Global Information Assurance Certification Paper

A Real Vulnerability: Rogue System Libraries and Binaries
Manny D. Peterson
December 14, 2000

The goal of this paper is to heighten the awareness regarding a real vulnerability. The vulnerability will be described through three examples of how the vulnerability could be exploited by an attacker. This paper does not attempt to suggest a solution nor does it examine all of the possible attacks that may be mounted as a result of this vulnerability as they are both beyond the scope of this paper. It is the responsibility of the reader to investigate what measures should be taken to protect his or her respective environments.

First Example: An Introduction

Imagine this scenario. Some early Monday morning a system administrator arrives at work, boots his computer and steps away to fetch his morning mug of java. With mug in hand, he returns to his computer to be greeted by a screen very familiar to all of us, the Microsoft Windows sign-on screen. Almost reflex like, he enters his user account credentials (username and password) only to be presented with a dialog box stating, "No domain server was available to validate your password." Immediately, the system administrator tries to enter his user account credentials again with no improvement. After several tries and verifying the availability of the domain controller with his peers, the frustrated system administrator resorts to reloading Windows or using another computer. What our friendly system administrator may never learn of is the fact that he has just supplied his user account credentials to a rogue sign-on screen, which possibly either stored or forwarded his user account credentials off to some remote location. In fact, not only were his user account credentials captured, the user account credentials of his peers were also captured while attempting to help the system administrator by offering to try their user account credentials.

Second Example

Our first example describes a fictitious, yet possible, scenario where system binaries were manipulated to display a rogue sign-on screen for the purpose of capturing the user account credentials of a privileged user. In this example we will introduce a different method of gaining privileged access through the manipulation of system libraries. Although the attack used in this example is quite aged, it was very successful due to its simplicity.

Thanks to a concept in the field of computer sciences known as code reuse, this particular form of attack is possible. Code reuse is a concept employed by programmers to reduce program development time and storage requirements. Code reuse is implemented in the form of shared code. When one or more programs share code that serves a similar function it is often placed in a library. In Unix and Unix like operating systems these libraries are referred to as shared objects (.so). In Windows based operating systems they go by a different name, dynamic link libraries (.dll). When a program is executed and makes a call to shared code, the library is loaded and linked to the program in real time (runtime). Once the library is loaded and linked to the program, the call can be completed and control returns to the caller. Under normal circumstances, this is how things occur. However, what if we change the circumstances and tell the program to look elsewhere for its library or simply replace the library all together? This is where our next example comes in.

First we need a rogue library to stand-in for our legitimate system library. To do this, we write and compile our own. In this example we wrote a library that replaces a standard C library function, getpass(). The getpass() function under normal conditions writes a small message to the screen prompting the user to provide a password then reads the password the user supplies and returns it to the caller. However, our rogue getpass() function will do nothing of the sort. Looking at *figure 1*, you will notice when called, our rogue getpass() function simply writes a small cute message to the screen, flushes the IO stream and invokes shell (/bin/sh) as a privileged user (root).

Figure 1

```
#include    char *getpass(const char *prompt) {
   printf("Welcome to the promised land!\n");
   fflush(stdout);
   system("/bin/sh");
}
```

Next, we need to get our rogue library onto the target system. This can be approached two ways. If you have an account on the system already simply dump the rogue library in a temporary directory such as /tmp or if you do not have an account on the target system FTP the rogue library into the FTP incoming directory.

Now we have a rogue library and it is in place on the target system. Next we need some privileged process to call it. To do that we must first convince the system to load our rogue library instead of the legitimate one, which can be done with the help of an environment variable used by the linker loader ld.so, LD_PRELOAD. If we have an account on the target system we can simply start the telnet client, set the LD_PRELOAD variable to point to the location of our rogue library (/tmp) and open a session to localhost. If we do not have an account on the target system, it is just as easy as if we did. Start the telnet client, set the LD_PRELOAD variable to point to the location of our rogue library on the target system (ftp/incoming) and open a session to the target host. Once you are connected you will be presented with a short message and given a shell prompt. You will have only 30 to 60 seconds before the telnet daemon times out, just long enough to setup camp.

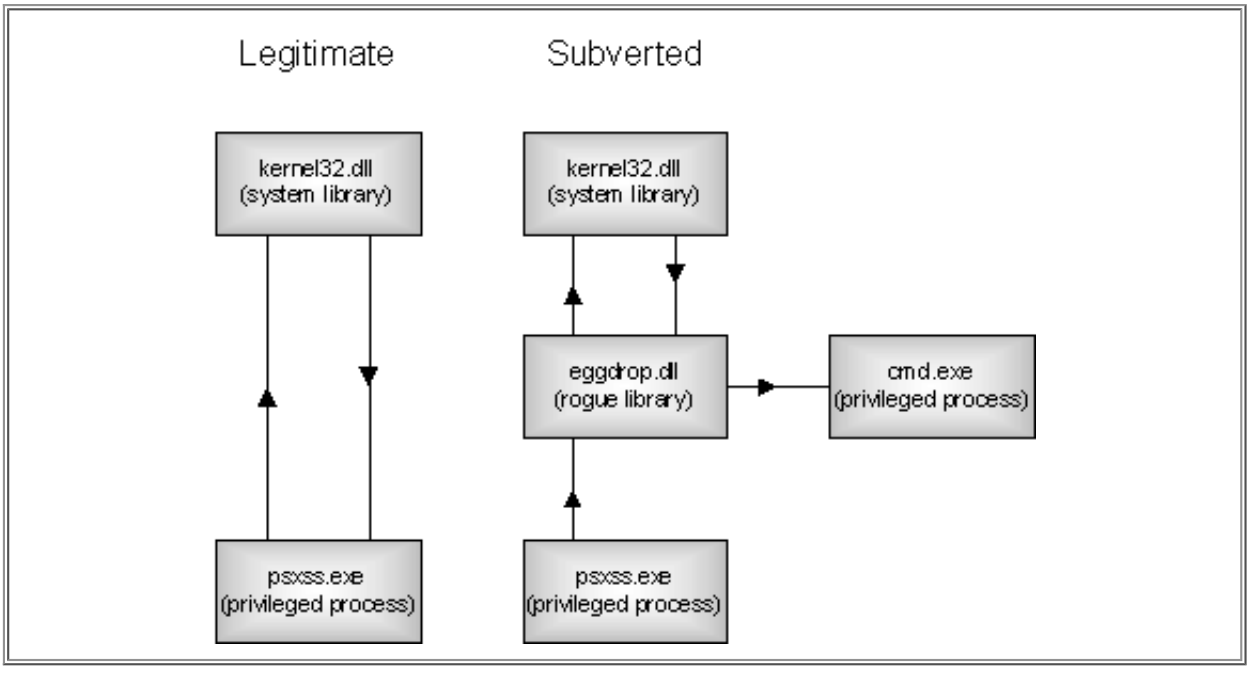
Third Example

Up to this point we have introduced two methods for manipulating system libraries and binaries for the purpose of gaining privileged user access. In the next example will reinforce what has been intruded in the previous example by describing a similar method of manipulating system libraries, which applies to the Microsoft Windows NT Operating System.

As a measure to improve the performance of applications when calling system libraries in the Microsoft Windows NT Operating System, Windows NT preloads the system libraries into a cache of file-mapping objects within the internal object namespace upon startup. Thus, when an application needs to call a function located with a system library, Windows NT does not need to read the location of the library from disk. Now, backing up briefly, Windows NT appropriately controls access to system libraries residing on disk. That is, with the help of properly configured NTFS access control lists. However, when Windows NT preloads the system library file-mapping objects into its internal object namespace it does something quite unexpected. Windows NT grants the Everyone group full control of the system library file-mapping objects cached within the internal object namespace. Thus, allowing an attacker to manipulate the file-mapping objects of system libraries. How is this done? Simple if you are familiar with the details of coding Win32 dynamic link libraries.

First, we need a rogue library to stand-in for the legitimate system library. The rogue library only needs to contain a function commonly called by Windows NT processes and hopefully, processes running as a privileged user. In the advisory published by L0pht, the function used is DllMain, a function common to all Windows NT libraries. A unique property of the L0pht's rogue library called eggdrop.dll is that it also acts as a DLL wrapper. Simply put, when the rogue library is called it forwards the call to the legitimate system library. Once the forwarded call to the legitimate system library returns, the rogue library completes the additional tasks necessary to achieve its goal, privileged access. Below, *figure 2* details the flow of a legitimate function call and a function call where a rogue library has been introduced.

Figure 2



Next, we need a small injector program to remove the legitimate system library file-mapping object from the cache and replace it with the rogue library. The L0pt uniquely calls this process poisoning the DLL cache.

With the rogue library and injector program in hand, providing all files are in place on the target system, we are ready. We first run our injector program to delete the legitimate system library file-mapping object from the cache and load our rogue library in its place. Now, all we need is a privileged process to call our rogue library. Say hello to PSXSS.EXE, our friendly POSIX subsystem (the OS/2 subsystem will work just as well). Everytime a subsystem is launched it runs as the local system account (NT AUTHORITY\SYSTEM). Immediately upon launch the POSIX subsystem calls our rogue library, which in turn calls the legitimate library. The legitimate library returns and leaves us with a shell (CMD.EXE) running as the local system account.

Summary

This paper described three methods for manipulating system libraries and binaries for the purpose of gaining privileged user access to a system. Two of the examples are real and were not included to hone your hacking skills. Rather, while reading through the examples given -- hopefully you have become more aware as to the ways an attacker could use similar methods to attack your system. I would highly recommend contacting the vendor of your system to find out ways you can safeguard critical system files and prevent their manipulation.

Lance J. Hoffman, "Rogue programs: viruses, worms, and Trojan" Wiley, John & Sons, Incorporated, November 1990 (12 December 2000)

"Telnetd Environmental Variable Passing Problem." (10 October 2000) URL: http://www.insecure.org/sploits/telnetd.LD_PRELOAD.enviropassing.html (12 December 2000)

"l0pht Security Advisory." (18 February 1999) URL: http://www.l0pht.com/advisories/dll_advisory.txt (12 December 2000)

"Somarsoft – Windows NT Security Issues." URL: http://escert.upc.es/cert/tech_tips/security.thml (12 December 2000)

"Telnetd Vulnerability." URL: http://www.outpost9.com/exploits/telnetdvuln.html (12 December 2000)