ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 2, No 5, May 2013

A Survey on Windows Component Loading Vulnerabilities

U.Haritha and V.Lokanadham Naidu

Dept of Information Technology, Sree Vidyaniketan Engineering College, Tirupati, Andhra Pradesh, India

Abstract- **Dynamic loading is a mechanism by which a computer program can, at run time, load a library memory, retrieve the addresses of functions and variables contained in library, execute those functions, and unload the library from memory. Unlike static linking and load time linking, this mechanism allows a computer program to startup in the absence of those libraries, to discover available libraries, and potentially gain additional functionality. Operating systems generally provide two resolution methods, by specifying the full path or the filename of the target component. This common component resolution strategy has an inherent security problem. As only a file name is given, unintended or malicious files with the same file name can be resolved instead. Therefore this paper presents the various vulnerabilities and work related to dynamic component loading vulnerabilities.**

Keywords- **Dynamic loading, Static linking, Component resolution.**

I. INTRODUCTION

Much of the functionality of the operating system is provided by dynamic link libraries (DLL) [1]. Additionally, when a program is run on one of the Windows operating systems, much functionality of the program may be provided by DLLs. The use of DLLs promotes modularization of code, efficient memory usage, and reduced disk space. Hence, the operating system and the programs load faster, run faster, and take less disk space on the computer. DLL's contain modular pieces of code that developers can call upon within their applications to perform various functions [2] [3]. Windows itself relies upon the same type of architecture and contains a plethora of DLLs that perform various functions..

Along with the DLL's built into Windows, application developers often create their own DLLs that contain the functions used by the program. When it is done, the developer created DLLs are packaged and installed with the application. The problem occurs based on how DLLs are loaded by applications. By default, when an application doesn't have a statically defined path to a DLL, it goes through a process to find it dynamically. The application first searches the directory it was executed from, then the system directory, next the 16-bit system directory, the windows directory, the current directory, and then the directories listed in the operating system PATH environment variable. In searching these paths, the application will use the DLL it finds first.

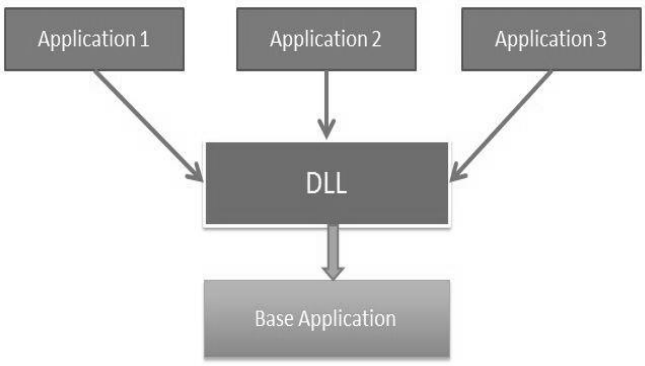

Fig 1.Applications using Dlls

For example when an application is executed, that must dynamically search for a required DLL upon loading. The application first immediately searches the path it was executed

from and finds a DLL that matches. Unfortunately for end user, the real DLL associated with the application is located in the Windows system directory [4]. The DLL that's been placed in the directory with the application is one that has been modified by an attacker to allow remote command shell access into the system. The application will never get to the real DLL because it's already found the match it needs.

II. DYNAMIC LOADING RELATED ATTACKS

Malicious component resolutions may cause an application to load unintended components. This issue had been known for a long time, but it had not been considered a serious threat because it requires local file system access on the victim host for exploitation. Recently, realistic attacks exploiting vulnerable component loading have been discovered, including the ones discovered by us. In this section, we describe these attack vectors.

A. DLL Hijacking

DLL hijacking [5] implies that either a DLL gets hijacked or something gets hijacked using a DLL. In large majority of binary planting vulnerabilities the binary (for instance, a DLL) in question does not exist - that is, until the attacker plants it. You can't hijack something that doesn't exist. A vulnerable application gets hijacked through a malicious *DLL* but then every vulnerability could be called hijacking of some sort. Before Windows XP SP2, the dynamic-link libraries search order [6] had the current working directory in the 2nd place that produced a lot of possibilities to actually hijack an existing DLL (e.g., one from the Windows system folder) by placing a malicious copy with the same name in the current working directory.

B. DLL Preloading

DLL preloading [7] implies that some presumably malicious DLL gets loaded in advance. We find no such advance-loading process taking place in the context of this vulnerability. The attack is that you find some DLL an application needs, make an evil twin, and put it in the same directory as a document, then provoke someone who you'd like to have running your code to open the document.

C. DLL Load Hijacking

DLL load hijacking [8] is a little better than DLL hijacking as it implies that it is the process of loading that gets

hijacked (and used for malicious purposes). And again - just like with insecure library loading -, loading is not a very suitable term for non-library executables (EXEs, COMs, etc.).

III. RELATED WORK

A. Systemic weakness in the COM security infrastructure

The Component object model (COM) is a software architecture that allows applications to built from binary software components.COM is a general architecture for component software, though Microsoft applying COM to address specific areas any developer can take advantage of structure and foundation that COM provides.

DLL Hell refers to the set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL) or a Component Object Model (COM) class [9]. Every COM object that is installed on the system is listed in the windows registry under a unique identifier called a Class ID (CLSID). On an average Windows system contains tens of thousands of COM objects registered. An application choosing to instantiate one of these objects simply needs to supply the corresponding CLSID and the desired type of interface to one of the appropriate Win32 APIs. Each of these APIs simply looks up the CLSID in the registry, loads the library listed in the registry for that object, and returns a handle to the object. Whichever interface was specified will allow access to a set of methods and properties corresponding to that interface.

B. Detecting unsafe dynamic component loadings

Dynamic loading is an important mechanism for software development as it allows an application the flexibility to dynamically link a component and use its exported functionalities. The problem of an unsafe dynamic loading had been known for a while, but then it had not been considered a serious threat because its exploitation requires local file system access on the victim host. This problem has started to receive more attention due to recently discovered remote code execution attacks. Taeho Kwon and Zhendong Su [10] presented an automatic technique to detect unsafe component loadings. This technique is implemented as a two phase dynamic analysis. The first phase, which is online, uses dynamic binary instrumentation to capture a program's sequence of events related to component loading. The second phase, which is offline, analyzes the captured profile to detect unsafe component resolutions. Their evaluation shows that unsafe component loadings are prevalent on both platforms and more severe on Windows platforms from a security perspective.

C. Windows API based vulnerability detection

Detection of zero day malware has been the great challenge for researchers from long time. Any program having the malicious intent can be classified as malware or computer infection program. It is a collective term for any malicious software that performs hidden unintended actions affects the integrity of the system and cause damage, loss without the knowledge of user.

The need for security is in fact a response to the increasing number of attacks led against information systems. Previous works and literature [11] [12] has shown that one single technique alone cannot detect all types of malware. Therefore researchers are working towards finding patterns or features which have unchangeable characteristics of malware even though the malware mutates or obfuscates itself. A statistical analysis of Windows API calls in malware reflects the behavior of a piece of code.

D. Discovery of API level exploits

Finding vulnerabilities in software components is different from finding exploits against them. Exploits that compromise the security often use several low-level details of the component, such as layouts of stack frames. A software component is vulnerable to an API-level exploit if its security can be compromised by invoking a sequence of API operations allowed by the component. Vulnerability in a software component is an error in its implementation that can possibly be used to alter the intended behavior of the component.

An exploit is a sequence of operations that attacks the vulnerability, with malicious intent and devastating consequences. Static analyzers such as BOON [13] and Percent-S [14] would benefit from an analysis that finds exploits for vulnerabilities they identify. These tools produce false positives because of imprecision in their analysis, and the process of classifying warnings as real vulnerabilities or a false positive is typically manual. A security exploit generated against a vulnerability identified by such tools offers several benefits. First, it provides evidence that the threat posed by the vulnerability is real. Second, the exploit can be used as a test case to stress the resilience of patched versions of the component. Finally, in cases where the analysis fails to produce a security exploit, the vulnerability can automatically be classified as a false positive, thus reducing the manual effort involved in classifying warnings.

E. Non-Control-Data Attacks Are Realistic Threats

Malicious attackers often break into computer systems by exploiting security vulnerabilities due to low-level memory corruption errors. Most memory corruption attacks follow a similar pattern known as the control-data attack: they alter the target program's control in order to execute injected malicious

code or out-of-context library. The attacks usually make system calls (e.g., starting a shell) with the privilege of the victim process. The possibility of these attacks has been suggested in previous work [15][16][17][18]. However, the applicability of these attacks has not been extensively studied, so it is not clear how realistic they are against real-world applications.

Control flow integrity may not be a sufficiently accurate approximation of software security. The general applicability of non-control-data attacks represents a realistic threat to be considered seriously in defense research.

IV. CONCLUSION

Windows is different from other operating systems is that it combines these two features; when a program instructs Windows to load a DLL, it looks in several different places for the library, including the current directory. When a file is loaded in Windows, it sets the current directory of the newly-started program to be the directory that contains the file. If an attacker knows the names of any of the DLLs the program tries to load, puts a DLL with one of those names adjacent to a data file, he can ensure that his DLL will be loaded whenever the program tries to open the data file. To detect vulnerable component loadings we have discussed some of the available papers to detect attacks bugs in the system. Therefore, these references could also help to detect a vulnerable component loading during the program execution.

REFERENCES

[1] http://en.wikipedia.org/wiki/Dynamic-link_library.

[2] http://sourceware.org/autobook/autobook/autobook_158.html.

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682599%28v=vs.85%29.aspx.

[4] http://www.vcode.no/web/resource.nsf/LFUG/249.htm.

[5] http://www.maravis.com/library/dll-hijacking-attacks/.

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682586%28v=vs.85%29.aspx.

[7] http://blogs.msdn.com/b/david_leblanc/archive/2008/02/20/dll-preloading-attacks.aspx.

[8] http://www.zdnet.com/blog/security/details-emerge-on-new-dll-load-hijacking-windows-attack-vector/7204.

[9] D. Dewey and P. Traynor, No Loitering: Exploiting Lingering Vulnerabilities in Default COM Objects, In *Proceedings of the ISOC Network & Distributed System Security Symposium (NDSS)*, 2011.

[10] T. Kwon and Z. Su, "Automatic Detection of Unsafe Component Loadings," Proc. 19th Int'l Symp. Software Testing and Analysis,pp. 107-118, 2010.

[11] Vinod, P., Jaipur, R., Laxmi, V. and Gaur, M., "Survey on Malware Detection Methods", Hack. 2009, 74.

[12] Mihai Christodorescu and Somesh Jha, "Testing Malware Detectors", in Proceedings of ISSTA'04, July 11 - 14, 2004, pages 33-44, Boston, MA USA, ACM Press.

[13] D.Wagner, J. S. Foster, E. Brewer, and A. Aiken. First steps towards automated detection of buffer overrun vulnerabilities. In *Proc. NDSS*. ISOC, 2000.

[14] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proc. 10*th *Security Symp.*USENIX, 2001.

[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium.* Washington, DC, August 2003.

[16] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA. October 2004.

[17] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[18] W. Young and J. McHugh. Coding for a believable specification to implementation mapping, In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.