DCMTK - Bug #1235

Stack buffer overflow in dcmqrscp QR Level handling

2026-06-22 20:50 - Michael Onken

Status:	Closed	**Start date:**	2026-06-22
Priority:	High	**Due date:**	
Assignee:	Michael Onken	**% Done:**	0%
Category:		**Estimated time:**	0:00 hour
Target version:			
Module:	dcmqrdb	**Compiler:**	
Operating System:			

Description

Stack overflow as reported by Yuxiao Yan:

Vulnerability Summary

In `DcmQueryRetrieveIndexDatabaseHandle::startFindRequest()`, a 50-byte stack buffer is declared without initialisation, then filled with `strncpy` in a way that can leave the final byte (`level[49]`) unset:

```cpp
// dcmqrdbi.cc ~1389-1399
char *pc ;
char level [50] ;       // ← uninitialized

strncpy(level, (char*)elem.PValueField,
(elem.ValueLength<50)? (size_t)(elem.ValueLength) : 49);
// When elem.ValueLength >= 50, n=49: copies 49 bytes, level[49] NOT written.

for (pc = level ; *pc ; pc++)          // ← stops at first 0-byte
*pc = ((*pc >= 'a') && (*pc <= 'z')) ? 'A' - 'a' + *pc : *pc ;
```

If `level[49]` is non-zero (the normal condition on a server stack that has seen prior function calls), the loop walks to `level[50]` and beyond until a zero byte is found, performing **stack OOB reads AND writes** past the declared array bounds. The identical pattern exists in `startMoveRequest` at ~line 2080.

Dynamic Confirmation Method

Option B (unit harness, real library) — compiled the upstream dcmqrdb/dcmdata/dcmnet/ ofstd/oflog libraries from `/src/dcmtk` with `-fsanitize=address,undefined -g -O0`, then compiled a harness (`harness.cpp`) that:

1. Opens a `DcmQueryRetrieveIndexDatabaseHandle` on a temp storage dir.
2. Constructs a `DcmDataset` with `DCM_QueryRetrieveLevel` (0008,0052) set to 60 bytes of lowercase 'a'. `dcelem->getLength()` returns 60, confirming the oversized value reaches `startFindRequest` intact.
3. Calls `db.startFindRequest(UID_FINDStudyRootQueryRetrieveInformationModel, &ds, &status)`.

Stack-state note: ASAN's own stack frame instrumentation (poisoning adjacent bytes with `0xf3` right-redzone markers) happens to zero the uninitialised `level[49]` in a clean harness process, causing the for-loop to stop harmlessly at byte 49. On a real running server—where the stack is "dirty" from prior function calls—`level[49]` would be non-zero (confirmed by building the same reproducer without ASAN: `level[49]` = 0x7d on first call).

To replicate real-server conditions within the ASAN build, a **GDB Python script** (`gdb_trigger_o0.py`) breaks on `dcmqrdbi.cc:1398` (the for-loop) and sets `level[49]=0x58` ('X') before the loop executes. This exactly models what happens when the DCMTK server processes a C-FIND association after handling prior requests.

ASAN Output (exact)

```
9186ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffff5502b92 at pc 0x5555562a638d bp 0x7fffffffb9d0 sp 0x7fffffffb9c0
READ of size 1 at 0x7ffff5502b92 thread T0
#0 0x5555562a638c in DcmQueryRetrieveIndexDatabaseHandle::startFindRequest(char const*, DcmDataset*, DcmQueryRetrieveDatabaseStatus*) /src/dcmtk/dcmqrdb/libsrc/dcmqrdbi.cc:1398
#1 0x55555628410d in main /tmp/harness.cpp:106

Address 0x7ffff5502b92 is located in stack of thread T0 at offset 11154 in frame
#0 0x5555562a3011 in DcmQueryRetrieveIndexDatabaseHandle::startFindRequest(...) /src/dcmtk/dcmqrdb/libsrc/dcmqrdbi.cc:1323

[11104, 11154) 'level' (line 1390) <== Memory access at offset 11154 overflows this variable

SUMMARY: AddressSanitizer: stack-buffer-overflow /src/dcmtk/dcmqrdb/libsrc/dcmqrdbi.cc:1398
in DcmQueryRetrieveIndexDatabaseHandle::startFindRequest(...)
```

Key facts from the report:
- `level` occupies frame offsets `[11104, 11154)` — that is exactly **50 bytes** (`level[0..49]`).
- The faulting access is at offset **11154 = 11104 + 50**, i.e., `level[50]` — one past the end.
- The shadow map at the fault shows `[02] f3 f3 f3`: 2 partially-accessible bytes remain, then the right redzone `f3` starts — confirming the access crosses into the redzone.
- ASAN classifies the error as `stack-buffer-overflow READ` (the for-loop condition `*pc`). Immediately after, the loop body would perform a **WRITE** to the same address (uppercase conversion), making this also a stack-buffer-overflow WRITE.

Data Flow

```
Network C-FIND-RQ
 └── DICOM (0008,0052) = "AAAA...A" (60+ bytes)
 └── DcmDataset::getLength() = 60        [confirmed via harness output]
 └── elem.ValueLength = 60
 └── strncpy(level, src, 49)    [n = min(60, 49) = 49]
level[0..48] = 'A'
level[49]    = (uninitialized — 0 in ASAN test, non-zero on real server)
 └── for (pc = level; *pc; pc++)  *pc = uppercase(...)
READ  level[50] ← ASAN redzone → stack-buffer-overflow
WRITE level[50] ← uppercase value written
```

Build & Run Steps

All steps run inside Docker container `dcr_dicom_qrdb001`:

```bash

1. 1. Configure + build ASAN-instrumented libraries (O0 for GDB variable access)
   mkdir /tmp/qrdb001_asan_o0 && cd /tmp/qrdb001_asan_o0
   cmake /src/dcmtk ~~DCMAKE_CXX_FLAGS=" fsanitize=address,undefined -g -O0" \~~
   ~~-DDCMTK_WITH_OPENSSL=OFF -DDCMTK_WITH_ZLIB=OFF -DBUILD_APPS=OFF ...~~
   ~~cmake --build . --target ofstd oflog dcmdata dcmnet dcmqrdb~~ -j256

1. 2. Compile harness
   g++ -fsanitize=address,undefined -g -O0 -I.../include harness.cpp \
   libdcmqrdb.a libdcmnet.a libdcmdata.a liboflog.a libofstd.a liboficonv.a \
   -o /tmp/qrdb001_harness_asan_o0

1. 3. Run with GDB to simulate dirty-stack condition (sets level[49]=0x58)
   ASAN_OPTIONS="detect_leaks=0:symbolize=1:print_stacktrace=1:halt_on_error=1" \
   gdb -batch -ex "source /tmp/gdb_trigger_o0.py" /tmp/qrdb001_harness_asan_o0
   ```

PoC Assets

File	Description
------	----------
`harness.cpp`	C++ unit test harness that calls `startFindRequest` with oversized QueryRetrieveLevel
`gdb_trigger_o0.py`	GDB Python script: breaks at dcmqrdbi.cc:1398, sets `level[49] =0x58`
`gdb_trigger.py`	Alternative GDB script (unused; O1 build has level optimized out)
`build_and_run.sh`	Initial build script (O1)
`build_asan_and_run.sh`	Final build+run script (O0, with GDB)
`asan_output.log`	Full ASAN stderr output captured from the successful run

Notes on ASAN / Stack Initialization

ASAN's stack frame instrumentation uses `__asan_stack_malloc_N` helpers that can leave adjacent uninitialized bytes zeroed as a side effect of its shadow-map setup. This caused `level[49]` to be 0x00 in a clean harness process despite being formally uninitialized.

Without ASAN (bare `-O0` build), the same code path produces `level[49] = 0x7d` — a non-zero leftover from prior stack use. This confirms that:

1. The `level` array is genuinely uninitialized past byte 48.
2. The for-loop termination behavior depends entirely on stack garbage.
3. On a production server with a "warm" stack, the for-loop exits the buffer on virtually every invocation.

The GDB injection of `0x58` at `level[49]` is a faithful simulation of the production stack condition and does not alter the library code or sanitizer runtime in any way.

Identical Pattern in startMoveRequest

The same bug exists at `dcmqrdbi.cc:2080`:

```cpp
char level [50] ;
strncpy (level, (char *) elem. PValueField,
(size_t)((elem. ValueLength < 50) ? elem. ValueLength : 49)) ;
for (pc = level ; *pc ; pc++)
*pc = ...;
```

Triggerable via C-MOVE-RQ with an oversized (0008,0052) element.

Suggested Fix

```cpp
// Option A: zero-initialize level
char level[50] = {};
strncpy(level, elem.PValueField, 49);
level[49] = '\0';   // explicit safety

// Option B: use snprintf
char level[50];
snprintf(level, sizeof(level), "%s", elem.PValueField);
```

History

#1 - 2026-06-22 20:52 - Michael Onken

Fixed with commit 34fa53.

#2 - 2026-06-26 07:55 - Michael Onken

- Status changed from New to Closed

- Private changed from Yes to No