Deprived Black Box Recognition of User Space Keystroke Logging

G.Selvakumari[#1],B.Tamilselvan [#2], K.Sridharan [#3]
[#1]Assistant Professor of Information Technology
[#2, #3]Student of Information Technology
SKP Engineering College, Tiruvannamalai-606611, Tamil Nadu.
E-mail: [#1]jayaselvi30@skpec.in, [#2]tamizhsri3@gmail.com, [#3]ksridharan91@gmail.com.

Abstract - Software keyloggers are fast growing invasive software. The rapid growth is possibility for unprivileged programs running in userspace to record all the keystrokes typed by the users of a system. It will be run in hidden mode facilitates their implementation, but, at the same time, one allows to understand their behavior in detail manner. We propose a new technique that monitored keystroke sequences in input and observes the behavior of the keylogger in output to unambiguously identify it among the processes.We have an idea of our technique as an unprivileged application, hence matching the same deployment of a keylogger executing in hidden mode. We have evaluated the underlying technique against the most common keyloggers. This confirms the development of our scenario in practical framework. We have also devised evasion techniques that may be adopted to evade our approach and proposed a new method to strengthen the effectiveness of our solution against more attacks.

Keywords –Keylogger, Black box, user space, keystroke, evasion technique.

I. INTRODUCTION

Keyloggers are inserted on a machine to deliberately monitor the user activity by logging keystrokes and sending them to a third party. Keyloggers are often maliciously utilized by attackers to pirate confidential information. Many passwords and credit card numbers have been stolen using key loggers which makes them one of the most hazardous types of spyware. Keyloggers can be implemented by tiny hardware devices or suitably in software. A software acting as a Keylogger can be implemented by two different techniques: as a kernel module or as a user-space. It is important to notice that, while a kernel Keylogger requires a confidential access to the system, a Keylogger can easily on documented sets of unprivileged API commonly available on operating systems. A user can be easily deceived in installing it, and, since no special authorization is required, the user can improperly regard it as a safe piece of software. On the contradictory, kernel level key loggers require a significant effort and knowledge for an effective and bug-free implementation.

In this paper, we propose a new technique to detect key loggers handling as unprivileged user-space processes. Our technique is solely implemented in an unprivileged process. The proposed detection technique does not depend on the internal structure of the keylogger or the particular set of APIs used to capture the keystrokes. On the contradictory, our solution is of general applicability, since it is based on behavioural attributes of common to all the keyloggers. Our approach has proven essential in all the cases. We have also evaluated the effect of false positives and false negatives in practical scenarios.

II. OUR APPROACH

Our approach is explicitly focused on designing a detection technique for unprivileged user-space keyloggers. Other category of keyloggers, a user-space keylogger is a background process which reveals operating-system-supported hooks to surreptitiously eavesdrop keystroke issued by the user into the current foreground application. Our objective is to check user-space keyloggers from stealing confidential data originally studied for a legitimate foreground application. Malicious foreground applications secretly logging user-issued keystrokes and application-specific keyloggers are outside our threat model and cannot be recognized using our recognition technique. Also, the background keylogger cannot spawn a foreground application and steal the current application focus on demand without the user immediately noticing.

Our model is based on these surveys and investigates the possibility of separating the keylogger in a controlled environment, where its performance is directly revealed to the recognition system. Our technique involves monitoring the keystroke events that the keylogger receives in input, and constantly monitoring the I/O action created by the keylogger in output. To maintain detection, we purchase the instinct that the connection between the input and output of the controlled environment can be modelled for most keyloggers. Anyway, the modifications of the keylogger executes, a expected pattern observed in the keystroke events in input shall somehow be replicated in the I/O action in output. When the input and the output are managed, we can recognize common I/O patterns and detection. Additionally, preselecting the input pattern can better avoid bogus detections and elusion efforts. To detect background key logging behaviour our technique encompasses a pre-processing step to vigorously move the focus to the background. This approach is also necessary to avoid flagging foreground applications that reasonably react to user-issued keystrokes as keyloggers.

The benefit of our technique is that it is centred around a black-box model that totally disregards the keylogger internals. Also, I/O monitoring is a nonintrusive mechanism and can be accomplished on multiple processes concurrently. As a result, our approach can deal with a large number of keyloggers clearly and enables a fully unprivileged recognition system able to vet all the processes running on a specific system. Our technique completely ignores the content of the input and the output data, and focuses completely on their transportation. Limiting the approach to a computable analysis, enables the ability to implement the recognition technique with only unprivileged medium, as we will better demonstrated. The fundamental model acquired, however, presents supplementary challenges. First, we must carefully deal with realizable data transformations that may determinate quantitative differences in the input and the output patterns. Second, the technique should be durable with respect to quantitative similarities identified in the output patterns of other legal system processes. The following architecture diagram which explains the keystroke recognition and the mouse events are monitored in the recognition technique.

III. ARCHITECTURE

Our design is based on five different components as depicted in Fig. injector, pattern translator, pattern generator, monitor, detector. The operating system at the bottom performs the needs of I/O and events. The Operating System domain does not introduce all the details to the upper levels without using confidential API calls. As a result, the monitor and the injector operate at another level, i.e., Stream Domain. In this level, keystroke events and the bytes output by a process arrive as a stream discharged at a specific rate. The work of the injector is to inject a keystroke stream to reproduce the behaviour of a user typing keystrokes on the keyboard. Similarly, the monitor document(s) a stream of bytes to continuously arrest the output behaviour of a specific process. A stream characterization is only disturbed with the distribution of keystrokes or bytes released over a given window of examination, without necessitate any additional qualitative information.

The injector accepts the input stream from the pattern translator, that acts as a link between the Pattern Domain and the Stream Domain. Similarly, the monitor transports the output stream registered to the pattern translator for furthermore surveys. In the Pattern Domain, the input stream and the output stream are both portrayed in theoretical form, termed Abstract Keystroke Pattern (AKP). A pattern in the AKP form is a separated and normalized representation of a stream. Adopting a concise and uniform representation is advantageous for various reasons. First, we allow the pattern generator to exclusively focus on generating an input pattern and that results a distribution of use. Details on how to inject a specific distribution of keystrokes into the system are off loaded to the pattern translator and the injector. And the next, the same input pattern can be reused to create and inject several input streams with different properties but following the same underlying distribution. Finally, the potential to reason over conceptual representations simplifies the role of the detector that only accepts an input pattern and an output pattern and makes the final decision whether detection should be triggered.

A. Injector

The role of the injector is to inject the input stream into the system, mimitating the behaviour of a replicated user at the keyboard. When the injector must satisfy several requirements. First, it should only depend on unprivileged API calls. Second, it should be ability of injecting keystrokes at variable rates to match the distribution of the input stream. Finally, the series of results of keystroke events produced should be no different than those triggered by a user at the keyboard. To address all these issues, we influence the same technique involved in automated testing. In the Windows-based operating systems, for example, this functionality is assumed by the API call Send Input, accessible for several versions of the OS. All the other Operating Systems supporting the X11 window server, the same performance is available via the API call X Test Fake Key Event, segment of the XTEST supplement library.

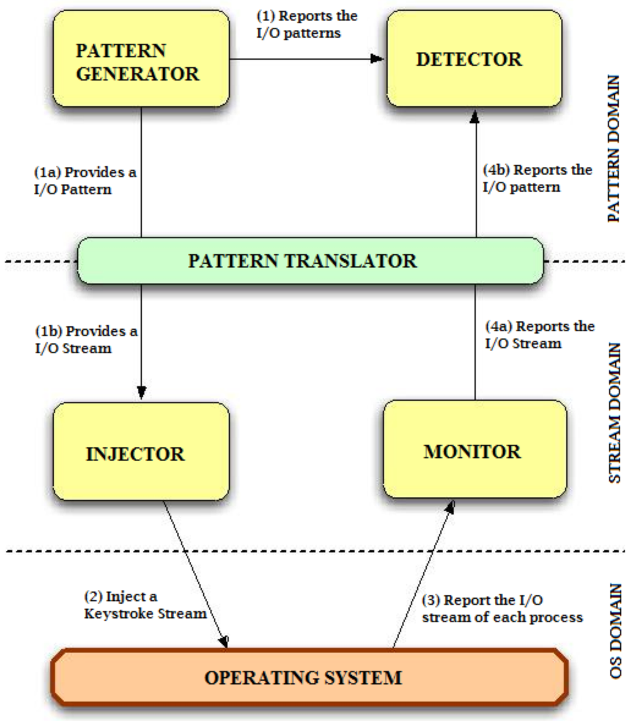
PATTERN GENERATOR
(1) Reports the I/O patterns
DETECTOR
(1a) Provides a I/O Pattern
(4b) Reports the I/O pattern
PATTERN DOMAIN
PATTERN TRANSLATOR
(1b) Provides a I/O Stream
(4a) Reports the I/O Stream
STREAM DOMAIN
INJECTOR
MONITOR
(2) Inject a Keystroke Stream
(3) Report the I/O stream of each process
OS DOMAIN
OPERATING SYSTEM

Fig 1 System Architecture

B. Monitor

The monitor is mainly responsible to record the output stream of the running operations and, over for the injector, we permit only deprived API calls. In addition, we favour a design to perform real-time monitoring with minimal overhead and the foremost degree of intention possible. Finally, we are focused in application level stats of I/O activities, to evade and dealing with system-level caching or other possible annoyance. Providentially, most operating systems provide deprived API calls to access performance token on a per process. If all the versions of Windows since Windows NT 4.0, this performance is provided by the Windows Management Instrumentation (WMI). In specific, the representation counters of each process are made accessible via the class Win32 Process, that carrys an client query based interface. All the representation counters are always supported up to date by kernel. In WMI, the counter Write Transfer Count includes the complete number of bytes the process wrote since its creation. To establish the output stream of a specified process, the monitor enquires the piece of information at regular time intervals, and documents the number of bytes written since the last query every time. The proposed approach is visibly customized to Windows operating systems. Nevertheless, we point out that identical strategies can be perceived in other OS.

C. Pattern Translator

The function of the pattern translator is to convert an AKP into a stream and respectively, specified a set of alignment parameters. A model in the AKP form can be modelled as a sequence of samples derived from a stream evaluated with a uniform time interval. A sample Pi of a design P is an theoretical presentation of the number of keystrokes released during the time interval i. Each sample is accumulated in a formalized form in the meanwhile ½0; 1_, where 0 and return the predefined minimum and maximum number of keystrokes in a defined time interval.

To modify an input pattern into a keystroke stream, the pattern translator appraises the following configuration parameters: N, the number of samples in the pattern; T, the constant time meanwhile linking for two consecutive samples; K max, the maximum number of keystrokes per sample allowed and K min, the minimum number of keystrokes per sample allowed between survey throughout the long term of data storage.

The translator understands a agreement between keystrokes and bytes and regards them equally as rest units of the input and output stream.

D. Detector

The outcome of our recognition algorithm lies in the ability to deduce a cause-effect relationship between the keystroke stream introduced in the system and the I/O action of a keylogger process, or, more partially, between the separate patterns in AKP form. While one must study every candidate process in the method, the recognition algorithm manages on a single process at a time, recognizing whether there is a strong resemblance between the input pattern and the output pattern acquired from the study of the I/O behaviour of the objective process. Specifically, a predefined input pattern and an output pattern of a specific process, the objective of the detection algorithm is to regulate whether there is a match in the patterns and the target process can be established as a keylogger with good possibility. The first step in the establishment of a detection algorithm comes down to the assumption of a acceptable metric to measure the resemblance between two specific patterns. Given K investigating deputations on K distinct data files from K different users, it is more superior for the TPA to batch these various tasks together and audit at one time. Maintaining this natural request in mind, we somewhat modify the protocol in a single user case, and achieve the gathering of K endorsement equations (for K checking tasks) into a single one. As a result, a secure batch checking protocol for concurrent auditing of various tasks is acquired.

IV. CONCLUSION

In this paper, we presented an deprived black-box approach for precise recognition of the most usual keyloggers, user-space keyloggers. We demonstrated the behaviour of a keylogger by surgically connecting the input (i.e., the keystrokes) with the output (i.e., the I/O patterns produced by the keylogger). In supplement, we enlarged our representation with the ability to manually inject carefully crafted keystroke patterns. We discussed the problem of selecting the best input pattern to upgrade our detection rate. Eventually, we proposed an execution of our detection technique on Windows, possibly the most vulnerable OS to the threat of keyloggers. To construct an OS autonomous architecture, we also gave implementation feature for other operating systems. We strongly assessed our prototype system against the most common free keyloggers. Other alternative results with a home grown Keylogger determined the effectiveness of our method in the general case.

REFERENCES:

[1] N. Grebennikov, “Keyloggers: How They Work and How to Detect Them,” http://www.viruslist.com/en/analysis?pubid=204791931,2012.

[2] Security Technology Ltd., “Testing and Reviews of Keyloggers, Monitoring Products and Spyware,” http://www.keylogger. org, 2012.

[3] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer,

[4] “Behavior-Based Spyware Detection,” Proc. 15th USENIX Security Symp., pp. 273-288, 2006.

[5] M. Aslam, R. Idrees, M. Baig, and M. Arshad, “Anti-Hook Shield against the Software Key Loggers,” Proc. Nat’l Conf. Emerging Technologies, pp. 189-191, 2004

[6] Y. Al-Hammadi and U. Aickelin, “Detecting Bots Based on Keylogging Activities,” Proc. Third Int’l Conf. Availability, Reliability and Security, pp. 896-902, 2008.