HeapDraw example

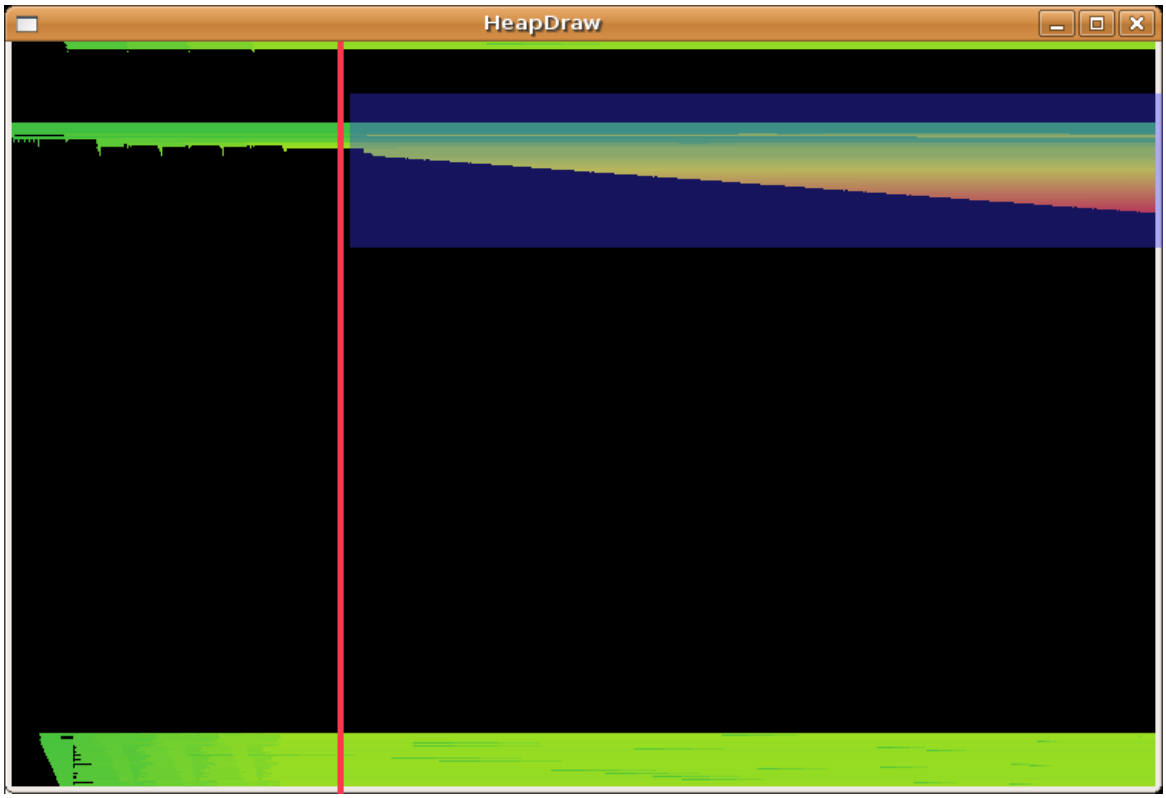
gera

(no fancy formatting, sorry)

This is an example, to show you how we use it to help us in developing exploits. If you've seen any of my Heap Massaging talks, you may (probably) understand a little bit more of this incoherent words I'm spitting here.

For this example I will be using the trace file dtlogin.Solaris 9.truss that you can find in the examples folder. The main picture looks like:

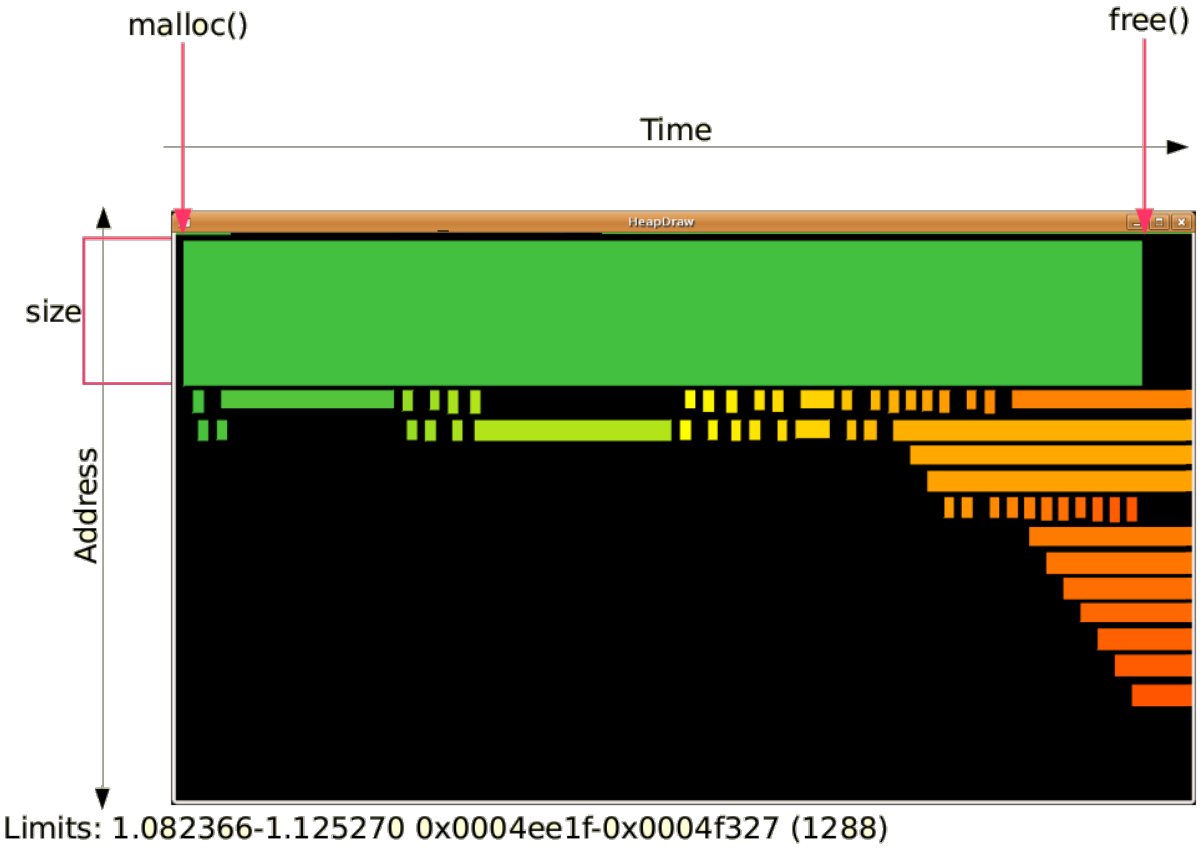
$ linux-native/heapdraw -t truss <Examples\ and\ Doc/dtlogin.Solaris\ 9.truss



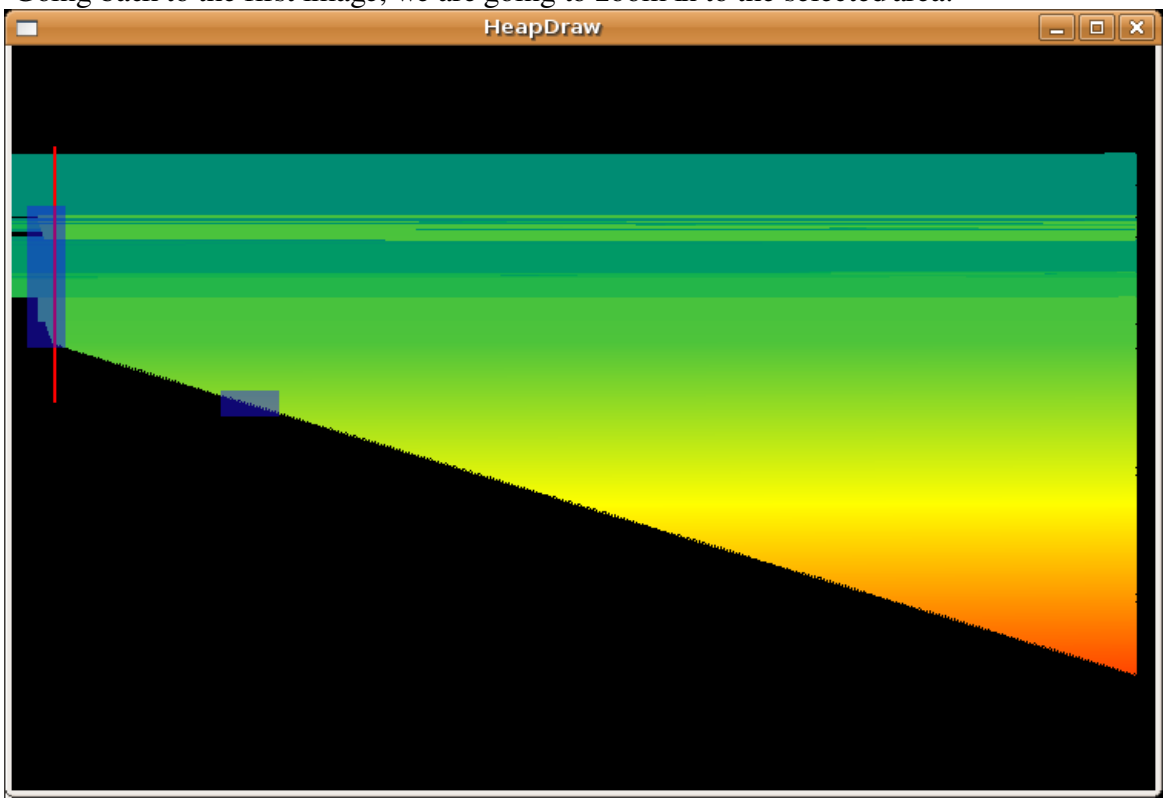
In the picture we can see how the heap evolved during the life of the application dtlogin, while targeting it with an exploit.

To the left of the red line, we can find all the heap activity for initialization. Everything to the right we know is due to our exploit. To understand a little bit better this picture, lets take a look at the next one from the 5 minutes presentation in the Documentation folder. This is not part of the trace from dtlogin.

Addresses are shown in the Y axis (vertically), with zero on the top. The time is shown in the X axis, growing to the left. A block in the drawing represents a heap chunk (block), delimited by its start and end address, and creating and deletion (free) time. The color also represents the creation time of the block, in a scale going from green to red through yellow.



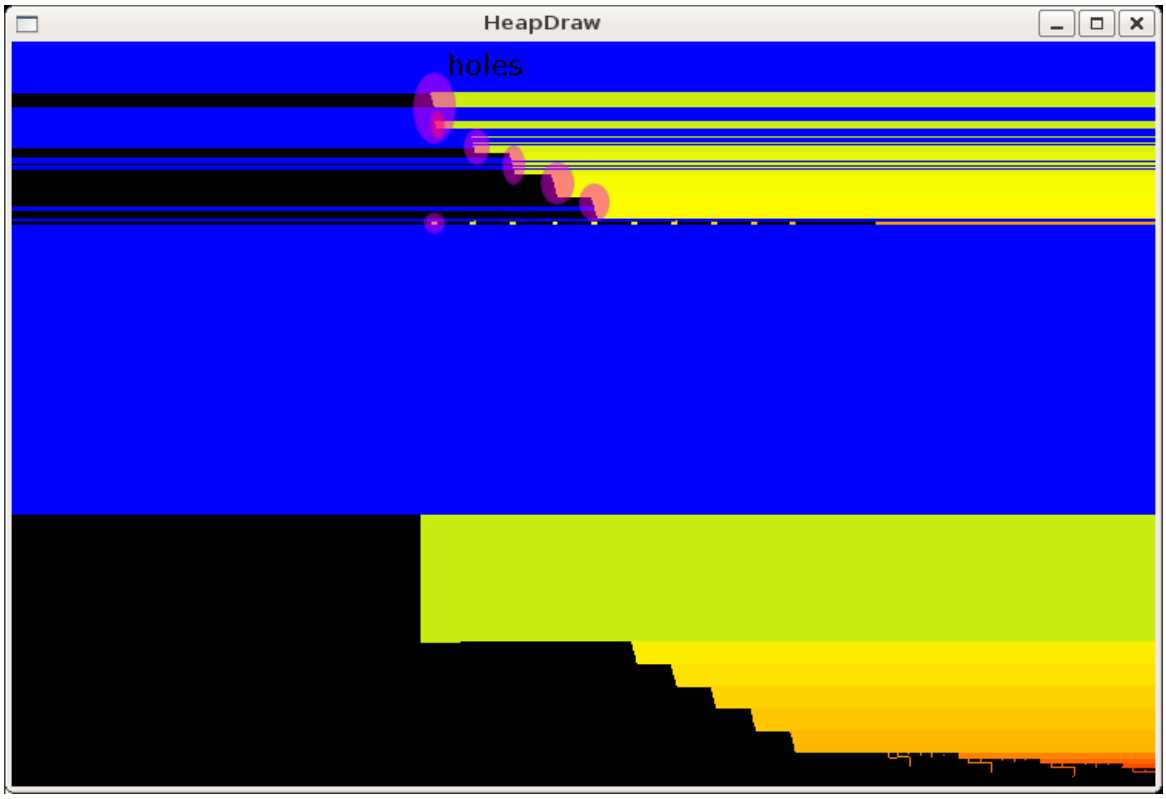
Going back to the first image, we are going to zoom in to the selected area:



Again, with a red line we separated two different phases, this time, of the exploit. To get good reliability in a heap exploit it's very important to start from a know heap structure, however, as the application may have been running for a long time, there is no real way to predict the actual heap layout at any given moment. To solve this problem it's very common that (good) exploits first try to fill the holes in the heap, forcing the layout to evolve to a [quite] predictable state.

To the left of the red line we can see this initialization (or better called hole filling) phase of the exploit. To the right, we can see the exploit itself: in this case it was a carefully crafted pointer-to-code-write primitive repeated ad nauseam, or until the exploit was successful.

The next two pictures will show first a zoom into the heap-filling phase, and then a zoom into the primitive exploitation phase:

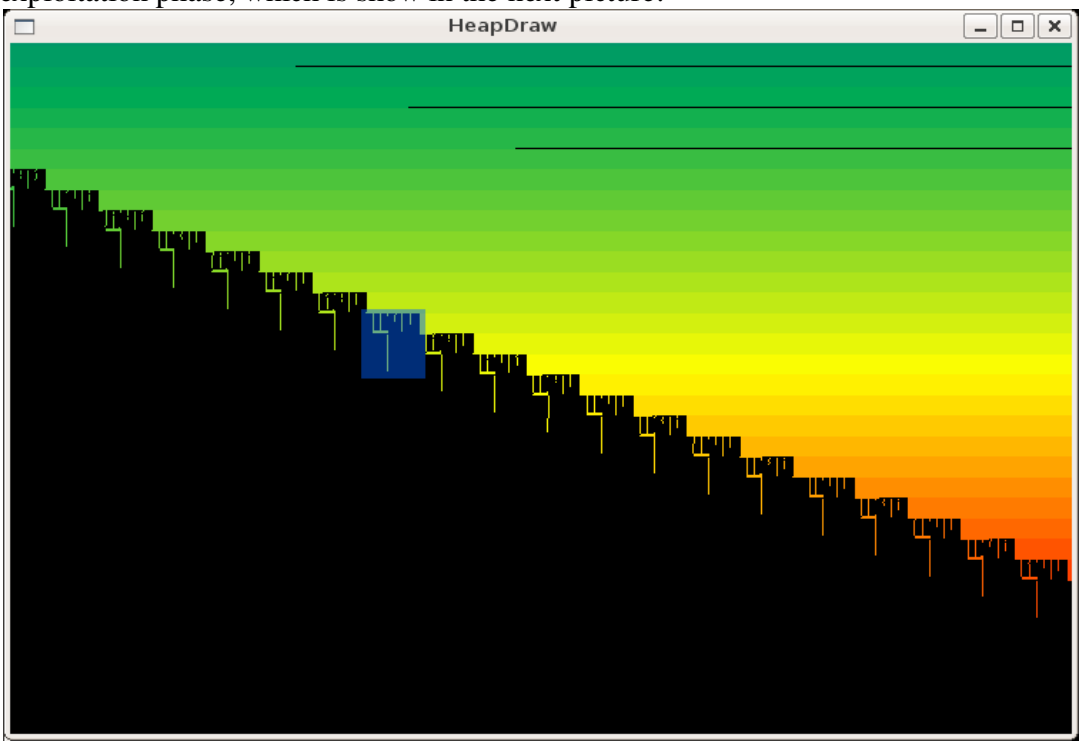


Every time you zoom in, the color scale is also zoomed in, recomputing the colors for every block, relative to the current view. The blue represet blocks which have been allocated to the left (earlier) than the zoomed in section.

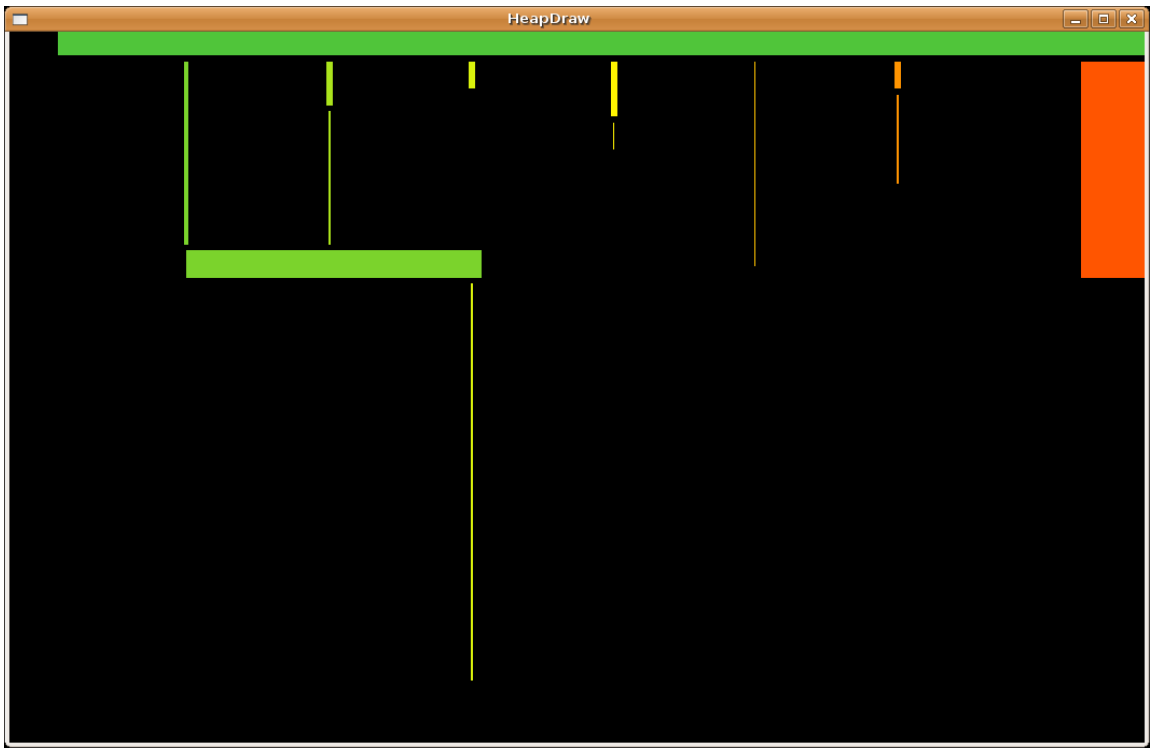
In the picture you can see you the exploit allocates more and more blocks on the heap, and how the first are located in the holes left by previous heap activity (some holes are highlighted). It's interesting to see the tiny block which is allocated and deallocated repeatedly (one of them per request).

We also highlighted in a darker oval near the top, for something that looks like a bug in heapdraw (and may actually be it): A block (yellow) is apparently allocated where there is already an allocated block (blue). This may be due to multiprocess applications (and showing the activity for all processes in the same window, or it may be due to heap activity preexisting the moment where we attached to the target application and stared tracing it).

After the initialization you can already see in this picture a different pattern, which is part of the primitive exploitation phase, which is show in the next picture:



Here we can see how the same patter repeats. We said that the primitive is used several times until the exploit is successful. For every try the exploit need to do several (7) heap operations which compose the basic pattern. In the next picture we zoom in again to see a single try of the pointer-to-code-write primitive:



In this last imge we can clearly see the 7 heap operations neededto perform a single pointer-to-code-write primitive:

1. Allocate 2 blocks, one big one smal, deallocate the bigger, leak the smaller.
2. Allocate and deallocate 2 blocks, one small one big. Both feet in the hole created in 1.
3. Allocate and deallocate 2 blocks, one small one big. The big one doesn't feet in the hole. Then deallocate the block leaker in 1.
4. Allocate and deallocate 2 blocks.
5. Allocate and deallocate 1 block.
6. Allocate and deallocate 2 blocks.
7. Allocate a big block (which actually contains the code and is never deallocated, as you can see in the bigger picture).

This is it. The tool is very useful when you want or need to see how the heap evolves, to see what possibilities a heap overflow gives the attacker.