Modular Behavior Profiles in Systems with Shared Libraries (Short Paper)

Carla Marceau and Matt Stillerman

ATC-NY, 33 Thornwood Drive, Ithaca NY 14850, USA
{carla, matt}@atc-nycorp.com

Abstract. Modern computing environments depend on extensive shared libraries. In this paper, we propose monitoring the calls between those libraries as a new source of data for host-based anomaly detection. That is, we characterize an application by its use of shared library functions and characterize each shared library function by its use of (lower-level) shared libraries. This approach to intrusion detection offers significant benefits, especially in systems such as Windows, much of which is implemented above the kernel as dynamically linked libraries (DLLs). It localizes anomalies to particular code modules, facilitating anomaly analysis and assessment and discouraging mimicry attacks. It reduces retraining after system updates and enables training concurrent with detection. The proposed approach can be used with various techniques for modeling call sequences, including N-grams, automata, and techniques that consider parameter values. To demonstrate its potential, we have studied how a DLL-level profiling IDS would detect two recent attacks on Windows systems.

Keywords: Anomaly detection, intrusion detection, behavior profile, shared libraries, dynamic link libraries.

1 Introduction

After ten years of research on host-based anomaly detection systems, anomaly detection is still a remote dream for applications that run on most desk-top systems. One reason for this is that modern applications, especially Windows applications, are huge and exhibit a very wide range of behaviors; as the set of legitimate behaviors grows, the probability of false negatives increases, as does the time needed to train a behavior profile. This problem is exacerbated by mimicry attacks [1], which imitate normal application behavior as seen by a given detector in order to defeat that detector. Second, as applications grow, training the anomaly detector takes longer. Worse, Windows systems are subject to frequent patches and updates, any one of which can invalidate the current behavior profile of an application and provoke retraining. Third, anomaly detectors indicate that something might be wrong, but they typically provide very little information for anomaly assessment and response. In particular, they cannot localize the anomaly to a specific program module, which might provide further information for assessment. For these reasons, most current approaches to application anomaly detection are unlikely to succeed for Windows applications.

In this paper, we propose a novel approach to application anomaly detection that addresses these difficulties. The basic idea is to exploit the use of shared libraries by applications to create profiles for the exported functions of each shared library. We model the behavior of the application by its calls to DLLs and the behavior of each DLL function by its calls to other DLLs. The result is a localized profile of each module (application binary or DLL). Figure 1 schematically represents this key idea. In the system of Figure 1, Application 1 is characterized by its calls to Kernel32.dll, AAA.dll, and BBB.dll. BBB.dll is characterized by its calls to CCC.dll, kernel32.dll, and DDD.dll.

It might seem that the use of shared libraries is too limited to profile applications. However, modern computing systems include extensive shared libraries that implement GUI components, display pictures, enable access to networks and databases, manage mail and other higher level protocols, and provide other reusable functionality. Much of the Windows operating system is implemented in well over a thousand DLLs that execute in user space and mediate access to the kernel. As one example, opening Outlook to open a single email exercises well over one hundred DLLs, of which up to five may be represented on the call stack at any one time. Furthermore, many vulnerabilities in Windows systems are located in DLLs, including a recently discovered vulnerability in the graphics rendering engine (gdi32.dll) that affects every Windows system shipped between 1990 and January 2006 [2, 3]. It is not surprising that most published Windows system vulnerabilities occur in DLLs, since DLLs are available for attackers to study and the payoff for cracking them (a large number of potential victims) is high.

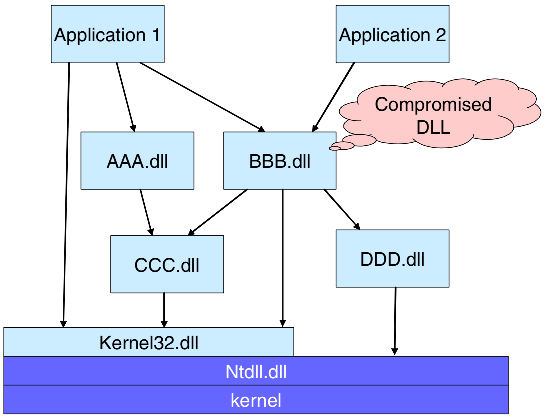

Fig. 1. Structure of a Windows application

This paper makes the following novel contributions:

- It defines DLL profiles and a class of intrusion detection systems based on DLL profiles
- It demonstrates that DLL profiles associate suspicious behavior with specific code modules

- It shows that identifying the locus of suspicious behavior opens new sources of data for analyzing anomalies
- It provides evidence that DLL profiles reduce false negatives and resist mimicry attacks
- It argues that DLL profiles can be used to minimize the burden of training and enable detection to proceed concurrently with retraining after updates

The paper is organized as follows. In Section 2, we review the structure of Windows processes and explain how DLL profiles can be used to detect anomalous program execution. In Section 3, we discuss related work. In Section 4, we briefly describe our experiments detecting two recent exploits on a small application. In Section 5, we substantiate the claimed benefits of DLL profiles. We conclude with suggestions for further research.

2 An Intrusion Detection Model Based on DLL Profiles

A Windows process comprises multiple (kernel-supported) threads, some of which are dedicated to GUI or system functions. Windows applications make extensive use of DLLs that implement the operating system and supply additional functionality. The Windows kernel API is defined by ntdll.dll. However, Windows applications rarely call ntdll.dll directly. Indeed, the Microsoft Visual Studio development environment does not support calls to ntdll.dll. Instead, kernel32.dll[1] defines the standard interface to the operating system, although a few DLLs call ntdll directly. Many calls to kernel32 are mediated through higher-level DLLs. As a result, the typical application cascades through layers of DLLs and results in multiple calls to ntdll and the kernel.

Ground-breaking work by Forrest, et al. [4, 5] showed that kernel-call traces capture application behavior. However, in systems and applications dominated by DLLs, much of the information in kernel-call traces characterizes the internal behavior of DLLs. Therefore, a single N-gram in such a trace often reflects the behavior of multiple DLLs. In the short execution of Outlook mentioned above, up to five DLLs at a time were represented on the call stack. Other characterizations of the behavior of the application as a whole also describe the combined behavior of many shared libraries.

In DLL profiling, we characterize each module (the application and the DLLs) by the calls it makes to other DLLs—not to the kernel. When one DLL calls another, their combined state can be represented with a *stack* of traces of calls between modules, one for each current invocation of a module. Figure 2 represents a snapshot of the stack. Each box represents a separate sequence that is *currently* being accumulated. In Figure 2, the most recent inter-module call by the application is to function f() in AAA, which in turn has called function c() in CCC. When function c returns, the current inter-DLL sequence for function c() is complete. If function f()

[1] The name “Kernel32” suggests that this DLL defines an interface to the kernel. Kernel32 provides very basic operating system functionality, but it accesses the kernel only through ntdll, which implements the kernel API. In this paper, we will commonly write DLL names without the .dll extension.

calls some other function in another DLL, a sequence for that function is pushed onto the stack. Note that since DLLs are reentrant, the stack may include multiple instantiations of a single module.

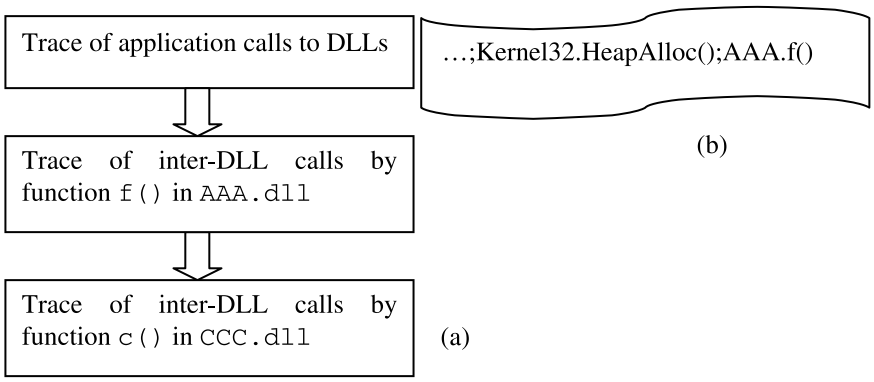
Trace of application calls to DLLs
Trace of inter-DLL calls by function f() in AAA.dll
Trace of inter-DLL calls by function c() in CCC.dll
(a)
…;Kernel32.HeapAlloc();AAA.f()
(b)

Fig. 2. (a) The stack of inter-DLL-call sequences in a thread. Each stack element is a trace (b) of calls from an exported function of a DLL (or the application main) to other DLLs.

DLL profiles support a new class of intrusion detection methods, depending on what information is recorded in the traces and the profile for each exported DLL function. For example, if the profile focuses on control flow, training traces record the identity of the called functions. N-grams, automata, or other methods may be used to represent the set of traces, as for kernel-call traces [4-11]. Alternatively, if the profile focuses on dataflow, the training traces can record not only the functions called, but also relations among the arguments to the function being profiled and the arguments of the functions it calls. The experiments described in this paper used N-grams, with N=6,[2] but most of our results are more generally applicable.

An IDS that uses the DLL stack model for intrusion detection can be realized in a straightforward way. We posit that the IDS maintains a profile of each function exported by a Windows system DLL, in addition to a profile of each application module (binary or DLL) to be protected. At run time, calls to each profiled DLL are captured, for example by mediating connectors [12, 13], and sent to the IDS. For each thread, the IDS maintains a stack of currently executing modules (DLLs or the main application). For each function in the stack, it records information about the external calls made by the function, as in Figure 2. When an exported function of a DLL is called from another DLL, the instrumentation informs the IDS of the call. The IDS notes the call in the trace at the top of the DLL stack for that thread, checks for anomalies against the profile of the calling function, and pushes a trace for the called function onto the stack. When the DLL function returns, its trace is popped off the DLL stack.

[2] Although Forrest's group used N=6 to model UNIX and Linux processes, a smaller value for N may be more appropriate for tracking behavior in terms of inter-DLL calls.

After an update to a DLL, the IDS continues to function but switches to training mode for the updated DLL. When an exported function from the newly updated DLL is called, the IDS pushes the DLL onto the stack, but instead of comparing the trace of the DLL function to the old profile, it collects the trace for input into a new profile. When the DLL function returns, the completed trace is added to the collection of traces for that function, and the profile creation module of the IDS processes it. At some point, the profile is deemed sufficiently mature to be used for detection. At that point, the IDS switches back into detection mode for that DLL function. Note that function profiles mature at different rates, depending on each function's range of behavior and on how frequently it is exercised.

3 Related Work

Much work has been done on profiling programs by sequences of calls, analyzing such sequences, and evading detection based on such sequences. The VtPath model of Feng et al. [14], who use much the same information as the process characterization of Figure 2. They exploit the call stack at each system call to record calls and returns between successive system calls. Like the VtPath model, DLL profiles are used to detect anomalies above the kernel-interface level. Our model differs from theirs in that (1) it records the thread history per calling DLL, rather than for the application as a whole, and (2) it is sparser in that it includes only calls between modules. At any one time, the expected number of functions on the DLL stack is much smaller than the number of functions on the call stack, because functions exported by a DLL are gateways to the DLL's entire functionality, much of which may be implemented in other functions. The exported function may make several calls within the DLL before some function makes a call to another DLL.

We note the difference between our approach and that of Sekar [15]; that approach characterizes an application as a whole by the sequence of its kernel calls augmented by a notation of the origin of the kernel call in the application itself. With Sekar's approach, it is possible to avoid characterizing library functions and focus on the behavior of the application itself; our approach also characterizes the application per se. However, by characterizing the intermediate shared library functions, we are able to identify attacks aimed precisely at these libraries. Indeed, this accounts for a very large number of attacks on Windows systems. Note that both [14] and [15] employ stack tracing in Linux to obtain data for the analysis. In Windows systems, stack tracing is often infeasible because of stack optimization, in which the compiler may use idiosyncratic stack structures within a DLL.

4 Experiments with DLL Profiles

To investigate DLL profiles, we created a DLL profile for a small Windows application and used it to detect two recent exploits. In this section, we describe the experiment.

The first exploit, introduced in the Fall of 2004, exploits a vulnerability present in most versions of gdiplus [3] up to Windows XP, Service Pack 1. It causes a heap overflow when gdiplus is used to display a malicious JPEG image. To study the exploit, we instrumented ImgViewer/32 [16], a freeware application for viewing pictures in GIF, JPEG, and other formats. Like many image viewing applications, ImgViewer/32 relies on Microsoft's graphics processing DLLs, gdi32 and gdiplus, and hence is vulnerable to the attack.

The gdiplus attack, as described in [17], occurs in two stages. In the first stage, a specially-crafted JPEG image header causes function GdipGetPropertyCount() to overwrite the heap with code contained in the "comments" section of the header. Later, during execution of the function GdiplusShutdown(), the overwritten code is executed. The version of the exploit that we used [18] takes advantage of the heap overflow to create a new user with administrative privileges.

Our experiment was conducted as follows. We created a profile of normal behavior by exercising the ImgViewer application on harmless JPEG images in thirty training runs. We then ran the application with a malicious image. The exploit traces were compared with the profiles to find anomalies, and the anomalies were analyzed. We obtained examples of harmless anomalies by exercising the ImgViewer application with JPEG comments against a profile that excluded images with comments.

The ImgViewer application exercises the application binary and 24 DLLs in several threads; we monitored only threads that were governed by the application, which used 14 DLLs. Using those threads, we constructed profiles as described in Section 2. Individual profiles were expressed as sets of N-grams.

We also profiled the effect of the recently discovered WMF exploit [19]. In January 2006, a vulnerability in gdi32.dll was discovered that had existed in all Windows systems since 1990. The vulnerability, which lies in the part of gdi32 that displays WMF pictures, enables a picture to specify arbitrary code to be executed when the picture is displayed. To exercise the vulnerability, we created a small WMF exploit that simply halts the process when invoked. We then used DLL profiles based on a short training period to detect the WMF exploit. Training consisted of the previous thirty executions of ImgViewer on JPEG images, followed by three executions of ImgViewer on benign WMF images.

We discuss the first example in some detail in Section 5. Results from the second example were similar.

We used two types of instrumentation in our experiments. Our first efforts were performed using mediating connectors [12, 13], which are wrappers placed at the point of entry into functions exported by a DLL. These connectors are ideal for intercepting calls into ntdll, but using them to capture calls from modules requires that the signature of each exported function of each DLL be known in advance. An alternative is to start with an application and automatically instrument each DLL as it is invoked; the result is a cascade of wrappers. We have implemented such a cascade and used it for our experiments. When the application or a DLL is linked, the instrumentation modifies its import table so that when a call is made, the instrumentation obtains control and writes a log entry.

5 Benefits of DLL Profiles

In this section, we claim several benefits for DLL profiles and illustrate them with experimental evidence from the gdiplus experiment.

Localization of anomalous behavior to code modules. The gdiplus exploit manifested in anomalies in traces of five exported DLL functions. First, GdipGetPropertyCount, in which the heap overflows, exhibited many calls to five functions that were not in its profile. Second, GdiplusShutdown, which executes the attack code, exhibited four anomalies—all to novel functions—as shown in Table 1. Third, during the execution of GdipGetPropertyCount, the HeapAlloc() function of kernel32 exhibited a call to RtlUnwind, which unwinds the stack after an exception. RtlUnwind() did not appear in the RtlUnwind's profile.

Table 1. Anomalous calls from GdiplusShutdown during the attack

DLL	Function	Comment
kernel32	LoadLibraryA	Loads netapi32
netapi32	NetUserAdd	Adds a new user for the machine
netapi32	NetLocalGroupAddMembers	Gives the new user privilege
kernel32	ExitProcess	End application

Two other functions—CoCreateInstance() in ole32.dll and NdrClientCall2() in rpcrt4.dll—also exhibited anomalies with respect to the available profiles. However, the profiles of those functions had not converged by the end of training. An IDS based on DLL profiles as described above would still be accumulating these two profiles. Thus, it would not (yet) be using them for detection.

False negatives and resistance to mimicry attacks. The DLL functions we have profiled typically have a narrow range of behavior. 90% of all traces are of length 6 or less, and half call just one other function. This dramatically reduces the chances of a false negative, since it is unlikely that attack behavior happens to fall into the narrow range of the function's normal behavior. For example, GdiplusShutdown() normally executes 10 functions in 2 DLLs, as shown in Table 2.

The narrow range of normal behavior reduces the probability of false negatives and makes mimicry attacks infeasible by making the target much smaller: 2 DLLs instead of 14 and 10 functions instead of over 800. Consider the gdiplus exploit, for example. Our exploit payload created a new user through calls to the netapi32 DLL. A clever attacker will avoid such blatantly malicious behavior, but will find himself constrained by the normal profile of the vulnerable function, in our case GdiplusShutdown. A mimicry attacker has to find a function that is not only vulnerable but also enables the desired functionality. In the case of GdiplusShutdown, it is hard to imagine any malicious behavior (other than crashing the application) that an attacker could accomplish using the functions in Table 2.

Anomaly analysis. Localizing anomalies to one or more DLLs makes it possible to draw on knowledge about the DLLs to analyze anomalies. Anomaly analysis in real time helps the IDS decide whether to treat the anomaly as novel application behavior or an attack. For example, the WMF exploit, described earlier, was in operating system code that had been stable for fifteen years. Suppose that an IDS based on

DLL behavior, as described here, had been in use for that whole time. The behavior profiles for that DLL would have been stable for much of that time. Thus, the anomalous behavior, when it appeared, would be very suspicious, in contrast with anomalies from a DLL whose profile has only recently converged.

Table 2. Functions invoked by GdiplusShutdown

DLL	Function	Comment
Kernel32	EnterCriticalSection	Wait for mutex object
	LeaveCriticalSection	Release mutex object
	SetEvent	Signal on (parameter) event
	WaitForSingleObject	Wait on locked object
	CloseHandle	Release object
	DeleteCriticalSection	Release resources for mutex
	HeapFree	Free block in heap
	HeapDestroy	Destroy user-created heap
Gdi32	DeleteObject	Delete object created by gdiplus
	DeleteDC	Delete gdi32 device context

Anomaly analysis can also consider the distance of the anomaly from the profile. For example, the gdiplus exploit called two functions in netapi32, which creates new users; netapi32 does not appear in the profile. Other factors of interest are the provenance and change history of the DLL.

Anomaly analysis can sometimes use information about functions to estimate their potential harm. A call to NetUserAdd() in the context of gdiplus is highly suspicious. We obtained examples of harmless anomalies by exercising the ImgViewer on JPEG images with comments against a profile based on images without comments. This resulted in three anomalous calls to the two functions of Table 3. An IDS armed with knowledge about common functions could guess that these two functions, which collect information about a device (the screen), are probably benign.

Table 3. Harmless anomalies

DLL	Function	Description
user32	GetDC	Retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen.
gdi32	GetDeviceCaps	Retrieves device-specific information for the specified device, specified by a handle.

Easing the burden of training (and retraining). In a system whose architecture is dominated by DLLs, when one DLL is updated the system-call profiles of all applications that use that DLL are invalidated, and must be retrained. In contrast, an IDS based on DLL profiles can retrain just the profile of the one changed DLL, and can continue to use all other profiles mature profiles for detection. Retraining a given DLL occurs less frequently and is quicker than training an entire application; it also allows detection of anomalies in other DLLs to continue while the updated DLL is being trained.

The rate at which profiles converge is apt to vary from one DLL function to another. In our experiments, the profiles for two functions never converged; we were nevertheless able to feel quite confident about anomalies detected in other functions whose profiles had converged quickly and unambiguously.

6 Conclusion and Future Work

We have presented a novel approach to host-based anomaly detection that relies on profiles of functions exported by shared libraries. We have argued and shown evidence that such profiles reduce false negatives, localize anomalies to code modules and provide opportunities analyze them, and reduce the burden of training.

Much research remains to be done to realize the potential benefits of DLL profiles. A major question is the performance cost of deploying various types of profiles and how to minimize that cost. Our preliminary measurements with Outlook suggest a performance penalty of 5-10%. This number assumes an IDS in which most anomaly checks are simple (like a table lookup) and the remainder represent state changes.

In addition, an IDS system based on DLL profiles faces non-trivial "bookkeeping" and security challenges. To make retraining practical, ways to base updated function profiles on the profiles for the previous version must be developed.

Finally, DLL profiles open new possibilities for anomaly analysis. Additional information about shared libraries and their functions, as well about connections between libraries (including static analysis of binaries) may lead to algorithms and heuristics for estimating the potential harmfulness of a large class of program anomalies.

Acknowledgements

This work was supported by the Army Research Office under contract DAAD 19-03-C-0060. We are grateful to Matthew Donovan and Ian Lenz for technical assistance. Mediating connectors used in the experiments were provided by Teknowledge Corporation.

References

1. Wagner, D. and P. Soto. "Mimicry Attacks on Host Based Intrusion Detection Systems," in *Proceedings of the Ninth ACM Conference on Computer and Communications Security*. 2002.
2. Allison, K., "Windows PCs Face 'Huge' Virus Threat," in Financial Times, January 2, 2006.

3. Microsoft (TM), "Microsoft Security Bulletin Ms05-053: Vulnerabilities in Graphics Rendering Engine Could Allow Code Execution (896424)," http://www.microsoft.com/technet/security/bulletin/MS05-053.mspx.
4. Forrest, S., S.A. Hofmeyr, and A. Somajayi. "A Sense of Self for UNIX Processes," in *Proceedings of the IEEE Symposium on Computer Security and Privacy*. 1996: IEEE Press.
5. Hofmeyr, S.A., S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security*, 1998. **6**(3): p. 151–180.
6. Debar, H., et al. "Fixed vs. Variable-Length Patterns for Detecting Suspicious Process Behavior," in *Proceedings of the ESORICS 98, 5th European Symposium on Research in Computer Security*. 1998. Louvain-la-Neuve, Belgium.
7. Warrender, C., S. Forrest, and B. Pearlmutter. "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the IEEE Symposium on Security and Privacy*. 1999, pp. 133-145.
8. Ghosh, A.K., A. Schwatzbard, and M. Shatz. "Learning Program Behavior Profiles for Intrusion Detection," in *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*. 1999. Santa Clara, California.
9. Marceau, C. "Characterizing the Behavior of a Program Using Multiple-Length N-Grams," in *Proceedings of the New Security Paradigms Workshop*. 2000. Ballycotton, Ireland.
10. Pfleger, K. "On-Line Cumulative Learning of Hierarchical Sparse N-Grams," in *Proceedings of the International Conference on Development and Learning*. 2004.
11. Michael, C.C. and A. Ghosh, "Simple, State-Based Approaches to Program-Based Intrusion Detection," *ACM Transactions on Information and System Security*, 2002. **5**(3): p. 203-237.
12. Balzer, R. and N. Goldman. "Mediating Connectors," in *Proceedings of the ICDCS Workshop on Electronic Commerce and Web-Based Applications*. 1999. Austin, TX, pp. 73-77.
13. Balzer, R. and N. Goldman. "Mediating Connectors: A Non-Bypassable Process Wrapping Technology," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. 1999.
14. Feng, H., et al. "Anomaly Detection Using Call Stack Information," in *Proceedings of the IEEE Security and Privacy*. 2003. Oakland, CA, USA.
15. Sekar, R., et al. "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proceedings of the IEEE Symposium on Security and Privacy*. 2001. Oakland, CA, pp. 144-155.
16. Arcata Pet, "Imgviewer/32," http://www.arcatapet.net/imgv32.cfm.
17. Ries, C., "Analysis of a Malicious JPEG Attack," http://www.vigilantminds.com/files/jpeg_attack_wp.pdf.
18. French Security Incident Response Team (FSIRT), "Windows JPEG GDI+ Overflow Administrator Exploit (Ms04-028)," http://www.frsirt.com/exploits/09232004.ms04-28-admin.sh.php.
19. Microsoft (TM) TechNet, "Microsoft Security Bulletin Ms06-001: Vulnerability in Graphics Rendering Engine Could Allow Remote Code Execution (912919)," http://www.microsoft.com/technet/security/bulletin/MS06-001.mspx.