

KLIMAX: Profiling Memory Write Patterns to Detect Keystroke-Harvesting Malware^{*}

Stefano Ortolani¹, Cristiano Giuffrida¹, and Bruno Crispo²

¹ Vrije Universiteit, De Boelelaan 1081, 1081HV Amsterdam, The Netherlands
{ortolani, giuffrida}@cs.vu.nl

² University of Trento, Via Sommarive 14, 38050 Povo, Trento, Italy
crispo@disi.unitn.it

Abstract. Privacy-breaching malware is an ever-growing class of malicious applications that attempt to steal confidential data and leak them to third parties. One of the most prominent activities to acquire private user information is to eavesdrop and harvest user-issued keystrokes. Despite the serious threat involved, keylogging activities are challenging to detect in the general case. From an operating system perspective, their general behavior is no different than that of legitimate applications used to implement common end-user features like custom shortcut handling and keyboard remapping. As a result, existing detection techniques that attempt to model malware behavior based on system or library calls are largely ineffective. To address these concerns, we introduce a novel detection technique based on fine-grained profiling of memory write patterns. The intuition behind our model lies in data harvesting being a good predictor for sensitive information leakage. To demonstrate the viability of our approach, we have designed and implemented KLIMAX: a Kernel-Level Infrastructure for Memory and eXecution profiling. Our system supports proactive and reactive detection and can be transparently deployed online on a running Windows platform. Experimental results with real-world malware confirm the effectiveness of our approach.

Keywords: Malware, Memory, Behavior, Keylogging, Detection.

1 Introduction

Malware is still one of the main reasons for security incidents [12]. Among different types of malware the one harvesting users' private information is increasing in terms of both impact and number of occurrences [17]. Stealing user confidential data serves for many illegal purposes, such as identity theft, banking and credit card frauds, software and services theft, disclosure of clinical records, just to name a few. A common activity performed by privacy-breaching malware is keylogging, that is the eavesdropping, harvesting, and leakage of user-issued keystrokes. To address the general problem of malware detection, a number of

^{*} The original publication is available at <http://www.springerlink.com>.

models and techniques have been proposed over the years. However, when applied to the specific problem of detecting malware with keylogging behavior, all existing solutions are unsatisfactory. Signature-based solutions have limited applicability since they can easily be evaded and also require to isolate and extract a valid signature before they are able to detect a new threat. Behavior-based detection techniques overcome some of these limitations. They aim at distinguishing between malicious and benign applications by profiling the behavior of legitimate programs [8] or malware [5]. Different techniques exist to analyze and learn the intended behavior, however most of them are based on which system calls or library calls are invoked at runtime. Unfortunately, characterizing keylogging behavior using system calls is a prohibitive task, since there are many legitimate applications (e.g., shortcut managers, keyboard remapping utilities) that intercept keystrokes in the background and exhibit a very similar behavior. These applications represent an obvious source of false positives. Using whitelisting to solve this problem is not an option, given the large number of programs of this kind and their pervasive presence in OEM software. Moreover, syscall-based keylogging behavior characterization is not immune from false negatives either. Consider the perfect model that can infer keylogging behavior from system calls that reveal explicit sensitive information leakage. This model will always fail to detect malware that harvests keystroke data in memory aggressively, and delays the actual leakage as much as possible. Since malicious applications strive to conceal their behavior, this scenario is the norm rather than the exception.

In this paper, we propose a new approach specifically tailored to detecting privacy-breaching malware containing any form of keylogging activities. Our approach is still behavior-based but it profiles memory writes rather than system or library calls. The basic idea is to analyze the correlation between the distribution of user-issued keystrokes and the resulting memory writes performed by the malware to harvest sensitive data. Following this intuition, we inject a carefully-chosen keystroke stream and observe the memory write patterns of the analyzed application. High correlation values translate to immediate detection.

Note that our approach does not rely on the observation of the actual leakage of sensitive data, but instead leverages the key intuition that identifying information harvesting is sufficient to infer malicious behavior. As a result, all malware evasion techniques that conceal or delay information leakage are not a concern for our detection technique. Another fundamental design choice is to adopt a fine-grained profiling strategy, to isolate the keylogging behavior from other concurrent activities. Our analysis shows that this is crucial to eliminate additional sources of false negatives, since privacy-breaching malware often performs many concurrent activities, possibly including those to actively disorient behavior-based detection strategies.

A much more effective concealment technique is given by trigger-based behavior, namely malware that only starts actively harvesting sensitive data when triggered by some, possibly external (e.g., bot command), events. This modus operandi poses a serious challenge to all the known behavior-based detection techniques, since failing to trigger the intended behavior either at learning or

detection time results in poor detection accuracy. The proposed design addresses this challenge allowing our detection strategy to work in both proactive and reactive mode. Proactive detection is activated directly by the user. In reactive mode, our behavior analysis is automatically activated on demand whenever a candidate malicious application is recognized at runtime. This strategy is feasible due to the distinctive runtime characteristics of the keylogging activity, as better explained later. All these countermeasures against evasion and concealment techniques allow our approach to achieve a very low false negative rate. In the remainder of the paper, we also show how careful design strategies allow our detection technique to achieve a minimum number of false positives as well. To summarize, the contributions of this paper are the following:

A new behavior-based detection model based on memory write pattern profiling, which is particularly suited for privacy-breaching malware exhibiting keylogging behavior.

Design and implementation of KLIMAX: a Kernel-Level Infrastructure for Memory And eXecution profiling based on our new model and ready to be transparently deployed online on a running Windows platform. The source code of the infrastructure is publicly available for download ³.

Evaluation against real-world malware and against legitimate applications that leverage keystroke-interception functionalities.

2 Background

Our behavioral model is based on the intuition that the malware actively harvests keystrokes and strives to conceal the related leakage. No assumption is made on the malware internals. Instead, to detect any possible form of keystrokes harvesting, we base our analysis on memory write patterns that necessarily emerge from the keylogging behavior.

Previously proposed approaches that attempted to build a profile of keylogging behavior in terms of I/O patterns [13] are not suitable to solve this problem. Unfortunately, malicious applications are determined to conceal their presence, for example by delaying or disguising their I/O activity. Nevertheless, we adopt two important concepts of that solution. First, we want to control the input of the system, i.e., the pattern of the issued keystrokes. By obtaining a detection environment where the input to the system is known, we can compare it to the memory write patterns a process exhibits. Second, we rely on the Pearson product-moment Correlation Coefficient (PCC from now on) to determine the correlation between the two patterns. The reason of this choice is twofold. First, the detailed analysis made in [13] provides a solid background to use PCC as a metric to infer malicious behavior. Second, the level of granularity of our detection technique advocates for a detection strategy that is robust against arbitrary data transformations that reflect the complexity of memory write activity. This

³ <https://klimax.few.vu.nl>

allows us to ignore the mere amount of bytes written due to an intercepted keystroke. However, in order to do any statistical reasoning, we must be able to map both the input pattern to a stream of keystrokes, and the amount of bytes written to an output pattern. We address this concern by adopting the same abstract keystroke representation introduced in [13] that discretized and normalized the stream. (we invite the reader to consult the paper for more details).

3 Our Approach

In our approach we aim to ascertain the correlation between the stream of issued keystrokes and the memory writes a process exhibits. In case a high correlation between those is found, the monitored process is flagged as malware with keylogging behavior. It is important to notice that in our approach we issue the keystrokes without any application on the foreground. This is to explicitly trigger any eavesdropping behavior in the background, and, at the same time, avoid the common case of a simple word-processing application raising false alarms. Malware that explicitly injects itself into a legitimate running process to eavesdrop keystrokes of a target foreground application is discussed in Section 6.

Profiling memory writes is a fairly complex task. First, even a simple program performs a huge amount of memory writes in a short period of time. Second, memory management in the modern x86 architecture is partly responsibility of the operating system (OS) and partly delegated directly to the hardware. While software-managed events like page faults are in complete control of the OS, tasks that occur more frequently like linear-to-physical address translations are performed directly by the hardware. The OS has no means to intercept or monitor these events. Performing differential analysis over multiple snapshots of the physical memory is another loose end: multiple writes performed on the same memory location would be detected as a single memory write.

The complexity of this challenge advocates for a low-level solution. Since we wanted our solution to be widely adopted and ready to be deployable in existing production systems, we ruled out the option of using any form of software or hardware virtualization support, and opted for a kernel-level solution. This choice is also crucial to access detailed information on execution contexts and memory regions that is only available in the kernel. Knowledge about the running thread and the DLL being used serves to our fine-grained analysis to better isolate and profile the keylogging behavior among the many possible concurrent activities performed by the malware. An obvious requirement for our solution is also the ability to access this information in a thread-safe manner.

In exchange for a low-level development environment, operating in kernel-space provides us with many advantages: we can intercept and to some extent control the memory management, override the kernel data structures, access real-time information, and most importantly, isolate our infrastructure from user-space threats thus adopting a limited trusted computing base (TCB). This allows us to target a broad class of malware, only ruling out kernel rootkits. In addition, kernel-level events can be intercepted and used to trigger malware analysis on

demand when using our detection technique in reactive mode, as better explained in Section 6. Figure 1 displays a high level view of our solution as a three-tier architecture. The three components are the monitor, the injector, and the detector, of which only the first two are designed to run in kernel space. Even if in our solution the detector is implemented as a user-space component, it can be easily moved into the kernel to further limit the TCB. The monitor exposes a memory write performance counter to the injector, and is divided into two sub-components, the shadower and the classifier. The former takes care of intercepting each memory write performed by the monitored process. The latter classifies which memory region has to be monitored, and which memory write has to be counted.

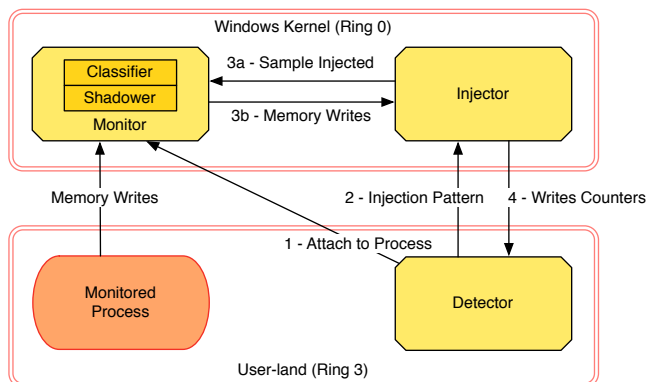


Fig. 1. High-level architecture.

Given a process to be analyzed for keylogging activities, our detection technique works as follows. First, we move the focus of the graphical user interface to the desktop. Then, the detector instructs the monitor to intercept the memory writes of the target process. The classifier classifies the memory regions of interest. Only for those memory regions the monitor instructs the shadower to intercept any memory access. The detector, after establishing the nature and length of the pattern to be used, sends its stream representation to the injector. The injector has now knowledge of the number of keystrokes it has to inject for each time interval. The detection process can now start: for each sample the injector issues the determined number of keystrokes to system, and notifies the monitor that the sample has been injected. The monitor then replies with the memory writes that took place. Upon injection of all the samples, the injector finally replies to the detector with the all the memory write counters. The detector transforms the write counters into patterns, and it computes their respective correlations against the pattern previously injected. If any of the correlations is statistically significant, the process is flagged as a keystroke-harvesting malware.

The solution hereby explained has been implemented for Windows XP 32-bit version, but the general design is applicable to other OSes as well. The

kernel has been configured to run in single processor mode and without taking advantage of the Physical Address Extension (PAE). All the components can be easily updated to handle PAE and SMP kernels. Porting the implementation to either Windows Vista or Windows 7 requires the user to disable the PatchGuard security protection.

3.1 Detector

The pattern generation is the most important task carried out by the detector. As we explained in Section 2, a pattern is defined in terms of multiple parameters (N , T , K_{min} , and K_{max}) and a characteristic function that describes the underlying pattern distribution. In order to generate a pattern representation from these input specifications we used the statistical suite R [15]. To obtain low predictability of the pattern in question, we leverage all the standard random distributions supported by R. Throughout our tests adopting different distributions and parameters yielded comparable accuracy results, as already confirmed in [13]. Upon completion of the injection, the detector receives a detailed report of the memory writes the process performed. The report includes a set of write patterns classified per code segment and thread. Each of these patterns is further categorized basing on the written memory regions (data, stack, or heap). The detection process terminates with a correlation test against all the output patterns found. The process is then flagged as malicious when at least one of those shows a $PCC \geq 0.70$.

3.2 Injector

The injector runs in kernel space and is implemented as a virtual keyboard driver. Once it receives the injection pattern sent by the detector, it converts it into a stream of keystrokes, and starts injecting the samples. After each sample it retrieves the write counters from the monitor. Once the whole injection terminates, it forwards the write results to the detector. It may be argued that simpler solutions exist. For instance, the library function `SendInput` would have allowed us to run the whole component in user space, thus reducing the overall complexity. However, in order to keep a limited TCB and a higher-priority injection we opted again for a kernel-level solution.

3.3 Shadower

In the x86 architecture a memory access is cooperatively handled by the CPU and the OS. Each time a linear address is referenced, the processor checks for its validity. When the physical page is either not present or its access is restricted, the processor asserts the page fault interrupt (0x0E). It also pushes in the thread's stack contextual information of the fault: the page fault error code, the faulting address, the current instruction pointer (EIP), and the `eflags` register's content. Finally, the control passes to the OS kernel. In Windows XP the

page fault is handled by the `KiTrap0E` handler. The handler’s task is to explicitly invoke `MmAccessFault` that is in charge to determine the nature of the occurred page fault. If the page in question is paged out to the disk, a page-in command is issued. The control can now safely return to the very same instruction that triggered the fault, and the program’s execution continues. If the page fault was due to an access violation (for instance because of an illegal address referenced), an exception record is built, and passed down to the user program. This will often result in the application abruptly terminating with a message informing the user of a protection fault.

KLIMAX places itself in the middle of this execution flow, and exploits its internals to track down each time a memory address is referenced. The main idea is to protect all the process’ address space, intercepting each time the processor asserts the page fault interrupt to signal the access violation. Once we identify the instruction liable for the access violation, we disassemble it and calculate the number of bytes the instruction attempted to write. The main issue is how we make the program gracefully recover from the error, and continue its execution. Obviously we need to unprotect that memory region (otherwise it would be impossible for the program to continue its execution). However, if another instruction later accesses the very same memory region, it will find no protection in place, thus we would not be able to intercept this memory access. The only viable instant to restore the protection is exactly after the execution of the first instruction. The `x86` architecture provides a built-in feature to notify the program after the processor has executed an instruction. This feature is known as “single step”, and can be enabled by setting the trap flag in the `eflags` register. When the flag is enabled, the process asserts the debug interrupt (`0x01`) prior execution of the following instruction. By leveraging this feature we are able to protect back a memory region exactly once the instruction referencing it completes its execution. If we programmatically execute all the steps we hereby outlined, a program’s execution can be thoroughly monitored by means of its memory accesses.

In KLIMAX the shadower is the component that implements the memory protection and handles all the memory accesses. KLIMAX installs two customized interrupt handlers for both `0x0E` and `0x01` interrupts by modifying the processor’s Interrupt Descriptor Table (IDT). These two handlers are the only entry points needed to selectively unprotect and protect the accessed memory regions. As soon as we instruct KLIMAX to monitor a process, the shadower asks the classifier which memory regions shall be protected, and hence monitored. The classifier reports back the corresponding set of page table entries (PTEs). The shadower creates a shadow copy of all the PTE’s `Owner` bit, i.e. it sets their bit to 0. It then flushes the TLB. This is mandatory in order to cope with the TLB caching address linear-to-physical resolutions. In case the referenced linear address is cached in the TLB, the OS needs not to walk the page tables. In contrast, if the TLB is flushed, any access to the memory referenced by these PTEs will result in an access violation. Figure 2(a) depicts this scenario. When this occurs, the shadower (i) reverse-lookups the PTE that references the

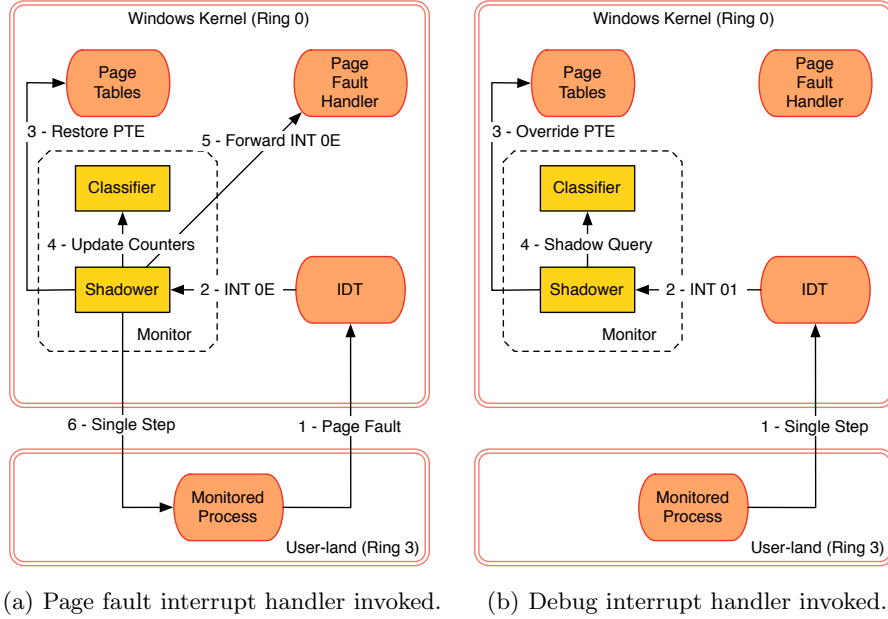


Fig. 2. The behavior of the shadower and classifier in both scenarios.

faulting address, then if the PTE is valid, it replaces the **Owner** bit with its original value; (ii) sets the trap flag in the pushed **eflags** register; (iii) stores the address that caused the page fault along with the current thread identifier in a private buffer. If the page fault error code reveals that the fault was because of a write attempt, the shadower invokes the classifier to update the performance counters. Before giving the control back to the OS we have to be sure that the current thread will be the next one to be executed. Otherwise any other thread being part of the same process would have access to an unprotected memory region. KLIMAX addresses thread safety by temporarily blocking, if present, all the other process' threads till the memory region is protected back. This may cause deadlocks if for some reason the same instruction causing a page fault blocks the current thread's execution. KLIMAX automatically intercepts these events, and restores the environment to safety by immediately protecting the memory region back, and by making the blocked threads runnable. Note that no memory write is lost in the entire process. Finally the control is given to the real interrupt handler **KiTrap0E**. The function **MmAccessFault** can now determine the real reasons of the page fault. In case no reason is found, that is the page was valid and the page fault took place only because of the shadower, the kernel gracefully resumes the program's execution. In any other case the kernel transparently executes all the steps required to resolve the page fault.

When the program resumes its execution, the very same instruction is executed for a second time. We point out that the same instruction may trigger

multiple page faults in case multiple memory regions are referenced. KLIMAX automatically handles these multiple page faults by following again the steps before outlined. Assuming that all the referenced memory regions are now unprotected, the execution continues till the following instruction, where, because of the set trap flag, the processor asserts the debug interrupt (Figure 2(b)). As a consequence, the shadower is again invoked (this time due to the 0x01 interrupt handler). It now checks in its private buffer which memory address previously faulted when the current thread was executing. It reverse-looks up the PTE and it replaces the `Owner` bit with the shadowed copy. Eventually it flushes the TLB entry by means of the `invlpg` instruction. There are cases in which the shadower does not have a shadow copy for that PTE yet. This happens when the original page fault occurred because the page was invalid. In such cases the classifier is once again invoked, and asked to determine whether the PTE shall be set protected. The program resumes its execution as soon as all the threads that KLIMAX previously blocked are restored to their original execution state.

3.4 Classifier

The classifier is invoked in two different courses of action: when the shadower needs to determine whether a PTE shall be protected, and to update the performance counters after a write took place. To determine if a PTE shall be shadowed, the classifier analyzes the PTE content. In a number of cases, the classifier replies negatively, for example when the PTE is not valid, or the PTE is not user accessible. In any other case it updates the PTE's shadow copy and replies affirmatively to the shadower. In case the classifier is invoked to update the performance counters, several steps are carried out. First, it uses the EIP to access the instruction that generated the page fault. It then disassembles it to extract the amount of bytes the instruction attempted to write. It also retrieves the original `ecx` register's value in case the faulting instruction was part of the `rep mov` family. This is a mandatory step because a `rep mov` instruction executes the `mov` instruction `ecx` times. Once the amount of bytes is computed, the classifier updates the performance counters. It uses the instruction to infer which executable component attempted to write (the main program or some DLLs). It also retrieves the current thread id, so it can discriminate writes performed by different threads. Depending on the particular memory location found, a memory write is recorded for the data region, the current thread's stack, or the heap.

4 Optimizing Detection Accuracy

In this section, we examine in detail how our design deals with potential sources of false negatives and false positives to maximize detection accuracy.

False negatives arise when a malicious application exhibiting keylogging behavior evades our technique and goes undetected. A first attempt for malware to evade detection is to spawn multiple processes and multiple threads and perform keylogging activity in any of newly created execution contexts. To deal

with this situation, our infrastructure supports simultaneous monitoring of multiple processes and multiple threads. Keylogging behavior is inferred from any highly-correlated memory write profile, put together on a per-thread basis.

Another important factor to consider is that malware authors strive to conceal the malicious behavior and exploit any possible information leakage channel available. To deal with this scenario effectively, KLIMAX monitors any memory writes performed by both the application code and the DLLs. This is crucial for two reasons. First, the keylogging activity may be implemented entirely in a DLL installed by the malicious application. Second, any form of information leakage that goes beyond harvesting keystroke-related data in memory must be mediated by the OS and typically exposed to the application via the library interface. We have experimented at length with many forms of information leakage, including storing keystroke-related data on the disk, recording information in the Windows registry, or sending data over the network. In all the cases, the memory write patterns exhibited by the system DLLs used to carry out these tasks showed extremely high correlation with our injected pattern.

A potential evasion strategy is to avoid using any system DLL and reimplement the API interface entirely without any significant memory writes that would otherwise trigger detection. While the concrete possibility of such a strategy remains to be explored—especially in multi-threaded contexts—, our implementation can be trivially extended to enrich the memory write profile with commonly used in-kernel performance counters that record and expose any form of I/O activity on a per-thread basis. In our analysis, however, we have not been able to identify any realistic example of this scenario in practice.

False positives arise when a legitimate monitored application shows high correlation with the injected pattern and triggers detection. In our preliminary experiments, we found many examples of benign applications showing high correlation when considering generic memory write patterns. In these cases, the application would typically register a callback to the kernel to intercept keystroke events, discriminate those of interest, and trigger some action (i.e. launch specific application) when a match against a predefined key sequence was identified. The high correlation was essentially triggered by the mechanics of invoking the programmer-provided callback—implemented in a system DLL (i.e. `USER32.dll` in the version of Windows we experimented with)—, and by transient memory write patterns observed on the stack at callback execution time.

To deal with these very common scenarios, our key observation is to concentrate the analysis exclusively on memory write patterns that clearly indicate a form of information harvesting or leakage. In this light, our implementation first avoids logging any memory writes performed by `USER32.dll`. As a result, this frequently-used system DLL becomes part of the TCB in our design. We believe this is not a serious limitation, since any common security suite solution constantly monitors system DLLs to detect any malicious attempt to replace them. As an option, our implementation can be trivially extended to perform similar integrity checks on core system DLLs and intercept attempts to replace

them. Note that `USER32.dll` does not expose any API that can be somehow exploited to leak keystroke-related data and potentially evade our technique.

Other sources of false positives are transient memory writes on the stack that are frequently used in the programmer-provided callback to implement the application logic. At a first glance, one might be tempted to exclude the stack from the analysis altogether. Unfortunately, an attacker could still leverage long-lived regions of the stack to harvest keystroke-related data and evade the resulting detection technique. Implementing this strategy is trivial and only involves allocating a sufficiently-large buffer on the stack in the entry point of the program (e.g. `main()`), and keeping a global pointer to access the buffer from the callback. To provide an effective solution to both problems, KLIMAX identifies long-lived regions of the stack during execution automatically and excludes any other stack region from the analysis.

To this end, we have designed an adaptive algorithm to safely identify long-lived stack regions for existing and newly created thread stacks. Initially, the entire stack is marked as long-lived and no memory write is excluded from the analysis. As the execution progresses, we sample the stack pointer of each thread under analysis at regular time intervals and update the deepest value found. This allows us to avoid any assumption on long-lived regions at thread initialization time when long-lived stack variables may not have been allocated yet. When a sampled value of the stack pointer falls behind the deepest value found, we finally observe the stack shrinking for the first time, and our adaptive identification strategy can safely start.

The first memory range we observe at the time when the stack first shrinks becomes the current long-lived region of the stack. As the stack keeps shrinking during execution, we update the long-lived region of the stack till convergence. This strategy follows the intuition that the stack pointer is always deeper than any long-lived stack variable used by the program with the exception of samples collected at thread initialization time. Our adaptive algorithm converges very quickly and causes only very few irrelevant memory writes on short-lived regions of the stack to be accounted for in the analysis at initial stages. Finally, note that ignoring short-lived regions of the stack in the analysis is hardly a concern for the generation of false negatives. An attacker can only temporarily harvest sensitive information on short-lived stack variables and any other global memory write pattern will still result in high correlation and trigger detection.

5 Evaluation

We have evaluated KLIMAX extensively, first with a synthetic keylogger to assess the ability to detect multiple forms of data harvesting, subsequently experimenting with realistic benign applications and malware to evaluate our detection accuracy in real-world scenarios. Our experiments were performed on a personal computer equipped with a 2.13GHz Intel Core i7 processor and 4 GB memory, running Windows XP Professional SP3.

5.1 Synthetic Evaluation

Our synthetic keylogger is a standard Windows application written in C++ in less than 100 lines of code. Our keylogger can be configured to emulate several forms of data harvesting, a feature which turned out to be very useful for evaluating the robustness of KLIMAX and for regression testing purposes during the development of the overall infrastructure.

Table 1. Synthetic test cases and resulting PCC values.

		Global+SLS	LLS	Disk	Network
keylogger.exe	Data	1	0	~1	0
	Stack	0	1	0	0
	Heap	1	0	~1	0
ntdll.dll	Data	-	-	0	0.76
	Stack	-	-	0	0
	Heap	-	-	~1	0.91
kernel32.dll	Data	-	-	0	0
	Stack	-	-	0	0
	Heap	-	-	~1	~1
mswsock.dll	Data	-	-	-	0
	Stack	-	-	-	0
	Heap	-	-	-	0.98
wshtcpip.dll	Data	-	-	-	0
	Stack	-	-	-	0
	Heap	-	-	-	0.94

In Table 1 we show the results of the most representative experiments conducted in common keystroke harvesting scenarios. In the table we represent every output distribution of interest showing at least one non-null value within the window of observation. Output distributions were produced at the finest level of granularity possible, to report PCC values for individual memory regions (i.e. data, stack, heap) of the program code (i.e. `keylogger.exe`) and of each DLL.

The first column of the table shows the correlation values estimated by KLIMAX for our synthetic keylogger configured to harvest every keystroke intercepted on the heap, on the data region, and on a stack variable allocated at callback execution time. As expected, full correlation is found on the heap and on the data region, while no activity was recorded and thus no correlation is shown for the short-lived stack variable.

The second column shows correlation results for our synthetic keylogger configured to harvest every keystroke intercepted on a long-lived stack buffer allocated in the entry point of the program. Thanks to the quick convergence of our adaptive algorithm to automatically track long-lived stack regions, full correlation is still found as a result of all the suspicious memory writes detected on the stack. We also tested our adaptive algorithm in several adverse conditions, for example, starting the analysis at initialization time or at thread creation time. In all the cases, the number of spurious writes in the initial stages of the algorithm was negligible and had no impact on the overall correlation values computed.

Finally, the last two columns of the table show correlation results for two other interesting scenarios: a keylogger logging every keystroke on the disk, and a keylogger sending every keystroke to a remote server. In both cases, the activity performed by the DLLs is reflected in very high correlation values that would

immediately trigger detection. Note that no DLL-originated memory write on the stack was recorded in any of test cases. Memory activity on the stack was only identified for short-lived variables, as expected. Also note that the high correlation values reported for memory write patterns on the heap and the data region in the third test case are actually produced by the C Run-Time Libraries, which on Windows are statically linked by default.

5.2 Malware Detection

To evaluate the effectiveness of our detection technique, we experimented KLIMAX with real-world malware. Our analysis started with obtaining a random sample of the malware dataset described in [16]. The original sample included 64 entries matching at least one keylogger-like label from all the results given by VirusTotal. Out of the 64 entries initially extracted, we isolated 23 malware samples that were categorized as active in the original dataset.

For all the identified entries, we conducted extensive analysis and manual inspection to determine the real nature of each sample and identify the presence of any relevant keystroke interception API used for keylogging purposes. Only in a few cases, the binary was neither packed nor obfuscated and basic static analysis was sufficient to extract the set of APIs used. In all the cases, however, we had to repeatedly perform dynamic malware analysis to determine whether any keylogging API was actually invoked at runtime. To carry out our analysis we experimented with the most common malware analyzers available online. In many cases, the analysis was made extremely difficult by malware trying to conceal and obfuscate their behavior, with explicit measures to evade several forms of static and dynamic analysis. We ran several experiments for each malware sample considered, even in cases when no keylogging API was detected by static or dynamic analysis. For these cases, it is important to assess whether any other malware activity could unexpectedly result in high PCC values and trigger detection. For all the other cases, high PCC values are to be expected every time a malware sample exhibits any form of keylogging behavior.

To simulate a realistic detection scenario, we assumed that no information was available on which of the running processes was the malware. To deal with this setting, we first waited to system to be idle, we then ran KLIMAX against all the processes for a limited amount of time ($N = 4$ and $T = 500$), and finally we flagged as candidate only the processes performing memory writes during a warm-up injection phase. This first step greatly reduced the number of candidate processes and allowed KLIMAX to examine only a few processes in a second step. In all our experiments (and in any realistic scenario on an idle system) the number of candidates rarely exceeded a handful of cases, thus allowing KLIMAX to later on analyze all the remaining processes in parallel, and minimize the detection time. During the second step of our analysis, we instead configured KLIMAX with $N = 20$ and $T = 500$, and triggered a successful detection in case of PCC values ≥ 0.70 . The remaining configuration parameters (K_{min} , K_{max} , and the underlying distribution of the pattern) played a negligible role in our experiments, hence producing similar results using different settings.

Table 2. Malware considered for analysis and resulting PCC values.

Malware Label	Keylogging API	API used	PCC
Backdoor.Win32.Poison.pg	✓	✓	~1
Trojan-Downloader.Win32.Zlob.vzd	-	-	negligible
Monitor.Win32.Perflogger.ca	-	-	negligible
Suspicious.Graybird.1	-	-	negligible
Trojan-Spy.Win32.SCKeYLog.am	-	-	negligible
Backdoor.Win32.IRCBot.ebt	-	-	negligible
Worm.MSIL.PSW.d	✓	✓	0.74
Worm.Win32.Fujack.cr	-	-	negligible
BackDoor.Generic9.MQL	✓	✓	~1
Trojan.Win32.Agent.arim	-	-	negligible
PSW.Agent.7.AH	✓	✓	0.78
Worm.Win32.AutoRun.adro	-	-	negligible
Trojan.Win32.Delf.eq	-	-	negligible
Net-Worm.Win32.Mytob.jxu	-	-	negligible
Trojan-Spy.Win32.SCKeYLog.au	-	-	negligible
Backdoor.Ciadoor	✓	✓	0.98
Backdoor.Win32.Agent.su	✓	-	negligible
Backdoor.Win32.G.Spot.20	-	-	negligible
Trojan-Spy.MSIL.KeyLogger.oa	✓	-	negligible
Downloader.Rozena	-	-	negligible
Downloader.Banload.BDRQ	-	-	negligible
Heur.Trojan.Generic	-	-	negligible
PSW.Generic7.BNDX	-	-	negligible

Table 2 shows the results of our evaluation for the set of malware samples considered. For each sample, we show: (i) the result of our static and dynamic analysis to identify any keylogging API; (ii) the result of our fine-grained analysis to determine whether the keylogging API was actually used at runtime; (iii) the maximum PCC value reported by KLIMAX for each process and each thread created by the malware sample at runtime. Negligible correlation is reported for PCC values below 0.1. The labels adopted to identify each malware sample are taken from common antivirus software—including Kaspersky, Symantec, and AVG—depending on availability and discrimination power.

As shown in the table, for 16 malware samples we were not able to identify any keylogging API and the resulting PCC values were always negligible, as expected. A manual inspection revealed that these samples were sometimes misclassified, in other cases we found downloaders instructed to download additional malicious software, in yet other cases we found privacy-breaching malware not exhibiting keylogging behavior (e.g. stored password stealers). Furthermore, in 5 cases, where the keylogging APIs were correctly identified and also used at runtime, KLIMAX always reported high correlation values triggering detection. Finally, in the 2 remaining cases, we identified the presence of keylogging APIs in the malware samples, but those APIs were never actually used at runtime. As a result, KLIMAX reported negligible correlation.

In both cases, we were able to easily analyze the runtime behavior of the malware and establish that no keylogging API was actually used. In the case of `Backdoor.Win32.Agent.su`, no memory write pattern could ever be recorded even when using very large windows of observation. The malicious application appeared to be completely idle and waiting for input from a remote server. In this case, it can be speculated that the keylogging behavior is only triggered on demand, when new input is received from the remote server. In the case of

`Trojan-Spy.MSIL.KeyLogger.oa`, intensive malicious activity was found in the memory write patterns recorded by KLIMAX, but not a single memory write was performed from the DLL that implements the keylogging API.

5.3 False Positive Analysis

We have evaluated KLIMAX with many common benign Windows applications to assess the robustness of our approach with respect to false positives. In the simplest cases, we experimented with applications not relying on any form of keystroke interception mechanism which always resulted in negligible correlation values, or, more often, no correlation at all. More interesting cases are those applications that do rely on some form of keystroke interception mechanism for legitimate purposes. This is the case for popular Windows shortcut managers, launchers, and key remappers. For this reason, we decided to concentrate our evaluation on these cases that are particularly prone to generating false positives.

We installed and tested a sample of the most popular free Windows applications in this category. For each application, we performed static binary analysis—and dynamic analysis when necessary—to extract the set of relevant Windows APIs used, all taken from `USER32.dll`. For our purposes, it is important to distinguish between generic keystroke interception APIs (e.g., `SetWindowsHookEx`, `GetKeyState`, `GetAsyncKeyState`), and hotkey registration APIs (i.e. `RegisterHotKey`). When `RegisterHotKey` is used, a programmer-provided callback is called only when the specified hotkey is detected by the kernel. Since `RegisterHotKey` only allows registering hotkeys with standard modifiers (i.e., `CTRL`, `ALT`, `SHIFT`, `WIN`), a carefully-chosen input stream adopted by the injector will essentially never trigger the execution of the programmer-provided callback and irrelevant correlation values are to be trivially expected.

Luckily, the majority of the hotkey managers we have encountered rely on both `RegisterHotKey` and some other standard keystroke interception API to provide a broader range of features. Testing applications that always make use of standard interception APIs is crucial to make our false positive analysis more effective. When necessary, we updated the default configuration of each application to trigger all the necessary code paths that forced the program to use standard keystroke interception APIs. Before running each experiment, we manually verified this assumption using dynamic analysis.

Table 3. Applications considered for false positive analysis and resulting PCC values.

Application	Standard API	RegisterHotKey	PCC
HoeKey 1.13	✓	✓	negligible
KeyTweak 2.3.0	✓	-	negligible
Hot Key Plus 1.01	✓	✓	negligible
AutoHotkey 1.0.96.00	✓	✓	~1
ZenKEY 2.3.9	✓	✓	negligible
Aquarius Soft Keyboard Hotkey 2.5	✓	✓	negligible
Hotkey Recorder Version 2	✓	-	negligible
HotKey Magic 1.3.0	✓	-	negligible

Table 3 shows the results of our analysis for the set of applications considered. For each application, we show the APIs identified using static and dynamic

analysis, and the resulting correlation values found. For brevity, we show a single correlation value for each application, which represents the maximum correlation value found over all the output distributions considered on a per-process per-thread basis. Negligible correlation is reported for PCC values below 0.1.

Our analysis shows that in only 1 case KLIMAX reported non-negligible correlation values. It is important to remark that in all the other cases high correlation values would have been still reported if we had not explicitly ignored any memory write patterns on short-lived stack regions or any memory writes generated by `USER32.dll`. In the case of AutoHotkey, arguably the most popular hotkey manager for the Windows platform, the high correlation value reported admittedly calls for immediate detection.

A closer inspection reveals that AutoHotkey stores all the keystrokes intercepted in a global buffer to implement advanced features and provide a scriptable interface for the user to handle the keystroke collected in the most convenient way. This experiment confirms the conservativeness of our approach, which aims to signal any form of sensitive data harvesting as dangerous, even without explicitly tracking down information leakage.

Ironically, the case of AutoHotkey shows that our analysis is rarely overly conservative. A quick web search reveals that the scriptable interface of AutoHotkey does allow the user to transfer the previously stored keystrokes elsewhere and implement a fully-fledged keylogger in as few as 8 lines of code.

6 Discussion

From the experiments presented, some important properties of our approach have distinctly emerged. First, we confirmed that in-memory keystroke data harvesting can be used as a good predictor to detect sensitive information leakage. Our detection strategy was successful in detecting all the malware samples examined that explicitly used keystroke interception APIs and exhibited keylogging behavior. The main strength of our detection strategy is to be able to detect keylogging behavior within short windows of observation even for malware buffering sensitive data in memory for a long time. In contrast, existing techniques that attempt to detect information leakage explicitly yield a higher number of false negatives in the general case, unless an indeterminately large window of observation can be possibly used. For example, an information leakage tracking mechanism would probably require a window of observation of days, if a malware were to use a sufficiently large buffer to harvest a substantial number of keystrokes before transferring all the data elsewhere.

Second, keystroke data harvesting, when identified correctly, leaves a small margin for false positives. Although it is not possible to draw final conclusions in the general case, we have only encountered a single hotkey manager that was signaled as suspicious. As mentioned earlier, this application can indeed be configured to behave like a keylogger and our detection result reflected its behavior. An important remark is that false positives are to be expected for benign applications that unnecessarily harvest sensitive data in global memory regions.

Consider, for example, a sloppy shortcut manager implementation that allocates all the temporary variables on the global data region. While it is impossible to rule out the existence of these cases in general, we have not encountered any example of realistic application in this category during our analysis. Furthermore, in cases where sensitive data harvesting were truly unnecessary, it would be straightforward to adapt the particular application under analysis to work with our detection technique. As far as false negatives are concerned, our technique, when used to proactively detect keylogging behavior, suffers from coverage problems common to existing solutions that attempt to build models based on dynamic malware behavior [6]. Namely, if the expected behavior is never triggered within the window of observation but somewhat later, the resulting model can potentially miss some of the fundamental properties intended. In our experimental analysis, we have seen only two candidate malware samples that could possibly belong to this category. In these two cases, we have speculated that the keylogging behavior might only be triggered when an event of a particular nature occurs. Under these circumstances, our proactive strategy may not be able to infer detection successfully within the window of observation.

While we believe that the problem of triggering a specific malicious behavior is orthogonal to our work and is focus of much prior research [11,2,3], our infrastructure design is intended to mitigate this issue. We explicitly designed KLIMAX to also support reactive detection with practically no runtime overhead. From the moment KLIMAX is installed into the kernel, some slowdown can only be perceived for the particular application under analysis. This means that we can leave KLIMAX inactive inside the kernel without any performance problem and reactively activate our analysis on a target application only when some particular event occurs. At the kernel level, we have the ability to support almost arbitrary detection policies driven by monitored system events. For example, a reactive detection policy might consider starting the analysis whenever a system call that registers a keystroke-interception callback is issued by a given application. This will immediately trigger a behavior analysis of the application. If no detection is found, another policy might consider repeating the same analysis on the same application every m minutes, to determine whether the behavior of the callback changes overtime in face of some particular event. Although we have not explicitly evaluated the performance of such policies at the system call level, we envision a negligible runtime overhead. The evaluation of policy-driven detection mechanisms is part of our ongoing work.

Another source of false negatives is given by malware trying to perform denial-of-service attacks or confuse our detection technique. A first important observation is that carrying out this attack successfully is not entirely trivial if we allow KLIMAX to perform a multi-stage analysis with different configuration parameters for each stage (i.e., typically increasing the size of the time interval at every stage). Second, we remind that the adopted correlation metric is known to be robust against attempts to break the correlation by disguise-ment. For example, in [13] we show that the PCC is not affected by keyloggers writing to a file a random number of bytes for each intercepted keystroke. Finally, a

malicious application performing any DOS attack should also avoid introducing an excessive delay not to miss subsequent keystrokes. This is the reason why buffering the intercepted keystrokes on the short lived stack for too long is also not an option to evade our detection technique.

7 Related Work

Malware detection has always proved to be a challenging task. If early detection mechanisms relied on signatures to counter this plague, code obfuscation or polymorphism easily affected the technique’s accuracy. To overcome this problem, behavior-based approaches [20] started to focus on sequences of system or library calls to profile the behavior deemed malicious. Unfortunately, since the sequence of syscalls only describes a certain implementation rather than a general behavior, building a malware evading this technique was a trivial task. Other approaches overcame this limitation by focusing on information flows rather than on mere sequences of syscalls. Malware profiles, by leveraging more-contextual information in terms of library [5] or system calls [9,6], started to grasp the semantics lying behind a malicious activity. However, mimicry attacks were still possible [7]. To address this concern, Lanzi et al. [8] recently proposed system-centric profiling of benign applications. This approach results in low false positives, without hindering the detection accuracy.

All the approaches hereby mentioned, however, can not cope with malware practically identical to benign applications in terms of system and library calls, without generating a significant number of false positives. As we showed in Section 5.3, malicious applications with keylogging abilities share huge portions of their logic with rather common user applications. In light of this concern, many approaches recently emerged to detect keylogging activities [1,4,13]. Instead of focusing on the APIs used to intercept the keystrokes, they have tried to measure the potential correlation with the APIs in charge of leaking this information. However, while this approach may be effective against commonly used keyloggers, they can not easily detect malicious applications concealing their presence by aggressively harvesting sensitive data and hiding leakage to any possible extent.

This clearly advocates for more fine-grained approaches. Unfortunately, even taint analysis proved itself ineffective in detecting malware harvesting user-issued keystrokes [18]. In our work, we ignore the concept of tainting, and instead leverage the behavior profiled by a fine-grained memory analysis. This is achieved by shadowing the entire memory address space of the monitored program. To our knowledge, similar approaches have only been adopted to evade rootkit detection [19] or to automatically unpack unknown malware [14]. Our memory monitoring strategy is similar, in spirit, to the technique proposed by Miller [10]. However, his solution did not monitor the whole address space, nor did it provide strong thread-safety guarantees. Since our infrastructure is to be used for malware analysis and detection, our design explicitly took into account every

memory write performed by any process' component to rule out the possibility of false negatives.

8 Conclusions

Traditional malware detection techniques are either signature-based or rely on coarse-grained behavioral profiles that model the interaction of a given application with the environment. In the present paper, we focused on detecting a particular class of malware exhibiting keylogging behavior, and argued that both models are ill-suited for the task. In addition, existing keylogger detection techniques are either not tailored to generic malware analysis and detection or heavily prone to generation of false positives. To address these concerns, we presented KLIMAX, a kernel-level infrastructure that we proposed to analyze and detect malware with generic keylogging behavior. Our prototype can be deployed on unmodified Windows-based production systems without interruption of service. To infer keylogging behavior, we inject a carefully-crafted keystroke stream into the system and observe the resulting memory write patterns of the target process.

The experimental results of our proactive detection technique show that our system leaves practically no margin for false positives and allows for no false negatives when the keylogging behavior is triggered within the window of observation. To address trigger-based keylogging behavior, our design supports policy-based reactive detection that allows for practically no false negatives in the general case. In our evaluation, we also found that almost every malware sample with keylogging behavior was misclassified by a number of antivirus programs. This suggests that our infrastructure can also be used in large-scale malware analysis and classification to help recognize and classify emerging privacy-breaching threats in a more accurate way. Finally, we believe that the general model proposed in this paper can potentially be reused to identify other classes of malware. Extending the scope of our detection technique to a broader range of malicious activities is part of our future work.

References

1. Al-Hammadi, Y., Aickelin, U.: Detecting bots based on keylogging activities. Proceedings of the Third International Conference on Availability, Reliability and Security pp. 896–902 (2008) [18](#)
2. Bowen, B., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. *Advances In Information Security* 36, 65–88 (2008) [17](#)
3. Bowen, B., Prabhu, P., Kemerlis, V., Sidirolou, S., Keromytis, A., Stolfo, S.: Botswindler: Tamper resistant injection of believable decoys in vm-based hosts for crimeware detection. Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID 2010) pp. 118–137 (2010) [17](#)

4. Han, J., Kwon, J., Lee, H.: Honeyid: Unveiling hidden spywares by generating bogus events. Proceedings of The IFIP TC11 23rd International Information Security Conference pp. 669–673 (2008) [18](#)
5. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. Proceedings of the 15th USENIX Security Symposium (SSYM '06) pp. 273–288 (2006) [2](#), [18](#)
6. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. Proceedings of the 18th USENIX Security Symposium (SSYM '09) pp. 351–366 (2009) [17](#), [18](#)
7. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. Proceedings of the 14th USENIX Security Symposium (SSYM '05) pp. 11–11 (2005) [18](#)
8. LANZI, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: AccessMiner: Using system-centric models for malware protection. Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010) pp. 399–412 (2010) [2](#), [18](#)
9. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A layered architecture for detecting malicious behaviors. Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008) pp. 78–97 (2008) [18](#)
10. Miller, M.: Memalyze: Dynamic analysis of memory access behavior in software. Uninformed Journal 7 (2007), <http://uninformed.org/?v=7&a=1> [18](#)
11. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. Proceeding of the 28th IEEE Symposium on Security and Privacy (SP '07) pp. 231–245 (May 2007) [17](#)
12. Open Security Foundation: DataLossDB. http://datalossdb.org/statistics?timeframe=last_month (April 2011) [1](#)
13. Ortolani, S., Giuffrida, C., Crispo, B.: Bait your hook: a novel detection technique for keyloggers. Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID 2010) pp. 198–217 (2010) [3](#), [4](#), [6](#), [17](#), [18](#)
14. Quist, D., Ames, C.: Temporal reverse engineering. Black Hat Briefings (2008) [18](#)
15. R Development Core Team: R: A language and environment for statistical computing. <http://www.R-project.org/> (2008) [6](#)
16. Rossow, C., Dietrich, C., Bos, H., Cavallaro, L., van Steen, M., Freiling, F., Pohlmann, N.: Sandnet: Network traffic analysis of malicious software. Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS 2011) (2011) [13](#)
17. Sharp, D.: Maine park users warned of credit card breach. http://www.mercurynews.com/california/ci_17691495 (April 2011) [1](#)
18. Slowinska, A., Bos, H.: Pointless tainting?: evaluating the practicality of pointer tainting. Proceedings of the Fourth ACM European Conference on Computer Systems (EuroSys '09) pp. 61–74 (2009) [18](#)
19. Sparks, S., Butler, J.: Shadow walker: Raising the bar for windows rootkit detection. Phrack Inc. 0x0b (2005) [18](#)
20. Xu, J.Y., Sung, A., Chavez, P., Mukkamala, S.: Polymorphic malicious executable scanner by api sequence analysis. Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS'04) pp. 378–383 (2004) [18](#)