Dynamic-Link Library Hijacking

Max “RIVAL”
www.siteofmax.com
rival@riseup.net

Abstract

The aim of this paper is to briefly discuss DLL Hijacking vulnerabilities and the techniques used to mitigate and fix them. This paper is aimed towards people with a basic understanding of Dynamic-Link Libraries and how they can be used in applications, however provides certain points of information for those who do not.

1. Introduction

Windows is statistically the most used Operating System around with Windows 7 taking around 55% of users. However, this does not mean that Windows is completely secure. DLL Hijacking, also known as DLL Preloading, is a fairly recent discovery from H.D. Moore in 2010 and although DLL hijacking is considered a big topic, there aren't a lot of resources that detail both the effects and the methods of defense for both users and developers. DLL Hijacking Vulnerabilities affect a lot of business applications and so can be of significant importance to security researchers and penetration testers. These vulnerabilities have also been known to be found in applications such as Windows Movie Maker and Windows Address Book [1]. In order to understand DLL Hijacking, you must understand how Windows Applications find their DLL files if not given a full path.

2. Windows DLL

Dynamic-Link Libraries are collections of data and executable code that are used by other applications and DLL files. The main reasons for using DLL files instead of just putting the functions and data into the executable itself are to both make it easier to update the software (instead of requiring any user wishing to update to re-download the entire executable, you'd only need to re-download one or two small files. You can imagine how useful this is for designers of games that may need to be frequently updated) and to reduce excess memory usage by allowing the DLL file's code to be shared between applications.

3. Windows DLL Search Order

Programmers often don't specify an absolute path to the DLL file they want to use. This would have caused the problem of the DLL not being found and used, however Microsoft came up with the Dynamic-Link Library Search Order, which runs at the application's load time, to solve this issue. By default, the first item found is the item that is used. The search order is as follows:

Directory of the Application
⇩
Current Directory
⇩
System Directory
⇩
16-bit System Directory
⇩
Windows Directory
⇩
Directories in PATH variable

Above: Diagram of Windows DLL Search. Depending on settings or your OS version, sometimes the path specified by lpPathName (for example, from the SetDllDirectory function) is used instead of the Directory of the Application. If SafeDllSearchMode is enabled, then the search order also differs so that “Current Directory” comes fifth in the search order.

4. Principles of DLL Hijacking

Now we understand how Windows goes about searching for the DLL file itself, we can understand DLL hijacking. For example, let's say the application requires “functions.dll”, a file that has not been specified with an absolute path, however is located within the System Directory. An attacker could place their own DLL files in an area accessed BEFORE the systems directory (such as the directory of the application). When a user opens the application, during the DLL search process, it will discover the attacker's file before it comes across the DLL in the Systems Directory and thus will load it instead, meaning any malicious code from the attacker will be executed.

5. Finding DLL Hijacking Vulnerabilities

Finding the vulnerability in a program is the first step in exploiting it. A good way to do this is to use Process Monitor to see when the program executes a search for a DLL file. When Process Monitor is loaded up, you can try to trigger a function from another DLL or wait until one is triggered. After this has been done, all that's required is to go to the Filter menu and add your filters. Below are the filters you should consider using to make your search a lot shorter:

- *Operation is QueryOpen then Include*
- *Process Name is vuln.exe then Include*
- *Path contains .dll then Include*

If you find something that looks similar to this:

Path	Result
C:\Windows\System32\vuln.dll	NAME NOT FOUND
C:\Windows\System\vuln.dll	NAME NOT FOUND
C:\Windows\vuln.dll	NAME NOT FOUND

Then you have most likely found a DLL hijacking vulnerability.

6. Finding function names from a DLL

In order to create a new DLL with malicious content, we must first know the function names that are used in it. On Windows this can be done with the DUMPBIN utility. With DUMPBIN we can use the /EXPORTS option. Below is an example of the output of an example DLL:

```
Dump of file C:\example.dll
File Type: DLL
  Section contains the following exports for example
    00000000 characteristics
    4FC31DEF time date stamp Mon Jun 05 18:32:49 2013
        0.00 version
           1 ordinal base
           3 number of functions
           3 number of names
    ordinal hint RVA      name
          1    0 00007BA0 output_data
          2    1 00007C40 show_integer
          3    2 00008940 add_integers
  Summary
        1000 .CRT
        1000 .bss
        1000 .data
        1000 .edata
        1000 .idata
        1000 .rdata
        1000 .reloc
        9000 .text
        1000 .tls
```

An alternative is the use of a debugger. Debuggers can show you plain text that is stored in the program. In our case, we will be able to see the name of any referenced DLL file stored as plain text.

7. Writing DLL exploits

Let's jump into the practical side of DLL Hijacking. Let's say we have a program that reads a particular format and uses the following segment of code to load a DLL:

```
DLL_FILE = LoadLibrary("test.dll");
```

The following code is used to execute the "output_text" function:

```
output_text = (DLLPROC)
GetProcAddress(DLL_FILE, "output_text");
```

A segment of the DLL code is as follows:

```
void output_text()
{
    cout << "All is going well!\n";
    return;
}
```

For this example we will assume the DLL file "test.dll" is located in the Windows directory. As we have not specified a full path, the search algorithm commences. The program finds the DLL in the Windows directory and executes the output_text function. The following is output:

```
All is going well!
```

We can create a new DLL file with the same function name as follows:

```
void output_text()
{
    cout << "Something's not right here!\n";
    return;
}
```

If we place our new file in the same location as one of a file the program is intended to open, when the application is executed via the file, the program will search in the folder that has called the program before the Windows directory. You can imagine how much of an impact this vulnerability can have. It's unlikely the victim will pay much attention to a DLL file that comes with their wanted file as they have no reason to fear it. When the application loads the wanted file, it will also load the attacker's DLL and execute the malicious function.

8. Attacking a victim remotely

A lot of people confuse DLL hijacking with replacement, where the user would have to replace the original file for the program itself. However, with hijacking that is not necessary as all that's required is for the user to execute the program from a location where the attacker's DLL is. It's all well and good saying that the user needs to have the DLL already on their computer in order for it to be used against the target, however it doesn't seem very simple. This is where Social Engineering can come in handy. Below are two examples of how an attacker could use DLL Hijacking on a target:

1. An attacker finds a vulnerability in a text file viewer. The attacker creates an archive file with both a text file, and a malicious DLL. A victim downloads the archive and opens the text file directly from the archive itself. The archive program extracts both files to a location and opens the text file with the default text file reader (in this case, the vulnerable program). The program searches for the DLL and first checks for it from where the file was opened (in this case, the temporary location the archive program extracted the files to). It finds the attacker's DLL and uses the functions from that. The malicious code is executed.
2. An attacker finds a vulnerability in an email viewer used by the company they work for. He uploads the malicious DLL, a few random files, and an email file to a shared folder used by the people within the company. A victim opens the email file and the email reader uses the malicious DLL file instead of the intended one which could be located in the Windows Directory. The malicious code is executed.

9. Defending against DLL Hijacking

The most important part of explaining a vulnerability is explaining how to defend against it. The responsibility of protecting the users falls to the developers of software themselves. Since this has brought a lot of attention towards the Windows Search Order, developers should learn to rely less upon the Operating System itself for support. As is the case with DLL Hijacking, the full path to the library needs to be specified in order to avoid this problem entirely. The alternative to this is to move the DLL files to a place that is checked earlier in the search order, however I myself do not recommend this option [2]. Yet another alternative is to create a checksum hash of the DLL and store it within the program itself so that on execution, when a DLL is found, it's checksum hash value must match the hash stored in the program. This can be improved by using a strong hashing algorithm alongside a salt or by using public and private keys in the program to encrypt the checksum outside the program and decrypt it within it. The users themselves also have the ability to protect themselves by taking certain precautions. The user can ensure that the files they are opening are not in the same directory as a DLL that may seem suspicious. For example, if a DLL comes with a picture you've downloaded, it's a good idea to remove the DLL before opening the picture. Users also have the ability to alter the registry key *CWDIllegalInDllSearch* [3]. Users can set two different registry keys themselves:

- *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\CWDIllegalInDllSearch*
- *HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\binaryname.exe\CWDIllegalInDllSearch*

Hopefully this has allowed you to understand DLL Hijacking and the algorithm used by the Windows Operating System. The more developers that are aware of this problem, the less likely it is that they will rely solely on the Operating System to handle processes such as this.

10. References

[1] Microsoft Security TechCenter, "Microsoft Security Advisory (2269637)", *http://technet.microsoft.com/en-us/security/advisory/2269637*

[2] Swiat, "More information about the DLL preloading remote attack vector", Technet, August 23rd 2010 *http://blogs.technet.com/b/srd/archive/2010/08/23/more-information-about-dll-preloading-remote-attack-vector.aspx*

[3] Microsoft Support, "A new CWDIllegalInDllSearch registry entry is available to control the DLL search path algorithm", *http://support.microsoft.com/kb/2264107*