Keyblock: a software architecture to prevent keystroke injection attacks

C. David B. Borges, J. Rafael B. de Araujo,
Robson L. de Couto, A. Márcio A. Almeida

Computer Engineering – Universidade Federal do Ceará (UFC)
Campus Mucambinho – Sobral – CE – Brazil

david.borges@protonmail.com, rafael.acurcio@gmail.com
robsoncouto@alu.ufc.br, marcio.albu@alu.ufc.br

Abstract. *This work investigates a solution to mitigate the threat of keystroke injection attacks. Current defense mechanisms often require relatively expensive hardware and time consuming configuration. We describe and test the effectiveness of a software layer between USB input hardware and processes. Our software, Keyblock, intercepts events from newly connected devices and uses keystroke dynamics analysis to detect whether an attack is in course. By detecting and immediately disabling devices with anomalous typing patterns, Keyblock provides a software-only automatic solution to prevent keystroke injection.*

1. Introduction

The widespread use of USB technology has advantages and disadvantages. One advantage is that plug-and-play, a requirement in the USB specification, makes the attachment and setup of new USB devices a straightforward task. Nevertheless, automatic hardware configuration can be exploited, specially when concerning human interface devices (HID), such as keyboards. It is possible to disguise a microcontroller as a USB keyboard by providing the target computer with a fake HID descriptor. An attacker can exploit this security flaw by setting up a device to enter predefined keystrokes into a shell terminal running on a target computer [SRLabs 2014]. This attack is known as keystroke injection.

In [SRLabs 2014], flaws in USB plug-and-play were exploited with the use of a microcontroller and a storage device coupled together. The authors demonstrated attacks in which an adversary could gain administrative access to Windows and GNU/Linux machines. These attacks were able to install a keylogger in the target computer, gain root access and infect other USB devices. These vulnerabilities also acquired popularity in the hacker community. A tool that exploits USB plug-and-play by HID emulation, known as Rubber Ducky, is available online. The device, which includes a 60 MHz 32-bit CPU and Micro SD storage, is disguised as a regular flash drive and programmed to perform keystroke injection attacks when connected to a computer. In fact, these attacks can be performed using any USB microcontroller capable of USB HID emulation.

It is important do develop defensive measures against this threat. In this paper, we describe, implement and test the effectiveness of a software layer between USB input hardware and processes. Our program, Keyblock, intercepts events from newly connected keyboards and detects whether an attack is in action by performing keystroke dynamics analysis. Our goal is to classify and automatically disable devices that present suspicious typing patterns, a characteristic of keystroke injection attacks. The main contribution of

© 2017 Sociedade Brasileira de Computação

this paper is a software-based keystroke injection detector with 0% error rate on prevalent attacks and near 0% false positives for standard keyboards. Moreover, Keyblock does not interfere with plug-and-play and requires no user configuration.

2. Related Work

The USB flaws that allow unwanted HID emulation and keystroke injection have drawn attention from the security research community and some protection mechanisms have been proposed. One measure to mitigate this threat is to require user interaction before allowing an HID to generate events that target other applications. This measure has been proposed and implemented in software and hardware.

[GData 2014] distributes a program that blocks events from new keyboards until permission is granted when the user clicks a pop-up message. Another, more robust, software-based approach is GoodUSB, a mediation architecture in the USB stack, proposed by [Tian et al. 2015]. It prompts the user to define the capabilities of new USB devices and restricts access to drivers that perform any other unwanted functionality, effectively preventing HID emulation.

Another solution, USBCheckIn, is a hardware-based HID protection system, proposed by [Griscioli et al. 2016], that asks the user to solve a random challenge before forwarding events to the computer's USB ports. Hardware interfaces between a computer system and USB peripherals are proposed by [Kang and Saiedian 2015] and [Loe et al. 2016]. They implement USBWall and SandUSB, respectively, systems that enumerate newly connected devices and allow the user, via software, to verify whether the peripheral is trusted before allowing it to operate. SandUSB has the advantage of detecting some attacks using keystroke dynamics analysis.

Different algorithms and applications have been proposed for keystroke dynamics analysis. User authentication by analysis of typing patterns is proposed by [Trojahn and Ortmeier 2013]. They classify users with attributes such as key press timing and interval between different key presses. Their results show that it's possible to obtain error rates below 2% using a mixture of keystroke and handwriting recognition. Not limited to physical keyboards, their study presents data related to touch screen devices including features such as pressure and swipe distances. [Killourhy and Maxion 2009] and [El-Abed et al. 2014] perform benchmarks of the main algorithms for anomaly detection and user biometrics, respectively, based on keystroke patterns. The use of keystroke analysis for detection of HID attacks was first proposed by [Barbhuiya et al. 2012]. The authors propose one possible architecture, useful keystroke parameters and metrics.

The majority of the cited solutions to alleviate keystroke injection risks depend on relatively expensive hardware, such as [Griscioli et al. 2016], [Loe et al. 2016] and [Kang and Saiedian 2015]. Others, such as [GData 2014] and [Tian et al. 2015], require a set of configurations steps to be performed for every new device. One of the goals of the proposed software, Keyblock, is to allow software-based keystroke injection detection with no required configuration steps. The work by [Barbhuiya et al. 2012] is the closest to our proposed solution, as it primarily uses keystroke dynamics to perform decisions. However, Keyblock does not require additional steps or configuration such as capturing keystroke samples and creating a user profile. In addition, our software uses a different detection architecture and a simpler classification method.

 © 2017 Sociedade Brasileira de Computação

3. Principles of Designing Secure Systems

Any device that complies with the USB specification is vulnerable to HID emulation and keystroke injection attacks. Current anti-virus software is not able to protect against this threat [Tian et al. 2015]. But these plug-and-play exploits exist because current USB standards ignore certain security principles. When considering computer security, it is important to continuously reexamine all assumptions about threat agents and the environment. The principle of complete mediation states that every access and every object should be checked every time [Beznosov 2015]. A system should regard the environment where it operates as inherently hostile and be reluctant to trust [McGraw 2013]. In the case of USB HID authentication, the object is a new USB device that requests access to the USB bus as an HID. While configuring a new device, the only verification is related to device class. However, because class information is provided by the device itself, it cannot be considered reliable. After the drivers are loaded and the device is configured, there is no further verification concerning its behavior. Therefore, current USB protocols neglect the principles of questioning all assumptions, complete mediation and reluctance to trust, thereby giving rise to a security flaw. Keyblock fills this hole and allows a system to continuously verify if a device should be trusted or not.

4. Keyblock and monitor threads

Keyblock was written in C/C++ and, currently, only supports GNU/Linux systems. Its purpose is to be run as a daemon, requiring no interaction or configuration by the user. The main loop of Keyblock continuously read the list of available input devices located in /dev/input/ and starts a monitor thread for every new device.

A monitor thread is responsible for reading every input event from a given device and decide whether it should be blocked or continue to operate normally. The operation of a monitor thread is summarized in figure 1. As a security measure, no key event is allowed to reach applications running on the X system, or graphical user interface, without analysis from the monitor thread. Keyblock uses the ioctl system call in order to configure a mechanism for monitor threads to grab and consume all events from a device. This prevents a key press from directly reaching any X application. The thread registers key press timestamps with nanosecond precision and only keeps in memory a window containing the last N events.

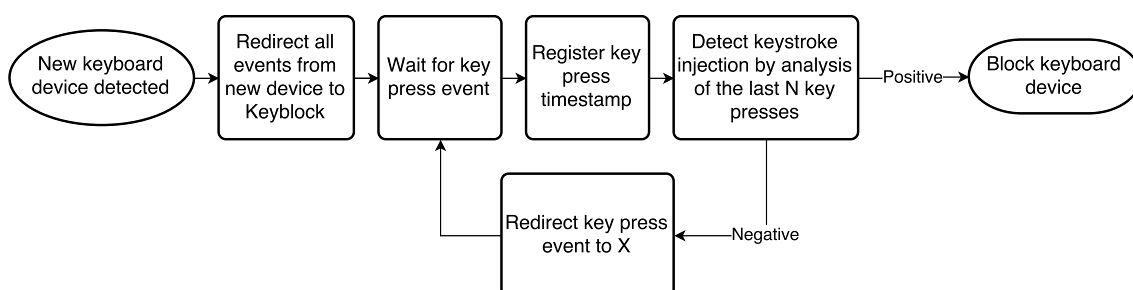

Figure 1. Monitor thread operation

When a new key press is registered, the monitor thread executes the detection phase. This phase is represented by a function that receives data about the last N key events and returns true, if it identifies a possible attack, or false, if there is no unusual

behavior. In the latter case, the key event is redirected to the X system. However, if unusual typing patterns are detected by a monitor thread, the corresponding device is automatically blocked until removal.

5. A simple detection model

The attack detection phase seen in figure 1 can be implemented in many ways. In principle, it is a function with a fixed length input vector that returns true or false. If true, the monitor thread will block the device, else, it will redirect the key press to the X system. This description allows great flexibility. One can use keystroke latencies or hold times as input and perform classification using any of the available methods in the literature. For our experiments, we developed two very simple classifiers, derived from empirical analysis of two keystroke latency datasets: one produced by humans while typing and another generated by our rogue microcontroller during keystroke injection tests.

The first dataset, $\mathcal{D}_1$, provides keystroke latencies from human writers and was collected by [Calot 2015]. It contains delays between combinations of successive key presses and releases, gathered during the period of one week. From this dataset, one is able to extract statistics of human keystroke latency and hold time. Following this approach, we generated a second dataset, $\mathcal{D}_2$, containing similar information to the first, in order to characterize keystroke injection attacks, using our test tool, an ATMEGA32U4.

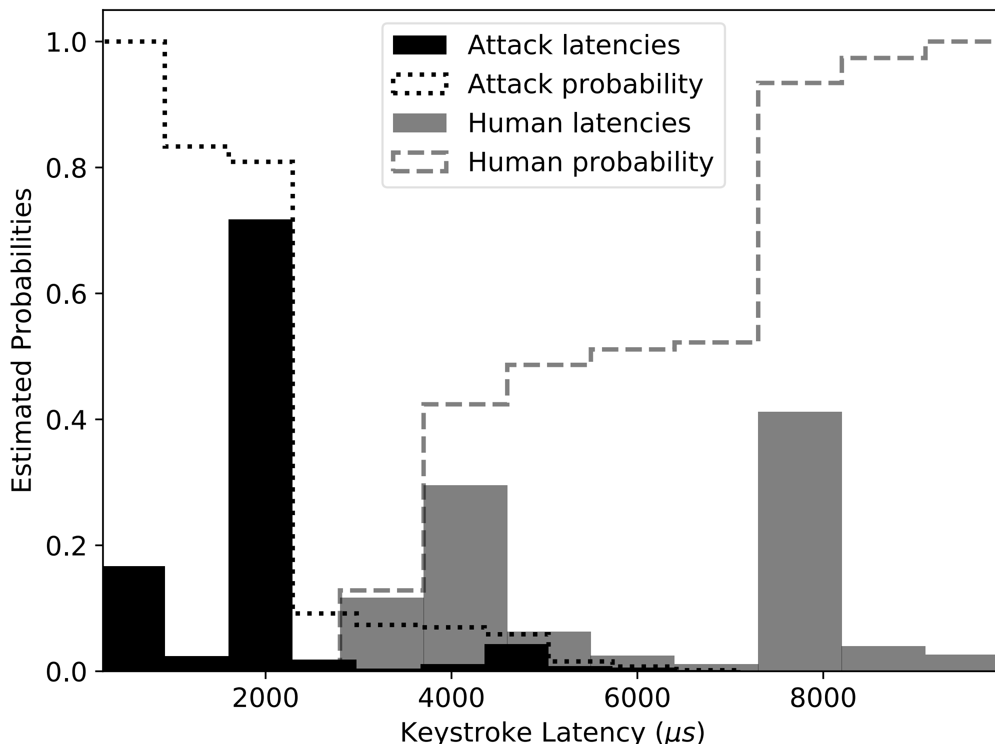

Figure 2. Estimated probabilities of keystroke injection attacks and human operators for latencies below 10 ms from $\mathcal{D}_1$ and $\mathcal{D}_2$

.

Our primary detection model derives from the observation that keystroke injection attacks usually have the purpose of introducing a payload as fast as possible, thus leaving

 © 2017 Sociedade Brasileira de Computação

a low latency footprint. Therefore, in principle, it's possible to discriminate between a human operator and an attack tool by defining a latency threshold. Figure 2 shows the lowest human typing latencies registered in $\mathcal{D}_1$, along with attack latencies in $\mathcal{D}_2$. The dashed line is a cumulative density function (CDF) matching the observed human latencies. The dotted line is a reverse CDF based on the latency distribution for keystroke injection attacks. From figure 2, according to latencies from $\mathcal{D}_1$ and $\mathcal{D}_2$, we see that there is an overlap between both distributions, specially in the interval from $3\ ms$ to $5\ ms$. For this reason, using latency alone, it is difficult to decide whether a set keystrokes are the product of an attack or a fast typing human.

If we fit a normal distribution, with mean μ and standard deviation σ, to the observed human latencies below $50\ ms$, the result is $\mu = 31.64\ ms$ and $\sigma = 9.63\ ms$. Our detection model takes into account how far the last N latency samples deviate from the expected human latency distribution. In this case, only left-side deviation from μ is considered, as arbitrary deviations to the right are common when a human operator is typing. Another characteristic of keystroke injection attacks is that keys are injected at an approximately steady rate. Our detector is improved by measuring the standard deviation of the last N keystrokes and comparing to the σ of an expected distribution.

Our detector is composed of two thresholds. The deviation threshold τ_d computes how far is the average of the last N latencies to μ; it is triggered if keystrokes are fed to the system with a much higher frequency than the average expected for a human. The stability threshold τ_s measures how stable are the last keystroke latencies; it is triggered if latencies have near constant values,producing a standard deviation much lower than the expected σ. The current Keyblock implementation defines $\tau_d = \mu - 3\sigma$ and $\tau_s = 0.1\sigma$.

6. Experiments

We tested how effectively Keyblock is able to detect keystroke injection attacks and how fast they are blocked using the model described in section 5. We used an ATMEGA32U4 to emulate a USB keyboard and inject a set of keystrokes in a target computer that runs Keyblock. Our simulated attacks mimic the behavior of standard keystroke injection tools, such as Rubber Ducky and Teensy. The ATMEGA32U4 was chosen as test platform, instead of the alternatives, Rubber Ducky and Teensy, because the former is cheaper and allows great programming flexibility. The ATMEGA32U4, which is USB 2.0 compliant, has one 8-bit processor core, can run up to 16 MIPS and has 32 kB of program flash memory [Atmel 2015].

We performed 100 keystroke injection experiments and registered whether each attack was successfully blocked and how many keystrokes were accepted before the offending device was disabled. Moreover, we tested if Keyblock could effectively identify and allow humans to operate normally with standard USB keyboards. The process of connecting a new keyboard and typing at different speeds was repeated 114 times. The resulting false negative and false positive rates are summarized in table 1. Our tests included two deviation threshold values, namely $\tau_d = \mu - 2\sigma$ and $\tau_d = \mu - 3\sigma$. A false negative happens when Keyblock incorrectly allows a keystroke injection attack to proceed. A false positive is registered if Keyblock erroneously disables a newly connected keyboard while a human is typing.

For $\tau_d = \mu - 2\sigma$, we performed 50 keystroke injection tests with an AT-

MEGA32U4 and 64 tests with typing humans. No attack was allowed to proceed, however we registered that 12 out of 64 human keyboard sessions were blocked, implying a 18.75% false positive rate. This means that almost 1 in 5 human sessions were erroneously closed. For $\tau_d = \mu - 3\sigma$, we carried out 50 keystroke injection tests and 50 human keyboard sessions. Again, there was a 0% false negative rate. However, the false positive rate was reduced to 2%, as only 1 human session was blocked. Additionally, in these experiments, not a single keystroke injected by an attack was incorrectly allowed to reach the X system.

Table 1. Results

	$\tau_d = \mu - 2\sigma$	$\tau_d = \mu - 3\sigma$
False negatives	0%	0%
False positives	18.75%	2%

7. Conclusion

The effectiveness of Keyblock's detection architecture (figure 1) was demonstrated in our experiments. This conclusion is drawn from the fact that Keyblock, using a relatively simple detection model, can acquire a 0% false negative rate while keeping false positive rates as low as 2%. The detection model used in the experiments was based on two thresholds, derived from latency distributions measured for standard keystroke injection attacks and human operators. Notwithstanding its simple classification method, it was able to block the most prevalent category of keystroke injection attack, in which key events are generated at speeds above that of a human, with a near constant frequency. More sophisticated attacks can be developed, such as imitating human typing patterns. Such an attack would hardly be blocked by a detection model based on latency only. However, our software architecture offers extensibility for implementation of new keystroke parameter measurements and classification techniques. The next step in this project is to develop and test other keystroke statistics, like key hold time, error rate and general typing habits, and more robust classification methods, such as linear discriminant analysis, support vector machines and neural networks. Although our current implementation is focused on GNU/Linux, the same architecture can be implemented as a tool for other operating systems, including Windows and Mac. [Keyblock 2017] is released as open-source software under the CC Attribution-NonCommercial 4.0 license.

References

Atmel (2015). Atmega32u4 datasheet. [Available] http://www.atmel.com/Images/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf [Access 30-07-2017].

Barbhuiya, F. A., Saikia, T., and Nandi, S. (2012). An anomaly based approach for hid attack detection using keystroke dynamics. In *Proceedings of the 4th International Conference on Cyberspace Safety and Security*, CSS'12, pages 139–152, Berlin, Heidelberg. Springer-Verlag.

Beznosov, K. (2015). Computer security: Principles of designing secure systems. [Available] http://courses.ece.ubc.ca/cpen442/sessions/08-design_principles.pdf [Access 30-07-2017].

 © 2017 Sociedade Brasileira de Computação

Calot, E. P. (2015). Keystroke dynamics keypress latency dataset. Database.

El-Abed, M., Dafer, M., and Khayat, R. E. (2014). Rhu keystroke: A mobile-based benchmark for keystroke dynamics systems. In *2014 International Carnahan Conference on Security Technology (ICCST)*, pages 1–4.

GData (2014). Usb keyboard guard: How to be sicher from usb attacks. [Available] https://www.gdatasoftware.com/en-usb-keyboard-guard [Access 30-07-2017].

Griscioli, F., Pizzonia, M., and Sacchetti, M. (2016). Usbcheckin: Preventing badusb attacks by forcing human-device interaction. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 493–496.

Kang, M. and Saiedian, H. (2015). Usbwall: A novel security mechanism to protect against maliciously reprogrammed usb devices. *Information Security Journal: A Global Perspective*.

Keyblock (2017). Keyblock. [Available] https://github.com/cosmonautd/Keyblock [Access 17-10-2017].

Killourhy, K. S. and Maxion, R. A. (2009). Comparing anomaly-detection algorithms for keystroke dynamics. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 125–134.

Loe, E. L., Hsiao, H. C., Kim, T. H. J., Lee, S. C., and Cheng, S. M. (2016). Sandusb: An installation-free sandbox for usb peripherals. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 621–626.

McGraw, G. (2013). Thirteen principles to ensure enterprise system security. [Available] http://searchsecurity.techtarget.com/opinion/Thirteen-principles-to-ensure-enterprise-system-security [Access 30-07-2017].

SRLabs (2014). Badusb: On accessories that turn evil. [Availbale] https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf [Access 30-07-2017].

Tian, D. J., Bates, A., and Butler, K. (2015). Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 261–270, New York, NY, USA. ACM.

Trojahn, M. and Ortmeier, F. (2013). Toward mobile authentication with keystroke dynamics on mobile phones and tablets. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 697–702.

© 2017 Sociedade Brasileira de Computação