

Investigation: Coverity CID 1668652

Summary

Field	Details
Coverity CID	1668652
Checker	UNINIT
Category	Uninitialized data
Reported function	<code>f_rename()</code>
Reported file	<code>/cpukit/libfs/src/fatfs/ff.c</code>
Reported operation	<code>memcpy(&djn, &djo, sizeof djo)</code>
Reported field	<code>djo.obj.id</code>
Current file	<code>contrib/cpukit/fatfs/ff.c</code>
FatFs version	R0.16
Classification	Coverity false positive
RTEMS-specific?	No
Upstream FatFs defect?	No

1. Coverity Finding

Coverity reports that `djo.obj.id` is uninitialized when the directory object `djo` is copied to `djn` in `f_rename()`:

```
** CID 1668652:      Uninitialized variables  (UNINIT)
/cpukit/libfs/src/fatfs/ff.c:5205 in f_rename()

memcpy(&djn, &djo, sizeof djo);
Using uninitialized value "djo". Field "djo.obj.id" is uninitialized when
calling "memcpy".
...
/cpukit/libfs/src/fatfs/ff.c:5227 in f_rename()
memcpy(&djn, &djo, sizeof (DIR));
```

The question is whether the uninitialized `id` field is actually consumed on the execution path of `f_rename()`.

2. Locate the Current `f_rename()`

The Coverity report uses the pre-refactoring path/line numbers. The current source is:

```
contrib/cpukit/fatfs/ff.c
```

The current function begins at line 5261.

The relevant source was obtained with:

```
grep -n -A80 -B5 "^FRESULT f_rename" contrib/cpukit/fatfs/ff.c
```

Relevant output:

```
5261:FRESULT f_rename (  
5262:  const TCHAR* path_old,  
5263:  const TCHAR* path_new  
5264:)  
5265:{  
5266:  FRESULT res;  
5267:  FATFS *fs;  
5268:  DIR djo, djn;  
...  
5273:  get_ldnumber(&path_new);      /* Snip the drive number of new name  
off */  
5274:  res = mount_volume(&path_old, &fs, FA_WRITE);  
5275:  if (res == FR_OK) {  
5276:      djo.obj.fs = fs;  
5277:      INIT_NAMEBUFF(fs);  
5278:      res = follow_path(&djo, path_old);  
...  
5302:      memcpy(&djn, &djo, sizeof djn);  
5303:      res = follow_path(&djn, path_new);  
...  
5327:      memcpy(buf, djo.dir, SZDIRE);  
5328:      memcpy(&djn, &djo, sizeof djn);  
5329:      res = follow_path(&djn, path_new);
```

There are two relevant copies: one in the exFAT path and one in the FAT/FAT32 path.

3. What Does `djo.obj` Contain?

`djo` is declared as:

```
DIR djo, djn;
```

Before `follow_path()` is called, `f_rename()` explicitly initializes:

```
djo.obj.fs = fs;
```

Then:

```
res = follow_path(&djo, path_old);
```

The `follow_path()` function initializes and updates the directory object as it traverses the path.

The Coverity warning specifically identifies:

```
djo.obj.id
```

as potentially uninitialized.

To determine whether that matters, the next step is to locate every use of `obj.id`.

4. Search for Uses of `obj.id`

The following search was performed:

```
grep -n -E "obj\.id|->id" contrib/cpukit/fatfs/ff.c
```

The relevant output was:

```
3665: fs->id = ++Fsid;  
3703: if (obj && obj->fs && obj->fs->fs_type && obj->id == obj->fs->id)  
3940: fp->obj.id = fs->id;  
4752: dp->obj.id = fs->id;
```

There are no references to:

```
djo.obj.id  
djn.obj.id
```

in `f_rename()`.

This means the fact that `memcpy()` copies the field does not by itself establish that an uninitialized value is subsequently used.

5. Inspect the Function That Uses `obj->id`

The search shows that `obj->id` is used by the `validate()` function.

The relevant code is:

```
if (obj && obj->fs && obj->fs->fs_type &&
    obj->id == obj->fs->id)
```

This is a consistency check used when validating an object.

Therefore, if `djo.obj.id` were later passed to `validate()`, the Coverity finding could potentially represent a real problem.

The critical question is whether the `f_rename()` path invokes `validate()` for `djo` or `djn`.

6. Inspect `follow_path()`

The relevant function was inspected with:

```
grep -n -A120 -B10 "^static FRESULT follow_path" contrib/cpukit/fatfs/ff.c
```

The function begins:

```
static FRESULT follow_path (
    DIR* dp,
    const TCHAR* path
)
{
    FRESULT res;
    BYTE ns;
    FATFS *fs = dp->obj.fs;
```

It then:

```
dp->obj.sclust = fs->cdir;
```

or initializes the object to the root directory, followed by path traversal through:

```
res = create_name(dp, &path);
```

and:

```
res = dir_find(dp);
```

The function finishes with:

```
return res;
```

There is no call to:

```
validate()
```

inside `follow_path()`.

This is important because `f_rename()` copies `djo` and then immediately passes `djn` to `follow_path()`.

7. Inspect the Relevant `follow_path()` Initialization

The current implementation contains:

```
#if FF_FS_EXFAT
    dp->obj.n_frag = 0;
    ...
#endif
```

and initializes the starting cluster:

```
dp->obj.sclust = fs->cdir;
```

or:

```
dp->obj.sclust = 0;
```

For exFAT, additional allocation metadata is initialized when required.

There is no requirement in `follow_path()` for:

```
dp->obj.id
```

because this field is not used by the function.

Thus `djo.obj.id` being uninitialized does not affect the path traversal performed by `f_rename()`.

8. Inspect `f_rename()` After the Copy

The exFAT branch performs:

```
memcpy(&djn, &djo, sizeof djn);  
res = follow_path(&djn, path_new);
```

The FAT/FAT32 branch similarly performs:

```
memcpy(&djn, &djo, sizeof djn);  
res = follow_path(&djn, path_new);
```

Afterward, the fields actually used by `f_rename()` include:

```
djn.obj.sclust  
djo.obj.sclust  
djn.dptr  
djo.dptr  
djn.dir  
djo.dir
```

For example:

```
res = (djn.obj.sclust == djo.obj.sclust &&  
       djn.dptr == djo.dptr) ? FR_NO_FILE : FR_EXIST;
```

There is no corresponding use of:

```
djn.obj.id
```

or:

```
djo.obj.id
```

in this logic.

9. Check How `id` Is Normally Initialized

The earlier search showed:

```
3940: fp->obj.id = fs->id;  
4752: dp->obj.id = fs->id;
```

This demonstrates that FatFs does initialize the `id` field for object types where the field is subsequently required.

For example, a file object receives:

```
fp->obj.id = fs->id;
```

and a directory object can receive:

```
dp->obj.id = fs->id;
```

The fact that `djo.obj.id` is not initialized in this particular path therefore does not imply that all uses of `FFOBJID.id` are unsafe.

It is specifically necessary to determine whether this directory object's `id` is consumed.

The source analysis above shows that it is not consumed on the `f_rename()` path.

10. Why `memcpy()` Does Not Automatically Make This a Defect

The Coverity report points to:

```
memcpy(&djn, &djo, sizeof djn);
```

Copying an object representation can copy bytes whose values were never initialized.

However, an uninitialized byte becoming part of the destination object is not by itself sufficient to establish a functional defect.

The relevant question is:

Is the copied indeterminate field subsequently read in a way that affects program behavior?

For this finding:

```

djo.obj.id
|
| memcpy()
v
djn.obj.id
|
X
not consumed on f_rename() path

```

The field is copied, but no later operation in the relevant path reads it.

The fields that `f_rename()` actually relies on are independently established and used.

11. Why `validate()` Does Not Change the Conclusion

`validate()` contains the `obj->id` comparison:

```
obj->id == obj->fs->id
```

Therefore it is the natural candidate for a possible downstream use.

However, the `f_rename()` execution path examined here is:

```

f_rename()
|
+-- djo.obj.fs = fs
|
+-- follow_path(&djo, path_old)
|
+-- memcpy(&djn, &djo, ...)
|
+-- follow_path(&djn, path_new)
|
+-- use sclust/dptr/dir

```

Neither `follow_path()` nor the relevant `f_rename()` logic invokes `validate()`.

Therefore there is no established read of the uninitialized `id` field along the reported path.

12. Classification

Is this an RTEMS issue?

No.

The finding concerns the implementation of the imported FatFs core in:

```
contrib/cpukit/fatfs/ff.c
```

There is no RTEMS-specific wrapper involved.

Is this an upstream FatFs issue?

No actionable defect has been established.

The field is copied but is not consumed on this execution path.

Is Coverity wrong?

Yes, in the sense relevant to this CID.

Coverity identifies the uninitialized field correctly at the copy operation, but the reported condition does not establish a harmful use. The static analyzer is treating the copied uninitialized field as significant without accounting for the fact that `djo.obj.id` / `djn.obj.id` are not subsequently read on this path.

13. Conclusion

CID 1668652 is a Coverity false positive.

`djo.obj.id` is not explicitly initialized before:

```
memcpy(&djn, &djo, sizeof djn);
```

and Coverity therefore reports an `UNINIT` condition.

However, source inspection shows that:

1. `f_rename()` initializes `djo.obj.fs`.
2. `follow_path()` does not use `obj.id`.
3. `follow_path()` does not invoke `validate()`.
4. The subsequent `f_rename()` logic does not read `djo.obj.id` or `djn.obj.id`.
5. The fields actually required by the rename operation, such as `sclust`, `dptr`, and `dir`, are handled independently.
6. `obj.id` is initialized in other FatFs object paths where it is actually required.

Therefore, copying the uninitialized `id` field does not establish an erroneous read or observable incorrect behavior on the reported `f_rename()` path.

Final classification

CID 1668652 — Coverity false positive (UNINIT). The `id` field is copied without initialization, but it is not subsequently consumed on the `f_rename()` path. No RTEMS fix and no upstream

FatFs fix are required.