Paul Campbell (108481554)

CSE 409 2-Week interim report

**The black magic of LD\_PRELOAD**

One of the techniques we had set out to use to perform the instrumentation of heap memory was using LD\_PRELOAD in Linux. This option seemed like it was a good idea because we could just do LD\_PRELOAD=/path/to/my/hook.so ./myprogram and we could hook onto any libc calls we wanted without root access and without modifying the binary itself. Initial testing of this went well; we created a shared object called libhook.so which was able to hook malloc. We achieved this by using dlysm to get the function pointer to the libc version of malloc. We then wrote our own wrapper around malloc that just included some print statements. We then called libc\_malloc inside of our wrapper and returned the pointer. To the program that this was being tested on nothing should see to be different. I made a sample program that made a simple call to malloc, which allocated space for two integers, assigned a value to one of those locations, and then freed the memory. We then tested it out using LD\_PRELOAD=./libhook.so ./sample which upon execution used our wrapper around malloc and internally used the libc\_malloc. Success!

This all seemed to be working well at first. Unfortunately when we added a hook to free the first set of problems seemed to manifest. After running with LD\_PRELOAD set to our shared object with free and malloc hooked, the program went into an infinite loop. Upon further inspection I noticed that free was being called many more times than my program had ever called it. These calls to free had NULL passed to it so NULL was attempting to be freed over and over until the program finally had a segmentation fault. The first attempt to solve this problem involved gdb; this is when another issue came about. To debug our program we had to set LD\_PRELOAD. This would cause gdb to then use our custom malloc and free as well. So debugging the program using gdb did not seem feasible. We then tried to use backtrace to see if we could get the parent function that was calling free but this just caused the program to also throw a segmentation fault.

To see if we could get anything done with this technique, in the wrapper we had temporarily set free to ignore calls to freeing NULL. We could then perform simple techniques like checking for double free, free memory that was not allocated with malloc etc. The next hurdle became detecting out of bounds access. After doing some research we found the function mprotect. The function mprotect allows you to change the access of memory. The idea was we would allocate more memory than we would need. Then we would set some padded value before and after to PROT\_NONE using mprotect so it would throw a SIGSEGV when someone tried to access memory out of bounds. This did not work as planned; mprotect can only change permissions on page aligned memory. We tried to get around this by setting up the memory so that it was page aligned using memalign and posix\_memalign. Now we would have tons of wasted space be in theory be able to change the permissions on the memory right? We mprotect can only change the permissions of a whole page even though it accepts a range (at least it seemed this way in Ubuntu 13.10). So even though mprotect accepts a range from the page offset it still set the whole memory page to PROT\_NONE.

We then came up with a final idea. We would allocate page aligned memory, add the padding before and after the space we wanted to, and then just set the whole page to PROT\_NONE as we had been doing. We then set up a signal handler using sigaction that would throw the signal SIGSEGV which would check to see if the address was valid, and if it was it would set the page to PROT\_READ | PROT\_WRITE. If not the program would print a message and exit. This works the first time memory access occurs, but we were unable to figure out a way to reset the memory page back to PROT\_NONE. This is currently where we are stuck using this technique.

**Debugging with ptrace**

Another technique we tried using later in our research was ptrace. Other programs like GDB are built on top of ptrace so this also seemed like it could possibly do what we wanted. This technique immediately seemed as good as LD\_PRELOAD because we would again run a binary without modifying it but could in turn affect the results that it produced. After spending many hours trying to understand how ptrace worked, we finally were able to set up a program that could count the amount of instructions run in a binary. The instruction count was insane (100,000 lines of code to print hello world using printf). To actually verify that this instruction count worked we created a simple program in at&t assembler which had 8 instructions to print out hello world. Having our instruction counting program run this hello world program we were able to verify that the instruction count was indeed correct. We then moved on to not only counting instructions using ptrace but actually getting the value of RIP(x64) or EIP(x86). This modified version of our ptrace program printed out the values of this register and other registers which we were able to confirm correct using other tools. This is the current point of our ptrace research.

**Conclusion**

While LD\_PRELOAD seemed to start off well, we feel that it is currently a dead end. Dynamirio was just very confusing to use and the documentation seemed to be lacking. Either ptrace or PIN seems to be the route we will take.