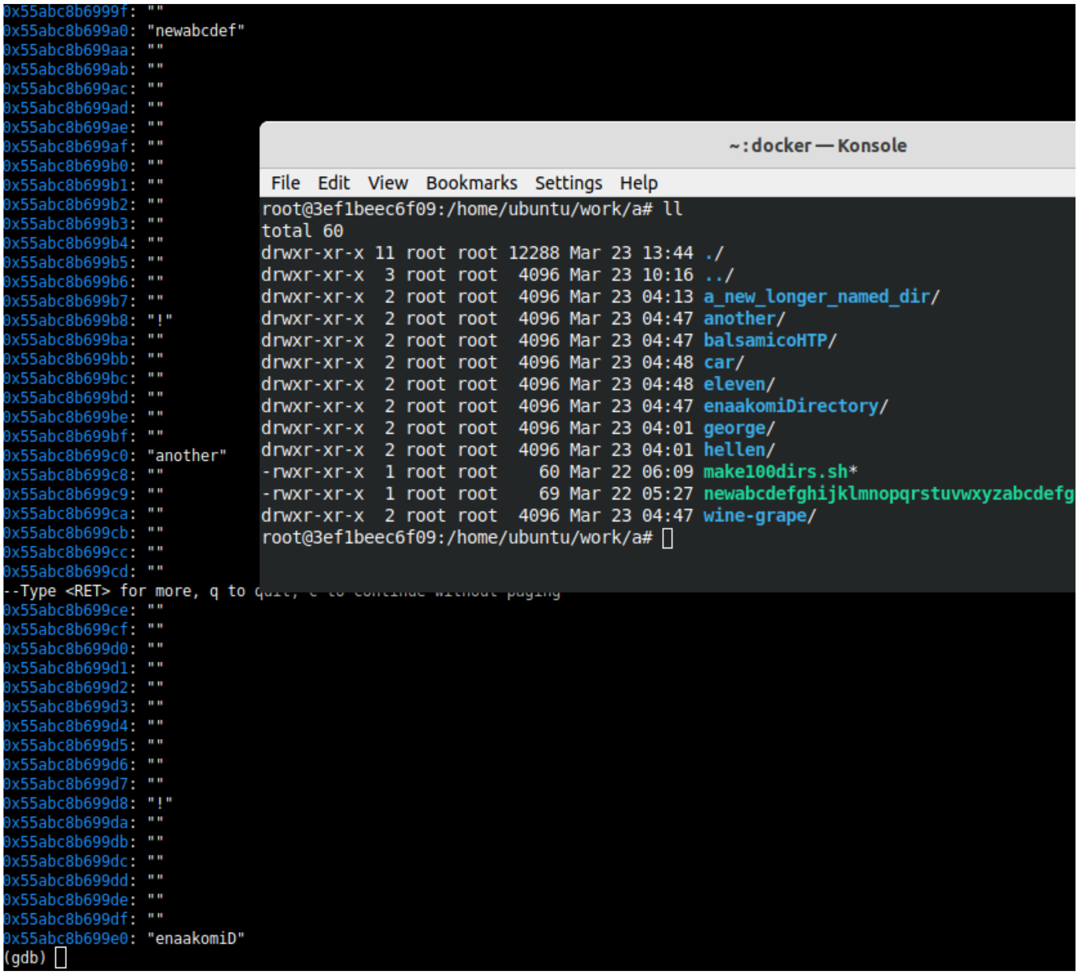
The absence of proper error handling for `malloc()` is a critical oversight, as it can lead to **undefined behavior** if the program attempts to dereference a `NULL` pointer. This behavior is not isolated to this instance; all calls to `malloc()` in the codebase exhibit the same lack of return value validation, indicating a systemic issue in the program's memory management practices. Error checking also should be placed in the return value of `opendir` and appropiate handling to be implemented to complement the functionality of the monitoring program. In general, all the return values of any interaction with the underlying system should be verified before letting the program continue further its execution, something that the executable is not performing.

B. Absence of `errno` usage in error handling

The failure handling logic in the code does not utilize the `errno` variable, which is a standard mechanism in C for identifying the cause of errors. By neglecting to check `errno`, the program forfeits valuable diagnostic information that could aid in debugging and resolving runtime issues. For example, in the event of a failed system call (e.g., `opendir()`), the program would be unable to distinguish between different failure modes, such as permission errors or invalid directory paths.

C. Truncation of directory names

Another notable observation relates to the handling of directory names. In `monitor.c`, the program copies the directory name into dynamically allocated memory but limits the copy to `(init_smaddr - 1)` characters (line 42). This truncation behavior means that if the directory name exceeds `(init_smaddr - 1)` characters, the full name will not be stored. While the program does not appear to rely on the full directory name for its functionality, this design choice could lead to subtle bugs or misinterpretations in scenarios where the complete path is required. This behavior is not explicitly addressed in the code, and while it does not currently impact the program's operation.



```
0x55abc8b6999f: ""
0x55abc8b699a0: "newabcdef"
0x55abc8b699aa: ""
0x55abc8b699ab: ""
0x55abc8b699ac: ""
0x55abc8b699ad: ""
0x55abc8b699ae: ""
0x55abc8b699af: ""
0x55abc8b699b0: ""
0x55abc8b699b1: ""
0x55abc8b699b2: ""
0x55abc8b699b3: ""
0x55abc8b699b4: ""
0x55abc8b699b5: ""
0x55abc8b699b6: ""
0x55abc8b699b7: ""
0x55abc8b699b8: "!"
0x55abc8b699ba: ""
0x55abc8b699bb: ""
0x55abc8b699bc: ""
0x55abc8b699bd: ""
0x55abc8b699be: ""
0x55abc8b699bf: ""
0x55abc8b699c0: "another"
0x55abc8b699c8: ""
0x55abc8b699c9: ""
0x55abc8b699ca: ""
0x55abc8b699cb: ""
0x55abc8b699cc: ""
0x55abc8b699cd: ""
--Type <RET> for more, q to q
0x55abc8b699ce: ""
0x55abc8b699cf: ""
0x55abc8b699d0: ""
0x55abc8b699d1: ""
0x55abc8b699d2: ""
0x55abc8b699d3: ""
0x55abc8b699d4: ""
0x55abc8b699d5: ""
0x55abc8b699d6: ""
0x55abc8b699d7: ""
0x55abc8b699d8: "!"
0x55abc8b699da: ""
0x55abc8b699db: ""
0x55abc8b699dc: ""
0x55abc8b699dd: ""
0x55abc8b699de: ""
0x55abc8b699df: ""
0x55abc8b699e0: "enaakomiD"
(gdb) 
```

```
root@3ef1beec6f09:/home/ubuntu/work/a# ll
total 60
drwxr-xr-x 11 root root 12288 Mar 23 13:44 ./
drwxr-xr-x  3 root root  4096 Mar 23 10:16 ../
drwxr-xr-x  2 root root  4096 Mar 23 04:13 a_new_longer_named_dir/
drwxr-xr-x  2 root root  4096 Mar 23 04:47 another/
drwxr-xr-x  2 root root  4096 Mar 23 04:47 balsamicoHTP/
drwxr-xr-x  2 root root  4096 Mar 23 04:48 car/
drwxr-xr-x  2 root root  4096 Mar 23 04:48 eleven/
drwxr-xr-x  2 root root  4096 Mar 23 04:47 enaakomiDirectory/
drwxr-xr-x  2 root root  4096 Mar 23 04:01 george/
drwxr-xr-x  2 root root  4096 Mar 23 04:01 hellen/
-rwxr-xr-x  1 root root    60 Mar 22 06:09 make100dirs.sh*
-rwxr-xr-x  1 root root    69 Mar 22 05:27 newabcdefghijklmnopqrstuvwxyzabcdefg
drwxr-xr-x  2 root root  4096 Mar 23 04:47 wine-grape/
root@3ef1beec6f09:/home/ubuntu/work/a# 
```

FIGURE 8. GDB examination of truncation of directory names when *init_smaddr* was set to 10.

XIII. References

- Ghidra - https://ghidra-sre.org/
- Valgrind - http://valgrind.org/
- GDB - https://www.gnu.org/software/gdb/

Appendix

Listing 3. **monitor.c**

```c
void monitor(char *dirname,int *smaddr)
{
  int iVar1;
  DIR *__dirp;
  void *oti_na_nai;
  dirent *pElements;
  int local_4c;
  int tmp;
  int tmp2;
  int local_40;
  int local_3c;
  int local_38;
  void *new_pointer;
  int init_smaddr;

  new_pointer = (void *)0x0;
  local_4c = 0;
  tmp = 0;
  printf("[*]␣monitoring␣directory:␣\percent␣s\n",dirname);
  while( true ) {
    init_smaddr = *smaddr;
    if (tmp == 0) {
      tmp = 10;
      new_pointer = malloc(0x50);
    }
    __dirp = opendir(dirname);
    if (__dirp == (DIR *)0x0) break;
    tmp2 = 0;
    while( true ) {
      pElements = readdir(__dirp);
      if (pElements == (dirent *)0x0) break;
      iVar1 = strcmp(pElements->d_name,".");
      if (iVar1 != 0) {
        iVar1 = strcmp(pElements->d_name,"..");
        if (iVar1 != 0) {
          if (tmp <= tmp2) {
            tmp = tmp << 1;
            new_pointer = realloc(new_pointer,(long)tmp * 8);
          }
          oti_na_nai = malloc((long)init_smaddr);
          *(void **)((long)tmp2 * 8 + (long)new_pointer) = oti_na_nai;
          strncpy(*(char **)((long)new_pointer + (long)tmp2 * 8),pElements->d_name,
                  (long)(init_smaddr + -1));
          *(undefined1 *)(*(long *)((long)new_pointer + (long)tmp2 * 8) + (long)init_smaddr + -1) = 0;
          tmp2 = tmp2 + 1;
        }
      }
    }
    closedir(__dirp);
    if (tmp2 == local_4c) {
      for (local_3c = 0; local_3c < tmp2; local_3c = local_3c + 1) {
        free(*(void **)((long)new_pointer + (long)local_3c * 8));
      }
    }
    else {
      puts("[!]␣directory␣change␣detected!");
      for (local_40 = 0; local_40 < local_4c; local_40 = local_40 + 1) {
        free(*(void **)((long)new_pointer + (long)local_40 * 8));
      }
      local_4c = tmp2;
    }
    sleep(5);
  }
  perror("opendir");
  for (local_38 = 0; local_38 < local_4c; local_38 = local_38 + 1) {
    free(*(void **)((long)new_pointer + (long)local_38 * 8));
  }
  free(new_pointer);
  return;
}
```

Listing 4. main.c

```
undefined8 main(int param_1,undefined8 *param_2)

{
  int shmem_id;
  void *__shmaddr;

  if (param_1 != 2) {
    printf("[*]␣usage:␣\percent␣s␣<directory>\n",*param_2);
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  shmem_id = shmget(0x6676,0x104,0x3b6);
  if (shmem_id == -1) {
    perror("shmget");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  __shmaddr = shmat(shmem_id,(void *)0x0,0);
  if (__shmaddr == (void *)0xffffffffffffffff) {
    perror("shmat");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  monitor(param_2[1],__shmaddr);
  shmem_id = shmdt(__shmaddr);
  if (shmem_id == -1) {
    perror("shmdt");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  return 0;
}
```