Rearing its Seven Ugly Heads: the DLL-Preload Attack

Version: 0.1

Research and Analysis:	**Haifei Li**	hfli@fortinet.com
Contributor and Editor:	**Guillaume Lovet**	glovet@fortinet.com

Abstract

In computer science and fashion alike, comebacks are often unavoidable, yet not always desirable (think “mullet”). But while the vagaries of fashion are impenetrable, trends in computer security follow logical rules. For instance, the implementation of mitigation technologies in modern OS (such as DEP and ASLR on Windows) has made leveraging a memory corruption bug into a working exploit a tremendously difficult task today. As a consequence, ancient exploitation techniques that don’t rely on memory corruption seem to become popular again. The DLL-Preload Attack is one of such.

This attack relies on a MS Windows system feature, which, in certain circumstances (read: when an application developer lacked caution or knowledge. But who never does?) can be abused to achieve escalation of privilege. Combined with either another exploit or simply a pinch of social engineering, it can even perfectly lead to execution of arbitrary code on the system by a remote attacker.

This paper’s aim is twofold: raise awareness on the issue - although this attack is not new, many applications, including very high profile ones, are subject (i.e. “vulnerable”) to it - and foster best practice for developers and testers.

Both aims are addressed by highlighting 7 typical mistakes in the development/QA process of applications that lead them to be vulnerable, identified via the analysis of 7 previously un-released instances of the vulnerability, in the following applications/OS: [pending disclosure], [pending disclosure], [pending disclosure], [pending disclosure], [pending disclosure], [pending disclosure], and [pending disclosure].

The paper concludes on the responsibility of application vendors in future instances of the vulnerability, as a system-wide solution that would not break backward compatibility is unlikely to exist.

I. Reminder: That attack principles

The attack actually relies on very simple principles, which boil down to abusing the flexibility provided by MS Windows when it comes down to using Dynamic Linked Libraries.

1. The Good: Implicit and Explicit Dynamic Linking

Indeed, on MS Windows, there are essentially two ways a process can dynamically load a library (.dll files) and map it into its address space:

- Either the library is loaded when the process itself is booted up by the Windows Loader - in which case the latter takes care of basically everything, including relocation, based on the information embedded in the PE header of the process' executable file. This is called **implicit linking**.
- Or the library is loaded "on demand" when the process explicitly issues calls to the Windows API functions dedicated to that task, such as *LoadLibrary*. This is called **explicit linking**.

Both methods have their pros and cons, but it is of course out of the scope of this paper to discuss those; it suffices to say that they provide programmers with flexible and varied ways to make use of shared functionalities bundled in libraries.

2. The Bad: Standard Search Order

One aspect of such a flexibility is that programmers do not actually need to provide the full path to a dll in order to have it loaded by the system at run time: Upon providing the sole name of the dll, the System will look for it in various places, in a pre-determined order. This order is called the **Standard Search Order** (SSO).

It is worth noting that in both implicit and explicit linking scenarios, the system resorts to the SSO to "resolve" a dll location from its bare name. Since MS Windows SP2, a System mode called "SafeDllSearchMode" is enabled by default. In which case, the SSO consists in the following:

1. The directory from which the application loaded.
2. The system directory.
Use the GetSystemDirectory function to get the path of this directory.
3. The 16-bit system directory.
There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory.
Use the GetWindowsDirectory function to get the path of this directory.

5. The current directory (**CWD**)
When an application is associated with a type of file (typically, a document or a media file), it may be automatically launched when action is taken on such a file (eg: double-clicking). The CWD is the directory where that file sits.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key.

The SafeDllSearchMode is controlled by the registry key:
HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode

However, despite its name, it doesn't make one safe from DLL-preloading attacks, as we will soon find out.

3. The Ugly: SearchPath

Although - as we've seen above - calling *LoadLibrary* with a bare name includes a location resolution operation by the System, it is also possible for programmers to call the location resolution part "manually" first, and then pass the returned full path to *LoadLibrary*. Example:

```
SearchPath(NULL, "test.dll", NULL, sizeof(szReturnBuff), szReturnBuff, (LPTSTR *)&tmp);
LoadLibrary(szReturnBuff);
```

Since a simple *LoadLibrary("test.dll")* would achieve basically the same goal, *SearchPath* is likely a "legacy" API function; surprisingly, it is however not totally absent of modern code.

The only difference with a direct call to LoadLibrary is that its search order is slightly different from the SSO:

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

Astute readers have probably spot the difference already: The CWD is in position 2 instead of position 5. In the scope of DLL-Preload attacks, it is interesting in the sense that it facilitates them, as we are about to see.

4. Typical attack scenarios

Indeed, the other locations in the search orders requiring a high level of privilege to be written to,

the most typical attack scenarios leverage the CWD.

Basics

Such a scenario may be broken down in the following steps:

- The attacker places a malicious dll in the same folder as a document or media file.
- The victim double-clicks on that file.
- The associated application is therefore launched with CWD set to the folder containing the file
- Likely, the associated application will require (implicitly or explicitly) some dlls to be loaded, and possibly, without giving their full path. Location resolution will therefore ensue.
- As a consequence, if the requested dll has the same name as the malicious dll and sits in a place that is searched **after** the CWD during resolution, the malicious dll will be loaded instead, hence permitting execution of arbitrary code with the privileges of the application.

This can be used for local privilege escalation, or for remote exploitation. At this point, the reason why SearchPath facilitates exploitation is now obvious: in the case of a direct call to LoadLibrary, places that are searched after the CWD during resolution boil down to “The directories that are listed in the PATH environment variable”, according to the SSO.

While in the case of a manual call to SearchPath , they consist in anywhere else than “The directory from which the application loaded”.

Challenges

An attacker is therefore confronted with a threefold challenge. For her attack to be successful, she must:

- Find a vulnerable application, that is to say an application that omits full path names for dlls that will happen to be located in a place coming after the CWD in the search order.
- Drop the malicious dll and a document/media file associated to the vulnerable application in a place accessible to the targeted system’s file explorer (namely, Windows Explorer).
- Get the victim to double-click on the document/media file.

Points 1. and 3. above are not outrageously difficult to deal with, the former requiring a good debugger and patience, and the latter a pinch of social engineering. The “hard” part in a DLL-Preload attack used to be placing the malicious dll on the victim’s file system. However:

- This can be achieved too by social engineering. A dll file does not trigger the same defiance from the victim than an executable.
- This can be achieved (and actually, has been achieved in the past, see for instance the Safari "Carpet Bombing" attack [4]) via a vulnerability in another application.
- In August 2010, security researchers from Slovak company Acros, while releasing a DLL-Preload flaw in iTunes they deemed "iTunes binary planting" [1], exposed an attack scenario where the malicious dll and the media file were located on a remote network share. They pointed that thanks to WebDAV, such shares were accessible via the internet from a default Windows configuration; this attack vector of course makes point 2 above significantly easier to deal with.

II. Mitigation

Microsoft responded to the scenario leveraging remote shares by releasing a security advisory essentially recommending to disable WebDAV [2]

The Redmond firm has been aware of the issue for quite a long time. As we have seen, they provided following mitigations as well as efforts.

1. SetDllDirectory

SetDllDirectory is an API function that effectively modifies the SSO, both in explicit and implicit dynamic linking scenarios. Once it was called by a process, for all future calls to LoadLibrary, and all future location resolution done by the Windows Loader within that process, the SSO is:

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

The CWD is totally rubbed out of the picture, which effectively knocks out most DLL-Preload based attacks. Therefore, SetDllDirectory is an efficient way for developers to harden their applications against possible DLL-Preload attacks. However, they must bear in mind that:

- If at some point after calling SetDllDirectory, their application is meant to use the CWD for loading a dll, it is of course not going to work anymore.
- Setting the lpPathName parameter to NULL will disable the effect of SetDllDirectory, and the SSO will remain unchanged (thus including the CWD). Thus for CWD disabling

purpose, it is recommended to set the lpPathName parameter to the empty string ("").

- In any case, the CWD is knocked out of the SSO only after the first call to SetDllDirectory. In particular, it means that this mitigation method will have no effect on the SSO of implicitly linked dlls of the application process itself. It will only protect implicit linking of subsequently loaded modules.

2. MS09-014/015

In response to the combined release of the "IE7 DLL-Load Hijacking" vulnerability by security researcher Aviv Raff in 2006 [3] (a classical DLL-Preload vulnerability) and of the Safari "Carpet Bomb" attack [4] allowing an attacker to drop an arbitrary file on the target's desktop, Microsoft released security bulletins MS09-014 and MS09-015 on April 14th, 2009 [5].

Beyond fixing the vulnerabilties in IE7, those updates provided two novelties to address the DLL-Preloading issue:

- A defense-in-depth protection for IE. This essentially calls SetDllDirectory("") early in IE's process, so as to thwart any subsequent DLL-Preload attack, be it on IE itself or one of the modules loaded afterwards (ActiveX objects, etc...). Unfortunately, that defense-in-depth protection is not enabled by default, so as to not break possibly existing modules relying on the CWD, and it's up to each user to enable it (a non-trivial process for the average user, by the way, see the FAQ in the bulletin).
- A new API for developers called SetSearchPathMode. Once called with argument TRUE, all subsequent calls to the infamous SearchPath API in the same process will search through the SSO for location resolution, exactly like LoadLibrary would do. This makes SearchPath "less insecure", in the sense that in the SSO, the CWD is almost in the last position... rather than in second position.

It should be noted that as of writing there is not a defense-in-depth protection for other Windows applications (than IE) in system level, this has been confirmed by the Microsoft Security Response Center.

III. Seven Mistakes, Seven Applications

[Pending disclosure]

1. Dangerous legacy code

[Pending Vendor Patch]

2. Incomplete testing environment Pt1: platforms

[Pending Vendor Patch]

3. Untested installation option

[Pending Vendor Patch]

4. Incomplete testing environment Pt2: versions

[Pending Vendor Patch]

5. Use of application PATH variable to locate libraries...

[Pending Vendor Patch]

6. ...Or browser plugins

[Pending Vendor Patch]

7. Beyond "DLL" Preloading: an example of how not to use the CWD

[Pending Vendor Patch]

References

[1] http://www.securityfocus.com/archive/1/513190

[2] https://www.microsoft.com/technet/security/advisory/2269637.mspx

[3] IE7 DLL-load hijacking Code Execution Exploit PoC, Aviv Raff,
http://aviv.raffon.net/2006/12/14/IE7DLLloadHijackingCodeExecutionExploitPoC.aspx

[4] Safari Carpet Bomb, Nitesh Dhanjani,
http://www.dhanjani.com/blog/2008/05/safari-carpet-b.html

[5] Microsoft Security Bulletin MS09-014,
http://www.microsoft.com/technet/security/bulletin/ms09-014.mspx