DUMA – Debugging memory on BREW®

airsource MOBILE SOFTWARE

1 About DUMA and this test

DUMA is an open source library that can detect the following memory access conditions:

- Illegal access beyond the top of an allocated area
- Illegal writes before the start of an allocated area at de-allocation time

DUMA stops the program under test at the exact instruction causing the fault

In addition, this test builds on the DUMA layer and:

- Stores four (configurable) levels of call stack at every allocation
- Tracks all allocations and deallocations to aid finding bugs

2 Running TestDUMA

Project files have been provided for Visual C++ 6.0, which rely on the environment variables BREWDIR and BREWSDKTOOLSDIR being correctly set. This project has only been tested on the BREW 3.1.5 SDK.

In order to run the application:

1. Ensure that the BREWDIR environment variable points to the location of the 3.1.5 SDK, e.g. C:\Program Files\BREW 3.1.5 SDK\sdk
2. Ensure that the BREWSDKTOOLSDIR environment variables points to the location of the BREW SDK Tools, e.g. C:\Program Files\BREW SDK Tools 1.0.1
3. Run the BREW Simulator from Visual Studio and make sure the MIF Directory setting is pointing to <testdumadir>. Use the device pack included in <testdumadir>.
4. Choose the TestDUMA application.
5. BREW loads the TestDUMA DLL and starts the application.
6. You can now generate various memory errors by following the instructions on-screen.

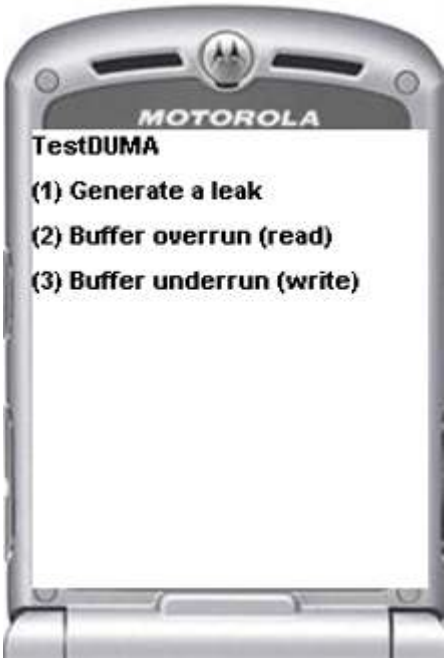

Figure 1 - Startup Screen

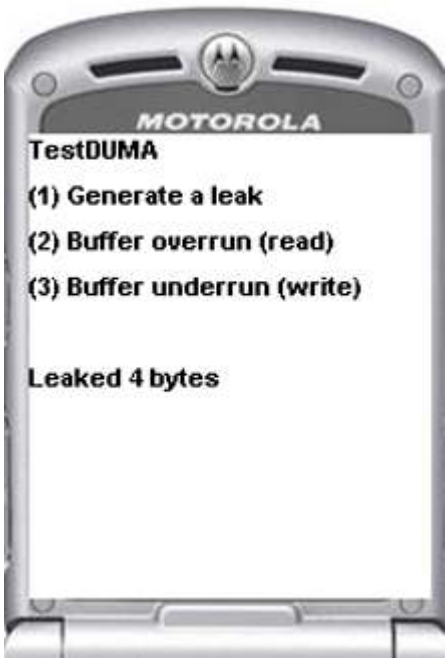

Figure 2 -Generating a leak

When the application runs, you have three choices. Press a key to choose one of the three error cases. Pressing any other key (except END) will redraw the screen. END will exit the application.

3 Leak Detection

Memory leaks are not instantly detected. You can search for potential memory leaks[1] or you can wait until application exit, when memory_check() is called in this test.

To trigger detection of a memory leak:

- Press 1 on the main menu, to leak 4 bytes
- Exit the application

Detection and analysis of a leak works as follows:

When memory_check() detects a leak, it triggers a breakpoint, and you will see a standard Visual Studio error box with the message 'User breakpoint called from code <location>'. Click OK, and you will see the following screen (Figure 3). The breakpoint was triggered by '__asm int 3', and as you can see, TestDUMA has found a leak. This is similar information to that which the BREW Simulator would give you normally, but we can now look further at the allocation, and find out who exactly allocated it.

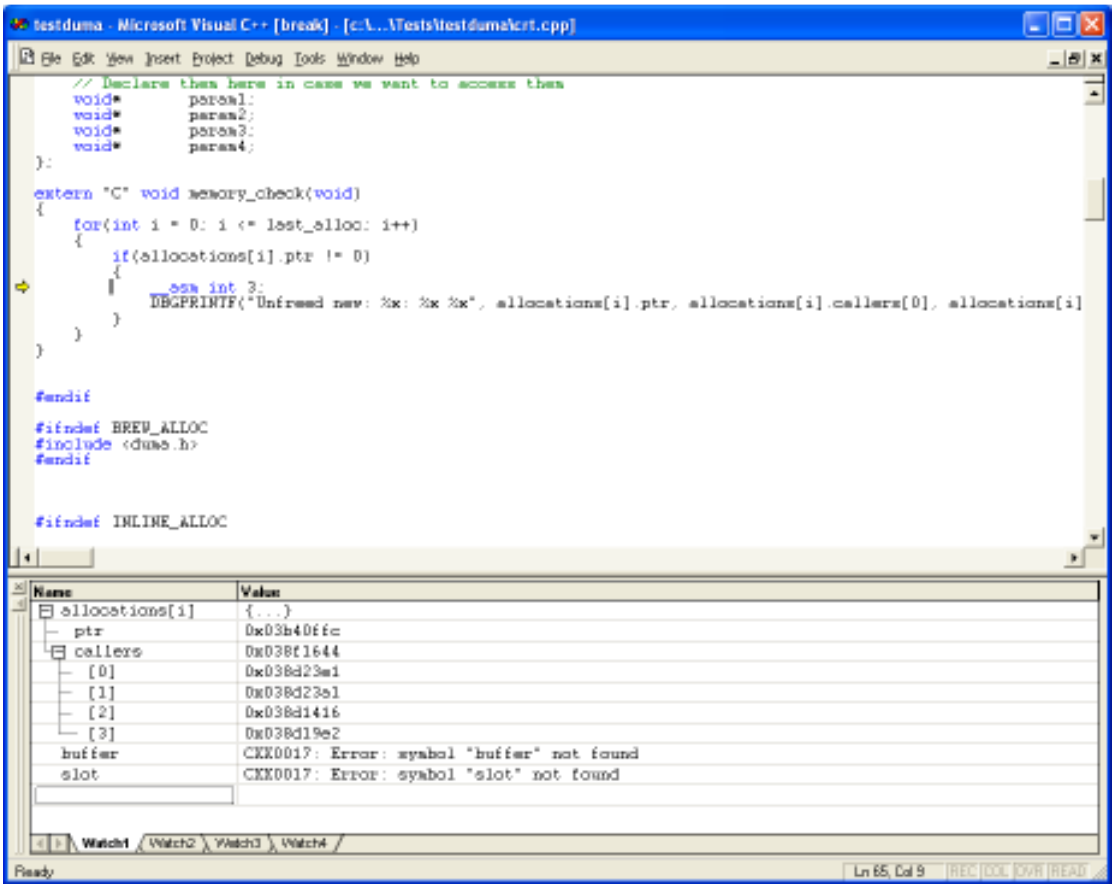

Figure 3 C++ listing for leak detection

[1] not implemented in this example, but easy to implement via calling memory_check()

Show the Assembly View, either by hitting Ctrl-F11 or right-clicking and selecting 'Go To Disassembly'. You will now see (Figure 4) an annotated assembly listing. Select one of the allocations[i].callers[] values, and drag it to the assembly listing, then release.

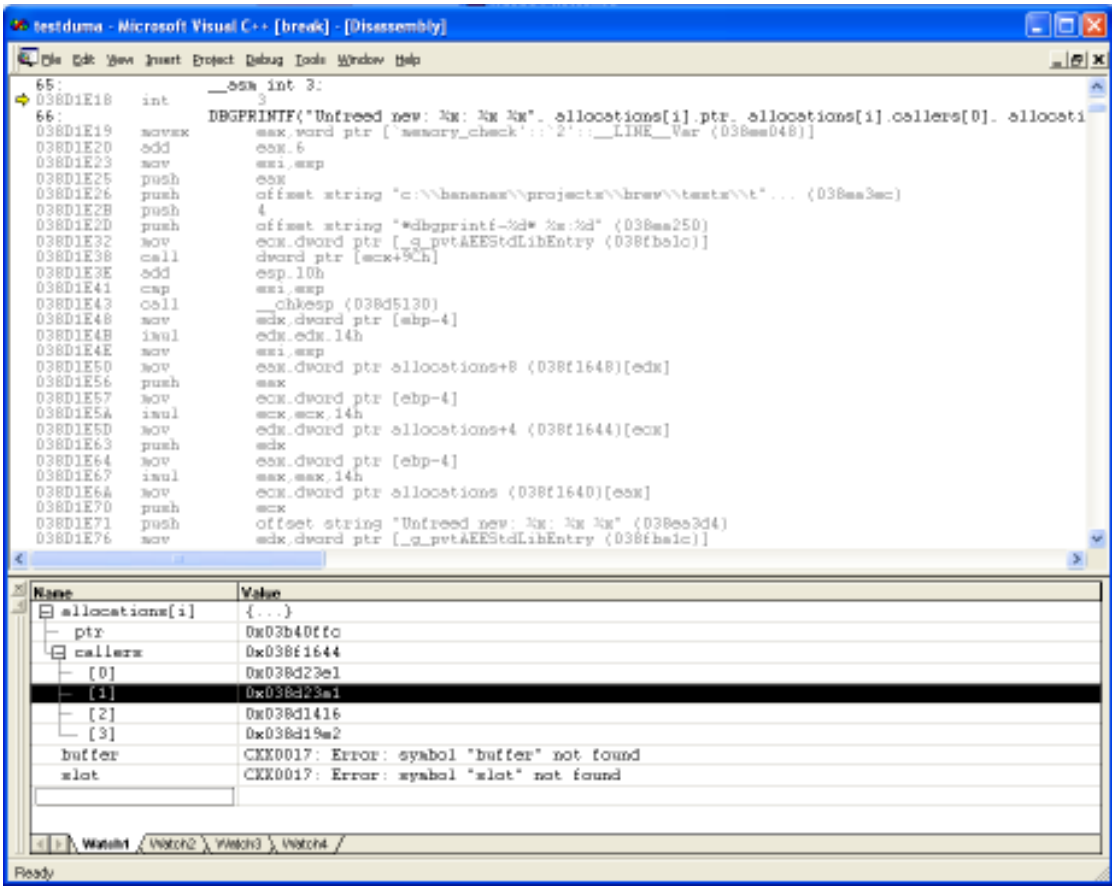

Figure 4 Assembly listing for leak detection

You will now see a new listing (Figure 5). In this case we selected the value for allocations[i].callers[2], so we are now looking at the method which called the method which called the method which allocated the memory. Or, if you like, we end up in this code when we return through three levels of call-stack from the allocation routine. This is considerably more useful information than the BREW Simulator gives you.

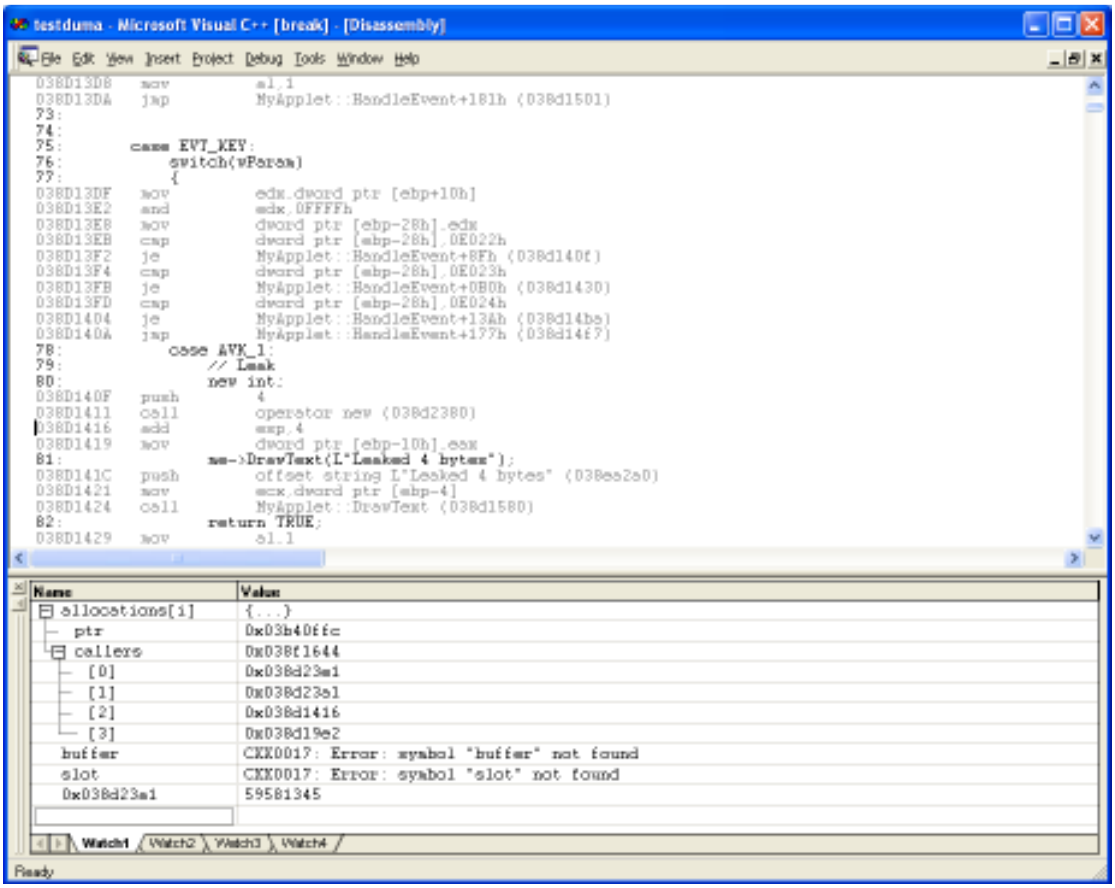

Figure 5 Assembly listing for original allocation

4 Buffer Overrun Detection

When a buffer overrun occurs, DUMA stops execution at the exact instruction causing the illegal memory access. You will be able to see the full call stack for the code being executed. In addition, provided you can identify the address of the allocation, which is generally pretty easy[2], you can look at where the address was allocated, and the call stack at that time.

When an illegal memory access is detected, you will see the following dialog (Figure 6). On Visual Studio 2005 there will be an option to break or continue, ignoring the error.[3]

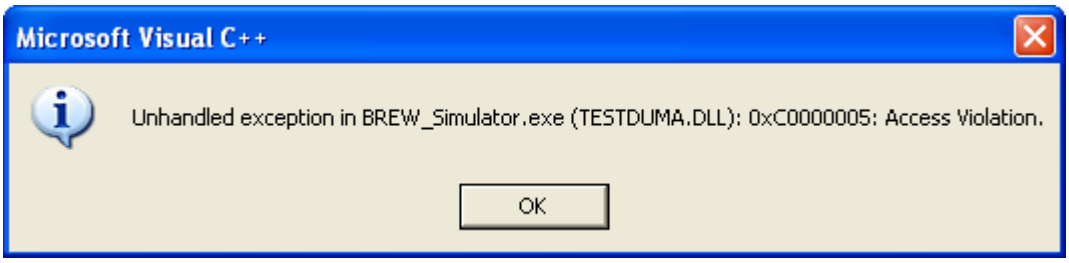

Figure 6 Breakpoint following illegal memory access

Click OK (break on VS 2005) and you will see the line of code triggering the error (Figure 7). TestDUMA includes a utility function find_allocation(void*) which will return the allocation information and call-stack for a pointer. In this case the buffer overrun occurs when access buf[5], so we call find_allocation(buf). If the pointer passed as a parameter to find_allocation() was not actually allocated on DUMA's heap, find_allocation() will return NULL.

[2] It would be possible to add a method to identify the allocation by looking backwards through memory for a magic marker.

[3] To ignore the error and continue in Visual C++ 6.0, click OK, then Ctrl-F11 to get the assembly view. Right click on the instruction *after* the illegal memory access and select "Set Next Statement". Then continue running with F5. You can do this in the C view as well, but it is not recommended, as you will be skipping more assembly instructions than necessary.

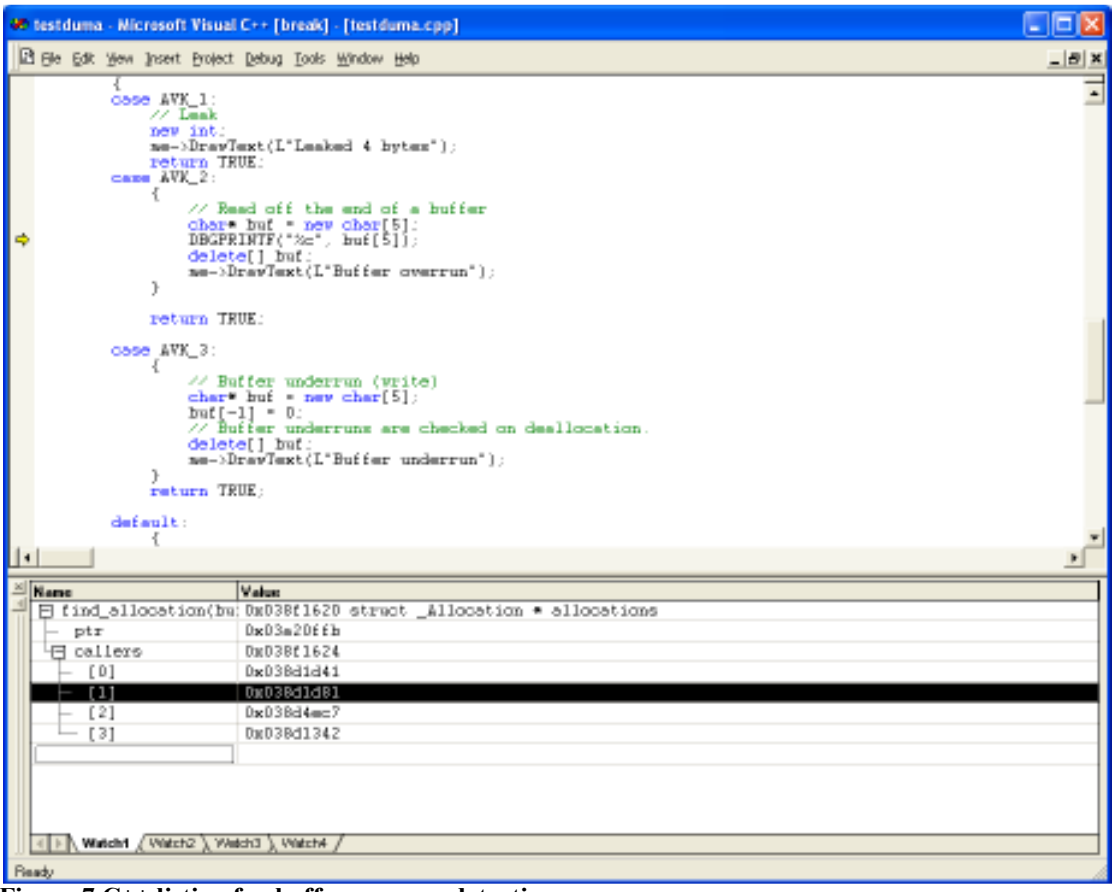

Figure 7 C++ listing for buffer overrun detection

It is important to note that find_allocation() only works on pointers allocated on DUMA's heap. It will not return call stack information for pointers allocated by other means, or for a pointer which addresses any address other than the base of a heap block. If the pointer has been modified, for example to point somewhere inside a heap block, find_allocation() will return NULL.

Having ascertained the call stack information in the watch window – find_allocation(buf).callers in this instance – you can now use this call stack information to view the code where the allocation took place. Follow the same procedure as listed in Leak Detection in the previous section.

5 Buffer Under-run Detection

Buffer under-runs are only detected on a write, not a read, and are detected at de-allocation time. At de-allocation, you will see a User Breakpoint dialog, and after clicking OK, you will be shown code in the DUMA_Abort routine. To find the actual code where the de-allocation takes place, you will need to go back up the call stack – in this case click on MyApplet::HandleEvent, which is the last visible line of call stack.

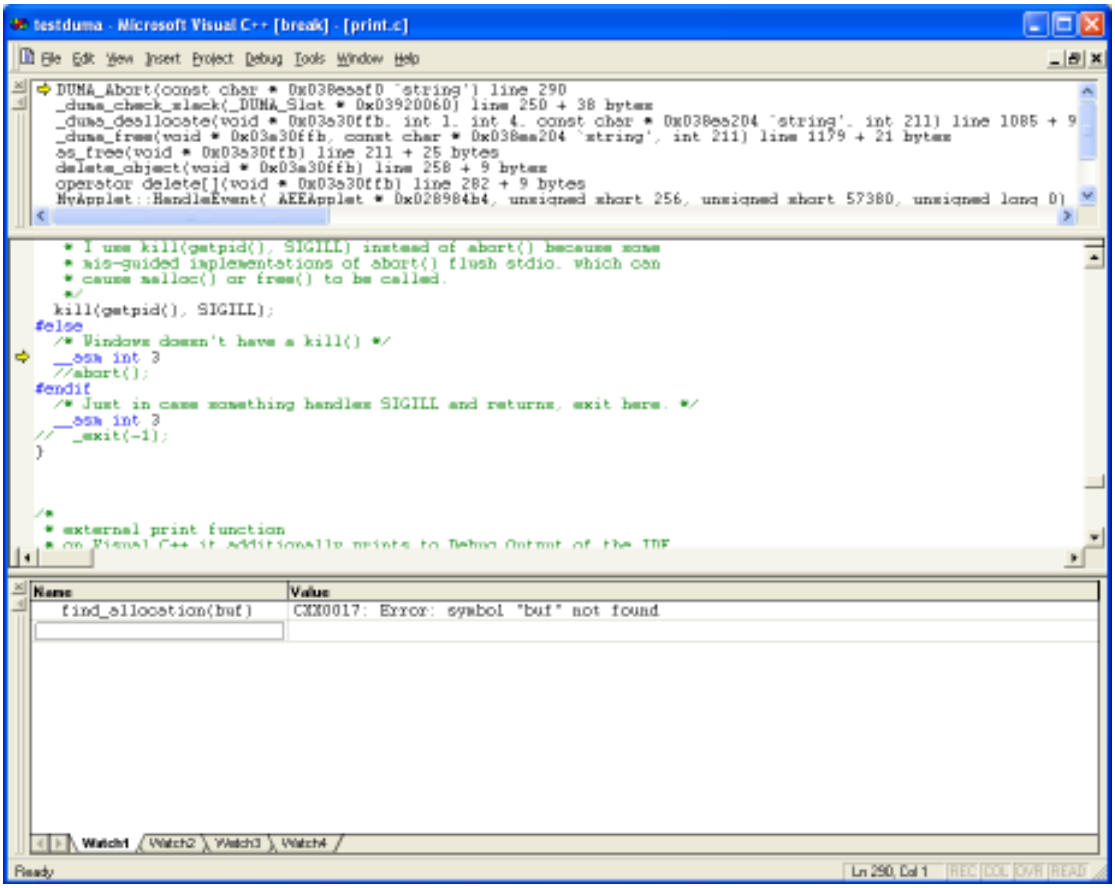

Figure 8 C++ listing for buffer under-run detection