Binary Instrumentation Interim Report by Paul Campbell, Michael Salerno, Shane Harvey

**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. The program that was being instrumented should not even notice. I made a sample program that made a simple call to malloc, it 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, more 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 was 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 to allocate more memory than we needed. 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 but we would still be be able to change the permissions on the memory right? Well 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 have 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 were 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.

**Pin - A Dynamic Binary Instrumentation Tool**

Another technique to perform instrumentation was using pin. Pin seemed like a good idea since it would not actually run the instrumented program, but emulate it. Pin also allowed us to modify it in real time. First, to get things rolling, we wrote a simple module to count the number of instructions a program executed. We did this by using the INS\_InsertCall object and incrementing a static variable. Next we wrote a module to hook malloc and free. We hooked these by checking for a routine named “malloc” every time an image was loaded. If we found a valid routine when an image was loaded we replaced it with our own rewritten version of malloc. This new version of malloc keeps track of the address spaces that were malloced and how big it was. Lastly, we detected if there was any reads or writes being done to the heap memory. If there was then we checked the address of what was being accessed and compared it with the list of addresses our new malloc function made. If the address was not in range of anything we malloced, we print out to tell the user that an invalid address is being accessed.

**Pin – Simple Stack Smashing Detection**

We would like to implement stack-smashing detection. Our basic strategy is to save the return instruction pointer on all call instructions and then check that this value is the same when we encounter a return (ret) instruction. From our experience so far we believe this is possible by using the function INS\_IsCall() and then passing the IARG\_RETURN\_IP as a parameter to a function that appends this value to a list. Then when we encounter a ret, using INS\_IsRet(), we can read the return address off the stack and check if it is the same as the last saved return address. We may want to change IARG\_RETURN\_IP to the current instruction pointer because that is what the call instruction saves on the stack.

We were able to make a Pin tool that could display the return\_ip’s at each call instruction(956 for a simple hello world c program) and the value at the stack by examining the value at REG\_ESP just before the ret. Our problem is that many of the IAR\_RETURN\_IP’s are zero and it looks like the values aren’t matching up the way we think they should. This is probably because of the different ways you can ret in x86, eg ret, retn, retf, ret with an immediate value.

**Conclusion**

While LD\_PRELOAD seemed to start off well, we feel that it is currently a dead end. We can do some more research on the benefits, if any, of using DynamoRio vs. Pin but we think ptrace or PIN seems to be the route we will take.