ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 3 Issue 10 October, 2014 Page No.8812-8820

UNPRIVILIGED BLACK BOX DETECTION OF USER-SPACE KEYLOGGERS

R.Suguna[1], R.Ramya[2]

[1]*Assistant Professor ,Department of Computer Science, PGP College of Arts and Science, Namakkal.*
Suguna.jeya@gmail.com
[2]*MPhil Scholar , Department of Computer Science ,PGP College of Arts and Science, Namakkal.*
Ramya8159@gmail.com

Abstract

Software keyloggers are a fast growing class of invasive software often used to harvest confidential information. One of the main reasons for this rapid growth is the possibility for unprivileged programs running in user space to eavesdrop and record all the keystrokes typed by the users of a system. The ability to run in unprivileged mode facilitates their implementation and distribution, but, at the same time, allows one to understand and model their behavior in detail. Leveraging this characteristic, we propose a new detection technique that simulates carefully crafted keystroke sequences in input and observes the behavior of the keylogger in output to unambiguously identify it among all the running processes. We have prototyped our technique as an unprivileged application, hence matching the same ease of deployment of a keylogger executing in unprivileged mode. We have successfully evaluated the underlying technique against the most common free keyloggers. This confirms the viability of our approach in practical scenarios. We have also devised potential evasion techniques that may be adopted to circumvent our approach and proposed a heuristic to strengthen the effectiveness of our solution against more elaborated attacks. Extensive experimental results confirm that our technique is robust to both false positives and false negatives in realistic settings.

Keywords: Invasive software, keylogger, security, black-box, PCC

1.INTRODUCTION

KEYLOGGERS are implanted on a machine to intentionally monitor the user activity by logging keystrokes and eventually delivering them to a third party [1]. While they are seldom used for legitimate purposes (e.g., surveillance/parental monitoring infrastructures), key loggers are often maliciously exploited by attackers to steal confidential information. Many credit card numbers and pass-words have been stolen using key loggers [2], [3],which makes them one of the most dangerous types of spyware known to date.

Key loggers can be implemented as tiny hardware devices or more conveniently in software. Software-based key-loggers can be further classified based on the privileges they require to execute. Keyloggers implemented by a kernel module run with full privileges in kernel space. Conversely, a fully unprivileged keylogger can be implemented by a simple user-space process. It is important to notice that a user-space key logger can easily rely on documented sets of unprivileged APIs commonly available on modern operating systems (OSs). This is not the case for a keylogger implemented as a kernel module. In kernel space, the programmer must rely on kernel-level facilities to intercept all the messages dispatched by the keyboard driver, undoubtedly requiring a considerable effort and knowledge for an effective and bug-free implementation. Furthermore, a keylogger implemented as a user-space process is much easier to deploy since no special permission is required. A user can erroneously regard the keylogger as a harmless piece of software and being deceived in executing it. On the contrary, kernel-space keyloggers require a user with superuser privileges to consciously install and execute unsigned code within the kernel, a practice often forbidden by modern operating systems such Windows Vista or Windows 7.In light of these observations, it is no surprise that 95 percent of the existing keyloggers run in user space [4]. Despite the rapid growth of keylogger-based frauds (i.e., identity theft, password leakage, etc.), not many effective and efficient solutions have been proposed to address this problem. Traditional defense mechanisms use fingerprinting strategies similar to those used to detect viruses and worms. Unfortunately, this strategy is hardly effective against the vast number of new keylogger variants surfacing every day in the wild.

In this paper, we propose a new approach to detect keyloggers running as unprivileged user-space processes. To match the same deployment model, our technique is entirely implemented in an unprivileged process. As a result, our solution is portable, unintrusive, easy to install, and yet very effective. In addition, the proposed detection technique is completely black-box, i.e., based on behavioral characteristics common to all keyloggers. In other words,

our technique does not rely on the internal structure of the keylogger or the particular set of APIs used. For this reason, our solution is of general applicability. We have prototyped our approach and evaluated it against the most common free keyloggers [5]. Our approach has proven effective in all the cases. We have also evaluated the impact of false positives in practical scenarios. In the final part of this paper, we further validate our approach with a homegrown keylogger that attempts to thwart our detection technique. Albeit already robust against the large majority of evasive behaviors, we also present and evaluate a heuristic against elaborated evasion strategies.

The structure of the paper is as follows: we start with an in-depth analysis of modern keyloggers in Section 2. We then introduce our approach in Section 3, detail its architecture in Section 4, and evaluate the resulting prototype in Section 5. Section 6 discusses the robustness against evasion techniques. We conclude with related work in Section 7 and final remarks in Section 8.

2. INTERNALS OF MODERN KEYLOGGERS

Breaching the privacy of an individual by logging his keystrokes can be perpetrated at many different levels. For example, an attacker with physical access to the machine might wiretap the hardware of the keyboard. A dishonest owner of an Internet cafe´, in turn, may find it more convenient to purchase a software solution, install it on all the terminals, and have the logs dropped on his own machine. Depending on the setting, a keylogger can be implemented in many different ways. For instance, external keyloggers rely on some physical property, either the acoustic emanations produced by the user typing [6], or the electromagnetic emanations of a wireless keyboard [7]. Hardware keyloggers are still external devices, but are implemented as dongles placed in between keyboard and motherboard. All these strategies, however, require physical access to the target machine.

To overcome this limitation, software approaches are more commonly used. Hypervisor-based keyloggers (e.g., BluePill [8]) are the straightforward software evolution of hardware-based keyloggers, literally performing a man-in-the-middle attack between the hardware and the operating system. Kernel keyloggers come second in the chain and are often implemented as part of more complex rootkits. In contrast to hypervisor-based approaches, hooks are directly used to intercept buffer-processing events or other kernel messages.

Albeit effective, all these approaches require privileged access to the machine. Moreover, writing a kernel driver—-hypervisor-based approaches pose even more challenges—-requires a considerable effort and knowledge for an effective and bug-free implementation (even a single bug may lead to a kernel panic). User-space keyloggers, on the other hand, do not require any special privilege to be deployed. They can be installed and executed regardless of the privileges granted.

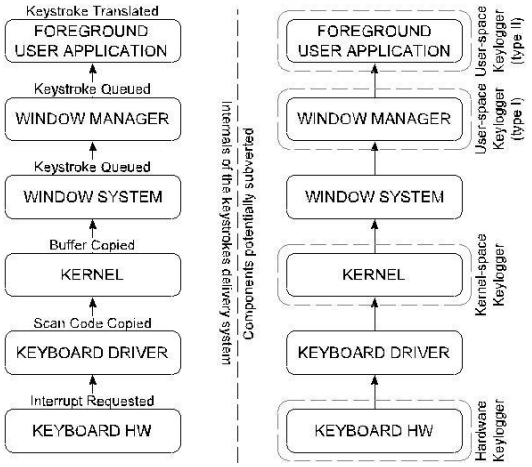

Fig.1.The delivery phases of a keystroke, and the components potentially subverted

This is a feat impossible for kernel keyloggers, since they require either superuser privileges or a vulnerability that allows arbitrary kernel code execution. Furthermore, user-space keylogger writers can safely rely on well-documented sets of APIs commonly available on modern operating systems, with no special programming skills required.

User-space keyloggers can be further classified based on the scope of the hooked message/data structures. Since a system hosts multiple applications, keystrokes can be intercepted either globally (i.e., for all the applications) or locally (i.e., within the application). We term these two classes of user-space keyloggers type I and type II. Fig. 1 shows the proposed classification: the left pane shows the process of delivering a keystroke to the intended application, whereas the right pane highlights the particular component subverted by each type of keylogger. Both types can be easily implemented in Windows, while the facilities available in Unix-like OSes—X11 and GTK required—allow for a straightforward implementation of the more invasive type I keyloggers.

Table 1: Presents a list of all the APIs that can be used to implement a user-space keylogger.
In brief, the SetWin-dowsHookEx() and gdk_window_add_filter() APIs are used to interpose the key logging procedure before a

Type	API	Comments
	Windows APIs	
Type I	`SetWindowsHookEx(WH_KEYBOARD_LL, ..., 0)`	The keylogging procedure is given as an argument.
	`GetAsyncKeyState()`	Poll-based.
Type II	`SetWindowsHookEx(WH_KEYBOARD, ..., 0)`	The keylogging procedure is given as an argument.
	`GetKeyboardState()`	Poll-based.
	`GetKeyState()`	Poll-based.
	`SetWindowLong(..., GWL_WNDPROC, ...)`	Overwrites the default procedure that deals with application messages.
	Intercepting `{Dispatch,Get,Translate}Message()`	Manual instrumentation of Win32 APIs.
	Unix-like APIs	
Type I	`gdk_window_add_filter(NULL, ...)`	The keylogging procedure is given as argument. `GTK` API.
	`inb(0x60)`	Poll-based and available only to the super-user.
	`XQueryKeymap()`	Poll-based. `X11` API.

Table 1

If the Scope of the API is Local, the Keylogger Must Inject Portions of Its Code in Each Application, e.g., Using a Library keystroke is effectively delivered to the target process. For SetWindowsHookEx(), this is possible by setting the last parameter (thread_id) to 0 (which subscribes to any keyboard event). For gdk_window_add_filter(), it is sufficient to set the handler of the monitored window to

NULL. The class of functions Get * State(), XQueryKeymap(), and inb(0x60) query the state of the keyboard and return a vector with the state of all (one in case of GetKeyState()) the keystrokes. When using these functions, the keylogger must continuously poll the keyboard in order to intercept all the keystrokes. The functions of the last class apply only to Windows and are typically used to overwrite the default address of key-stroke-related functions in all the Win32 graphical applica-tions. We have not found any example of this particular class of keyloggers in Unix-like OSes.

Since some of the APIs have just local scope, Type II keyloggers need to inject part of their code in a shared portion of the address space to have all the processes execute the provided callback. The only exception is with a Type II keylogger that uses either GetKeyState() or GetKeyboardState(). In these cases, the keylogging process can attach its input queue (i.e., the queue of events used to control a graphical user application) to other threads by using the procedure AttachtreadInput(). As a tentative counter-measure, Windows Vista recently eliminated the ability to share the same input queue for processes running in two different integrity levels. Unfortunately, since higher integ-rity levels are assigned only to known processes (e.g., Internet Explorer), common applications are still vulnerable to these interception strategies.

We can draw three important conclusions from our analysis. First, all user-space keyloggers are implemented by either hook-based or polling mechanisms. Second, all APIs are legitimate and well-documented. Third, all modern operating systems offer (a flavor of) these APIs. In particular, they always provide the ability to intercept keystrokes regardless of the application on focus. This design choice is dictated by the necessity to support such functionalities for legitimate applications. The following are three simple scenarios in which the ability to intercept arbitrary key-strokes is a functional requirement: 1) keyboards with additional special-purpose keys; 2) window managers with system-defined shortcuts; 3) background user applications whose execution is triggered by user-defined shortcuts (for instance, an application handling multiple virtual work-spaces requires hot keys that must not be overridden by other applications).

3.OUR APPROACH

Our approach is explicitly focused on designing a detection technique for unprivileged user-space keyloggers. Unlike other classes of keyloggers, a user-space keylogger is a background process which registers operating-system-supported hooks to surreptitiously eavesdrop (and log) every keystroke issued by the user into the current foreground application. Our goal is to prevent user-space keyloggers from stealing confidential data originally intended for a (trusted) legitimate foreground application. Malicious foreground applications surreptitiously logging user-issued keystrokes (e.g., a keylogger spoofing a trusted word processor application) and application-specific keyloggers (e.g., browser plugins surreptitiously performing keylogging activities) are outside our threat model and cannot be identified using our detection technique.

Our model is based on these observations and explores the possibility of isolating the keylogger in a controlled environment, where its behavior is directly exposed to the detection system. Our technique involves controlling the keystroke events that the keylogger receives in input, and constantly monitoring the I/O activity generated by the keylogger in output. To assert detection, we leverage the intuition that the relationship between the input and output of the controlled environment can be modeled for most keyloggers with very good approximation. When the input and the output are controlled, we can identify common I/O patterns and flag detection. Moreover, pre-selecting the input pattern can better avoid spurious detections and evasion attempts.

To detect background keylogging behavior our technique comprises a preprocessing step to forcefully move the focus to the background. This strategy is also necessary to avoid flagging foreground applications that legitimately react to user-issued keystrokes (e.g., word processors) as keyloggers.

The key advantage of our approach is that it is centered around a black-box model that completely ignores the keylogger internals. I/O monitoring is a nonintrusive procedure and can be performed on multiple processes simultaneously. As a result, our technique can deal with a large number of keyloggers transparently and enables a fully unprivileged detection system able to vet all the processes running on a particular system in a single run.

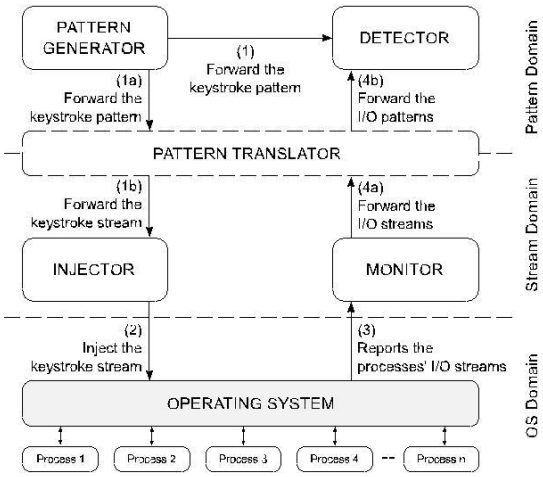

Fig. 2. The different components of our architecture.

Our approach completely ignores the content of the input and the output data, and focuses exclusively on their distribution. Limiting the approach to a quantitative analysis enables the ability to implement the detection technique with only unprivileged mechanisms, as we will better illustrate later. The underlying model adopted, however, presents additional challenges. First, we must carefully deal with possible data transformations that may introduce quantitative differences between the input and the output patterns. Second, the technique should be robust with respect to quantitative similarities identified in the output patterns of other legitimate system processes. In the following, we discuss how our approach deals with these challenges.

4. ARCHITECTURE

Our design is based on five different components as depicted in Fig. 2: injector, monitor, pattern translator, detector, pattern generator. The operating system at the bottom deals with the details of I/O and event handling. The OS Domain does not expose all the details to the upper levels without using privileged API calls. As a result, the injector and the monitor operate at another level of abstraction, the Stream Domain. At this level, keystroke events and the bytes output by a process appear as a stream emitted at a particular rate.

The task of the injector is to inject a keystroke stream to simulate the behavior of a user typing at the keyboard. Similarly, the monitor records a stream of bytes to constantly capture the output behavior of a particular process. A stream representation is only concerned with the distribution of keystrokes or bytes emitted over a given window of observation, without entailing any additional qualitative information. The injector receives the input stream from the pattern translator, which acts as bridge between the Stream Domain and the Pattern Domain. Similarly, the monitor delivers the output stream recorded to the pattern translator for further analysis. In the Pattern Domain, the input stream and the output stream are both represented in a more abstract form, termed Abstract Keystroke Pattern (AKP). A pattern in the AKP form is a discretized and normalized representation of a stream. Adopting a compact and uniform representation is advantageous for several reasons.
First, this allows the pattern generator to exclusively focus on generating an input pattern that follows a desired distribution of values. Details on how to inject a particular distribution of keystrokes into the system are offloaded to the pattern translator and the injector. Second, the same input pattern can be reused to produce and inject several input streams with different properties but following the same underlying distribution. Finally, the ability to reason over abstract representations simplifies the role of the detector that only receives an input pattern and an output pattern and makes the final decision on whether detection should or should not be triggered.

4.1.Injector

The role of the injector is to inject the input stream into the system, simulating the behavior of a user at the keyboard. By design, the injector must satisfy several requirements. First, it should only rely on unprivileged API calls. Second, it should be capable of injecting keystrokes at variable rates to match the distribution of the input stream. Finally, the resulting series of keystroke events produced should be no different than those generated by a real user. In other words, no user-space keylogger should be somehow able to distinguish the two types of events. To address all these issues, we leverage the same technique employed in automated testing. On Windows-based operating systems this functionality is provided by the API call key-bd_event.

4.2.Monitor

The monitor is responsible to record the output stream of all the running processes. As done for the injector, we allow only unprivileged API calls. In addition, we favor strategies to perform realtime monitoring with minimal overhead and the best level of resolution possible. Finally, we are interested in application-level statistics of I/O activities, to avoid dealing with filesystemlevel caching or other potential nuisances. Fortunately, most modern operating systems provide unprivileged API calls to access performance counters on a per-process basis. On all the versions of Windows since Windows NT 4.0, this functionality is provided by the Windows Management Instrumentation (WMI). In particular, the performance counters of each process are made available via the class Win32_Process, which supports an efficient query-based interface. The counter WriteTransferCount contains the total number of bytes written by the process since its creation. Note that monitoring the network activity is also possible, although it requires a more recent version of Windows, i.e., at least Vista. To construct the output stream of a given process, the monitor queries this piece of information at regular time intervals, and records the number of bytes written since the last query every time. The proposed technique is obviously tailored to Windows-based operating systems. Nonetheless, we point out that similar strategies can be realized in other OSes; both Linux and OSX, in fact, support analogous performance counters which can be accessed in an un-privileged manner; the reader may refer to the iotop utility for usage examples.

4.3.Detector

The success of our detection algorithm lies in the ability to infer a cause-effect relationship between the keystroke stream injected in the system and the I/O behavior of a keylogger process, or, more specifically, between the respective patterns in AKP form. While one must examine every candidate process in the system, the detection algorithm operates on a single process at a time, identifying whether there is a strong similarity between the input pattern and the output pattern obtained from the analysis of the I/O behavior of the target process. Specifically, given a predefined input pattern and an output pattern of a particular process, the goal of the detection algorithm is to determine whether there is a match in the patterns and the target process can be identified as a keylogger with good probability.
The first step in the construction of a detection algorithm comes down to the adoption of a suitable metric to measure the similarity between two given patterns. In principle, the AKP representation allows for several possible measures of dependence that compare two discrete sequences and quantify their relationship. In practice, we rely on a single correlation measure motivated by the properties of the two patterns. The proposed detection algorithm is based on the Pearson product-moment correlation coefficient (PCC), one of the most widely used correlation measures [9]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [9]sample standard deviations, and P and Q are sample means.The PCC has been widely used as an index to measure bivariate association for different distributions in several applications including pattern recognition, data analysis, and signal processing [10]. The values given by the PCC are always symmetric and ranging between _1 and 1, with 0 indicating no correlation and 1 or _1 indicating complete direct (or inverse) correlation. To measure the degree of association between two given patterns we are here only interested in positive values of correlation. Hereafter, we will always

refer to its absolute value. Our interest in the PCC lies in its appealing mathematical properties. In contrast to other correlation metrics, the PCC measures the strength of a linear relationship between two series of samples, ignoring any nonlinear association. In our setting, a linear dependence well approximates the relationship between the input pattern and an output pattern produced by a keylogger. The intuition is that a keylogger can only make local decisions on a per-keystroke basis with no knowledge about the global distribution. Thus, in principle, the resulting behavior will linearly approximate the original input stream injected into the system.
An interesting application of the location invariance property is the ability to mitigate the effect of buffering. When the keylogger uses a fixed-size buffer whose size is comparable to the number of keystrokes injected at each time interval, it is easy to show that the PCC is not significantly affected.

4.4 Pattern Generator

Our pattern generator is designed to support several pattern generation algorithms. More specifically, the pat-tern generator can leverage any algorithm producing a valid pattern in AKP form. We now present a number of pattern generation algorithms and discuss their properties.

The first important issue to consider is the effect of variability in the input pattern. Experience shows that correlations tend to be stronger when samples are dis-tributed over a wider range of values [11]. In other words, the more the variability in the given distributions, the more stable and accurate the resulting PCC computed. This suggests that a robust input pattern should contain samples spanning the entire target interval ½0; 1&. The level of variability in the resulting input stream is also similarly influenced by the range of keystroke rates used in the pattern translation process. The higher the range delimited by the minimum keystroke rate and maximum keystroke rate, the more reliable the results.

A robust pattern generation algorithm should allow for a minimum number of false positive. When the chosen input pattern happens to closely resemble the I/O behavior of some benign process, the PCC may report a high value of correlation for that process and trigger a false detection. For this reason, it is important to focus on input patterns that have little chances of being confused with output patterns generated by legitimate processes. Fortunately, studies show that the correlation between realistic I/O workloads for PC users is generally considerably low over small time intervals. The results presented in [13] are derived from 14 traces collected over a number of months in realistic environments used by different categories of users. The authors show that the value of correlation given by the PCC over 1 minute of I/O activity is only 0.046 on average and never exceeds 0.070 for any two given traces. This suggests that the I/O behavior of one or more processes is in general very poorly correlated with other I/O distributions.

The problem of designing a pattern generation algorithm that minimizes false positives under a given known workload can be modeled as follows: we assume that traces for the target workload can be collected and converted into a series of patterns (one for each process running on the system) of the same length N. All the patterns are generated to build a valid training set for the algorithm. Under the assumption that the traces collected are representative of the real workload available at detection time, our goal is to design an algorithm that learns the characteristics of the training data and generates a maximally uncorrelated input pattern. Concretely, the goal of our algorithm is to produce an input pattern of length N that minimizes the PCC measured against all the patterns in the training set. Without any further constraints on the samples of the target input pattern, it can be shown that this problem is a nontrivial nonlinear optimization problem. In practice, we can relax the original problem by leveraging some of the assumptions discussed earlier. As motivated before, a robust input pattern should present samples distributed over a wide range of values. To assume the widest range possible, we can arbitrarily constrain the series of samples to be uniformly distributed over the target interval ½0; 1&. This is equivalent to consider a set of N samples of the form Prior research has shown how to transform this particular problem into an equivalent Quadratic Assignment Problem (QAP) that can be very efficiently solved with a standard QAP solver when the global minimum is known in advance [15]. In our solution, we have implemented a similar approach limiting the approach to a maximum number of iterations to guarantee convergence since the minimum value of the PCC is not known in advance. In practice, for a reasonable number of samples N and a modest training set, we found that this is rarely a concern. The algorithm can usually identify the optimal pattern in a bearable amount of time.

To conclude, we now more formally propose two classes of pattern generation algorithms for our generator. First, we are interested in workload-aware generation algorithms. For this class, we focus on the optimization algorithm we have just introduced—we refer to this pattern generation algorithm with the term WLD—assuming a number of representative traces have been made available for the target workload. Moreover, we are interested in workload-agnostic pattern generation algorithms. With no assumption made on the nature of the workload, they are more generic and easier to implement. In this class, we propose the following algorithms:

Random (RND). Each sample is generated at random.
Random with fixed range (RFR). The pattern is a random permutation of a series of samples uniformly distributed over the interval ½0; 1&. This is to maximize the amount of variability in the input pattern.
Impulse (IMP). Every sample 2i is assigned the value of 0 and every sample 2i þ 1 is assigned the value of 1.
This algorithm attempts to produce an input pattern with maximum variance and idle periods at minimum.
Sine wave (SIN). The pattern generated is a discrete sine wave distribution oscillating between 0 and 1.
The sine wave grows or drops with a fixed step of 0.1. This algorithm explores the effect of constant increments and decrements in the input pattern.

5 EVALUATION

To evaluate the proposed detection technique, we implemented a prototype based on the ideas described in the paper. Written in C# in 7,000 LoC, it runs as an unprivileged application for the Windows OS. It also collects simultaneously all the processes' I/O patterns, thus

allowing us to analyze the whole system in a single run. Although the proposed design can easily be extended to other OSes, we explicitly focus on Windows for the significant number of keyloggers available. In the following, we present several experiments to evaluate our approach. The ultimate goal is to understand the effectiveness of our technique and its applicability to realistic settings. For this purpose, we evaluated our prototype against many publicly available keyloggers. We also developed our own keylogger to evaluate the effect of particular conditions more thoroughly. Finally, we collected traces for different realistic PC workloads to evaluate the effectiveness of our approach in real-life scenarios. We ran all of our experiments on PCs with a 2.53 Ghz Core 2 Duo processor, 4 GB memory, and 7,200 rpm SATA II hard drives. Every test was performed under Windows 7 Professional SP1, while the workload traces were gathered from a number of PCs running several different versions of Windows. Since the performance counters are part of the default accounting infrastructure, monitoring the processes' I/O came at negligible cost: for reasonable values of T , i.e., > 100 ms, the load imposed on the CPU by the monitoring phase was less than 2 percent. On the other hand, injecting high keystroke rates introduced additional processing overhead throughout the system. Experimental results showed that the overhead grows approximately linearly with the number of key-strokes injected per sample. In particular, the CPU load imposed by our prototype reaches 25 percent around 15,000 keystrokes per sample and 75 percent around 47,000. Note that these values only refer to detection-time overhead. No runtime overhead is imposed by our technique when no detection is in progress.

5.1.Keylogger Detection

To evaluate the ability to detect real-world keyloggers, we experimented with all the keyloggers from the top monitoring free software list [5], an online repository continuously updated with reviews and latest develop-ments in the area. To carry out the experiments,

Keylogger	Detection	Notes
Refog Keylogger Free 5.4.1	✓	focus-based buffering
Best Free Keylogger 1.1	✓	-
Iwantsoft Free Keylogger 3.0	✓	-
Actual Keylogger 2.3	✓	focus-based buffering
Revealer Keylogger Free 1.4	✓	focus-based buffering
Virtuoza Free Keylogger 2.0	✓	time-based buffering
Quick Keylogger 3.0.051	✓	-
Tesline KidLogger 1.4	✓	-

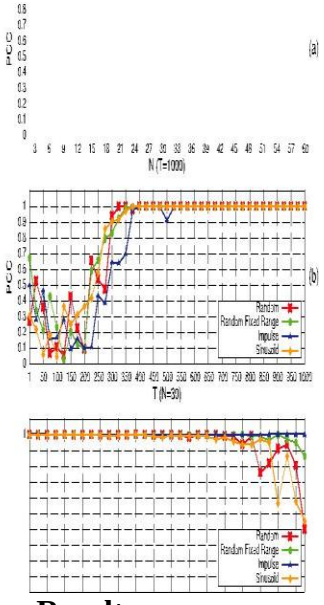

Table 2: Detection Result

Table 2 shows the keyloggers used in the evaluation and summarizes the detection results. All the keyloggers were detected within a few seconds without generating any false positives; in particular, no legitimate process scored PCC values _ 0:3. Virtuoza Free Keylogger required a longer window of observation to be detected; this sample was indeed the only keylogger to store keystrokes in memory and flush out to disk at regular time intervals. Nevertheless, we were still able to collect consistent samples from flush events and report high PCC values.

In a few other cases, keystrokes were kept in memory but flushed out to disk as soon as the keylogger detected a change of focus. This was the case for Actual Keylogger, Revealer Keylogger Free, and Refog Keylogger Free. To deal with this common strategy, our detection system enforces a change of focus every time a sample is injected. In addition, some of the keyloggers examined included support for encryption and most of them used variable-length encoding to store special keys. As Section 5.2 demonstrates, our approach deal with these nuisances transparently with no effect on the resulting PCC.

5.2.False Negatives

In our approach, false negatives may occur when the output pattern of a keylogger scores an unexpectedly low PCC value. To test the robustness of our approach against false negatives, we made several experiments with our own artificial keylogger. Our evaluation starts by analyzing the impact of the number of

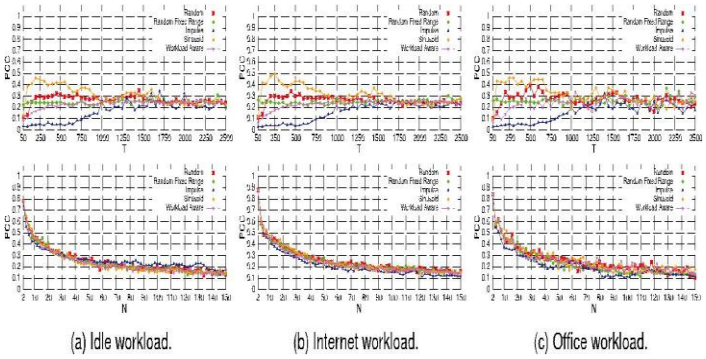

Fig. 6. Impact of different classes of noise on the PCC

samples N and the time interval T on the final PCC value. For each pattern generation algorithm, we plot the PCC measured with our prototype keylogger which we configured so that no buffering or data transformation was taking place. Figs. 3a and 3b depict our findings with K_{min} ¼ 1 and K_{max} ¼ 1;000. We observe that when the keylogger logs each keystroke without introdu-cing delay or additional noise, the number of samples N does not affect the PCC value. This behavior should not suggest that N has no effect on the production of false negatives. When noise in the output stream is to be expected, higher values of N are indeed desirable to produce more stable PCC values and avoid false negatives.

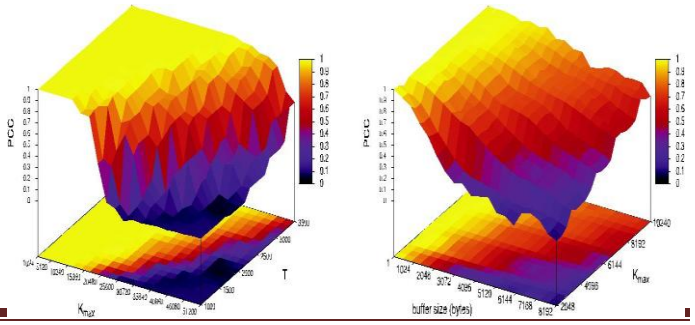

Fig. 4. Impact of K_{max} and T on the PCC. Fig. 5. Detection of a keylogger buffering its outpu

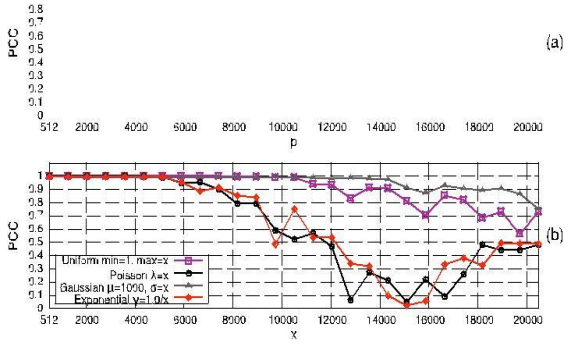

Fig. 7. Impact of N and T on the PCC measured with our prototype keylogger against different workloads.

In a more advanced version of our keylogger, we also simulated the effect of several possible input-output transformations. First, we experimented with a keylogger using a nontrivial fixed-length encoding for keystrokes. Fig. 5a depicts the results for different values of padding p with N ¼ 30, K_{min} ¼ 1, and K_{max} ¼ 1,024. A value of p ¼ 1,024 simulates a keylogger writing 1,024 bytes on the disk for each eavesdropped keystroke. As discussed in Sec-tion 4.4,

the PCC should be unaffected in this case and presumably exhibit a constant behavior. The figure confirms this intuition, but shows the PCC decreasing linearly after p _ 10;000 bytes. This behavior is due to the limited I/O throughput that can be achieved within a single time interval. We previously encountered similar problems when choosing suitable values for K_{max}. Note that in this scenario both K_{min} and K_{max} are affected by the padding introduced, thus yielding a more significant impact on the PCC.

The result is that each of these transformations can be always approximated by a linear transformation with constant scaling. We conclude our analysis by verifying the impact of a keylogger buffering the eavesdropped data before leaking it to the disk. Although we have not found many real-world examples of this behavior in our evaluation, our technique can still handle this class of keyloggers correctly for reasonable buffer sizes. Fig. 6 depicts our detection results against a keylogger buffering its output through a fixed-size buffer. The figure shows the impact of several possible choices of the buffer size on the final PCC value. We can observe the pivotal role of K_{max} in successfully asserting detection. For example, increasing K_{max} to 10,240 is necessary to achieve sufficiently high PCC values for the largest buffer size proposed. This experiment demonstrates once again that the key to detection is inducing the pattern to distinctly emerge in the output distribution, a feat that can be easily obtained by choosing a highly variable injection pattern with low values for K_{min} and high values for K_{max}. We believe these results are encouraging to acknowledge the robustness of our detection technique against false nega-tives, even in presence of complex data transformations.

5.3.False Positives

In our approach, false positives may occur when the output pattern of some benign process accidentally scores a significant PCC value. If the value happens to be greater than the selected threshold, a false detection is flagged. This section evaluates our prototype keylogger to investigate the likelihood of this scenario in practice.

6 .EVASION AND COUNTERMEASURES

In this section, we speculate on the possible evasion techniques a keylogger may employ once our detection strategy is deployed on real systems.

6.1.Aggressive Buffering

A keylogger may rely on some forms of aggressive buffering, for example flushing a very large buffer every time interval t, with t being possibly hours. While our model can potentially address this scenario, the extremely large window of observation required to collect a sufficient number of samples would make the resulting detection technique impractical. It is important to point out that such a limitation stems from the implementation of the technique and not from a design flaw in our detection model.

6.2.Trigger-Based Behavior

A keylogger may trigger the keylogging activity only in face of particular events, for example when the user launches a particular application. Unfortunately, this trigger-based behavior may successfully evade our detection technique. This is not, however, a shortcoming specific to our approach, but rather a more fundamental limitation common to all the existing detection techniques based on dynamic analysis [17]. While we believe that the problem of triggering a specific behavior is orthogonal to our work and already focus of much ongoing research, we point out that the user can still mitigate this threat by periodically reissuing detection runs when necessary (e.g., every time a new particularly sensitive context is accessed). Since our technique can vet all the processes in a single detection run, we believe this strategy can be realistically used in real-world scenarios.

6.3.Discrimination Attacks

Mimicking the user's behavior may expose our approach to keyloggers able to tell artificial and real keystrokes apart. A keylogger may, for instance, ignore any input failing to display

known statistical propertiese.g., not akin to the English language. However, since we control the input pattern, we can carefully generate keystroke scancode sequences displaying the same statistical properties (e.g., English text) expected by the keylogger, and therewith perform a separate detection run thwarting this evasion technique. About the case of a keylogger ignoring key-strokes when detecting a high (nonhuman) injection rate. This strategy, however, would make the keylogger prone to denial of service: a system persistently generating and exfiltrating bogus keystrokes would induce this type of keylogger to permanently disable the keylogging activity. Recent work demonstrates that building such a system is feasible in practice (with reasonable overhead) using standard two facilities [18].

6.4.Decorrelation Attacks

Decorrelation attacks attempt at breaking the correlation metric our approach relies on. Since of all the attacks this is specifically tailored to thwarting our technique, we hereby propose a heuristic intended to vet the system in case of negative detection results. This is the case, for instance, of a keylogger trying to generate I/O noise in the background and lowering the correlation that is bound to exist between the pattern of keystrokes injected I and its own output pattern O. In the attacker's ideal case, this translates to PCCðI; OÞ _ 0. To approximate this result in the general case, however, the attacker must adapt its disguisement strategy to the pattern generation algorithm in use, i.e., when switching to a new injection I^0 6¼ I, the output pattern should reflect a new distribution O^0 6¼ O. The attacker could, for example, enforce this property by adapting the noise generation to some input distribution-specific variable (e.g., the current keystroke rate). Failure to do so will result in random noise uncorrelated with the injection, a scenario which is already handled by our PCC-based detection technique. At the same time, we expect any legitimate process to maintain a sufficiently stable I/O behavior regardless of the particular injection chosen. The conclusion is that analyzing a sufficiently large number of samples is crucial to obtain accurate results when estimating the similarity between different distributions.

7. RELATED WORK

While ours is the first technique to solely rely on unprivi-leged mechanisms, several approaches have been recently proposed to detect privacy-breaching malware, including keyloggers. Behavior-based spyware detection has been first introduced by Kirda et al. in [21]. Their approach is tailored to malicious Internet Explorer loadable modules. In parti-cular, modules monitoring the user's activity and disclosing private data to third parties are flagged as malware. Their analysis models malicious behavior in terms of API calls invoked in response to browser events. Those used by keyloggers, however, are also commonly used by legitimate programs. Their approach is therefore prone to false positives, which can only be mitigated with continuously updated whitelists.

Other keylogger-specific approaches have suggested detecting the use of well-known keystroke interception APIs. Unfortunately, all these calls are also used by legitimate applications (e.g., shortcut managers) and this approach is again prone to false positives. Xu et al. [23] push this technique further, specifically targeting Windows-based operating systems. They rely on the very same hooks used by keyloggers to alter the message type from WM_KEYDOWN to WM_CHAR. A keylogger aware of this countermeasure, however, can easily evade detection by also switching to a new message type or periodically registering a new hook to obtain the highest priority in the hook chain.

Closer to our approach is the solution proposed by Al-Hammadi et al. [24]. Their strategy is to model the keylogging behavior in terms of the number of API calls issued in the window of observation. To be more precise, they observe the frequency of API calls invoked to 1) intercept keystrokes, 2) writing to a file, and 3) sending bytes over the network. A keylogger is detected when two of these frequencies are found to be highly correlated. Since no bogus events are issued to the system (no injection of crafted input), the correlation may not be as strong as expected. The resulting value would be even more impaired in case of any delay introduced by the keylogger. Moreover, since their analysis is solely focused on a specific bot, it lacks a proper discussion on both false positives and false negatives. In contrast to their approach, our quantitative analysis is performed at the byte granularity and our correlation metric (PCC) is rigorously linear. As shown earlier, linearity makes our technique completely resilient to several common data transformations performed by keyloggers.

A similar quantitative and privileged technique is sketched by Han et al. [25]. Unlike the solution presented in [24], their technique does include an injection phase. Their detection strategy, however, still models the key-logging behavior in terms of API calls. In practice, the assumption that a certain number of keystrokes results in a predictable number of API calls is fragile and heavily implementation-dependent. In contrast, our byte-level analysis relies on finer grained measurements and can identify all the information required for the detection in a fully unprivileged way. Complementary to our work, recent approaches have proposed automatic identification of trigger-based behavior, which can potentially thwart any detection technique based on dynamic analysis. In parti-cular, in [17], [26] the authors propose a combination of concrete and symbolic execution for the task. Their strategy aims to explore all the possible execution paths that a malware can possibly exhibit during execution. As the authors in [17] admit, however, automating the detection of trigger-based behavior is an extremely challenging task which requires advanced privileged tools. The problem is also undecidable in the general case.

8. CONCLUSIONS

In this paper, we presented an unprivileged black-box approach for accurate detection of the most common keyloggers, i.e., user-space keyloggers. We modeled the behavior of a keylogger by surgically correlating the input (i.e., the keystrokes) with the output (i.e., the I/O patterns produced by the keylogger). In addition, we augmented our model with the ability to artificially inject carefully crafted keystroke patterns. We then discussed the problem of choosing the best input pattern to improve our detection rate. Subsequently, we presented an implementation of our detection technique on Windows, arguably the most vulnerable OS to the threat of keyloggers. To establish an OS-independent architecture, we

also gave implementation details for other operating systems. We successfully eval-uated our prototype system against the most common free keyloggers [5], with no false positives and no false negatives reported. Other experimental results with a homegrown keylogger demonstrated the effectiveness of our technique in the general case. While attacks to our detection technique are possible and have been discussed at length in Section 6, we believe our approach considerably raises the bar for protecting the user against the threat of keyloggers.

REFERENCES

T. Holz, M. Engelberth, and F. Freiling, "Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones," Proc. 14th European Symp. Research in Computer Security, pp. 1-18, 2009.

L. Zhuang, F. Zhou, and J.D. Tygar, "Keyboard Acoustic Emanations Revisited," ACM Trans. Information and System Security, vol. 13, no. 1, pp. 1-26, 2009.

M. Vuagnoux and S. Pasini, "Compromising Electromagnetic Emanations of Wired and Wireless Keyboards," Proc. 18th USENIX Security Symp., pp. 1-16, 2009.

J. Rutkowska, "Subverting Vista Kernel for Fun and Profit," Black Hat Briefings, vol. 5, 2007.

J.L. Rodgers and W.A. Nicewander, "Thirteen Ways to Look at the Correlation Coefficient," The Am. Statistician, vol. 42, no. 1, pp. 59-66, Feb. 1988.

J. Benesty, J. Chen, and Y. Huang, "On the Importance of the Pearson Correlation Coefficient in Noise Reduction," IEEE Trans. Audio, Speech, and Language Processing, vol. 16, no. 4, pp. 757-765, May 2008.

L. Goodwin and N. Leech, "Understanding Correlation: Factors that Affect the Size of r," Experimental Education, vol. 74, no. 3,

249-266, 2006.

J. Aldrich, "Correlations Genuine and Spurious in Pearson and Yule," Statistical Science, vol. 10, no. 4, pp. 364-376, 1995.

W. Hsu and A. Smith, "Characteristics of I/O Traffic in Personal Computer and Server Workloads," IBM System J., vol. 42, no. 2,
347-372, 2003.

H.W. Kuhn, "The Hungarian Method for the Assignment Problem," Naval Research Logistics Quarterly, vol. 2, pp. 83-97, 1955.