HeapCheck 1.2

Thanassis Tsiodras, Dr.-Ing
ttsiod@softlab.ntua.gr

HeapCheck is a Debugging Library for Win32 environments. What sets it apart from other heap-debugging libraries is its ability to pinpoint invalid read accesses to the heap, not just heap corruptions. It also detects memory leaks and reports memory usage statistics. Access checks are done through the paging hardware of your CPU, and don't introduce any CPU overhead. HeapCheck debug versions run at almost the same speed as normal debug versions - with performance hits noticeable only when allocating and deallocating.

1 Disclaimer

I am NOT responsible for any damages this causes to your computer, company, fame, payroll, etc. I am releasing my humble efforts to the public so that some programmer's life gets a little easier. If it leads you on a wild goose chase and you waste two weeks debugging something, too bad. If you can't deal with the above, please don't use the software! I've written this in an attempt to help other people, not to get myself sued or prosecuted.

2 Features

HeapCheck is a Win32 Debugging Allocator with the following features:

- Detection of invalid read/write accesses to memory allocated with heap functions.
  Let me stress this out: READ as well as write accesses!
- Detection of memory leaks
- Detection of invalid read/write accesses to freed memory
- Thread safe
- File/Line info reporting, based on `imagehlp.dll`
- A GNU license (which means that it's free - read `GnuLicense`)

If you are eager to check it out, go to the Usage section. Just make sure you eventually read everything, or you'll be missing some goodies (like the ones in section 8 (Stack techniques)).

3 Introduction

As you program with C or C++, you are bound to hit upon a heap-related bug sooner or later. What's a heap, you ask? Well, it's the system resource from which you remove chunks when you call `malloc()`, `calloc()`, `new()`, etc. Unfortunately, it is associated with a lot of really tough bugs. You see, in this code:

```
main()
{
        char *p = (char *) malloc(5);
        strcpy( p, "12345");
        puts(p);
}
```

...you probably get away clean in the debugging phase. You only allocated 5 bytes, you are writing 6 (5 for the string plus one trailing 0), but nothing seems to happen, right?

Wrong.

The extra byte you are writing moves your program into 'undefined behavior' domain. Some operating systems/languages tolerate such things, others don't. Even those who seem to do, eventually don't. You see, the extra byte is written into memory not owned by the allocation. What is stored there can be anything from debugging info to application data (in some operating systems, even application code).

Then again, we don't free the memory we allocated. It seems we forgot to:

```
free(p);
```

Notice also that the call to `puts()` makes the system read 6 bytes, since `puts()` is probably implemented in the C runtime library with something like this:

```
while(*p)
     putchar(*p++);
```

...which means that not only your code, but also system and RTL (run-time library) code accesses memory it shouldn't. This can make your program behave in a non-predictable way - crash sometimes, work OK as long as you don't run WinAmp... (you get the idea).

If you want to quickly find heap-related bugs in your Win32 programming, you can use this library in addition to any others you have found/bought. It is a simple little library, which I have found useful in my programming efforts. Why do I use it instead of others (like the builtin debugging heap of Visual C++ 6.0)? It's small (and thus, easily configurable) and it supports C++. It also lacks any complicated interfaces, and provides you with a quick and easy way to detect invalid READ as well as write accesses (this last reason is probably the strongest point of the library).

HeapCheck employs a technique invented in the Unix's Electric Fence, created by Bruce Perens. It uses the virtual memory hardware of your computer to place an inaccessible memory page immediately after (or before, at a `#define` option) each memory allocation. When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then trivial to find the erroneous statement since Visual C++ will point to the code doing the invalid access. In a similar manner, memory that has been released is made inaccessible, and any code that touches it will get a segmentation fault.

HeapCheck also checks for any allocations you forgot to free/delete, and prints a report when your application exits to the Output window of Visual C++. This report tells you also the allocation requirements of your program (how much memory it needed in the previous run).

4 Usage

You can easily use HeapCheck in your own programs in three easy steps:

1. In the source files where you want the checks to take place, include the `swaps.h` header file. Make sure that it is the first include file you are using, *after* the system and RTL include files, e.g.

   ```
   #include <xxx.h>
   ...
   #include <yyy.h>
   #include "swaps.h"
   #include "myfile1.h"
   ...
   ```

   This file `#define`s the calls to heap-related functions in your code to calls in the library. It also provides the prototypes for the new allocation and deletion operators.

2. If your code is in C only, add `HeapCheck.c` in the project, or
   If your code is in C++, add `HeapCheck.c` and `CplusCheck.cpp` in the project.

3. Add `imagehlp.lib` in the libraries you link with.

That's it. Your code will be automatically checked for heap errors.

- To get correct file/line information, make sure you are using the correct versions of the debugging libraries (the Windows 2000 ones). I used `dbghelp.dll` version 5.0.2195.1, and `imagehlp.dll` version 5.0.2195.1. Maybe `SymGetLineFromAddr()` works with other combinations, but I used these DLLs successfully under both NT 4.0sp6 and Windows 2000.

- By default, range checks will be done for post-block accesses. After checking that everything works OK in this setup, modify your project so that *PRE_CHECK* is defined, or even edit `swaps.h` and define it on top. Recompile everything, and this way, pre-block checking will take place.

- HeapCheck messages appear on the Debugging Output of Visual C++. If you are running the program outside the IDE, use DebugView to see them (you can download it from *http://www.sysinternals.com*).

- If you program with MFC, make sure you use it in the form of a dynamic DLL *(Project - Settings - General - Microsoft Foundation Classes - Use MFC in a Shared DLL)*. Otherwise, you 'll get link problems, since MFC defines a global `operator new`, just as HeapCheck. If you use it in the form of a DLL, MFC sticks to its own `operator new`, while your code calls HeapCheck's.

When you have completed debugging your program, recompile in Release mode. The debugging checks will melt away automatically.

5 Configuration

In `HeapCheck.c`, you can change the way the library behaves by changing:

1. *MAX_ALLOCATIONS:*

   Total concurrent allocations possible. If your program needs more, you'll get an assertion at runtime telling you to increase this.

2. *MAX_ALLOCATABLE_BLOCK:*

   Total heap available to the application. If your program needs more, you'll get an assertion at runtime telling you to increase this.

3. *NO_VC_HEAP_ERRS:*

   It is possible to write code that bypasses the mechanisms of the library, and allocates from the standard VC-heap. An example is code inside the runtime library which is pre-compiled and uses different allocation mechanisms. Example: an enhanced `new()` operator in `iostrini.cpp` (line 21) and `cerrinit.cpp` (line 21) where `_new_crt()` is used. `_new_crt()` maps through a `#define` to a `new()` operator that takes extra parameters. HeapCheck can't catch this call, so when the time comes for the deletion of these blocks, the library's delete operator doesn't find them in it's tables. It is capable to understand that these are VC-heap blocks though, through the use of `_CrtIsValidHeapPointer`. If you `#define NO_VC_HEAP_ERRS`, the library won't complain for such situations. This is the default, but if your code issues direct calls to `_malloc_dbg(), _calloc_dbg()`, etc, you should disable this.

   If the previous text sounds like Chinese, leave it as it is.

6 Debugging - Other tools

1. Visual C++ 6.0 Runtime Heap Debugging:

   This only checks for heap corruption (i.e. destructive side-effects of *writing* outside the allocations) and even then, it only catches writings in the *NO_MANS_LAND_SIZE* (default:4 bytes) on either side of the blocks. The detection of these errors is done whenever the user (or the system) calls `_CrtCheckMemory()`. HeapCheck catches read accesses as well, at the EXACT place they are made, and in a much larger block (a page in i386 systems is 4096 bytes).

2. BoundsChecker:

   This is a very good debugging tool, capable of catching almost every bug. However, CPU cycles are used in order to perform all these checks, and especially in *instrumentation* mode and *maximum memory checking*, the program runs really slow (some programs, like protocol stacks, won't run correctly if they run too slowly). HeapCheck can only catch heap-related errors, but it uses the paging hardware to do the checking. The net result is that HeapCheck debug versions of programs run almost as fast as normal debug versions (with speed impact noticeable only during allocations and deallocations). In other words, you can leave it in your project and forget about it.

7 For UNIX gurus and fanatics only

If you have a history using Electric Fence in UNIX, you'll remember that the only thing required was linking with libefence.a. Why can't we do something like this under Win32 for HeapCheck, too?

1. Win32 ain't UNIX. I guess you knew that, but there's this little fact about libraries that makes them almost useless: if a library uses any function of the C run-time library (like memchr, strcpy, etc) it has to link with one of the six versions of the run-time library (RTL):

   - Release, static, no multithreading
   - Release, static, multithreading
   - Release, dynamic, multithreading
   - Debug, static, no multithreading
   - Debug, static, multithreading
   - Debug, dynamic, multithreading

   Now suppose that your library is linked with one of these RTLs. When you use your library inside one of your own projects, your project MUST be linked with the *same* RTL! Amazing. I don't know about you, but I prefer to just add two source files in the project and be done with it, instead of building six HeapCheck libraries.

   I could replace calls to the C run-time library with my own versions of things, but that would lead to the next problem:

2. If HeapCheck was a library with implementations of standard functions (malloc, realloc, etc) then the projects using it would have to choose between using HeapCheck and using the C runtime library. Unless I am doing something wrong, the linker can't tolerate two libraries providing implementations of the same function. How is he to choose? The UNIX ld implements a first-serve attitude: Whichever library provides it first in the link line, is chosen.

If anyone of you comes up with any ideas on these matters, do share. This is why HeapCheck is under the GNU license - I made it, it's your turn now to make it better!

8 Stack space checks

If you are like me, you probably use code like this sometimes:

```
void f()
{
        char a[100];
        ...
```

Unfortunately, you don't have HeapCheck's automatic range checking for buffers allocated this way. There is a way around this, if you code in C++. Place this template/macro in a commonly accessed header file:

```
template <class T>
class Array {
public:
    Array(T *p):_p(p)
    ~Array() { delete [] _p; }
    operator T*() { return _p; }
    T& operator [](int offset) { return _p[offset]; }
private:
    T *_p;
};
#define ARRAY(x,y,z) Array<x> y(new x[z])
```

and then, when you need a temporary array, use it like this:

```
void f() {
        ARRAY(char,a,100);
```

or, directly:

```
void f() {
        Array<char> a(new char[100]);
```

As you see, this way heap space is used for your arrays, and it is automatically freed when the variable goes out of scope, just like a stack-based array. So, you get the best of both worlds: automatic freeing (through the template) and bounds checking (through HeapCheck). This technique also lowers your stack usage, which could mean something if you use recursive algorithms.

9 Known issues

HeapCheck uses the paging hardware of your machine to place an extra guard page before or after each allocation your program makes. A page is 4096 bytes for the vast majority of today's CPUs, which means that the memory requirements of your program can increase beyond usual. Of course these requirements vanish when you compile your Release versions, but still, make sure you have plenty of physical memory and/or swapfile for your Debug versions. For example, the default settings of HeapCheck create an 8MB heap. This consists of 8*1048576/4096 = 2048 pages. If you make 1024 allocations of 1 byte each, you'll exhaust this heap (each allocation reserves one data and one guard page). This is the worst case scenario, of course, but it shows how quickly memory fills up with HeapCheck. From version 1.2 onward, guard pages no longer occupy physical storage, which will lower your actual memory requirements by up to 50%. However, you'll probably have to increase *MAX_ALLOCATABLE_BLOCK* in most cases... (you'll know when to do this, because HeapCheck will give you an assertion if the heap is exhausted).

10 Contacting the author

Why? :)
As you can see in the disclaimer, I am not responsible for any disasters HeapCheck causes to the universe. Seriously though, I am interested in any bug reports/comments. You can e-mail me at *ttsiod@softlab.ntua.gr*