CS341l Fall 2007 Lab 7

The purpose of this lab is to understand the basics about how dynamic linking occurs using a global offset table (GOT) and procedure linkage table (PLT). Make sure you are working on a 32-bit machine. Some of this is further reading for those interested in security, you only have to answer questions in italics, the rest is FYI.

compile helloworld.c with this command: **"gcc helloworld.c -o helloworld"**

(1) Now run the debugger with "**gdb helloworld".** Type **"break main"** to set a breakpoint at main and then **"disassemble main"** to disassemble it. *Write the entire instruction that is the call to printf*:

(2) Now disassemble the 5-byte instruction that is at the address main() is actually calling. I used **"disassemble 0x080482b0 0x080482b0+5"** because my main was calling **"0x80482b0 <_init+56>"**. *Write the entire instruction down*. (Note that **_init** is a mis-labeling, we're actually accessing the PLT which comes after the _init function in memory by chance).

(3) Type **"x/5b 0x080482b0"** (but insert the appropriate address for your binary if different) and *write down what the five bytes of machine code for the x86 instruction you wrote for problem #2 is*.

(4) Now type **"run"** to run to the breakpoint we set at main() and type **"disassemble printf"** to see printf disassembled from the library portion of memory. *Is the call to printf from main going directly to printf?* [Note, if your PLT entry disassembles as "something *0x80495d0" then it's loading a pointer from the address at 0x80495d0 and somethinging to that pointer, not somethinging (forgive me for verbing a noun) to 0x80495d0, use "x/1w 0x80495d0" to get that pointer]

(5) Now through a combination of **"dissasemble addr addr+20"** and **"x/1w addr"** track down where control flow is ultimately going. When you do the final disassemble command necessary you should see the name of the function. It looks like "_dl_map...", that's the first part of it, *write down the name of the whole function.*

(6) The _dl_map... function is part of a whole system that will resolve the printf symbol, which is undefined because the compiler did not know where the libraries would be loaded in memory. Set another breakpoint at the end of main (sometime after the printf, like at the "leave" or "ret" by typing **"break *addr"** where addr is the address of the instruction you want to breakpoint at. Run until this second breakpoint and then check out the PLT and GOT entries again. *Where would a second call (and all subsequent calls) to printf go to?*

(7) Quit out of gdb with **"quit"** and then run **"objdump -x helloworld"**. *Where are the PLT and GOT resolved to?* Does this jive with your answers to earlier questions? (If not you might have done the earlier questions wrong). *What does the line about the symbol printf say?*

(8) There's a very good reason why I did printf("%s", "Hello World\n") rather than printf("Hello World\n"). While the latter is not a security vulnerability, a printf(str) where str is a string variable that the attacker gives as input is an opportunity for them to break into your system

Check out the Wikipedia article on format string vulnerabilities:
http://en.wikipedia.org/wiki/Format_string_vulnerabilities

...and browse scut's paper if you're interested in more details:
http://doc.bughunter.net/format-string/exploit-fs.html

Typical format string exploits overwrite pointers on the stack, but suppose there was some defense mechanism against this. *Which structure that we've seen today, the PLT or the GOT, would be another place that an attacker could overwrite a function pointer and hijack control flow to execute malicious code?*

An example of this where the attacker wouldn't need to execute their own machine code is that if rename() is dynamically linked and so is execv() then an attacker could overwrite rename()'s pointer with execv()'s and then, assuming they were attacking an FTP server, they could ask to rename "/bin/sh" to some garbled pointer represented as a string and effectively get execv("/bin/sh", {0, "/bin/sh"}) which executes a shell and gives them access to the system. You can read the details of this example here (note that when I wrote this the PLT and GOT had only one level of indirection):
http://www.cs.unm.edu/~crandall/wuformat.txt

Didn't think Hello World in C could be so complicated, did you? ;-)