Rootkits — Title TBA

Christopher Wood
Rochester Institute of Technology
Department of Software Engineering
Rochester, New York, USA
caw4567@rit.edu

Rajendra K. Raj
Rochester Institute of Technology
Department of Computer Science
Rochester, New York, USA
rkrics@rit.edu

Abstract

Keylogging is a technique used to track user input often for malicious purposes. As people use computers for a variety of activities including online banking and other personal correspondence, keylogging poses a serious threat to user privacy and security.

Current approaches to keylogging can be based both in hardware and software, and can work within or outside a network. Although keylogging approaches can be ighly dynamic, techniques used to monitor such attacks tend to be structural, not behavior-based.

To improve the monitoring of keylogging, it would be useful to consider the behavior of the malware, not just its structure. We have designed and implemented an approach to monitoring keylogging by looking for differences in the patterns of keyboard entry sequences and the corresponding changes in memory and disk.

This paper describes our approach to monitoring keylogging, its design and its implementation, and presents initial results of comparisons with existing techniques.

1 Introduction

Malicious software, or malware, that attempts to retrieve personal information on a computer can easily do so by a technique known as keystroke eavesdropping (cite keystroke eavesdropping paper), which is the process of covertly stealing user input and relaying it to an unauthorized location. This is typically done by targeting the keyboard, which is the most common interface between a computer and its user. Since most of the information entered by a user is private, any input leak (define input leak?) caused by malware such as rootkits, viruses, and trojan horses pose a significant threat to the safety and security of the user.

The remote retrieval of keystroke information is a twofold process. That is, they require the malware to be able to stealthily retrieve user input from the keyboard. For a malware developer, this is most easily accomplished by tapping into the flow of data to and from the physical keyboard device from inside either the user mode (ring 3) or kernel mode (ring 0) of the host operating system (OS) or a lower, more hardware specific level. Then, the malware must be able to successfully transmit the information recorded from the user to the desired location, whether that is on the local host computer or a remote computer. This can be accomplished in many different ways and the approach implemented is dependent entirely upon the developer and his personal agenda and overall objective.

Hoglund and Butler [2] explore some of the functional aspects of keyloggers implemented as rootkits. Rootkits are a type of malware that consist of a k̇itöf small programs used by attackers to gain permanent, undetectable access to the root of the target system. They are typically deployed in the kernel of the host OS where they can directly modify the kernel memory and objects or host OS to achieve stealth and provide future functionality for the attacker. Keyloggers targeting Microsoft Windows PCs are often layered on top of the keyboard device driver stack and employ kernel hooks. Layered drivers can easily intercept information from the keyboards and access privileged kernel-mode memory that is typically inaccessabile to user-mode processes, both of which are useful for stealthy keyloggers.

When dealing with malware that is well-crafted and platform specific it is more ideal to use a dynamic instead of a static detection technique. This is because that any well-crafted piece of malware will not be unique and original in some way and will not match any typical malware signatures. The behavior and actions of the malware are more helpful in the detection process because any piece of malware, no matter how sophisticated, will leave a footprint somewhere in the system. Depending on the implementation of the malware, dynamic detection algorithms can be implemented to look for suspicious changes in memory, system behavior or overhead, and function calls. When dealing with keystroke-monitoring malware, I found (should this be here?) that the flow of keystroke information throughout a computer using certain system functions, protocols, and pathways is a fairly successful indicator of the presence of malware. Specifically, by dynamically monitoring the system for suspicious behavior exhibited by objects with keystroke data I have found (have I really found this?) that malware can be detected with moderate success.

In this paper I present a keystroke-monitoring malware detection technique and algorithm that is specific for Microsoft Windows (NT) PCs. There are two main parts to this technique. Firstly, keystrokes entered by the user are extracted and stored temporarily in a buffer. The design and implementation of this component is detailed in section 4. Secondly, a comparison thread takes an image of the keystroke buffer and uses it perform matching algorithms with data being sent around the system, such as data that is being written into memory or sent over the TCP/IP protocol. The interval between each comparison iteration is large enough so that overhead created by the comparisons isn't noticeable (true yet?). Section 5 discusses the implementation of this algorithm and section 7 presents performance evaluations taken during time of use (weird phrasing, rephrase).

2 Theory and Limitations

TODO: talk about why this technique should work, what class of malware it applies to, etc..

- reliable and trusted data from keyboard (TODO: throw assumption that malware doesn't modify keystrokes into assumptions section) - efficient and accurate comparison algorithm to compare and contrast data - limited by the hardware of system (performance of algorithm) and medium through which keyloggers operate (local hard drives and basic TCP/IP protocols)

3 Overview

This section will discuss the assumptions made by the detection technique, deliverable implementation overview, and a brief discussion of each phase of the development of the deliverables.

The filter device driver approach consists of two stages: keystroke extraction and storage and multi-threaded comparison algorithm. The keystroke extraction phase retrieves user input from the keyboard, converts the device scancode (for the purpose of this paper, only US keyboard layouts are considered), and stores the data in a queue. Simultaneously, there is a background thread that retrieves the information from this queue and iteratively populates a character array called the buffer queue (BQ) that is essentially a circular list. The multi-threaded comparison phase will create a copy of the BQ, along with any other data retrieved from system function calls, and perform thorough string pattern checks. The objective is to determine if common string patterns begin to emerge between the original, user-entered data, and information flowing throughout the system. Such similar patterns are a good indication that keystrokes are being used by authorized programs.

3.1 Assumptions and Considerations

In order for this technique to operate correctly, there are several assumptions made about the host OS and nature of keylogging malware that are important to its design and implementation. Firstly, it is assumed that the keystroke-monitoring malware does not use any hand-crafted protocols to send data across a network to a remote user. Such protocols do not necessarily use the same system routines that typical TCP/IP enabled keyloggers use and thus will be more difficult to detect (true?). Secondly, it is assumed that the malware does not implement any form of sophisticated stenanography to hide data that is sent to remote users. Such stenanographic approaches involve the use of complex algorithms for decryption and thus are out of the scope of this paper and project. Lastly, it is assumed that there are an average amount of write routines being executed on by the OS while the malware and detection driver are loaded on the system. Since the functionality of the technique ultimately depends on user input and write routines throughout the system, a greater than average amount would increase the computational overhead in the comparison component of the driver and lead to undesired results (is this the case? tests will prove it..)

TODO: go more in depth about assumptions, talk about conditions

4 Design and Architecture

TODO: component responsibility and role with reasoning and relationship between rest of driver

Each phase of this technique is handled by a single Windows NT filter device driver. This is because important data can be shared and used throughout the program without worrying about managing multiple modules and concurrency issues. The functionality of the filter driver is broken up into several different components, including the driver load/unload, keyboard hook, comparison, threading, and utility modules, each of which is highly cohesive and handles a single set of tasks.

The driver load/unload component handles the vital routines necessary for the driver to be dynamically loaded and unloaded from system, as well every basic I/O request not pertaining to read requests. Much of the configuration and set-up for the driver takes place inside this component since it is the main entry point. Since this driver is not a piece of malware (although it may be perceived as one by AV software), it does not make any attempt to hide registry settings or hide its processes from the host or user. The keyboard hook component creates the driver device and layers it onto the keyboard device driver stack so that it can easily intercept I/O Request Packets (IRPs) as they flow to and from the keyboard device. However, its primary responsibility is to handle the kernel-mode hooked routines, such as the ZwWriteFile hooked function, that are a necessary part of the overall technique. It does this through the use of C macros that were published as part of (**GET PUBLISHED THING FROM ROOTKIT BOOK**) (cite).

The comparison and thread components are the most essential parts of the driver because they implement the main functionality. The comparison component performs the character string comparisons between collections of characters retrieved from the keyboard and the host OS. To do this it uses a modified version of the Boyer-Moore string matching algorithm (cite) that accounts for special cases and conditions that can arise depending on system activity. For example, if there are other read or write operations being performed simultaneously with keystroke character data writing to a file, the character data structure created from the parameters of ZwWriteFile can become more difficult to search (explain better). Equally important is the threading component of the driver, which is essentially the controller behind the overall technique. It is responsible for storing user keystrokes and executing character string comparisons on a clock-based schedule based on the tick of kernel timer object. To execute the comparisons, this component takes a "snapshot" of the structures holding the keystroke characters, creates temporary storage for them, and passes them along to the comparison component for inspection. (switch order of comparison and thread... YES)

Finally there is the basic utility component, which contains helper functions for testing, debugging, and data manipulation that are used by several of the other components.

For instance, this module implements the function that traverses a circularly-linked LIST_ENTRY structure list.

A filter device driver was also extremely useful for this technique because it is implemented and loaded in the kernel of the host OS where it has full-permissions and access to vital system routines, tables, memory, and data. Permission to manipulate system call tables is necessary in order to implement the hooking portion of this technique (briefly talk about pros and cons for filter driver).

To provide the extra functionality and flexibility needed for this technique, the driver was built using the Windows Driver Model (WDM), which is superset of the Windows Drive Foundation that all Windows drivers are typically written in.

*mutli-threaded design with pictures depicting how different components of driver are used for the different phases

5 Driver Implementation

Aside from the driver load/unload and utility components of the driver there is some level of complexity in the design and implementation of the rest of the driver that merits further explanation. Much of this technique depends on the ability of my driver to hook vital system routines to modify their control flow through the system. Hooking is the process of rerouting a function from its original location to one that a developer implements (use hooking image used for poster) and then re-patching the location back to the original (better description..). This plays a large role with the write operations provided by kernel-mode driver support routines, specifically ZwWriteFile and ZwCreateFile. ZwWriteFile writes data to an open file that has been opened by ZwCreateFile. By hooking these functions, any data that is provided as an argument to the routine by any other processes on the system, including malware, can be used and analyzed by this dynamic detection technique. However, for the purposes of this project, the only argument (or parameter) that is of value is the user-allocated buffer passed to the ZwWriteFile routine, which contains the data that is to be written to the specified file. Of course, the target file could also be analyzed for suspicious use, but there are too many possibilities that malware developers could choose for a file and without a clear set of rules attempting to do so would be pointless. The buffer of data that is to be written to a file is simply copied by the driver and stored in a KEY_DATA structure that is then enqueued into a queue that is dealt with by the threading component of this driver. By hooking these two functions we are essentially *getting data used and written to files (that might be malware)...

*insert image of keyboard hook *expand upon..

Getting the actual keystroke character from the keyboard is also a topic that merit's further discussion. The filter driver is implemented as a layered driver so that it can easily attach itself to the top of the keyboard driver stack. This way it can have access to all I/O requests and IRPs to/from the keyboard device. IRPs that are passed to the keyboard from the I/O subsystem of the OS are tagged for completion by this component so that once they are dealt with by the keyboard they will revisit this component on their way back up the driver stack. Drivers that are lower on the stack will handle direct input from the keyboard and populate the IRP's SystemBuffer(?) with the scancode that corresponds to the key that was pressed. When the hooking component receives the IRP that is tagged for completion it simply extracts the scancode from its buffer, coverts it to the appropriate character based on the US keyboard layout, and stores it in a KEY_DATA structure that is then enqueued into a queue that is dealt with elsewhere (threader).

*insert image of layered driver and extraction *expand upon..

6 Comparison Algorithm

-¿ string matching algorithms discussion, time complexity (with proof if possible) for this algorithm, breakdown and walkthrough of algorithm with pseudocode

*Pseudocode for algorithm (10 lines max so it's not too much for reader) *discussion about optimization attempts and downfalls (gaps in writeBuffer, for example, need to be accounted for due to hooking approach)

The comparison component contains the set of functions (modify this when code is implemented to include specific functions) that are used to check character data flowing throughout the system. This task essentially boils down to a string pattern matching problem with a non-trivial solution. Common string pattern matching algorithms, such as the Knuth-Morris-Pratt and Boyer-Moore algorithms, are not suitable for this task because they each involve searching for fixed pattern target inside another source. In the context of this technique, string matching must be more comprehensive and thorough by treating what is typically the pattern target as a pool of possible pattern targets. Therefore, the overall time complexity of this algorithm will inevitably increase.

There are also several system issues that affect the performance of this comparison algorithm (will i talk about one or many?). For instance, the source character buffer is populated by data that is passed into typical system write routines. If the target system is undergoing many write routines, most of which are not issued from malware, then much of the character data that is filled into the source buffer is not useful. At this point the comparison component makes no attempt to filter out such data from the source buffer so the comparison algorithm must be updated accordingly to account for for useless data inside the source buffer. The proposed solution effectively checks target combinations with all possible source combinations based on direct character comparisons. This is achieved using a recursive-inspired iterative solution to loop through the possible source combinations.

$$INSERTPSEUDOCODEFORALGORITH$$

As shown in the pseudo code in Figure (FIGURE NN), a stack is used to maintain the location of the last matching characters. As soon as a mismatch occurs, the algorithm jumps back to the last match location, increments the pointer location, and continues searching for matches. This approach effectively ensures that the source is thoroughly checked for a target pattern character in case any non-malware-generated characters were received and filled into the buffer by the threading component.

7 Initial Evaluation

(problems, soon-todo testing, future work, etc.)

TODO: discussion of results/limitations thus far and future work

8 Conclusion

TODO: conclude results and findings.

For now, let's cite a bunch of papers here, and we can redistribute them as needed. - RKR

References

[1] S. Embleton, S. Sparks, and C. Zou. Smm rootkits: a new breed of os independent malware. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–12, New York, NY, USA, 2008. ACM.

[2] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[3] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[4] M. E. Kabay. Some notes on malware. *Ubiquity*, 2005(August):1–1, 2005.

[5] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] M. Moffie, W. Cheng, D. Kaeli, and Q. Zhao. Hunting trojan horses. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 12–17, New York, NY, USA, 2006. ACM.

[7] S. Sagiroglu and G. Canbek. Keyloggers: Increasing threats to computer security and privacy. In *Technology and Society Magazine, IEEE*, pages 10–17. IEEE, 2009.

[8] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[9] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.